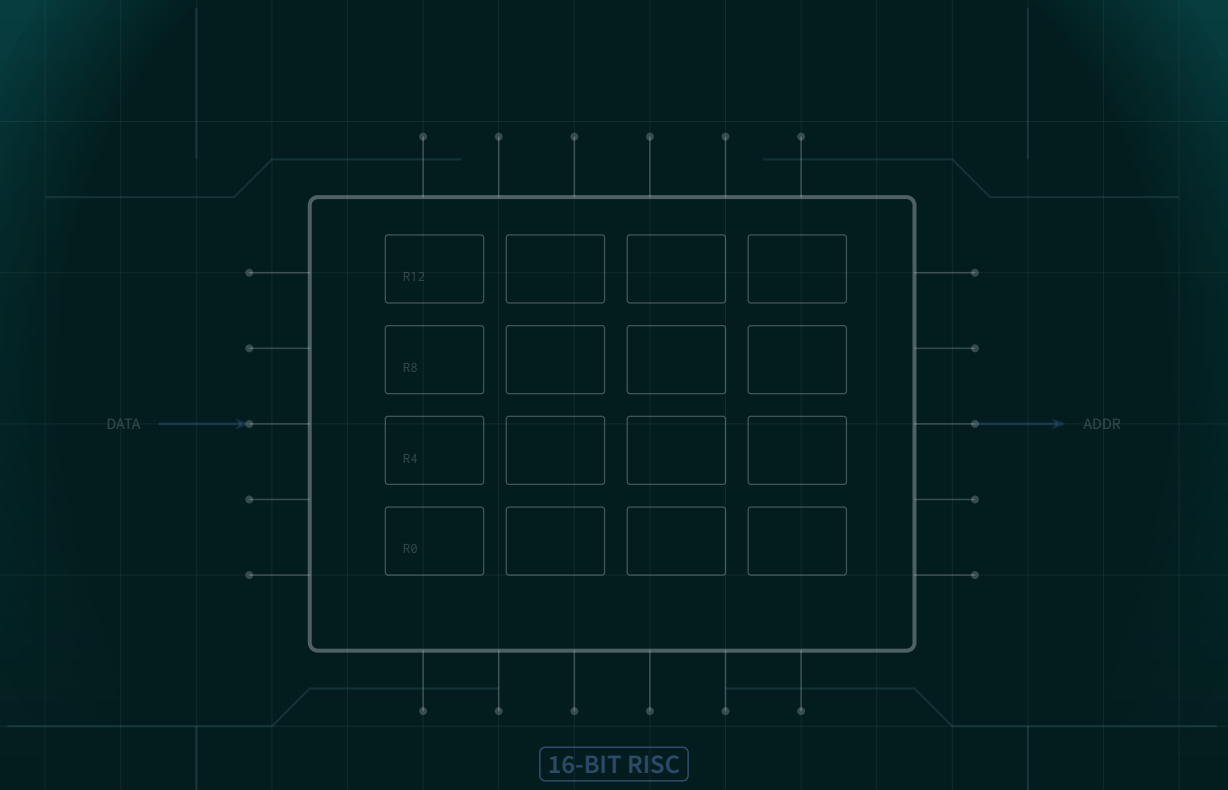
The Sampo CPU Handbook

Designing, Programming, and Building a 16-bit RISC CPU



Alex Jokela

March 2026 | Sampo CPU v1.0

The Sampo CPU Handbook
Designing, Programming, and Building a 16-bit RISC CPU

Copyright © 2026 Alex Jokela.
All rights reserved.

Covers Sampo CPU v1.0.

First edition, March 2026.

The Sampo CPU source code is available at:
https://github.com/ajokela/sampo

Contents

I Welcome to Sampo 1

1 Why Build a CPU? 3

1.1 The Case for Building Your Own CPU . . . 3

1.1.1 The Abstraction Problem . . . 4

1.1.2 Why Not Start with an Existing Architecture? . . . 4

1.1.3 What Kind of Project Is This? . . . 4

1.2 What You'll Learn . . . 6

1.2.1 Assembly Language Programming . . . 6

1.2.2 Systems Programming in Rust . . . 6

1.2.3 Hardware Description with Verilog . . . 7

1.2.4 FPGA Implementation . . . 8

1.3 The Sampo Project Overview . . . 8

1.3.1 About the Name . . . 8

1.3.2 Design Goals . . . 9

1.3.3 Project Structure . . . 9

1.3.4 The Architecture at a Glance . . . 10

1.4 Z80 Programmer's RISC: Bridging CISC and RISC . . . 11

1.4.1 The CISC–RISC Spectrum . . . 11

1.4.2 Feature-by-Feature Comparison . . . 11

1.4.3 The Two-Tier Encoding Strategy . . . 11

1.4.4 What Stays from the Z80 . . . 12

1.4.5 What Gets Cleaned Up . . . 12

1.5 How to Use This Book . . . 13

1.5.1 Path 1: The Programmer . . . 13

1.5.2 Path 2: The Hardware Designer . . . 13

1.5.3 Path 3: The Full Stack . . . 14

1.5.4 Prerequisites . . . 14
1.5.5 Conventions Used in This Book . . . 14
1.6 Exercises . . . 15

2 The Sampo Architecture 17
2.1 The Register File . . . 18
2.1.1 Register Layout and Conventions . . . 18
2.1.2 R0: The Hardwired Zero Register . . . 18
2.1.3 R1–R3: Architectural Registers . . . 19
2.1.4 R4–R15: Working Registers . . . 19
2.1.5 Special Registers . . . 20
2.2 The Alternate Register Bank . . . 21
2.2.1 What Gets Swapped . . . 21
2.2.2 Hardware Implementation . . . 21
2.2.3 Interrupt Handler Pattern . . . 22
2.3 Memory Model . . . 23
2.3.1 Address Space . . . 23
2.3.2 Memory Map . . . 23
2.3.3 Memory Interface Signals . . . 23
2.3.4 Word Alignment . . . 24
2.4 Instruction Formats . . . 25
2.4.1 Format R: Register–Register Operations . . . 25
2.4.2 Format I: Immediate Operations . . . 25
2.4.3 Format S: Store Operations . . . 26
2.4.4 Format B: Branch Operations . . . 26
2.4.5 Format J: Jump Operations . . . 27
2.4.6 Format X: Extended Instructions (32-bit) . . . 28
2.4.7 The Primary Opcode Map . . . 29
2.5 The FLAGS Register . . . 29
2.5.1 Which Instructions Set Which Flags . . . 29
2.5.2 The Carry Convention for Subtraction . . . 30
2.5.3 Flag Access Instructions . . . 31
2.6 Addressing Modes . . . 31
2.6.1 Register Direct . . . 31
2.6.2 Immediate . . . 32
2.6.3 Register Indirect with Offset . . . 32
2.6.4 PC-Relative . . . 32
2.7 Comparison: Z80 vs. RISC-V vs. ARM Thumb vs. Sampo . . . 33
2.7.1 Instruction Encoding Philosophy . . . 33
2.7.2 Register File Comparison . . . 33

2.7.3 I/O and Memory Model . . . 34
2.7.4 Where Sampo Sits . . . 34
2.8 Exercises . . . 35

3 Setting Up Your Workshop 37
3.1 Installing Rust and Cargo . . . 37
3.1.1 Installing rustup . . . 38
3.1.2 Building the Assembler . . . 38
3.1.3 Building the Emulator . . . 39
3.2 Installing Icarus Verilog and GTKWave . . . 40
3.2.1 macOS (Homebrew) . . . 40
3.2.2 Ubuntu/Debian Linux . . . 40
3.2.3 Verifying the Installation . . . 40
3.3 Installing the FPGA Toolchain . . . 41
3.3.1 The OSS CAD Suite . . . 41
3.3.2 Individual Tool Installation . . . 41
3.3.3 Verifying the FPGA Tools . . . 42
3.3.4 The ULX3S Board . . . 42
3.4 Building the Project from Source . . . 43
3.4.1 Building the Software Tools . . . 43
3.4.2 Building the Verilog Simulation . . . 43
3.4.3 The Makefile at a Glance . . . 44
3.5 Running hello.s End to End . . . 44
3.5.1 Step 1: Assemble . . . 44
3.5.2 Step 2: Run in the Emulator . . . 45
3.5.3 Step 3: Convert to Verilog Hex Format . . . 46
3.5.4 Step 4: Simulate in Verilog . . . 46
3.5.5 Step 5: View the Waveforms . . . 47
3.5.6 Step 6: Run on the FPGA (Optional) . . . 48
3.5.7 The Complete Pipeline . . . 48
3.6 Exercises . . . 49

II Programming the Sampo 51

4 Hello, Sampo! 53
4.1 Anatomy of a Sampo Assembly Program . . . 53
4.1.1 The Big Picture . . . 54
4.1.2 Registers Used in This Program . . . 55
4.2 The Assembler: sasm Directives . . . 55

4.2.1 .org — Set Origin Address . . . 56
4.2.2 .equ — Define a Constant . . . 56
4.2.3 .db — Define Bytes . . . 56
4.2.4 .dw — Define Words . . . 56
4.2.5 .ascii and .asciz — Define Strings . . . 57
4.3 Labels, Comments, and Expressions . . . 57
4.3.1 Labels . . . 57
4.3.2 Comments . . . 58
4.3.3 Numeric Literals . . . 58
4.3.4 Register Names and Aliases . . . 59
4.4 Writing "Hello, Sampo!" . . . 59
4.4.1 Setting Up: Loading the String Address . . . 59
4.4.2 The Character Loop . . . 60
4.4.3 The Transmit-Ready Busy Wait . . . 60
4.4.4 Sending the Character . . . 61
4.4.5 Halting . . . 61
4.4.6 The String Data . . . 62
4.5 Running in the Emulator . . . 62
4.5.1 Assembling and Running . . . 62
4.5.2 Trace Mode . . . 63
4.5.3 Interactive Mode . . . 63
4.6 The TUI Debugger . . . 64
4.6.1 Starting the TUI . . . 64
4.6.2 TUI Controls . . . 65
4.6.3 Stepping Through Hello, Sampo! . . . 65
4.6.4 What the TUI Reveals . . . 66
4.7 Exercises . . . 66

5 Arithmetic, Logic, and Flags 69
5.1 Three-Operand Arithmetic: ADD, SUB, ADC, SBC . . . 69
5.1.1 ADD — Add Two Registers . . . 69
5.1.2 SUB — Subtract Two Registers . . . 70
5.1.3 ADC — Add with Carry . . . 71
5.1.4 SBC — Subtract with Carry . . . 71
5.2 Bitwise Operations: AND, OR, XOR, NOT, NEG . . . 72
5.2.1 AND — Bitwise AND . . . 72
5.2.2 OR — Bitwise OR . . . 73
5.2.3 XOR — Bitwise Exclusive OR . . . 73
5.2.4 NOT — Bitwise Complement . . . 74
5.2.5 NEG — Two's Complement Negate . . . 74

5.3 Immediate Arithmetic: ADDI . . . 75
5.3.1 Common ADDI Idioms . . . 76
5.4 The FLAGS Register in Depth . . . 77
5.4.1 Flag Layout . . . 77
5.4.2 Which Instructions Affect Which Flags . . . 77
5.4.3 Reading and Writing Flags Directly . . . 78
5.5 Carry Semantics: ARM-Style Inverted Carry . . . 79
5.5.1 Carry on Addition . . . 79
5.5.2 Carry on Subtraction: Inverted Borrow . . . 79
5.5.3 Why This Convention Matters . . . 80
5.5.4 Overflow Flag (V) . . . 80
5.6 CMP and TEST . . . 81
5.6.1 CMP — Compare . . . 81
5.6.2 TEST — Bitwise Test . . . 82
5.7 Worked Examples . . . 83
5.7.1 Absolute Value . . . 83
5.7.2 Maximum of Two Values . . . 84
5.7.3 16-Bit Population Count . . . 84
5.7.4 32-Bit Addition . . . 84
5.7.5 Checking a Bit Field . . . 85
5.8 Exercises . . . 85

6 Memory and the Stack 87
6.1 Load/Store Architecture . . . 87
6.1.1 LW — Load Word . . . 87
6.1.2 LB and LBU — Load Byte . . . 88
6.1.3 SW — Store Word . . . 89
6.1.4 SB — Store Byte . . . 89
6.1.5 A Complete Load/Store Round-Trip . . . 90
6.2 Offset Addressing . . . 90
6.2.1 Load with Offset . . . 90
6.2.2 Store with Offset . . . 91
6.3 Extended Load/Store: LWX and SWX . . . 92
6.3.1 LWX — Load Word Extended . . . 92
6.3.2 SWX — Store Word Extended . . . 92
6.4 Loading 16-Bit Immediates: LIX . . . 93
6.4.1 LIX vs. ADDI for Small Constants . . . 93
6.5 The Stack . . . 94
6.5.1 Stack Organization . . . 94
6.6 PUSH and POP . . . 95

6.6.1 PUSH . . . 95
6.6.2 POP . . . 95
6.6.3 PUSH/POP Round-Trip . . . 95
6.6.4 LIFO Ordering . . . 96
6.6.5 SP Preservation . . . 96
6.7 Stack Frames and Local Variables . . . 97
6.7.1 The Calling Convention . . . 97
6.7.2 Function Prologue . . . 97
6.7.3 Function Epilogue . . . 98
6.7.4 Accessing Local Variables . . . 98
6.7.5 A Complete Function Example . . . 99
6.8 Exercises . . . 100

7 Control Flow 103
7.1 Branch Instructions: All 16 Conditions . . . 103
7.1.1 Branch Encoding . . . 103
7.1.2 The Complete Condition Table . . . 104
7.1.3 Testing the Branches . . . 104
7.1.4 Signed Comparison Example: BLT and BGE . . . 105
7.1.5 Unsigned Comparison Example: BLTU and BGEU . . . 106
7.1.6 The BMI/BPL Pair . . . 107
7.1.7 Compound Conditions: BGT, BLE, BHI, BLS . . . 107
7.2 Branch Encoding . . . 108
7.2.1 The Offset Calculation . . . 108
7.2.2 The Assembler Computes Offsets Automatically . . . 109
7.3 Jump Instructions: J, JR, JALR . . . 109
7.3.1 J — PC-Relative Jump . . . 109
7.3.2 JR — Jump to Register . . . 110
7.3.3 JALR — Jump and Link Register . . . 110
7.4 Extended Jumps: JX and JALX . . . 111
7.4.1 JX — Jump Extended (Absolute) . . . 111
7.4.2 JALX — Jump and Link Extended . . . 112
7.5 Subroutines and the Calling Convention . . . 113
7.5.1 Calling a Subroutine . . . 113
7.5.2 Inside the Subroutine . . . 114
7.6 Function Prologue and Epilogue . . . 114
7.6.1 General Template . . . 114
7.6.2 Stack Frame Layout . . . 115
7.7 Worked Example: Recursive Factorial . . . 115
7.7.1 Tracing the Execution . . . 117

7.7.2 Why Recursion Works . . . 117
7.7.3 Iterative Alternative . . . 118
7.8 Exercises . . . 118

8 The Full Instruction Set 121
8.1 Shift and Rotate Instructions . . . 121
8.1.1 Shift-by-One Instructions . . . 121
8.1.2 Rotate Operations . . . 123
8.1.3 SWAP — Byte Swap . . . 123
8.1.4 Shift Carry Behavior . . . 123
8.2 Extended Shifts . . . 124
8.2.1 Shift-by-4 and Shift-by-8 . . . 124
8.2.2 Extended Shifts with Arbitrary Amounts: SLLX, SRLX, SRAX . . . 126
8.3 Multiply and Divide . . . 126
8.3.1 MUL and MULH . . . 127
8.3.2 DIV and DIVU . . . 128
8.3.3 REM and REMU . . . 129
8.3.4 DAA — Decimal Adjust . . . 129
8.4 Block Operations . . . 130
8.4.1 LDI and LDD — Block Transfer (Single Step) . . . 130
8.4.2 LDIR and LDDR — Block Transfer (Repeat) . . . 130
8.4.3 CPIR — Compare and Search . . . 131
8.4.4 FILL — Fill Memory . . . 131
8.5 I/O Instructions . . . 132
8.5.1 Immediate Port I/O: INI and OUTI . . . 132
8.5.2 Register Port I/O: IN and OUT . . . 132
8.5.3 Extended I/O: INX and OUTX . . . 132
8.6 System Instructions . . . 133
8.6.1 NOP . . . 133
8.6.2 HALT . . . 134
8.6.3 DI and EI . . . 134
8.6.4 RETI . . . 134
8.6.5 SWI — Software Interrupt . . . 134
8.6.6 SCF and CCF . . . 134
8.6.7 EXX — Exchange Register Sets . . . 135
8.7 The MC6850 ACIA . . . 136
8.7.1 ACIA Register Map . . . 137
8.7.2 Status Register Bits . . . 137
8.8 Busy-Wait UART Output . . . 137
8.9 Exercises . . . 139

III Inside the Assembler 141

9 Lexing and Parsing 143
9.1 Overview of `sasm` . . . 143
9.1.1 A Zero-Dependency Rust Assembler . . . 143
9.1.2 The Three-Stage Pipeline . . . 144
9.1.3 Command-Line Interface . . . 145
9.1.4 Module Structure . . . 146
9.2 The Lexer . . . 146
9.2.1 Token Types . . . 147
9.2.2 The Lexer Structure . . . 148
9.2.3 The Main Tokenization Loop . . . 149
9.2.4 Whitespace and Comment Handling . . . 149
9.2.5 The Token Dispatcher . . . 150
9.2.6 Number Formats . . . 151
9.2.7 The Minus Sign Ambiguity . . . 153
9.2.8 Register Names and Aliases . . . 153
9.2.9 String Literals and Escape Sequences . . . 154
9.2.10 Character Literals . . . 156
9.2.11 Directives . . . 156
9.3 The Parser . . . 157
9.3.1 Parser State . . . 157
9.3.2 Two Utility Methods: `check` and `expect` . . . 158
9.3.3 The Parse Loop . . . 158
9.3.4 Statement Parsing . . . 159
9.3.5 Parsing Directive Arguments . . . 160
9.3.6 Parsing Operands . . . 161
9.4 Abstract Syntax: Operand Types . . . 164
9.4.1 The `Operand` Enum . . . 164
9.4.2 The `Statement` Enum . . . 165
9.4.3 The `DirectiveArg` Enum . . . 165
9.4.4 The `Program` Structure . . . 166
9.4.5 Putting It Together . . . 166
9.5 Error Handling and Line Tracking . . . 167
9.5.1 Lexer Error Reporting . . . 167
9.5.2 Parser Error Reporting . . . 168
9.5.3 Error Propagation Through the Pipeline . . . 168
9.5.4 A Taxonomy of Errors . . . 169
9.6 Hands-On: Trace Through Assembling a Program . . . 169
9.6.1 Step 1: Lexing . . . 170

9.6.2 Step 2: Parsing . . . 171
9.6.3 What Happens Next . . . 172
Exercises . . . 173

10 Code Generation **175**
10.1 Three-Pass Assembly . . . 175
10.1.1 Why Three Passes? . . . 176
10.1.2 Code Generator State . . . 176
10.1.3 Pass 1: Label Collection . . . 177
10.1.4 Instruction Size Determination . . . 179
10.1.5 Pass 2: Code Emission . . . 179
10.2 Instruction Encoding . . . 180
10.2.1 R-Format: Three-Register ALU Operations . . . 180
10.2.2 I-Format: Add Immediate . . . 181
10.2.3 Load and Store Format . . . 182
10.2.4 Two-Register Operations . . . 183
10.2.5 Stack and Miscellaneous Operations . . . 184
10.2.6 System Instructions . . . 185
10.2.7 Pseudo-Instructions: NEG and NOT . . . 185
10.3 Immediate Encoding and Sign Extension . . . 186
10.3.1 Sign Extension in Hardware . . . 186
10.3.2 Truncation for Encoding . . . 186
10.4 Branch Offset Calculation . . . 187
10.4.1 Branch Encoding Format . . . 187
10.4.2 The `emit_branch` Method . . . 187
10.4.3 The 16 Branch Conditions . . . 188
10.4.4 Branch Range . . . 188
10.5 Jump Encoding . . . 189
10.5.1 Register Jumps: JR and JALR . . . 190
10.6 Extended Instruction Encoding . . . 190
10.6.1 Extended Format Layout . . . 190
10.6.2 Encoding **`LIX`**: Load Immediate Extended . . . 191
10.6.3 Encoding I/O Instructions . . . 192
10.7 Symbol Table and Forward References . . . 193
10.7.1 The Fixup Mechanism . . . 193
10.7.2 Pass 3: Applying Fixups . . . 194
10.7.3 The Offset Formula in Detail . . . 195
10.7.4 Little-Endian Byte Emission . . . 196
10.7.5 Directive Emission . . . 196
10.8 Hands-On: Encode Instructions by Hand . . . 197

10.8.1 Example 1: Simple ALU (from `examples/simple.s`) . . . 197
10.8.2 Example 2: Branch Offset (Forward Reference) . . . 198
10.8.3 Example 3: Extended Instruction (from `examples/tests/test_extended.s`) 199
10.8.4 Example 4: HALT and PUSH . . . 199
Exercises . . . 199

11 The Binary Toolchain **201**
11.1 Binary Output Format . . . 201
11.1.1 What the Assembler Produces . . . 201
11.1.2 The Origin Padding . . . 202
11.1.3 Little-Endian Byte Order . . . 202
11.2 `bin2hex.py`: Binary to Verilog Hex . . . 203
11.2.1 Why Convert? . . . 203
11.2.2 The Complete Script . . . 203
11.2.3 How It Works . . . 204
11.2.4 Output Format . . . 205
11.2.5 Integration with the RAM Module . . . 205
11.2.6 The Makefile Pipeline . . . 206
11.3 `hex_loop_patch.py`: Patching for FPGA . . . 206
11.3.1 The Problem: HALT on an FPGA . . . 206
11.3.2 The Patching Strategy . . . 207
11.3.3 The Delay Loop . . . 207
11.3.4 The Jump Encoding Challenge . . . 208
11.3.5 The Patching Process . . . 209
11.3.6 Verification Output . . . 210
11.3.7 Build Integration . . . 210
11.4 The Address-Shift Bug: A Cautionary Tale . . . 211
11.4.1 The Symptom . . . 211
11.4.2 The Root Cause . . . 211
11.4.3 The Fix . . . 211
11.4.4 Lessons Learned . . . 212
11.5 Building a Disassembler . . . 212
11.5.1 The Decoding Algorithm . . . 212
11.5.2 Sign Extension Helpers . . . 214
11.5.3 The Shared-Opcode Trap . . . 214
11.5.4 Extended Instruction Handling . . . 214
11.5.5 Output Format . . . 215
11.5.6 Round-Trip Verification . . . 216
Exercises . . . 216

IV The Emulator **217**

12 The CPU Emulation Loop **219**
12.1 Fetch–Decode–Execute in Software . . . 219
12.1.1 Fetching a Word . . . 220
12.1.2 The Run Loop . . . 221
12.2 The CPU Struct . . . 222
12.2.1 Register File . . . 223
12.2.2 Flags Register . . . 223
12.2.3 Memory . . . 224
12.2.4 Initialization and Reset . . . 225
12.3 Instruction Decoding: Opcode Dispatch . . . 226
12.3.1 Field Extraction . . . 227
12.3.2 The Opcode Map . . . 227
12.3.3 Extended Instructions . . . 227
12.4 ALU Operations with Flag Computation . . . 229
12.4.1 Addition Flags . . . 229
12.4.2 Subtraction Flags and ARM-Style Carry . . . 230
12.4.3 Logic Flags . . . 231
12.4.4 The ADDI Quirk . . . 232
12.5 Memory Operations . . . 232
12.5.1 Load Instructions . . . 232
12.5.2 Store Instructions . . . 234
12.5.3 Byte-Level Memory Access . . . 234
12.5.4 Block Transfer Instructions . . . 235
12.6 Branch and Jump Logic . . . 236
12.6.1 Conditional Branches . . . 236
12.6.2 Jump Instructions . . . 237
12.7 I/O Simulation . . . 238
12.7.1 The UART Ports . . . 239
12.7.2 Cycle Counting . . . 240
12.8 Shift and Miscellaneous Operations . . . 240
12.8.1 Multiply and Divide . . . 241
12.8.2 System Instructions . . . 242
12.9 Hands-on: Trace Execution of hello.s . . . 243
12.9.1 Running with Trace Output . . . 244
12.9.2 Stepping Through the First Iteration . . . 245
12.9.3 Final State Dump . . . 246

13 The TUI Debugger **249**

13.1 The Ratatui TUI Framework . . . 249
13.1.1 Terminal Setup and Teardown . . . 250
13.1.2 The Main Loop Structure . . . 252
13.1.3 Application State . . . 252
13.2 The Layout System . . . 253
13.3 Register Display Panel . . . 255
13.3.1 Register Naming . . . 256
13.4 Memory View with Hex Dump . . . 257
13.5 Disassembly View . . . 258
13.5.1 The Disassembler . . . 260
13.6 The Stack Panel . . . 261
13.7 Keyboard Controls . . . 262
13.7.1 Single-Step Mechanics . . . 264
13.7.2 Reset Behavior . . . 264
13.7.3 Keyboard Input Forwarding . . . 264
13.8 Serial Output Capture . . . 265
13.8.1 The Terminal Emulator . . . 265
13.8.2 Character Processing . . . 266
13.8.3 ANSI Escape Sequence Handling . . . 267
13.8.4 Output Buffering and Flow Control . . . 268
13.8.5 Rendering the Terminal Panel . . . 269
13.9 The Status Bar . . . 270
13.9.1 Performance Metrics . . . 271

14 Peripheral Simulation **273**
14.1 I/O Port Abstraction . . . 273
14.1.1 The Port Array . . . 274
14.1.2 Port Access Instructions . . . 274
14.1.3 Extended Port Instructions . . . 276
14.2 UART Simulation . . . 276
14.2.1 The Status Register . . . 276
14.2.2 The TX Data Path . . . 278
14.2.3 The RX Data Path . . . 278
14.2.4 TUI Integration . . . 279
14.3 Adding New Peripherals . . . 280
14.3.1 The Peripheral Pattern . . . 280
14.3.2 Example: A Simple Timer . . . 281
14.3.3 Example: GPIO Port . . . 283
14.4 Connecting to Host I/O . . . 284
14.4.1 Stdout and Stdin . . . 284

14.4.2 File-Backed Peripherals . . . 284
14.4.3 The Default Port Handler . . . 285
14.4.4 Port Address Map Summary . . . 285
14.5 Peripheral Design Principles . . . 286
14.5.1 Match the Hardware Interface . . . 286
14.5.2 Keep Reads Side-Effect-Free (Mostly) . . . 286
14.5.3 Document the Register Map . . . 286
14.5.4 Test with Real Programs . . . 287

V Digital Logic Foundations 289

15 Combinational Logic 291
15.1 Logic Gates: The Primitive Building Blocks . . . 291
15.1.1 The NOT Gate (Inverter) . . . 291
15.1.2 The AND Gate . . . 292
15.1.3 The OR Gate . . . 293
15.1.4 The XOR Gate . . . 293
15.1.5 The NAND and NOR Gates . . . 294
15.2 Boolean Algebra and Truth Tables . . . 294
15.2.1 Fundamental Laws . . . 295
15.2.2 De Morgan's Theorems . . . 295
15.2.3 Truth Tables and Minterm Expansion . . . 295
15.2.4 Karnaugh Maps . . . 296
15.3 Multiplexers and Decoders . . . 297
15.3.1 The Multiplexer . . . 297
15.3.2 The Decoder . . . 298
15.4 Adders: From Half to Ripple-Carry . . . 300
15.4.1 The Half Adder . . . 300
15.4.2 The Full Adder . . . 301
15.4.3 The Ripple-Carry Adder . . . 301
15.4.4 Subtraction via Two's Complement . . . 302
15.5 The Arithmetic Logic Unit Concept . . . 303
15.5.1 ALU Architecture . . . 303
15.5.2 Operations and the Select Signal . . . 304
15.5.3 Flag Generation . . . 305
15.5.4 A Complete Combinational Pipeline . . . 306
15.6 Exercises . . . 307

16 Sequential Logic 309

16.1 Flip-Flops: The Memory Element . . . 309
16.1.1 The SR Latch . . . 309
16.1.2 The D Flip-Flop . . . 310
16.1.3 The JK Flip-Flop . . . 311
16.2 Registers: Groups of Flip-Flops . . . 312
16.2.1 A 16-Bit Register . . . 312
16.2.2 The SAMPO Register File . . . 312
16.3 Counters and Shift Registers . . . 314
16.3.1 Binary Counters . . . 314
16.3.2 Shift Registers . . . 315
16.4 Clock Domains and Timing . . . 316
16.4.1 The System Clock . . . 316
16.4.2 Clock Edges and Synchronous Design . . . 316
16.4.3 Multiple Clock Domains . . . 317
16.5 Setup and Hold Time . . . 318
16.5.1 The Timing Constraint . . . 318
16.5.2 Critical Paths in the SAMPO . . . 318
16.5.3 Hold Time and Clock Skew . . . 319
16.6 RAM: SRAM Cells and Address Decoding . . . 319
16.6.1 SRAM Cells . . . 319
16.6.2 The SAMPO RAM Module . . . 320
16.6.3 RAM Capacity and Organization . . . 322
16.7 Exercises . . . 322

17 Verilog Fundamentals 325
17.1 Module Structure: Ports, Wires, and Regs . . . 325
17.1.1 Module Declaration . . . 325
17.1.2 Wires vs. Regs . . . 326
17.1.3 Internal Signals . . . 327
17.1.4 Module Instantiation . . . 327
17.2 Continuous Assignment vs. Procedural Blocks . . . 328
17.2.1 Continuous Assignment . . . 328
17.2.2 Procedural Blocks . . . 329
17.3 Blocking vs. Non-Blocking Assignment . . . 331
17.3.1 Blocking Assignment (=) . . . 331
17.3.2 Non-Blocking Assignment (<=) . . . 331
17.4 Combinational vs. Sequential Always Blocks . . . 333
17.4.1 Combinational Always Blocks . . . 333
17.4.2 Sequential Always Blocks . . . 334
17.5 Parameters and Generate Statements . . . 335

17.5.1 Parameters . . . 335
17.5.2 The Package Header . . . 337
17.5.3 Generate Statements . . . 338
17.6 The Icarus Verilog Simulator . . . 339
17.6.1 Compilation and Execution . . . 339
17.6.2 Using the Makefile . . . 339
17.7 Testbench Structure . . . 341
17.7.1 Anatomy of a Testbench . . . 341
17.7.2 The Check Task . . . 342
17.7.3 Test Stimulus . . . 342
17.7.4 The System Testbench . . . 343
17.8 Waveform Viewing with GTKWave . . . 345
17.8.1 Generating Waveform Data . . . 345
17.8.2 Launching GTKWave . . . 345
17.8.3 Reading Waveforms . . . 345
17.8.4 Debug Tracing . . . 346
17.9 Exercises . . . 347

VI Building the CPU in Verilog 349

18 The Arithmetic Logic Unit 351
18.1 ALU Design Overview . . . 351
18.1.1 The Datapath at a Glance . . . 352
18.2 Operation Selection via Function Code . . . 352
18.3 Addition and Subtraction with Carry . . . 354
18.3.1 Extended-Width Intermediate Signals . . . 354
18.3.2 The ADD Operation . . . 355
18.3.3 The SUB Operation and ARM-Style Carry . . . 355
18.3.4 Multiplication and Division . . . 356
18.4 Flag Generation: N, Z, C, V . . . 358
18.4.1 The N and Z Flags . . . 358
18.4.2 The C Flag . . . 359
18.4.3 The V Flag . . . 359
18.5 Overflow Detection . . . 359
18.5.1 Addition Overflow . . . 359
18.5.2 Subtraction Overflow . . . 360
18.5.3 The Overflow and Branch Connection . . . 361
18.6 Pass-Through and Bitwise Operations . . . 361
18.6.1 Pass-Through: PASS_A and PASS_B . . . 361

18.6.2 Bitwise AND, OR, and XOR . . . 362
18.6.3 NOT: Bitwise Complement . . . 363
18.6.4 NEG: Two's Complement Negation . . . 363
18.6.5 Shift Operations in the ALU . . . 364
18.7 Testing the ALU . . . 365

19 The Register File 369
19.1 Register Array Design . . . 369
19.2 Read and Write Ports . . . 371
19.2.1 Dual Read Ports . . . 371
19.2.2 The General Write Port . . . 371
19.2.3 The Dedicated SP Write Port . . . 372
19.3 Ro Hardwired to Zero . . . 372
19.4 EXX Bank Switching . . . 374
19.4.1 The Swap Mechanism . . . 374
19.4.2 What EXX Does Not Touch . . . 375
19.4.3 EXX vs. Regular Writes . . . 375
19.4.4 Verifying the Double Swap . . . 376
19.5 Synchronous Write, Combinational Read . . . 377
19.5.1 The Read-After-Write Question . . . 377
19.5.2 Reset Behavior . . . 377
19.5.3 FPGA Synthesis Considerations . . . 378

20 The Shifter 381
20.1 Barrel Shifter Design . . . 381
20.1.1 The 16 Shift Functions . . . 382
20.2 Shift Functions: SLL, SRL, SRA . . . 383
20.2.1 Shift Left Logical (SLL) . . . 384
20.2.2 Shift Right Logical (SRL) . . . 385
20.2.3 Shift Right Arithmetic (SRA) . . . 385
20.3 Rotate Functions: ROL, ROR . . . 386
20.3.1 Rotate Left (ROL) . . . 386
20.3.2 Rotate Right (ROR) . . . 387
20.3.3 Rotate Through Carry: RCL and RCR . . . 387
20.4 SWAP: Byte Swap . . . 388
20.5 Carry Output for Each Operation . . . 389
20.5.1 Carry as a Multi-Word Shift Connector . . . 390
20.6 Integration with the CPU . . . 391

21 The Instruction Decoder 393

21.1 Opcode Field Extraction . . . 393
21.2 Control Signal Generation . . . 395
21.3 Register Selection: Rd, Rs1, Rs2 . . . 397
21.4 Immediate Extraction and Sign Extension . . . 398
21.4.1 The 8-Bit Signed Immediate . . . 398
21.4.2 The Branch Offset . . . 398
21.4.3 The Jump Offset . . . 399
21.5 Extended Instruction Detection . . . 399
21.6 Load/Store and Branch/Jump Control . . . 401
21.6.1 Load Instructions . . . 401
21.6.2 Store Instructions . . . 402
21.6.3 Branch Instructions . . . 402
21.6.4 Jump Instructions . . . 403
21.7 I/O Control Signals . . . 403
21.8 The Misc Opcode: A Kitchen Drawer . . . 404
21.9 The J/JR/JALR Decode Conflict . . . 406
21.9.1 The Three Jump Forms . . . 406
21.9.2 The Conflict . . . 406
21.9.3 JR R1 as Return . . . 407
21.10 System Instructions . . . 408

22 The CPU State Machine **411**
22.1 Multi-Cycle CPU Design Philosophy . . . 411
22.2 The 8-State FSM . . . 412
22.3 State Transitions and Timing . . . 413
22.3.1 The Execute Exit Decision . . . 415
22.4 FETCH and FETCH_EXT . . . 415
22.4.1 The Fetch Cycle . . . 415
22.4.2 Fetching Extended Instructions . . . 416
22.5 EXECUTE: ALU, Branches, and Register Writes . . . 417
22.5.1 ALU Input Selection . . . 417
22.5.2 PC Update Logic . . . 418
22.5.3 Branch Condition Evaluation . . . 419
22.5.4 Flag Updates . . . 420
22.5.5 Execution Result Selection . . . 421
22.6 MEMORY and WRITEBACK . . . 422
22.6.1 The MEMORY State . . . 422
22.6.2 The WRITEBACK State . . . 424
22.7 The Pipeline Hazard: The MEMORY to FETCH Bug . . . 425
22.7.1 The Original Design . . . 425

22.7.2 The Problem . . . 426
22.7.3 The Fix . . . 426
22.8 PUSH/POP: Dual Write . . . 427
22.8.1 PUSH Execution . . . 427
22.8.2 POP Execution . . . 428
22.9 Instruction Execution Walkthroughs . . . 429
22.9.1 Walkthrough: ADD R4, R5, R6 . . . 429
22.9.2 Walkthrough: LIX R4, 0x1234 . . . 430
22.9.3 Walkthrough: SW (R5), R4 . . . 430
22.9.4 Walkthrough: BEQ (taken) . . . 431

23 System on Chip 433
23.1 SoC Architecture Overview . . . 433
23.2 Memory Bus: addr, rdata, wdata, we, be . . . 435
23.3 Synchronous RAM . . . 436
23.3.1 Memory Organization . . . 438
23.4 Address Decoding and Byte Enables . . . 438
23.4.1 Program Loading with $readmemh . . . 439
23.5 The UART Controller . . . 440
23.5.1 The Status Register . . . 440
23.5.2 Transmit Logic . . . 441
23.5.3 Receive Logic . . . 442
23.5.4 CPU Read Path . . . 443
23.6 Baud Rate Generation . . . 444
23.7 Status Register and LED Outputs . . . 444
23.8 Putting It All Together . . . 445

VII Simulation and Verification 449

24 Testbench Design 451
24.1 Testbench Anatomy . . . 451
24.1.1 The Module Shell . . . 451
24.1.2 Instantiating the DUT . . . 452
24.1.3 Clock Generation . . . 453
24.1.4 The Reset Sequence . . . 454
24.1.5 Applying Stimulus . . . 455
24.1.6 Termination and Summary . . . 456
24.2 Self-Checking Testbenches . . . 457
24.2.1 The Check Task Pattern . . . 457

24.2.2 Using the Check Task . . . 458
24.2.3 The Register File Check Pattern . . . 459
24.2.4 System-Level Self-Checking . . . 461
24.3 VCD Waveform Generation and Analysis . . . 462
24.3.1 Generating VCD Files . . . 462
24.3.2 Viewing Waveforms with GTKWave . . . 463
24.3.3 Debug Trace Output . . . 463
24.3.4 Reading VCD Signals . . . 465
24.4 Parameterized Testbenches . . . 465
24.4.1 Parameters for Configuration . . . 465
24.4.2 Parameter Overrides from the Command Line . . . 466
24.4.3 The Makefile Pattern . . . 466
24.4.4 Benefits of Parameterization . . . 467
24.5 The `tx_ready=1` Lesson . . . 467
24.5.1 The Symptom . . . 467
24.5.2 The Root Cause . . . 468
24.5.3 The Fix . . . 468
24.5.4 Capturing UART Output . . . 470
24.5.5 The Broader Lesson . . . 470
Exercises . . . 471

25 The ISA Test Suite **473**
25.1 Test Strategy . . . 473
25.2 Test Framework Helpers . . . 475
25.2.1 The `print_str` Routine . . . 475
25.2.2 The `check_eq` Routine . . . 476
25.3 UART-Based Test Reporting . . . 477
25.3.1 The Reporting Protocol . . . 477
25.3.2 The Full Output Flow . . . 478
25.3.3 Testbench Integration . . . 478
25.4 Test Coverage Table . . . 479
25.5 Writing test_alu.s . . . 481
25.5.1 Program Structure . . . 481
25.5.2 Arithmetic Tests . . . 482
25.5.3 Flag Tests . . . 483
25.5.4 String Data . . . 484
25.6 Writing test_loadstore.s: Finding the Pipeline Hazard . . . 484
25.6.1 Basic Store/Load Round-Trip . . . 485
25.6.2 Byte Access and Sign Extension . . . 485
25.6.3 Offset Addressing . . . 487

25.6.4 The Hazard That Hid Here . . . 487
25.7 Makefile Automation . . . 488
25.7.1 The Test List . . . 488
25.7.2 The Assembly Pipeline . . . 488
25.7.3 Running Individual Tests . . . 489
25.7.4 Running All Tests . . . 490
25.7.5 The Precious Directive . . . 490
25.7.6 Adding a New Test . . . 490
Exercises . . . 491

26 Debugging Hardware **493**
26.1 Common Verilog Bugs . . . 493
26.1.1 Blocking vs. Non-Blocking Assignment . . . 493
26.1.2 Incomplete Sensitivity Lists . . . 495
26.1.3 Latches from Incomplete Case Statements . . . 495
26.1.4 Uninitialized Signals . . . 497
26.1.5 Width Mismatches . . . 497
26.2 Waveform Debugging . . . 498
26.2.1 Setting Up the Waveform View . . . 498
26.2.2 The Backward Trace Method . . . 499
26.2.3 Reading the State Machine . . . 499
26.3 Signal Tracing Through the State Machine . . . 500
26.3.1 Fetch (State 1) . . . 500
26.3.2 Decode (State 3) . . . 500
26.3.3 Execute (State 4) . . . 501
26.3.4 Memory (State 5) . . . 501
26.3.5 Writeback (State 6) . . . 501
26.4 The MEMORY to WRITEBACK Fix in Detail . . . 501
26.4.1 The Symptom . . . 502
26.4.2 The Original State Machine . . . 502
26.4.3 The Timing Diagram . . . 502
26.4.4 The Waveform Evidence . . . 503
26.4.5 The Fix . . . 504
26.4.6 Why WRITEBACK Works . . . 504
26.4.7 The PUSH Side Effect . . . 505
26.5 Simulation vs. Synthesis Mismatches . . . 506
26.5.1 Initial Value Differences . . . 506
26.5.2 Memory Initialization . . . 506
26.5.3 Timing-Dependent Behavior . . . 507
26.6 Timing Violations and Metastability . . . 507

26.6.1 Setup and Hold Time . . . 507
26.6.2 Clock Domain Crossings . . . 508
26.6.3 Reset Synchronization . . . 508
26.6.4 Debugging Timing Issues . . . 509
Exercises . . . 509

VIII FPGA Implementation **511**

27 FPGA Fundamentals **513**
27.1 FPGA Architecture Overview . . . 513
27.1.1 How FPGAs Differ from Simulation . . . 514
27.2 Look-Up Tables: Any N-Input Function . . . 515
27.2.1 A Concrete Example: 2-to-1 Multiplexer . . . 515
27.2.2 LUT Utilization in the SAMPO . . . 516
27.3 Carry Chains for Arithmetic . . . 517
27.3.1 Impact on Timing . . . 518
27.4 Block RAM vs. Distributed RAM . . . 518
27.4.1 Block RAM . . . 519
27.4.2 Distributed RAM . . . 520
27.4.3 Block RAM Utilization . . . 520
27.5 Clock Management: PLLs and Dividers . . . 521
27.5.1 The Clock Distribution Network . . . 521
27.5.2 Phase-Locked Loops . . . 521
27.5.3 Our Approach: Toggle Divider . . . 522
27.6 The ECP5 Family . . . 522
27.6.1 ECP5 Architecture . . . 523
27.6.2 ECP5 Family Members . . . 523
27.6.3 ECP5 I/O . . . 523
27.6.4 ECP5 Block RAM Details . . . 524
27.6.5 ECP5 PLL Resources . . . 525
27.6.6 Open-Source Toolchain Support . . . 526

28 The ULX3S Board **529**
28.1 ULX3S Hardware Overview . . . 529
28.2 Pin Constraints File . . . 530
28.2.1 Anatomy of a Constraints File . . . 530
28.2.2 Button Constraints . . . 531
28.2.3 LED Constraints . . . 532
28.2.4 FTDI UART Constraints . . . 532

28.3 Clock Configuration . . . 533
28.3.1 From 25 MHz to 12.5 MHz . . . 533
28.3.2 Clock Domain Considerations . . . 533
28.4 FTDI USB-Serial Interface . . . 534
28.4.1 The Communication Path . . . 534
28.4.2 The Hardware UART Transmitter . . . 534
28.4.3 The Hardware UART Receiver . . . 535
28.4.4 The MC6850-Compatible Interface . . . 537
28.5 LEDs, Buttons, and GPIO . . . 538
28.5.1 LED Status Display . . . 538
28.5.2 Buttons and Reset . . . 539
28.6 JTAG Programming via openFPGALoader . . . 541
28.6.1 openFPGALoader . . . 541
28.6.2 SRAM vs. Flash Programming . . . 541
28.6.3 Verifying the Connection . . . 542

29 Synthesis and Place & Route **545**
29.1 The Open-Source FPGA Toolchain . . . 545
29.2 Yosys: RTL Synthesis . . . 546
29.2.1 The Synthesis Command . . . 546
29.2.2 What Yosys Does . . . 547
29.2.3 Reading the Synthesis Log . . . 547
29.3 nextpnr-ecp5: Placement and Routing . . . 549
29.3.1 The Place and Route Command . . . 549
29.3.2 What nextpnr Does . . . 550
29.3.3 The nextpnr Timing Model . . . 550
29.4 ecppack: Bitstream Generation . . . 551
29.5 Reading Timing Reports . . . 552
29.5.1 The Critical Path . . . 552
29.5.2 Interpreting the Critical Path . . . 552
29.5.3 Slack . . . 553
29.6 Resource Utilization . . . 553
29.6.1 Utilization Summary . . . 553
29.6.2 Where Do the LUTs Go? . . . 554
29.7 The Makefile Build Pipeline . . . 555
29.7.1 Source File Definitions . . . 555
29.7.2 FPGA Configuration . . . 556
29.7.3 The Full FPGA Pipeline . . . 556
29.7.4 The Per-Test FPGA Pipeline . . . 557
29.7.5 Build, Program, and Capture . . . 557

29.7.6 Running All Tests . . . 558

30 Running on Real Hardware 561
30.1 Programming the FPGA . . . 561
30.1.1 Observing the LEDs . . . 562
30.2 Opening the Serial Port . . . 563
30.3 Hello, Sampo! on Real Hardware . . . 564
30.4 The Looping Problem . . . 564
30.4.1 Why Not Just Open the Port First? . . . 565
30.4.2 Why Not Use a Hardware Trigger? . . . 565
30.4.3 The Solution: Make It Loop . . . 565
30.5 hex_loop_patch.py: Replacing HALT with a Delay Loop . . . 565
30.5.1 The Delay Loop . . . 565
30.5.2 Why Address `0x0000`? . . . 566
30.5.3 Finding and Replacing HALT . . . 567
30.6 The Address-Shift Bug: The Full Detective Story . . . 567
30.6.1 The Encoding Conflict . . . 567
30.6.2 The Problem . . . 568
30.6.3 The Detective Work . . . 568
30.6.4 The Fix: Conflict Avoidance . . . 569
30.6.5 The Offset Calculation . . . 570
30.7 Clock Divider Selection . . . 570
30.7.1 The Timing Constraint . . . 571
30.7.2 Divider Options . . . 571
30.7.3 PLL-Based Alternatives . . . 571
30.7.4 The Impact on UART Timing . . . 571
30.7.5 Capturing UART Output . . . 572

IX Hardware Testing on the FPGA 577

31 FPGA Test Infrastructure 579
31.1 The FPGA Testing Pipeline . . . 579
31.1.1 End-to-End Timing . . . 580
31.2 fpga_capture.py: Serial Port Communication . . . 581
31.2.1 The Iteration Detection Strategy . . . 581
31.2.2 Command-Line Interface . . . 583
31.2.3 The Per-Read Timeout . . . 584
31.3 fpga_test.py: Automated Test Execution . . . 584
31.3.1 The FTDI Problem . . . 585

31.3.2 Two Strategies for Capture . . . 585
31.3.3 Serial Port Discovery . . . 587
31.3.4 FPGA Programming . . . 588
31.3.5 Exit Code Semantics . . . 588
31.4 Makefile Targets: fpga-%, fpga-tests . . . 589
31.4.1 The ISA Test List . . . 589
31.4.2 The fpga-% Pattern Rule . . . 589
31.4.3 The fpga-tests Aggregate Target . . . 591
31.4.4 Protecting Intermediate Files . . . 592
31.5 Per-Test FPGA Builds . . . 592
31.5.1 Why Per-Test Builds? . . . 592
31.5.2 The Build Chain for a Test . . . 593
31.5.3 The Hex Loop Patch . . . 594
31.5.4 The bin2hex.py Converter . . . 596
31.6 Handling the FTDI Buffer . . . 597
31.6.1 Stale Data in the Receive Buffer . . . 597
31.6.2 The USB Re-enumeration Window . . . 597
31.6.3 Output Interleaving . . . 598
31.6.4 UART TX Buffer Overflow . . . 598
31.6.5 The Double-Read Pattern . . . 599

32 132 Tests, Zero Failures 601
32.1 Running the Complete Suite . . . 601
32.1.1 What Happens During a Run . . . 602
32.1.2 Running Individual Suites . . . 603
32.2 Interpreting the Results . . . 603
32.2.1 The Test Output Format . . . 604
32.2.2 What Each Test Suite Verifies . . . 605
32.3 Simulation vs. Hardware Differences . . . 608
32.3.1 Timing and Clock Domains . . . 608
32.3.2 Reset Behavior . . . 609
32.3.3 Memory Initialization . . . 609
32.3.4 The MEMORY→FETCH Hazard . . . 610
32.4 The Full Results Table . . . 610
32.4.1 Test Coverage Summary . . . 614
32.5 Regression Testing . . . 615
32.5.1 When to Run the Suite . . . 615
32.5.2 Simulation-First Workflow . . . 616
32.5.3 Tracking Test Results Over Time . . . 616
32.6 Continuous Integration for Hardware . . . 617

32.6.1 The CI Pipeline for SAMPO . . . 617
32.6.2 Setting Up a Lab Runner . . . 617
32.6.3 A Minimal CI Script . . . 618
32.6.4 What CI Catches . . . 619

X The Amaranth Implementation 621

33 Python HDL with Amaranth 623
33.1 What is Amaranth HDL? . . . 623
33.2 Signals, Modules, and Elaboration . . . 624
33.2.1 Signals . . . 624
33.2.2 Modules and Elaboratable . . . 626
33.2.3 Submodules . . . 626
33.2.4 Elaboration . . . 627
33.3 Combinational vs. Synchronous Domains . . . 627
33.3.1 Combinational Assignments . . . 628
33.3.2 Synchronous Assignments . . . 628
33.3.3 Conditional Logic . . . 629
33.3.4 Enumerations . . . 630
33.3.5 The FSM Helper . . . 631
33.4 The Amaranth Simulator . . . 632
33.4.1 Setting Up a Simulation . . . 632
33.4.2 Testing Combinational Logic . . . 633
33.4.3 Testing Sequential Logic . . . 634
33.4.4 VCD Waveform Output . . . 636
33.4.5 Full System Simulation . . . 636
33.5 Generating Verilog from Python . . . 637
33.5.1 Using the Verilog Back-End . . . 637
33.5.2 Understanding the Generated Output . . . 638
33.5.3 Using the RTLIL Back-End . . . 639
33.5.4 Platform-Targeted Builds . . . 639
33.6 Comparison with Direct Verilog . . . 639
33.6.1 Code Volume . . . 640
33.6.2 Safety and Error Detection . . . 640
33.6.3 Testability . . . 641
33.6.4 Synthesis Quality . . . 641
33.6.5 Learning Curve . . . 641
33.6.6 When to Choose Which . . . 641
Exercises . . . 641

What's Next . . . 642

34 Sampo in Amaranth **645**
34.1 The Amaranth CPU Module . . . 645
34.1.1 Module Interface . . . 645
34.1.2 Submodule Instantiation and Wiring . . . 646
34.1.3 The State Machine . . . 648
34.2 ALU, Register File, and Decoder . . . 651
34.2.1 The ALU . . . 651
34.2.2 The Shifter . . . 653
34.2.3 The Register File . . . 654
34.2.4 The Instruction Decoder . . . 656
34.3 Side-by-Side Comparison with Verilog . . . 659
34.3.1 Module Declaration . . . 659
34.3.2 ALU Operation: ADD . . . 659
34.3.3 State Machine Structure . . . 660
34.3.4 SoC Integration . . . 660
34.4 When to Use Amaranth vs. Verilog . . . 663
34.4.1 Choose Amaranth When… . . . 663
34.4.2 Choose Verilog When… . . . 663
34.4.3 The Hybrid Approach . . . 664
34.5 Testing with Python Testbenches . . . 664
34.5.1 Test Runner Structure . . . 664
34.5.2 Unit Testing the ALU . . . 665
34.5.3 Integration Testing the SoC . . . 666
34.5.4 Running Tests . . . 667
34.6 Generating Verilog for FPGA Synthesis . . . 669
34.6.1 Manual Verilog Generation . . . 669
34.6.2 Platform-Targeted Builds . . . 670
34.6.3 The RAM Initialisation Pattern . . . 671
34.6.4 Synthesis Results . . . 672
Exercises . . . 673
What's Next . . . 673

XI Beyond the Basics **675**

35 Performance and Pipelining **677**
35.1 Current Performance Analysis . . . 677
35.1.1 Cycle Counts per Instruction . . . 677

35.1.2 Cycles per Instruction in Practice . . . 679
35.1.3 The Clock Speed Ceiling . . . 680
35.1.4 Throughput: Where the Time Goes . . . 681
35.2 Pipelining Concepts . . . 681
35.2.1 The Laundry Analogy . . . 682
35.2.2 The Classic Five-Stage Pipeline . . . 682
35.2.3 Pipeline Timing . . . 682
35.3 Data Hazards and Forwarding . . . 683
35.3.1 Read-After-Write Hazards . . . 683
35.3.2 The Stall Solution . . . 683
35.3.3 Data Forwarding . . . 684
35.3.4 Load-Use Hazards . . . 685
35.3.5 WAW and WAR Hazards . . . 686
35.4 Control Hazards and Branch Prediction . . . 686
35.4.1 The Branch Problem . . . 686
35.4.2 Static Branch Prediction . . . 687
35.4.3 Dynamic Branch Prediction . . . 688
35.4.4 Jumps and the Delay Slot Alternative . . . 688
35.5 A Pipeline Design for SAMPO . . . 689
35.5.1 Pipeline Register Definitions . . . 689
35.5.2 The Fetch Stage . . . 689
35.5.3 The Decode Stage . . . 690
35.5.4 Execute, Memory, and Writeback . . . 691
35.5.5 Handling Extended Instructions . . . 692
35.5.6 Pipeline Control: Stalls and Flushes . . . 692
35.5.7 The **EXX** Problem . . . 693
35.5.8 The **PUSH**/**POP** Dual-Write . . . 693
35.6 A Pipeline Design for SAMPO — Putting It Together . . . 694
35.7 Area vs. Speed Tradeoffs on ECP5 . . . 695
35.7.1 Current Resource Usage . . . 695
35.7.2 Pipeline Overhead . . . 695
35.7.3 The Timing vs. Area Tradeoff . . . 696
35.7.4 When Pipelining Is Not Worth It . . . 697
35.7.5 A Pragmatic Middle Ground . . . 697
Exercises . . . 698
What's Next . . . 698

36 Towards a Compiler Target **699**
36.1 The LLVM Backend Concept . . . 699
36.1.1 Why LLVM? . . . 699

36.1.2 The Compilation Pipeline . . . 700
36.1.3 What an LLVM Backend Contains . . . 700
36.1.4 Existing Small-Target Backends . . . 700
36.2 Target Description . . . 701
36.2.1 Register Definitions . . . 701
36.2.2 Instruction Patterns . . . 703
36.2.3 The Calling Convention . . . 704
36.2.4 Handling 16-Bit Limitations . . . 706
36.2.5 The Data Layout String . . . 707
36.3 From Rust to Assembly . . . 708
36.3.1 A Minimal Rust Function . . . 708
36.3.2 A Function with Control Flow . . . 709
36.3.3 A Function with Stack Usage . . . 710
36.3.4 Instruction Selection Challenges . . . 711
36.4 no_std Rust on SAMPO . . . 712
36.4.1 Why no_std? . . . 712
36.4.2 A Minimal no_std Program . . . 712
36.4.3 The Linker Script . . . 714
36.4.4 Runtime Initialization . . . 715
36.4.5 Inline Assembly on SAMPO . . . 716
36.5 The Full Loop: `cargo build` to FPGA . . . 717
36.5.1 The Complete Build Pipeline . . . 717
36.5.2 The Target Specification . . . 717
36.5.3 Building with Cargo . . . 718
36.5.4 Synthesizing with the Program . . . 719
36.5.5 The Complete Makefile Target . . . 719
36.5.6 What About the Assembler? . . . 720
36.5.7 Practical Limitations . . . 720
36.5.8 Alternative Approaches . . . 721
Exercises . . . 722
What's Next . . . 722

A ISA Quick Reference **725**
A.1 Instruction Summary . . . 725
A.1.1 Arithmetic Instructions . . . 725
A.1.2 Logic Instructions . . . 726
A.1.3 Multiply/Divide Instructions . . . 726
A.1.4 Shift and Rotate Instructions . . . 726
A.1.5 Load Instructions . . . 727
A.1.6 Store Instructions . . . 727

A.1.7 Branch Instructions . . . 728
A.1.8 Jump Instructions . . . 728
A.1.9 Stack Instructions . . . 729
A.1.10 Block Instructions . . . 729
A.1.11 Miscellaneous Instructions . . . 729
A.1.12 I/O Instructions . . . 730
A.1.13 System Instructions . . . 730
A.1.14 Extended Instructions (32-bit) . . . 730
A.2 Register Conventions . . . 731
A.2.1 General-Purpose Registers . . . 731
A.2.2 Special Registers . . . 731
A.2.3 Alternate Register Set . . . 731
A.2.4 Block Operation Register Conventions . . . 731
A.3 Flag Effects per Instruction . . . 731
A.3.1 FLAGS Register Layout . . . 731
A.3.2 Flag Effect Summary . . . 733
A.4 Branch Condition Truth Table . . . 733
A.4.1 Branch Offset Encoding . . . 735
A.5 Memory Map . . . 736
A.5.1 Address Space Layout . . . 736
A.5.2 Memory Properties . . . 736
A.5.3 Power-On Defaults . . . 736
A.5.4 I/O Port Space . . . 736
A.5.5 UART Port Assignments . . . 737
A.6 Opcode Map . . . 737
A.7 Instruction Format Summary . . . 739
A.8 Quick Reference: At a Glance . . . 739
A.8.1 Instruction Categories . . . 739
A.8.2 Register Aliases . . . 739
A.8.3 Key Constants . . . 739

B Instruction Encoding **741**
B.1 Instruction Formats . . . 741
B.1.1 Format R: Register–Register . . . 741
B.1.2 Format I: Immediate . . . 742
B.1.3 Format S: Store . . . 742
B.1.4 Format B: Branch . . . 742
B.1.5 Format J: Jump . . . 743
B.1.6 Format X: Extended (32-bit) . . . 743
B.2 R-Format Instructions . . . 744

B.2.1 Opcode 0x0: ADD . . . 744
B.2.2 Opcode 0x1: SUB . . . 744
B.2.3 Opcode 0x2: AND . . . 744
B.2.4 Opcode 0x3: OR . . . 744
B.2.5 Opcode 0x4: XOR . . . 745
B.2.6 Opcode 0xA: Shift and Rotate . . . 745
B.2.7 Opcode 0xB: Multiply and Divide . . . 746
B.2.8 Opcode 0xC: Miscellaneous . . . 746
B.3 I-Format Instructions . . . 747
B.3.1 Opcode 0x5: ADDI . . . 747
B.4 Load and Store Instructions . . . 748
B.4.1 Opcode 0x6: Load . . . 748
B.4.2 Opcode 0x7: Store . . . 749
B.5 B-Format: Branch Instructions . . . 749
B.6 J-Format: Jump Instructions . . . 750
B.6.1 J: PC-Relative Jump . . . 750
B.6.2 JR and JALR: Register Jumps . . . 751
B.7 X-Format: Extended Instructions . . . 751
B.8 I/O Instructions . . . 752
B.8.1 Opcode 0xD: I/O . . . 752
B.9 System Instructions . . . 753
B.9.1 Opcode 0xE: System . . . 753
B.10 Encoding Map Summary . . . 754
B.11 Encoding Examples . . . 754
B.11.1 Example 1: ADD R4, R5, R6 . . . 755
B.11.2 Example 2: ADDI R4, -1 . . . 755
B.11.3 Example 3: LW R4, (R5) . . . 756
B.11.4 Example 4: BEQ +8 . . . 756
B.11.5 Example 5: PUSH R4 . . . 757
B.11.6 Example 6: LIX R4, 0x1234 . . . 758
B.11.7 Example 7: JR RA . . . 758
B.11.8 Example 8: JALX 0x0400 . . . 759
B.12 Byte Order and Memory Layout . . . 759
B.13 Quick Decoding Procedure . . . 760

C Verilog Module Reference **763**
C.1 The soc Module . . . 763
C.1.1 Parameters . . . 763
C.1.2 Port Table . . . 764
C.1.3 LED Assignment . . . 764

C.1.4 Submodule Instances . . . 764
C.1.5 Instantiation Example . . . 765
C.2 The cpu Module . . . 765
C.2.1 Parameters . . . 765
C.2.2 Port Table . . . 766
C.2.3 Internal Architecture . . . 766
C.2.4 FSM State Transitions . . . 767
C.2.5 Submodule Instances . . . 767
C.2.6 Instantiation Example . . . 767
C.3 The alu Module . . . 768
C.3.1 Port Table . . . 768
C.3.2 ALU Operations . . . 769
C.3.3 Instantiation Example . . . 769
C.4 The regfile Module . . . 770
C.4.1 Port Table . . . 770
C.4.2 Internal Storage . . . 770
C.4.3 Key Behaviors . . . 771
C.4.4 Instantiation Example . . . 771
C.5 The shifter Module . . . 771
C.5.1 Port Table . . . 772
C.5.2 Shift Functions . . . 772
C.5.3 Instantiation Example . . . 773
C.6 The decode Module . . . 773
C.6.1 Input Ports . . . 773
C.6.2 Extracted Field Outputs . . . 773
C.6.3 Computed Value Outputs . . . 773
C.6.4 Memory Control Outputs . . . 773
C.6.5 Control Flag Outputs . . . 773
C.6.6 Instantiation Example . . . 773
C.7 The ram Module . . . 775
C.7.1 Parameters . . . 775
C.7.2 Port Table . . . 775
C.7.3 Internal Organization . . . 775
C.7.4 Instantiation Example . . . 776
C.8 The uart Module . . . 776
C.8.1 Port Table . . . 776
C.8.2 I/O Port Map . . . 776
C.8.3 TX and RX Handshake . . . 776
C.8.4 Instantiation Example . . . 777
C.9 Module Hierarchy Summary . . . 777

D Assembler Reference **781**
D.1 Complete Mnemonic List . . . 781
D.1.1 Arithmetic . . . 781
D.1.2 Logic . . . 782
D.1.3 Multiply/Divide . . . 782
D.1.4 Shift and Rotate . . . 783
D.1.5 Load and Store . . . 783
D.1.6 Branch . . . 783
D.1.7 Jump . . . 784
D.1.8 Stack and Miscellaneous . . . 785
D.1.9 I/O . . . 785
D.1.10 System . . . 786
D.1.11 Extended (32-bit) Instructions . . . 786
D.2 Directive Reference . . . 786
D.2.1 `.org` — Set Assembly Origin . . . 787
D.2.2 `.equ` — Define Symbolic Constant . . . 787
D.2.3 `.db` — Define Bytes . . . 788
D.2.4 `.dw` — Define Words . . . 788
D.2.5 `.ascii` — Define ASCII String . . . 788
D.2.6 `.asciz` — Define Null-Terminated String . . . 788
D.3 Expression Syntax . . . 789
D.3.1 Numeric Literals . . . 789
D.3.2 Labels . . . 789
D.3.3 Register Names . . . 789
D.3.4 Operand Formats . . . 790
D.3.5 Comments . . . 791
D.3.6 Case Sensitivity . . . 791
D.3.7 String Escape Sequences . . . 791
D.4 Two-Pass Assembly Process . . . 791
D.5 Known Limitations and Workarounds . . . 792
D.5.1 INI/OUTI Always Emit Extended Format (4 bytes) . . . 793
D.5.2 NOT Emitted as XORIX (4 bytes) . . . 793
D.5.3 NEG Emitted as SUB . . . 793
D.5.4 No Expression Evaluation . . . 793
D.5.5 Limited Shift Mnemonics . . . 794
D.5.6 Many Extended Mnemonics Not Yet Implemented . . . 794
D.5.7 No Macro Support . . . 794
D.5.8 No .include Directive . . . 795
D.5.9 JAL Uses JALX Encoding (4 bytes) . . . 795
D.5.10 Branch Range Checking . . . 795

D.5.11 Duplicate Labels Silently Overwritten 796
D.6 Complete Assembly Example 796

E Glossary **799**
E.1 Terms A–M 799
E.2 Terms N–Z 809

Part I

Welcome to Sampo

Chapter 1

Why Build a CPU?

Every programmer, at some point, stares at a line of code and wonders: *what actually happens when this runs?* Not in the abstract sense of interpreters and compilers, but at the level of electrons flowing through silicon—what makes a processor tick? One way to answer that question is to build one yourself.

This book is the story of the SAMPO CPU: a 16-bit RISC processor designed, assembled, emulated, synthesised to hardware, and verified—start to finish. By the time you reach the final chapter, you will have written assembly language programs, built an assembler and emulator in Rust, implemented a complete CPU in Verilog, run it in simulation, and loaded it onto a real FPGA board. Along the way, you will learn how every layer of a computing system connects, from the instruction set specification all the way down to lookup tables and flip-flops.

But before we write a single line of code, let's talk about *why* you might want to do this—and what you will get out of the journey.

1.1 The Case for Building Your Own CPU

There is no shortage of CPU designs in the world. RISC-V has an open specification. ARM cores power billions of devices. x86 runs the cloud. Why would anyone design a new one?

The answer depends on your goals. If you need a production CPU for a product, you should use an existing, proven architecture. But if your goal is *understanding*—really understanding, at every level—then designing your own is the single most effective thing you can do.

1.1.1 The Abstraction Problem

Modern computing stacks are deep. A Python program calls into a C runtime, which calls operating system APIs, which trigger privileged instructions, which activate hardware state machines, which orchestrate transistor switching. Each layer exists to hide the complexity of the layer below it. That is good engineering, but it makes the whole system opaque.

Building your own CPU collapses these abstractions. You see the instruction set from both sides: as the programmer writing assembly, and as the hardware designer implementing the decoder. You see memory from both sides: as the load/store instructions in your program, and as the address bus and byte-enable signals on the wire. Nothing is hidden because you built all of it.

1.1.2 Why Not Start with an Existing Architecture?

You certainly *could* implement a RISC-V core from the specification, and many people have. The advantage of designing your own ISA is that every decision is yours to make. When you choose how many registers to have, or whether to include hardware multiply, or how to encode branch offsets, you confront the same tradeoffs that professional architects face. You learn *why* architectures are designed the way they are, not just *how* they work.

Learning by Designing

Following a specification teaches you the "what." Designing your own teaches you the "why." Both are valuable, but the insights from making your own mistakes—and debugging your own hardware—are hard to get any other way.

1.1.3 What Kind of Project Is This?

The SAMPO project is not a toy. It is a full-stack CPU implementation comprising:

- An **instruction set architecture** with approximately 66 instructions, fixed 16-bit and 32-bit encoding, and a clean register file
- An **assembler** (`sasm`), written in Rust, with a proper lexer, parser, and code generator
- An **emulator** (`semu`), also in Rust, with trace output, an interactive debugger, and a full terminal user interface
- A **Verilog RTL implementation** with ALU, register file, decoder, shifter, and CPU state machine
- An **FPGA build** targeting the ULX3S board with an ECP5 chip, using the fully open-source Yosys/nextpnr toolchain
- An **Amaranth HDL implementation** for those who prefer Python-based hardware description

- A comprehensive **test suite** with 132 tests covering every instruction, verified both in simulation and on real hardware

Here is the actual "Hello, World" program that runs on SAMPO, in hardware, on an FPGA:

Listing 1.1: The SAMPO "Hello, World" (`examples/hello.s`)

```
; Hello World for Sampo CPU
; Outputs "Hello, Sampo!" to serial port

        .org 0x0100

; Serial port definitions (MC6850 ACIA style)
.equ    ACIA_STATUS 0x80
.equ    ACIA_DATA   0x81
.equ    TX_READY    0x02

start:
        LIX  R4, message        ; Load address of message

loop:
        LBU  R5, (R4)           ; Load byte from string
        CMP  R5, R0             ; Compare with zero
        BEQ  done               ; If null, we're done

wait_tx:
        INI  R6, ACIA_STATUS    ; Read serial status
        AND  R7, R6, R6         ; Copy to R7
        ADDI R7, -2             ; Subtract TX_READY (2)
        BNE  wait_tx            ; Wait if not ready

        OUTI ACIA_DATA, R5      ; Write character
        ADDI R4, 1              ; Next character
        J    loop               ; Continue

done:
        HALT

message:
        .asciz "Hello, Sampo!\n"
```

This program uses several features that reveal SAMPO's character: the **`LIX`** extended load immediate (a 32-bit instruction), port-based I/O with **`INI`** and **`OUTI`**, the hardwired zero register `R0` for null comparison, and a **`HALT`** instruction to stop the processor. We will trace through every instruction of

this program later in the book (Chapter 4). For now, notice how readable the assembly is—that is by design.

1.2 What You'll Learn

Building a CPU from scratch is inherently multidisciplinary. This book covers four major areas, each building on the last.

1.2.1 Assembly Language Programming

You will learn to program in SAMPO assembly, from basic arithmetic and control flow to stack manipulation, subroutine calling conventions, block memory operations, and interrupt handling. The assembly language is designed to be approachable: if you have programmed in any language, you will find SAMPO assembly readable within a few pages.

The instruction set draws from the best of both worlds. Like RISC-V, it has a clean, orthogonal register file with a hardwired zero register. Like the Z80, it includes port-based I/O, alternate registers, and block copy instructions. The combination is surprisingly practical.

1.2.2 Systems Programming in Rust

The SAMPO assembler and emulator are written in Rust. You will see how a real assembler works from the inside—lexical analysis, parsing, symbol resolution, and binary code generation—and how a CPU emulator implements the fetch-decode-execute cycle in software. The assembler's pipeline mirrors what happens in hardware:

Listing 1.2: The assembler pipeline (`sasm/src/main.rs`)

```
// Lexical analysis
let mut lexer = Lexer::new(&source);
let tokens = match lexer.tokenize() {
    Ok(t) => t,
    Err(e) => {
        eprintln!("Lexer error: {}", e);
        std::process::exit(1);
    }
};

// Parsing
let mut parser = Parser::new(tokens);
let program = match parser.parse() {
    Ok(p) => p,
    Err(e) => {
        eprintln!("Parser error: {}", e);
```

```
        std::process::exit(1);
    }
};

// Code generation
let mut codegen = CodeGen::new();
let binary = match codegen.generate(&program) {
    Ok(b) => b,
    Err(e) => {
        eprintln!("Code generation error: {}", e);
        std::process::exit(1);
    }
};
```

Source text goes in; binary machine code comes out. The emulator then loads that binary and runs it:

Listing 1.3: The emulation loop (`semu/src/main.rs`)

```
fn run(cpu: &mut Cpu) {
    loop {
        match cpu.step() {
            Ok(true) => {} // Continue
            Ok(false) => {
                println!("\nCPU halted at 0x{:04X}", cpu.get_pc());
                break;
            }
            Err(e) => {
                eprintln!("\nError at 0x{:04X}: {}", cpu.get_pc(), e);
                cpu.dump_state();
                std::process::exit(1);
            }
        }
    }
    cpu.dump_state();
}
```

Each call to `cpu.step()` fetches an instruction from memory, decodes it, executes it, and updates the CPU state—the same sequence the hardware performs, expressed in Rust instead of Verilog.

1.2.3 Hardware Description with Verilog

The heart of this book is the Verilog implementation of SAMPO. You will learn digital logic fundamentals—combinational and sequential circuits—and then apply them to build each module of the CPU:

- The **ALU** (`verilog/rtl/alu.v`) performs arithmetic, logic, and multiply/divide operations
- The **register file** (`verilog/rtl/regfile.v`) manages 16 registers plus alternate bank storage
- The **shifter** (`verilog/rtl/shifter.v`) handles shifts, rotates, and byte-swap operations
- The **decoder** (`verilog/rtl/decode.v`) translates instruction bits into control signals
- The **CPU core** (`verilog/rtl/cpu.v`) ties everything together with a finite state machine
- The **SoC wrapper** (`verilog/rtl/soc.v`) adds RAM and a UART for a complete system

The CPU state machine is the conductor. Every instruction passes through its states—RESET, FETCH, DECODE, EXECUTE, MEMORY, WRITEBACK—and the control signals generated at each state orchestrate the datapath. Here is how the states are defined in the actual Verilog:

Listing 1.4: CPU state definitions (`verilog/rtl/sampo_pkg.vh`)

```
// CPU states
`define ST_RESET     4'h0
`define ST_FETCH     4'h1
`define ST_FETCH_EXT 4'h2
`define ST_DECODE    4'h3
`define ST_EXECUTE   4'h4
`define ST_MEMORY    4'h5
`define ST_WRITEBACK 4'h6
`define ST_HALTED    4'h7
```

Eight states, four bits. That is the entire control spine of the processor. Each of these states will be explored in detail when we build the CPU in **??**.

1.2.4 FPGA Implementation

The final step is running SAMPO on real hardware. The book covers the full open-source FPGA workflow: synthesis with Yosys, place-and-route with nextpnr, bitstream packing with ecppack, and programming the device with openFPGALoader. You will see your CPU running on a ULX3S board, printing "Hello, Sampo!" over a real serial port.

This is not a hypothetical exercise. The SAMPO Verilog has been synthesised, loaded onto hardware, and tested with 132 individual test programs. Every instruction works. The gap between simulation and reality—which can be significant in FPGA development—is one you will learn to close.

1.3 The Sampo Project Overview

1.3.1 About the Name

The SAMPO is a magical artifact from Finnish mythology, central to the epic poem *Kalevala* compiled by Elias Lönnrot in 1835. According to legend, the Sampo was forged by Ilmarinen, a legendary

blacksmith and sky god, from a swan's feather, a grain of barley, a ball of wool, a drop of milk, and a shaft of a distaff.

The Sampo took the form of a magical mill that could produce flour, salt, and gold endlessly—bringing riches and good fortune to its holder. The exact nature of the Sampo has been debated by scholars since 1818, with over 30 theories proposed, ranging from a world pillar to an astrolabe to a decorated shield. This mystery makes it a fitting namesake for a CPU architecture: something that transforms inputs into useful outputs, whose inner workings invite exploration.

1.3.2 Design Goals

The SAMPO architecture was designed with five guiding principles:

1. **RISC-inspired instruction set**: Clean, orthogonal design with Z80-friendly extensions
2. **16-bit native word size**: Registers, ALU, and memory addressing are all 16 bits wide
3. **Efficient for interpreters and compilers**: Stack operations, indirect addressing, and block operations make it practical for real workloads
4. **Straightforward to implement**: Suitable for both FPGA hardware and software emulation
5. **Z80-workload compatible**: Port-based I/O, BCD arithmetic, block operations, and alternate registers enable porting retro computing programs

These goals pull in different directions. A pure RISC design would not include block copy instructions or BCD support. A Z80 clone would have far more addressing modes and implicit register usage. SAMPO finds a middle ground: RISC regularity where it matters for implementation clarity, with targeted CISC conveniences where they enable practical workloads.

1.3.3 Project Structure

The SAMPO project is organised into several directories, each representing a distinct layer of the system:

Table 1.1: The SAMPO project directory structure

Directory	Language	Purpose
`sasm/`	Rust	Assembler: lexer, parser, code generator
`semu/`	Rust	Emulator: CPU core, TUI, interactive debugger
`verilog/rtl/`	Verilog	Hardware: ALU, register file, decoder, CPU, SoC
`verilog/tb/`	Verilog	Testbenches for simulation
`verilog/fpga/`	Verilog	FPGA-specific modules (UART, reset, top-level)
`rtl/`	Python	Amaranth HDL implementation
`examples/`	Assembly	Example programs

The assembler transforms source code into binaries. The emulator runs those binaries in software. The Verilog implementation runs them in hardware. All three consume the same binary format, so a program assembled once can be tested in all three environments.

One Binary, Three Targets

A single `sasm examples/hello.s -o hello.bin` produces a binary that runs identically in the Rust emulator, the Verilog simulation, and on a physical FPGA. This is the power of a clean architecture: the software and hardware agree because they implement the same specification.

1.3.4 The Architecture at a Glance

Before we dive deep into the architecture in Chapter 2, here is a high-level summary:

- **Word size**: 16 bits
- **Registers**: 16 general-purpose (`R0`–`R15`), with `R0` hardwired to zero
- **Address space**: 64KB, byte-addressable, little-endian
- **Instruction encoding**: 16-bit base, 32-bit extended (prefix `0xF`)
- **Instruction count**: Approximately 66, plus extended 32-bit forms
- **I/O**: 256 ports, port-based (separate from memory address space)
- **Flags**: N (Negative), Z (Zero), C (Carry), V (Overflow), H (Half-carry), I (Interrupt enable)

The register conventions follow a RISC-V-inspired pattern:

Table 1.2: SAMPO register conventions

Register	Alias	Convention	EXX Swap?
`R0`	`ZERO`	Always zero	No
`R1`	`RA`	Return address	No
`R2`	`SP`	Stack pointer	No
`R3`	`GP`	Global pointer	No
`R4`–`R7`	`A0`–`A3`	Arguments/returns	Yes
`R8`–`R11`	`T0`–`T3`	Temporaries	Yes
`R12`–`R15`	`S0`–`S3`	Saved (callee)	No

Notice the **EXX Swap** column. The **`EXX`** instruction swaps only `R4`–`R11` with their shadow copies. The zero register, return address, stack pointer, global pointer, and callee-saved registers are never touched. This deliberate choice means an interrupt handler can **`EXX`** into the alternate bank, do its work using eight fresh registers, **`EXX`** back, and **`RETI`**—without saving or restoring a single register on the stack.

1.4 Z80 Programmer's RISC: Bridging CISC and RISC

SAMPO occupies an intentional middle ground between classic CISC and modern RISC designs. To understand why, it helps to see where it sits on the spectrum.

1.4.1 The CISC–RISC Spectrum

The Z80, released in 1976, is a quintessential CISC processor. It has over 300 instructions, variable-length encoding (1 to 4 bytes), many addressing modes, and registers with implicit roles (the A register is always the accumulator, HL is always the memory pointer for block operations). This design made hand-written assembly efficient but hardware complex.

Modern RISC processors—MIPS, RISC-V, ARM—went the other direction. Fewer instructions, fixed-width encoding, orthogonal registers, and load/store memory access. This makes the hardware straightforward to pipeline and verify, at the cost of occasionally needing more instructions to express the same operation.

SAMPO takes the RISC foundation and adds targeted CISC features where they provide genuine value:

A Z80 Programmer's RISC

SAMPO is a load/store RISC architecture with fixed 16/32-bit instruction encoding and an orthogonal register file, augmented with Z80-inspired features: port-based I/O, alternate registers (**`EXX`**), block memory operations (**`LDIR`**, **`LDDR`**, **`FILL`**), and BCD support (**`DAA`**). The result is an architecture that is clean enough to implement in straightforward hardware, yet familiar and practical for programmers coming from the 8-bit world.

1.4.2 Feature-by-Feature Comparison

The following table shows how SAMPO relates to its architectural ancestors:

1.4.3 The Two-Tier Encoding Strategy

One of SAMPO's most distinctive features is its encoding system. Unlike pure 32-bit RISC architectures (where every instruction is 4 bytes) or fully variable-length CISC architectures (where you need complex state machines to determine instruction boundaries), SAMPO uses a disciplined two-tier approach:

- **16-bit instructions** for common operations: ALU, branches, loads, stores, shifts
- **32-bit instructions** for operations requiring full 16-bit immediates, always signalled by a `0xF` prefix in the top nibble

Table 1.3: Architectural comparison: Z80, RISC-V, ARM Thumb, and SAMPO

Feature	Z80	RISC-V	ARM Thumb	SAMPO
Word size	8-bit	32/64-bit	32-bit	16-bit
Instruction width	1–4 B	4 B (RV32I)	2/4 B	2/4 B
Registers	8 + alt	32	16	16 + alt
Zero register	No	`x0`	No	`R0`
I/O model	Port	Memory-mapped	Memory-mapped	Port
Block operations	Yes	No	No	Yes
Hardware multiply	No	M extension	Yes	Yes
BCD support	DAA	No	No	DAA
Instruction count	∼300+	∼50 base	∼100	∼66

The decoder checks one nibble to determine instruction length. If bits 15:12 are 1111, the instruction is 32 bits wide and the CPU fetches a second word. Otherwise, it is 16 bits. No alignment analysis, no look-ahead buffer, no variable-length complexity.

This echoes ARM's Thumb-2 approach, where 16-bit and 32-bit instructions coexist, but SAMPO's mechanism is even more regular: there is exactly one prefix value (`0xF`) for all extended instructions, and the sub-opcode in bits 3:0 of the first word determines the operation.

1.4.4 What Stays from the Z80

The Z80 features that SAMPO retains are not arbitrary. Each one addresses a real need:

Port-based I/O (`IN`/`OUT`)
: Separating I/O from memory simplifies the memory map and makes peripheral access explicit. The 256-port address space maps naturally to UART controllers, SPI peripherals, and GPIO.

Alternate registers (`EXX`)
: Fast context switching without stack overhead. An interrupt handler can swap to a fresh set of eight registers in a single cycle, do its work, and swap back. No `PUSH`/`POP` cascade required.

Block operations (`LDIR`, `LDDR`, `FILL`, `CPIR`)
: Memory copy and fill are fundamental operations for any system that handles strings, buffers, or display memory. While a RISC purist would use a loop, hardware-assisted block operations can be significantly faster and more compact.

BCD arithmetic (`DAA`)
: Decimal Adjust After Addition supports calculator and financial applications where binary-coded decimal representation matters.

1.4.5 What Gets Cleaned Up

The things SAMPO *drops* from the Z80 tradition are equally important:

Implicit register usage
: Z80's LD instruction has dozens of forms with implicit source and destination registers. SAMPO uses explicit three-operand instructions: `ADD Rd, Rs1, Rs2` always writes to `Rd`, reads from `Rs1` and `Rs2`. No guessing.

Complex addressing modes
: Z80 supports indexed, relative, indirect, and combinations thereof. SAMPO has four addressing modes: register direct, immediate, register indirect with offset, and PC-relative. Four is enough.

Variable-length encoding
: Z80 instructions range from 1 to 4 bytes with prefix bytes that modify behaviour. SAMPO instructions are 2 or 4 bytes, period. The hardware never has to speculatively fetch additional bytes.

The result is an architecture that a Z80 programmer will recognise but a RISC-V designer will respect.

1.5 How to Use This Book

This book is written as a sequential narrative: each chapter builds on the previous ones. However, not every reader needs every chapter. The book is organised into eleven Parts, and there are three natural reading paths depending on your background and goals.

1.5.1 Path 1: The Programmer

If your primary interest is assembly language programming and software tools:

1. **Part I** (Chapters 1–3): Architecture overview and environment setup
2. **Part II** (Chapters 4–8): Programming the SAMPO in assembly
3. **Part III** (Chapters 9–11): Inside the assembler
4. **Part IV** (Chapters 12–14): The emulator and debugger

This path covers everything from writing your first program through understanding how the assembler and emulator work internally. By the end, you will be comfortable writing non-trivial assembly programs, using the debugger to trace execution, and understanding the complete software toolchain.

1.5.2 Path 2: The Hardware Designer

If you already know assembly and want to focus on the digital logic and FPGA implementation:

1. **Part I** (Chapters 1–3): Architecture overview and setup
2. **Part V** (Chapters 15–17): Digital logic foundations
3. **Part VI** (Chapters 18–23): Building the CPU in Verilog
4. **Part VII** (Chapters 24–26): Simulation and verification
5. **Part VIII** (Chapters 27–30): FPGA implementation
6. **Part IX** (Chapters 31–32): Hardware testing on the FPGA

This path takes you from gates and flip-flops through a working CPU on an FPGA board. You will build each hardware module, write testbenches, debug waveforms, and run the design on real silicon.

1.5.3 Path 3: The Full Stack

Read the entire book, front to back. This is the path for those who want to understand every layer—from the instruction set specification to the FPGA bitstream, with the assembler and emulator in between. It is the most time-intensive path, but also the most rewarding. By the end, there will be no part of the SAMPO system that you do not understand.

Suggested Approach for the Full-Stack Path

Even on the full-stack path, consider reading Part II (Programming) before Part V (Digital Logic Foundations). Writing assembly programs first gives you an intuition for what the hardware needs to do. When you later implement the ALU in Verilog, you will already know what **`ADD`** and **`SUB`** mean from the programmer's perspective.

1.5.4 Prerequisites

This book assumes:

- **Some programming experience**: You should be comfortable in at least one language. Python, C, JavaScript—it does not matter which. We will teach the Rust and Verilog you need.
- **Basic understanding of binary and hexadecimal**: You should know what `0xFF` means and be able to convert between binary and hex. If not, any introductory computer science resource will get you there in an hour.
- **A willingness to work at the bit level**: This book routinely discusses individual bits in instruction encodings, flag registers, and memory addresses. Comfort with bit manipulation will develop as you read.
- **No prior hardware experience required**: Parts V through IX assume no previous knowledge of digital logic, Verilog, or FPGAs. We build up from first principles.

1.5.5 Conventions Used in This Book

Throughout this book, we use consistent formatting to make technical content scannable:

- **Register names** appear in monospace: `R0`, `R4`, `SP`
- **Instruction mnemonics** appear in bold monospace: **`ADD`**, **`LW`**, **`BEQ`**
- **Hexadecimal values** appear in monospace: `0x0100`, `0xFF`
- **Binary values** appear in monospace: `1111`, `0000_1010`
- **I/O ports** appear in monospace: `0x80`, `0x81`
- **File paths** appear in monospace: `verilog/rtl/cpu.v`

- **FSM states** appear in small caps: FETCH, DECODE, EXECUTE
- **CLI flags** appear with dashes: `--trace`, `--tui`
- **Code listings** are drawn from actual project source files, with file path and language indicated in the caption

Coloured boxes highlight different types of information:

- **Blue boxes** contain important notes and observations
- **Grey boxes** provide formal definitions of key terms
- **Green boxes** offer practical tips and best practices
- **Amber boxes** warn about common pitfalls
- **Red boxes** flag critical warnings (especially hardware-damaging mistakes)

Code Examples Are Real

Every code listing in this book comes from the actual SAMPO project source files. If you find a discrepancy between the book and the source code, the source code is authoritative. We indicate the source file path in each listing's caption so you can always check the original.

1.6 Exercises

These exercises are designed to orient you before we dive into the architecture in the next chapter.

1. **Read the Hello World.** Open `examples/hello.s` from the SAMPO repository and read through it line by line. For each instruction, write down what you *think* it does based on the mnemonic and comments. You do not need to be right about everything—the goal is to build initial intuition. How many distinct instruction types can you identify?

2. **Count the registers.** The SAMPO architecture has 16 general-purpose registers. Using the register convention table in this chapter (Table 1.2), answer: how many registers are available for general computation in a typical function (not counting `R0`, `RA`, `SP`, and `GP`)? How many of those are caller-saved versus callee-saved?

3. **Compare instruction counts.** The Z80 has over 300 instruction encodings, while SAMPO has approximately 66 base instructions. Research the RISC-V RV32I base integer instruction set: how many instructions does it define? Where does SAMPO fall on the spectrum between Z80 and RISC-V in terms of instruction count? What might account for SAMPO having more instructions than RISC-V but far fewer than the Z80?

4. **Map the encoding.** The SAMPO uses a 4-bit primary opcode (bits 15:12 of each instruction). That gives 16 possible primary opcodes. Using the opcode map from `ARCHITECTURE.md`, list all 16 opcode values and their categories. The value `0xF` is special—what does it signal, and why is this design choice useful for a 16-bit architecture?

5. **Choose your path.** Review the three reading paths described in Section 1.5. Decide which path best matches your goals and write down the chapter numbers you plan to read. If you are unsure, start with the full-stack path—you can always skip ahead once you find your footing.

What's next. Now that you know what the SAMPO project is and why it exists, it is time to get concrete. In Chapter 2, we will lay out the full architecture in detail: the register file, the memory model, every instruction format, the FLAGS register, and addressing modes. By the end of that chapter, you will have a complete mental model of the machine you are about to build.

Chapter 2

The Sampo Architecture

Open the file `verilog/rtl/regfile.v` from the SAMPO project and look at the first few lines of the module declaration:

Listing 2.1: Register file module interface (`verilog/rtl/regfile.v`)

```
module regfile (
    input  wire        clk,
    input  wire        rst_n,
    input  wire [3:0]  rd_addr1,
    input  wire [3:0]  rd_addr2,
    input  wire [3:0]  wr_addr,
    output wire [15:0] rd_data1,
    output wire [15:0] rd_data2,
    input  wire [15:0] wr_data,
    input  wire        wr_en,
    input  wire        exx,         // Exchange R4-R11 with alternates
    // Second write port for SP updates (used by PUSH/POP)
    input  wire [15:0] sp_wr_data,
    input  wire        sp_wr_en
);
```

That is the entire interface to the SAMPO register file. Four-bit addresses, 16-bit data, an exchange signal, and a dedicated stack pointer write port. From this handful of signals, we can read the architecture's priorities: 16 registers addressed by 4 bits, a 16-bit data path, a hardware-supported register swap mechanism, and special treatment of the stack pointer. Every architectural decision described in this chapter ultimately manifests as signals on wires like these.

Let's trace through the complete architecture, from the register file to the instruction encoding to the FLAGS register, and build a precise mental model of the machine.

2.1 The Register File

The SAMPO has 16 general-purpose registers, each 16 bits wide. They are named `R0` through `R15`, and every one is visible to the programmer. Unlike the Z80, where registers have implicit roles and asymmetric capabilities, SAMPO registers are orthogonal: any register can be used as the source or destination of any instruction.

2.1.1 Register Layout and Conventions

Table 2.1: The SAMPO register file

Encoding	Register	Alias	Convention	Saved By
`0x0`	`R0`	`ZERO`	Always reads as zero	—
`0x1`	`R1`	`RA`	Return address	Caller
`0x2`	`R2`	`SP`	Stack pointer	—
`0x3`	`R3`	`GP`	Global pointer	—
`0x4`	`R4`	`A0`	Argument 0 / Return value 0	Caller
`0x5`	`R5`	`A1`	Argument 1 / Return value 1	Caller
`0x6`	`R6`	`A2`	Argument 2	Caller
`0x7`	`R7`	`A3`	Argument 3	Caller
`0x8`	`R8`	`T0`	Temporary 0	Caller
`0x9`	`R9`	`T1`	Temporary 1	Caller
`0xA`	`R10`	`T2`	Temporary 2	Caller
`0xB`	`R11`	`T3`	Temporary 3	Caller
`0xC`	`R12`	`S0`	Saved 0	Callee
`0xD`	`R13`	`S1`	Saved 1	Callee
`0xE`	`R14`	`S2`	Saved 2	Callee
`0xF`	`R15`	`S3`	Saved 3	Callee

Each register is encoded in 4 bits, fitting neatly into the instruction format. The aliases (`ZERO`, `RA`, `SP`, etc.) are recognised by the assembler and can be used interchangeably with the numbered names.

2.1.2 R0: The Hardwired Zero Register

`R0` always reads as zero. Writes to `R0` are silently discarded. This is not a convention—it is enforced in hardware. From the register file implementation:

Listing 2.2: Hardwired zero in the register file (`verilog/rtl/regfile.v`)

```
// Read ports - R0 always returns 0
assign rd_data1 = (rd_addr1 == 4'h0) ? 16'h0000 : regs[rd_addr1];
assign rd_data2 = (rd_addr2 == 4'h0) ? 16'h0000 : regs[rd_addr2];
```

And on the write side:

Listing 2.3: Write protection for R0 (`verilog/rtl/regfile.v`)

```
if (wr_en && (wr_addr != 4'h0)) begin
    // Write to register (except R0)
    regs[wr_addr] <= wr_data;
end
```

Why a Zero Register?

A hardwired zero register eliminates an entire class of instructions. Need to compare a value with zero? **CMP** R5, R0. Need to move a register? **ADD** Rd, Rs, R0. Need to negate? **SUB** Rd, R0, Rs. The zero register is a multiplier: it makes existing instructions more versatile without adding new opcodes. This idea comes from MIPS (`$zero`) and RISC-V (`x0`).

2.1.3 R1–R3: Architectural Registers

Three registers have special architectural significance:

R1 / RA — Return Address
: The **JAL** (Jump and Link) and **JALR** instructions store the return address in R1. Subroutine return is performed by **JR** RA. The calling convention requires the caller to save RA if it calls another function.

R2 / SP — Stack Pointer
: The **PUSH** and **POP** instructions implicitly use R2 as the stack pointer. The register file has a dedicated second write port (`sp_wr_data`, `sp_wr_en`) so that **POP** can update both the destination register and the stack pointer in a single clock cycle.

R3 / GP — Global Pointer
: Reserved for pointing to global data. Its use is optional and depends on the program's memory layout.

2.1.4 R4–R15: Working Registers

The remaining twelve registers are divided into three groups by calling convention:

- **R4–R7 (A0–A3)**: Function arguments and return values. The first four arguments are passed in registers; additional arguments go on the stack. Return values use `A0` (and `A1` for 32-bit returns). These are caller-saved.
- **R8–R11 (T0–T3)**: Temporaries for scratch computation. Caller-saved—a called function may freely overwrite them.
- **R12–R15 (S0–S3)**: Callee-saved registers. A function that uses these must save them on the stack in its prologue and restore them in its epilogue.

The calling convention follows RISC-V's model: arguments in low registers, saved registers in high registers, and clear ownership rules. A typical function prologue and epilogue look like this:

Listing 2.4: Function prologue and epilogue

```
; Function prologue
my_function:
    PUSH RA          ; Save return address
    PUSH S0          ; Save callee-saved registers

    ; ... function body ...

    ; Function epilogue
    POP  S0          ; Restore callee-saved
    POP  RA          ; Restore return address
    JR   RA          ; Return to caller
```

2.1.5 Special Registers

Beyond the general-purpose register file, SAMPO has three non-addressable special registers:

Table 2.2: SAMPO special registers

Register	Width	Description
`PC`	16-bit	Program counter. Points to the current instruction. Updated by the state machine on every cycle.
`FLAGS`	8-bit	Status flags register (N, Z, C, V, H, I). Modified by arithmetic, logic, and comparison instructions.
`IVR`	16-bit	Interrupt vector register. Holds the address of the interrupt service routine.

The program counter starts at the reset vector (`0x0100` by default). In the Verilog implementation, this is a parameter:

Listing 2.5: Reset vector parameter (`verilog/rtl/cpu.v`)

```
module cpu #(
    parameter RESET_VECTOR = 16'h0100
) (
    input  wire        clk,
    input  wire        rst_n,
    // ...
```

2.2 The Alternate Register Bank

One of SAMPO's most distinctive features is the alternate register bank—a shadow copy of registers `R4` through `R11`. The **`EXX`** instruction swaps the primary and alternate copies in a single clock cycle.

2.2.1 What Gets Swapped

The swap is selective. Only eight of the sixteen registers participate:

Table 2.3: EXX swap scope

Registers	Swapped by EXX?	Rationale
`R0` (ZERO)	No	Hardwired to zero
`R1`–`R3` (RA, SP, GP)	No	Must survive context switch
`R4`–`R7` (A_0–A_3)	Yes	Arguments/scratch
`R8`–`R11` (T_0–T_3)	Yes	Temporaries/scratch
`R12`–`R15` (S_0–S_3)	No	Callee-saved, stable across calls

This is a deliberate departure from the Z80, where **`EXX`** swaps *all* main registers (BC, DE, HL). By leaving the zero register, return address, stack pointer, global pointer, and saved registers untouched, SAMPO's **`EXX`** is safe to use in interrupt handlers without corrupting the call stack or return path.

2.2.2 Hardware Implementation

In the Verilog register file, the alternate bank is stored as a separate 8-entry array:

Listing 2.6: Alternate register storage (`verilog/rtl/regfile.v`)

```
// Main register file (R0-R15)
reg [15:0] regs [0:15];

// Alternate registers (R4'-R11', stored as indices 0-7)
reg [15:0] alt_regs [0:7];
```

When the exx signal is asserted, the swap is performed simultaneously for all eight registers:

Listing 2.7: The EXX swap logic (`verilog/rtl/regfile.v`)

```
// Handle EXX: swap R4-R11 with alternate set
if (exx) begin
    // Swap R4-R11 with alt_regs[0-7]
    for (i = 0; i < 8; i = i + 1) begin
        regs[i + 4] <= alt_regs[i];
        alt_regs[i] <= regs[i + 4];
    end
end
```

Simultaneous Swap

Because Verilog non-blocking assignments (`<=`) all take effect at the end of the clock edge, this loop performs a true simultaneous swap. The old values of `regs[4]` through `regs[11]` are captured into the alternates at the same moment the alternates are loaded into the mains. No temporary storage is needed.

2.2.3 Interrupt Handler Pattern

The canonical use of **EXX** is in interrupt handlers. The handler swaps to the alternate bank at entry and swaps back before returning:

Listing 2.8: Interrupt handler using EXX

```
irq_handler:
        EXX                     ; Swap to alternate R4-R11
        ; ... handle interrupt using R4'-R11' ...
        ; Primary registers preserved automatically
        EXX                     ; Swap back
        RETI                    ; Return from interrupt
```

Without **EXX**, the handler would need to **PUSH** and **POP** every register it touches. For a handler that uses four registers, that is eight memory accesses saved—a significant performance win on an architecture without hardware pipeline or cache.

2.3 Memory Model

2.3.1 Address Space

The SAMPO has a flat, 64KB address space. Every address is 16 bits wide, supporting addresses from `0x0000` to `0xFFFF`. Memory is byte-addressable and little-endian: the least significant byte of a 16-bit word is stored at the lower address.

Little-Endian Byte Order

In a little-endian system, a 16-bit value `0x1234` stored at address A places `0x34` at address A (the low byte) and `0x12` at address $A + 1$ (the high byte). This convention matches the Z80, x86, and RISC-V, and means that byte and word accesses to the same address produce consistent results for the low byte.

2.3.2 Memory Map

The suggested memory map divides the 64KB space into four regions:

Table 2.4: Suggested SAMPO memory map

Address Range	Size	Purpose
`0x0000–0x00FF`	256 B	Interrupt vectors and reset area
`0x0100–0x7FFF`	∼32KB	Program ROM/Flash
`0x8000–0xFEFF`	∼32KB	RAM
`0xFF00–0xFFFF`	256 B	Memory-mapped I/O (optional)

The reset vector at `0x0100` places programs above the interrupt vector table. The single interrupt vector resides at `0x0004`. Note that port-based I/O is the primary I/O mechanism; the memory-mapped region at `0xFF00` is available but optional.

2.3.3 Memory Interface Signals

In the Verilog implementation, the CPU accesses memory through a synchronous bus:

Listing 2.9: Memory interface signals (`verilog/rtl/cpu.v`)

```
// Memory interface
output reg  [15:0] mem_addr,
input  wire [15:0] mem_rdata,
output reg  [15:0] mem_wdata,
output reg         mem_we,
output reg  [1:0]  mem_be,
```

```
output reg         mem_valid,
input  wire        mem_ready,
```

The key signals are:

- `mem_addr`: The 16-bit address bus
- `mem_rdata`/`mem_wdata`: 16-bit read and write data buses
- `mem_we`: Write enable (high for stores, low for loads)
- `mem_be`: Byte enables for byte-level access (`01` = low byte, `10` = high byte, `11` = full word)
- `mem_valid`/`mem_ready`: Handshake signals for the memory bus

2.3.4 Word Alignment

16-bit word accesses should be aligned to even addresses. The architecture supports unaligned byte access (using **LB**, **LBU**, **SB**), but word loads and stores (**LW**, **SW**) expect alignment. The hardware handles byte extraction from word-aligned reads:

Listing 2.10: Byte extraction for byte loads (`verilog/rtl/cpu.v`)

```
if (dec_mem_byte) begin
    if (mem_addr[0]) begin
        // Odd address: high byte
        mem_result <= dec_mem_signed ?
            {{8{mem_rdata[15]}}, mem_rdata[15:8]} :
            {8'h00, mem_rdata[15:8]};
    end else begin
        // Even address: low byte
        mem_result <= dec_mem_signed ?
            {{8{mem_rdata[7]}}, mem_rdata[7:0]} :
            {8'h00, mem_rdata[7:0]};
    end
end else begin
    mem_result <= mem_rdata;
end
```

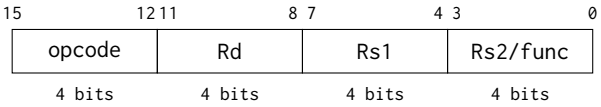
The address bit `mem_addr[0]` selects between the high and low byte of the word fetched from the aligned address. For signed byte loads (**LB**), the byte is sign-extended to 16 bits; for unsigned loads (**LBU**), it is zero-extended.

2.4 Instruction Formats

Every SAMPO instruction is either 16 bits (one word) or 32 bits (two words). The 16-bit base instructions use six distinct formats, and the 32-bit extended instructions always begin with the `0xF` prefix. Let's examine each format.

2.4.1 Format R: Register–Register Operations

Format R encodes three-register ALU operations, shifts, and miscellaneous instructions:



The opcode field (bits 15:12) identifies the instruction category. For R-type instructions, the bottom 4 bits carry either a third register (`Rs2`) or a function code that selects the specific operation within the category. For example, opcode `0x0` is **ADD**: `Rd = Rs1 + Rs2`. Opcode `0xA` is the shift category, where the `func` field selects between **SLL**, **SRL**, **SRA**, **ROL**, **ROR**, **SWAP**, and the fixed-amount variants.

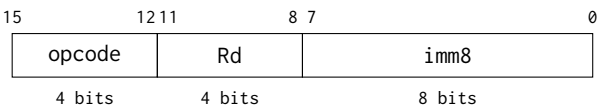
Encoding example: **ADD** R4, R5, R6

opcode	Rd	Rs1	Rs2
0000	0100	0101	0110

→ `0x0456`

2.4.2 Format I: Immediate Operations

Format I encodes operations with an 8-bit immediate value:



The 8-bit immediate is sign-extended or zero-extended depending on the instruction.

ADDI is a 2-Operand Instruction

Unlike RISC-V, where **ADDI** is `Rd = Rs + imm`, SAMPO's **ADDI** is two-operand: `Rd = Rd +` sign_ext(imm8). The register specified in bits 11:8 serves as *both* source and destination. This is a direct consequence of the 16-bit instruction width: with only 8 bits available after the opcode and Rd fields, there is no room for a separate Rs field. The assembler enforces this: `ADDI R4, 10` adds 10 to R4 and stores the result back in R4.

Encoding example: `ADDI R4, 10`

```
opcode    Rd      imm8
 0101    0100   0000 1010

→ 0x540A
```

2.4.3 Format S: Store Operations

Store instructions need a base register, a data register, and a function code (which encodes a small offset). Stores do not have a destination register, so the bits 11:8 field carries the source data register instead:

Here `Rs2` is the value to store, `Rs1` is the base address, and the function code selects the offset and access width. The load and store opcodes share a function code scheme for offsets:

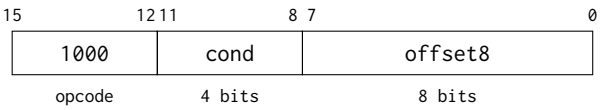
Table 2.5: Load/Store function codes and offsets

func	Load Mnemonic	Store Mnemonic	Offset
0x0	**LW** Rd, (Rs1)	**SW** (Rs1), Rs2	0
0x1	**LB** Rd, (Rs1)	**SB** (Rs1), Rs2	0 (byte)
0x2	**LBU** Rd, (Rs1)	**SW** 2(Rs1), Rs2	+2
0x3	**LW** Rd, 2(Rs1)	**SW** 4(Rs1), Rs2	+4
0x4	**LW** Rd, 4(Rs1)	**SW** 6(Rs1), Rs2	+6
0x5	**LW** Rd, 6(Rs1)	**SW** −2(Rs1), Rs2	−2
0x6	**LW** Rd, −2(Rs1)	**SW** −4(Rs1), Rs2	−4
0x7	**LW** Rd, −4(Rs1)	—	—
0x8	**LUI** Rd, Rs1	—	(special)

The small set of fixed offsets (0, ±2, ±4, ±6) handles the most common stack frame and structure access patterns. For arbitrary offsets, the extended **LWX** and **SWX** instructions provide a full 16-bit offset.

2.4.4 Format B: Branch Operations

Conditional branches use a condition code and an 8-bit signed offset:



The branch target is computed as PC + 2 + sign_extend(offset8) × 2. The multiplication by 2 gives word-aligned targets, extending the effective range to −256 to +254 bytes from the instruction following the branch.

With 4 bits for the condition code, SAMPO supports 16 branch conditions:

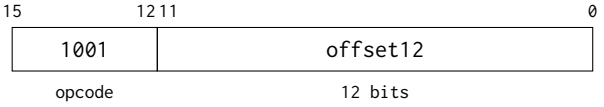
Table 2.6: SAMPO branch conditions

Code	Mnemonic	Condition	Flags Tested	Meaning
`0x0`	**BEQ**	Z=1	Zero	Equal
`0x1`	**BNE**	Z=0	Zero	Not equal
`0x2`	**BLT**	N≠V	Negative, Overflow	Signed less than
`0x3`	**BGE**	N=V	Negative, Overflow	Signed greater or equal
`0x4`	**BLTU**	C=0	Carry	Unsigned less than
`0x5`	**BGEU**	C=1	Carry	Unsigned greater or equal
`0x6`	**BMI**	N=1	Negative	Minus (negative)
`0x7`	**BPL**	N=0	Negative	Plus (positive or zero)
`0x8`	**BVS**	V=1	Overflow	Overflow set
`0x9`	**BVC**	V=0	Overflow	Overflow clear
`0xA`	**BCS**	C=1	Carry	Carry set
`0xB`	**BCC**	C=0	Carry	Carry clear
`0xC`	**BGT**	Z=0 ∧ N=V	Zero, Negative, Overflow	Signed greater than
`0xD`	**BLE**	Z=1 ∨ N≠V	Zero, Negative, Overflow	Signed less or equal
`0xE`	**BHI**	C=1 ∧ Z=0	Carry, Zero	Unsigned higher
`0xF`	**BLS**	C=0 ∨ Z=1	Carry, Zero	Unsigned lower or same

The condition encoding uses the same logic as ARM processors. Signed comparisons test the relationship between the N and V flags (N≠V means "less than" in two's complement). Unsigned comparisons test the Carry flag, which for **SUB** uses ARM-style inverted borrow: C=1 means *no* borrow occurred (i.e., the subtraction did not underflow).

2.4.5 Format J: Jump Operations

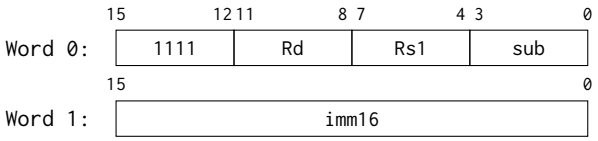
Unconditional jumps use a 12-bit signed offset:



The target is PC + 2 + sign_extend(offset12) × 2, giving a range of −4096 to +4094 bytes. For register-indirect jumps (**JR**, **JALR**), the same opcode `0x9` is used with a different bit pattern that distinguishes the register form from the offset form.

2.4.6 Format X: Extended Instructions (32-bit)

When the top nibble is `0xF`, the instruction is 32 bits. The first word contains the opcode prefix, register fields, and a 4-bit sub-opcode. The second word contains a full 16-bit immediate:



The sub-opcode (bits 3:0 of the first word) selects among 16 extended operations. These are defined in `verilog/rtl/sampo_pkg.vh`:

Listing 2.11: Extended instruction sub-opcodes (`verilog/rtl/sampo_pkg.vh`)

```
// Extended sub-opcodes (bits 3:0 when opcode = 0xF)
`define EXT_ADDIX   4'h0
`define EXT_SUBIX   4'h1
`define EXT_ANDIX   4'h2
`define EXT_ORIX    4'h3
`define EXT_XORIX   4'h4
`define EXT_LWX     4'h5
`define EXT_SWX     4'h6
`define EXT_LIX     4'h7
`define EXT_JX      4'h8
`define EXT_JALX    4'h9
`define EXT_CMPIX   4'hA
`define EXT_INX     4'hB
`define EXT_OUTX    4'hC
`define EXT_SLLX    4'hD
`define EXT_SRLX    4'hE
`define EXT_SRAX    4'hF
```

Encoding example: **`LIX`** `R4, 0x1234`

Word 0: `1111 0100 0000 0111 = 0xF407` (sub=0x7 for LIX)
Word 1: `0001 0010 0011 0100 = 0x1234` (immediate value)

When to Use Extended Instructions

Extended instructions are needed when an 8-bit immediate is not enough. Loading an arbitrary 16-bit address into a register requires **`LIX`**. Jumping to an absolute address requires **`JX`** or **`JALX`**. Adding a 16-bit offset uses **`ADDIX`**. For most arithmetic and control flow, the base 16-bit instructions suffice—keeping code compact.

2.4.7 The Primary Opcode Map

The 4-bit primary opcode gives 16 instruction categories:

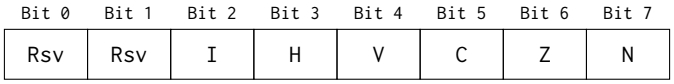
Table 2.7: The SAMPO primary opcode map

Opcode	Category	Format	Key Instructions
`0x0`	ADD	R	**ADD**, **ADC**, **NOT**, **NEG**, **MOV**
`0x1`	SUB	R	**SUB**, **SBC**
`0x2`	AND	R	**AND**
`0x3`	OR	R	**OR**
`0x4`	XOR	R	**XOR**
`0x5`	ADDI	I	**ADDI**
`0x6`	Load	R	**LW**, **LB**, **LBU**, **LUI**
`0x7`	Store	S	**SW**, **SB**
`0x8`	Branch	B	**BEQ**, **BNE**, **BLT**, **BGE**, …
`0x9`	Jump	J/R	**J**, **JR**, **JALR**
`0xA`	Shift	R	**SLL**, **SRL**, **SRA**, **ROL**, **SWAP**
`0xB`	Mul/Div	R	**MUL**, **DIV**, **REM**, **DAA**
`0xC`	Misc	R	**PUSH**, **POP**, **CMP**, **EXX**, **LDIR**
`0xD`	I/O	I/R	**INI**, **OUTI**, **IN**, **OUT**
`0xE`	System	—	**NOP**, **HALT**, **DI**, **EI**, **RETI**
`0xF`	Extended	X	**LIX**, **JX**, **ADDIX**, **LWX**, …

Giving each major ALU operation (**ADD**, **SUB**, **AND**, **OR**, **XOR**) its own primary opcode is a deliberate choice. It simplifies the decoder: the top 4 bits directly select the ALU operation for opcodes `0x0`–`0x4`, without needing an additional function code lookup. Compare this with RISC-V, where all R-type ALU operations share a single opcode and are distinguished by `funct3` and `funct7` fields.

2.5 The FLAGS Register

The SAMPO FLAGS register is 8 bits wide. Six bits are defined; two are reserved:



2.5.1 Which Instructions Set Which Flags

Not every instruction modifies the flags. The rules are:

- **Arithmetic** (**ADD**, **SUB**, **ADC**, **SBC**, **ADDI**, **CMP**, extended arithmetic): Sets **N, Z, C, V**

Table 2.8: FLAGS register bit definitions

Bit	Name	Description
7	N (Negative)	Set to the most significant bit of the result. Indicates a negative result in two's complement.
6	Z (Zero)	Set when the result is exactly zero.
5	C (Carry)	Set on unsigned overflow for addition; set when *no borrow* occurs for subtraction (ARM-style inverted borrow).
4	V (Overflow)	Set on signed overflow: when addition of two positive numbers yields a negative result, or subtraction produces an out-of-range result.
3	H (Half-carry)	Carry from bit 3 to bit 4 (lower nibble to upper nibble). Used by the **DAA** instruction for BCD arithmetic.
2	I (Interrupt)	Interrupt enable. Set by **EI**, cleared by **DI**. When clear, maskable interrupts are ignored.
1–0	—	Reserved, read as zero.

- **Logic** (**AND**, **OR**, **XOR**, **NOT**, **TEST**): Sets **N, Z**; clears **C, V**
- **Shift/Rotate**: Sets **N, Z, C**; clears **V**
- **Multiply/Divide**: Sets **N, Z** only
- **Load/Store, Branch, Jump, I/O**: Do *not* modify flags
- **System**: **SETF** writes the FLAGS register directly; **EI**/**DI** modify only the I bit; **SCF**/**CCF** modify only the C bit

2.5.2 The Carry Convention for Subtraction

This is one of SAMPO's most important details, and a common source of confusion. For subtraction, the Carry flag uses the ARM convention: C=1 means *no borrow occurred*, and C=0 means a borrow was needed.

In the ALU Verilog:

Listing 2.12: Inverted borrow for SUB (`verilog/rtl/alu.v`)

```
`ALU_SUB: begin
    result = sub_result[15:0];
    // Carry is set when there's NO borrow (a >= b for unsigned)
    flag_c = ~sub_result[16];
    flag_v = sub_overflow;
end
```

The `sub_result[16]` is the raw borrow bit from the 17-bit subtraction. Inverting it (`~sub_result[16]`) gives the ARM-style carry. This means:

- After **CMP** Ra, Rb (which computes Ra − Rb):
- C=1 ⇒ Ra ≥ Rb (unsigned)—no borrow needed
- C=0 ⇒ Ra < Rb (unsigned)—borrow occurred

Carry Sense Differs from Z80

The Z80 uses the opposite convention: its Carry flag is set when a borrow *does* occur during subtraction. If you are porting Z80 code, branch conditions after **CMP** may need to be inverted. In SAMPO, "branch if unsigned less than" is **BLTU** (C=0), not the Z80's "branch if carry set."

2.5.3 Flag Access Instructions

Two instructions provide direct access to the FLAGS register:

- **GETF** Rd: Copies the 8-bit FLAGS register into Rd (zero-extended to 16 bits)
- **SETF** Rs: Writes the low 8 bits of Rs into the FLAGS register

These are useful for saving and restoring flags across function calls, or for directly manipulating interrupt enable state.

2.6 Addressing Modes

SAMPO supports four addressing modes. This number is deliberately minimal—every mode must be implementable with straightforward combinational logic in the decoder and ALU.

2.6.1 Register Direct

The operand is the value in a register.

Listing 2.13: Register direct addressing

```
ADD  R4, R5, R6          ; R4 = R5 + R6
NOT  R8, R9              ; R8 = ~R9
MOV  R12, R4             ; R12 = R4
```

This is the most common mode, used by all R-type instructions. The hardware reads the register file during the DECODE state and latches the values into `rs1_val` and `rs2_val` for use in EXECUTE.

2.6.2 Immediate

The operand is a constant encoded within the instruction.

Listing 2.14: Immediate addressing

```
ADDI R4, 42             ; R4 = R4 + 42
ADDI R5, -1             ; R5 = R5 - 1
LIX  R6, 0x1234         ; R6 = 0x1234 (extended immediate)
```

For 16-bit base instructions, the immediate is 8 bits (signed for **ADDI**, range -128 to $+127$). For 32-bit extended instructions, the immediate is a full 16 bits.

2.6.3 Register Indirect with Offset

The effective address is the sum of a base register and a small offset encoded in the instruction.

Listing 2.15: Register indirect with offset

```
LW   R4, (R5)           ; R4 = mem[R5]        (offset 0)
LW   R4, 2(R5)          ; R4 = mem[R5 + 2]    (offset +2)
SW   -4(R3), R7         ; mem[R3 - 4] = R7    (offset -4)
LWX  R4, 0x100(R5)      ; R4 = mem[R5 + 256]  (extended)
```

The base 16-bit load/store instructions encode a small set of fixed offsets (0, ± 2, ± 4, ± 6) in the function code field. For arbitrary offsets, the extended **LWX** and **SWX** instructions provide a full 16-bit offset in the second word.

2.6.4 PC-Relative

Branch and jump targets are expressed relative to the current program counter.

Listing 2.16: PC-relative addressing

```
BEQ  done               ; if Z=1, PC = PC + 2 + offset*2
J    loop               ; PC = PC + 2 + offset*2
```

The assembler calculates the offset automatically from the label. The programmer writes labels; the hardware sees signed offsets.

No Absolute Addressing Without Extended Instructions

The base 16-bit instruction set has no absolute addressing mode. All branches and jumps are PC-relative, which makes code position-independent by default. Only the 32-bit extended **JX** and **JALX** instructions use absolute 16-bit addresses.

2.7 Comparison: Z80 vs. RISC-V vs. ARM Thumb vs. Sampo

Having laid out the SAMPO architecture in detail, let's put it in context. The following comparison highlights the design decisions that define each architecture.

2.7.1 Instruction Encoding Philosophy

Table 2.9: Instruction encoding comparison

Aspect	Z80	RISC-V (RV32I)	ARM Thumb-2	SAMPO
Instruction width	1–4 bytes	4 bytes fixed	2 or 4 bytes	2 or 4 bytes
Length determination	Complex prefix decoding	Always 4 bytes	Check top bits of halfword	Check top nibble for `0xF`
Opcode field	Varies by prefix	7-bit opcode	Varies	4-bit (bits 15:12)
Max immediate (base)	8 bits	12 bits (I-type)	8 bits (Thumb-1)	8 bits
Max immediate (ext.)	16 bits (LD nn)	20 bits (U-type)	12 bits	16 bits

2.7.2 Register File Comparison

Table 2.10: Register file comparison

Aspect	Z80	RISC-V	ARM Thumb	SAMPO
GP registers	8 (7 usable)	32 (31 usable)	16 (13 usable)	16 (15 usable)
Width	8-bit	32-bit	32-bit	16-bit
Zero register	None	`x0`	None	`R0`
Alternate bank	Full set	None	None	R4–R11 only
Encoding bits	3	5	3–4	4

SAMPO sits in a sweet spot: more registers than the Z80 (which constantly spills to memory), fewer than RISC-V (which has more registers than most programs use), and the same count as ARM Thumb. The 4-bit register encoding fits naturally in the 16-bit instruction word.

Table 2.11: I/O and memory model comparison

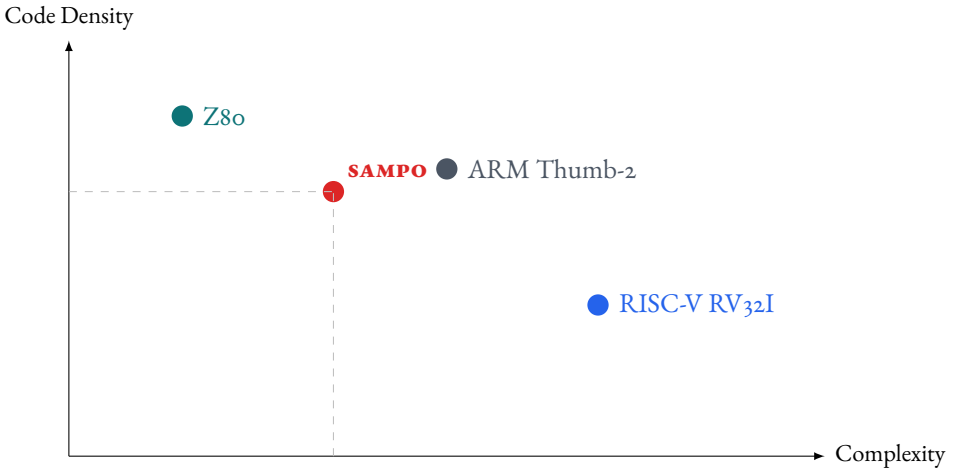
Feature	Z80	RISC-V	ARM	SAMPO
I/O model	Port-based	Memory-mapped	Memory-mapped	Port-based
Port space	256 ports	—	—	256 ports
Address space	64KB	32-bit+	32-bit	64KB
Endianness	Little	Little	Bi-endian	Little
Block ops	LDIR, LDDR	None	LDM/STM	LDIR, LDDR, FILL

2.7.3 I/O and Memory Model

SAMPO's port-based I/O keeps the memory map clean and matches the Z80's model, making peripheral access explicit. The 64KB address space is a deliberate constraint: it keeps addresses 16 bits wide, matching the register and data path width.

2.7.4 Where Sampo Sits

To visualise SAMPO's position in the design space:



The Z80 achieves extreme code density through its variable-length encoding and many addressing modes, but at the cost of a complex decoder. RISC-V prioritises decode regularity at the expense of code density (every instruction is 4 bytes in RV32I). ARM Thumb-2 balances both well. SAMPO aims for a similar balance to Thumb-2 but with a lower-complexity implementation—the 4-bit primary opcode and single-nibble extension check keep the decoder tractable while the 16-bit base encoding keeps code compact.

2.8 Exercises

1. **Encode by hand.** Encode the following instructions as hexadecimal machine code using the encoding rules from this chapter. Check your answers against the encoding examples in `ENCODING.md`.

   - **SUB** R7, R8, R9
   - **ADDI** R10, -5
   - **BNE** with an offset of $+6$ bytes
   - **LIX** R12, 0xBEEF

2. **Flag analysis.** After executing the sequence **LIX** R4, 0x7FFF followed by **ADDI** R4, 1, what are the values of the N, Z, C, and V flags? What does this tell you about the result? (Hint: think about signed overflow.)

3. **EXX register mapping.** An interrupt occurs while a function is using R4–R7 for local computation. The interrupt handler calls **EXX**, uses R4–R6 to handle the interrupt, then calls **EXX** again and **RETI**. Draw a diagram showing the contents of the main and alternate register banks at each step.

4. **Addressing mode analysis.** For each of the four addressing modes, give two example instructions that use that mode and explain when a programmer would choose each.

5. **Compare with RISC-V.** The RISC-V `ADDI` instruction has the form `ADDI rd, rs1, imm12` —three operands with a 12-bit immediate. SAMPO's **ADDI** has the form **ADDI** Rd, imm8—two operands with an 8-bit immediate. What are the tradeoffs of each design? When would you miss the third operand, and when would you appreciate the smaller instruction size?

What's next. You now have a complete picture of the SAMPO architecture: the register file, memory model, instruction formats, FLAGS register, and addressing modes. Before we start writing programs, we need to set up the development environment. In Chapter 3, we will install the Rust toolchain for the assembler and emulator, the Verilog simulator for hardware testing, and the FPGA toolchain for running on real hardware. By the end of that chapter, you will have assembled and run the "Hello, World" program end to end.

Chapter 3

Setting Up Your Workshop

A CPU project spans multiple languages and toolchains. Before we write our first program, we need to assemble the tools: Rust for the assembler and emulator, Icarus Verilog for hardware simulation, and the open-source FPGA toolchain for synthesis. This chapter walks through each installation and ends with a satisfying proof that everything works: assembling, simulating, and running the "Hello, World" program from Listing 1.1.

The good news is that every tool in the SAMPO workflow is open source and available on macOS, Linux, and Windows (via WSL). There are no license servers, no proprietary IDEs, and no vendor lock-in.

3.1 Installing Rust and Cargo

The SAMPO assembler (`sasm`) and emulator (`semu`) are both written in Rust, using the 2021 edition. The assembler has zero external dependencies—it is pure Rust standard library code. The emulator depends on three crates for its terminal user interface:

Listing 3.1: Emulator dependencies (`semu/Cargo.toml`)

```
[dependencies]
ratatui = "0.29"
crossterm = "0.28"
sysinfo = "0.32"
```

3.1.1 Installing rustup

The recommended way to install Rust is through `rustup`, the official Rust toolchain manager. On macOS or Linux:

Listing 3.2: Installing Rust via rustup

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Follow the on-screen prompts and accept the defaults. After installation, restart your shell or run:

Listing 3.3: Activating Rust in the current shell

```
source "$HOME/.cargo/env"
```

Verify the installation:

Listing 3.4: Verifying Rust installation

```
rustc --version
cargo --version
```

You should see version 1.70 or later. The SAMPO tools use Rust edition 2021 features, which are stable across all recent Rust releases.

Windows Users: Use WSL

On Windows, the recommended approach is to install the Windows Subsystem for Linux (WSL 2) with an Ubuntu distribution, then install all tools inside the Linux environment. The Verilog and FPGA toolchains are native Linux tools, and WSL 2 provides full compatibility with minimal overhead.

3.1.2 Building the Assembler

From the root of the SAMPO repository, build the assembler in release mode:

Listing 3.5: Building the assembler

```
cd sasm && cargo build --release
```

The compiled binary lands in `sasm/target/release/sasm`. Test it with the help flag:

Listing 3.6: Checking the assembler

```
cargo run --release -- --help
```

You should see:

Listing 3.7: Assembler help output

```
Sampo Assembler (sasm) v0.1.0

Usage: sasm <input.s> [-o output.bin]

Options:
  -o <file>    Output file (default: input with .bin extension)
  -h, --help   Show this help message
```

3.1.3 Building the Emulator

The emulator builds the same way:

Listing 3.8: Building the emulator

```
cd semu && cargo build --release
```

The first build will download and compile the `ratatui` and `crossterm` crates for the TUI interface. Subsequent builds are fast.

Listing 3.9: Checking the emulator

```
cargo run --release -- --help
```

Listing 3.10: Emulator help output

```
Sampo Emulator (semu) v0.1.0

Usage: semu <program.bin> [options]

Options:
  -t, --trace       Trace execution
  -i, --interactive Interactive CLI debugger
      --tui         TUI mode with graphical interface
  -h, --help        Show this help message
```

The emulator has three modes of operation:

- **Default**: Runs the program to completion and prints UART output
- **Interactive** (`--interactive`): Command-line debugger with step, dump, and memory inspection
- **TUI** (`--tui`): Full terminal user interface showing registers, memory, disassembly, and UART output simultaneously

3.2 Installing Icarus Verilog and GTKWave

Icarus Verilog (`iverilog`) is the Verilog simulator we use for all hardware simulation and testing. GTKWave is a waveform viewer for inspecting signal traces. Together, they form the core hardware debugging environment.

3.2.1 macOS (Homebrew)

Listing 3.11: Installing Icarus Verilog and GTKWave on macOS

```
brew install icarus-verilog
brew install --cask gtkwave
```

3.2.2 Ubuntu/Debian Linux

Listing 3.12: Installing Icarus Verilog and GTKWave on Ubuntu

```
sudo apt update
sudo apt install iverilog gtkwave
```

3.2.3 Verifying the Installation

Check that both tools are available:

Listing 3.13: Verifying Verilog tools

```
iverilog -V
gtkwave --version
```

Icarus Verilog version 11.0 or later is recommended. The SAMPO Verilog uses standard IEEE 1364 features with Verilog-2001 syntax.

Verilator as an Alternative

The SAMPO Makefile also includes a `verilate` target for building with Verilator, which compiles Verilog to C++ for faster simulation. Verilator is optional—Icarus Verilog handles everything we need—but it can be useful for running long simulations more quickly.

3.3 Installing the FPGA Toolchain

Running the SAMPO on a physical FPGA requires four tools: Yosys for synthesis, nextpnr-ecp5 for place-and-route, ecppack for bitstream generation, and openFPGALoader for programming the board. All four are open source.

FPGA Tools Are Optional

If you are following the Programmer reading path from Section 1.5.1, you do not need the FPGA toolchain for Parts I through IV. You can skip this section and return to it when you reach Part VIII. The Verilog simulation with Icarus Verilog does not require any FPGA tools.

3.3.1 The OSS CAD Suite

The easiest way to install the entire open-source FPGA toolchain is through the **OSS CAD Suite**, a pre-built collection of all the tools you need. Download the latest release for your platform:

Listing 3.14: Installing the OSS CAD Suite

```
# Download and extract (adjust URL for your platform)
# Visit: https://github.com/YosysHQ/oss-cad-suite-build/releases

# Linux x86_64 example:
wget https://github.com/YosysHQ/oss-cad-suite-build/releases/download/2024-02-14/oss-
    cad-suite-linux-x64-20240214.tgz
tar xzf oss-cad-suite-linux-x64-20240214.tgz

# Add to PATH
export PATH="$HOME/oss-cad-suite/bin:$PATH"
```

Add the `export PATH` line to your shell profile (`.bashrc`, `.zshrc`, etc.) to make it permanent.

3.3.2 Individual Tool Installation

Alternatively, install the tools individually:

On macOS (Homebrew):

Listing 3.15: Installing FPGA tools on macOS

```
brew install yosys
brew install nextpnr
brew install ecppack
brew install openfpgaloader
```

On Ubuntu/Debian:

Listing 3.16: Installing FPGA tools on Ubuntu

```
sudo apt install yosys nextpnr-ecp5 fpga-icestorm
sudo apt install openfpgaloader
```

3.3.3 Verifying the FPGA Tools

Check that all four tools are accessible:

Listing 3.17: Verifying FPGA toolchain

```
yosys --version
nextpnr-ecp5 --version
ecppack --version
openFPGALoader --version
```

3.3.4 The ULX3S Board

The SAMPO FPGA implementation targets the ULX3S development board, which carries a Lattice ECP5-85F FPGA. The Makefile in `verilog/Makefile` is preconfigured for this board:

Listing 3.18: FPGA target configuration (`verilog/Makefile`)

```
DEVICE = 85k
PACKAGE = CABGA381
LPF = $(FPGA_DIR)/ulx3s_v20.lpf
```

The ULX3S connects via USB and appears as a serial device for both FPGA programming and UART communication. On macOS, the serial port is typically `/dev/cu.usbserial-*`; on Linux, `/dev/ttyUSB0` or `/dev/ttyACM0`.

No FPGA Board Yet?

You can complete the entire Verilog implementation and simulation workflow (Parts V through VII) without physical hardware. Icarus Verilog simulates the design cycle-accurately, and GTKWave lets you inspect every signal. The FPGA board is only needed for Parts VIII and IX.

3.4 Building the Project from Source

With all tools installed, let's build the complete SAMPO project. Start by cloning the repository:

Listing 3.19: Cloning the Sampo repository

```
git clone https://github.com/alexjokela/sampo.git
cd sampo
```

3.4.1 Building the Software Tools

Build both the assembler and emulator:

Listing 3.20: Building the assembler and emulator

```
cd sasm && cargo build --release && cd ..
cd semu && cargo build --release && cd ..
```

3.4.2 Building the Verilog Simulation

The Verilog build uses a Makefile in the `verilog/` directory. The default target compiles the testbench and runs the Hello World simulation:

Listing 3.21: Building and running the Verilog testbench

```
cd verilog && make test
```

Under the hood, this runs two steps. First, Icarus Verilog compiles all the RTL source files and the testbench into a simulation binary:

Listing 3.22: What `make test` does (compilation step)

```
iverilog -Irtl -o build/sampo_tb \
    rtl/alu.v rtl/shifter.v rtl/regfile.v \
    rtl/decode.v rtl/cpu.v rtl/ram.v \
    rtl/uart.v rtl/soc.v tb/sampo_tb.v
```

Then the simulation binary runs:

Listing 3.23: What `make test` does (execution step)

```
cd build && ./sampo_tb
```

3.4.3 The Makefile at a Glance

The Verilog Makefile provides a comprehensive set of targets for both simulation and FPGA builds:

Table 3.1: Key Makefile targets

Target	Category	Description
`make test`	Simulation	Run Hello World in simulation
`make test-alu`	Simulation	Run ALU unit tests
`make test-regfile`	Simulation	Run register file unit tests
`make isa-tests`	Simulation	Run all ISA instruction tests
`make wave`	Debug	Open waveforms in GTKWave
`make synth`	FPGA	Yosys synthesis
`make pnr`	FPGA	nextpnr place-and-route
`make bit`	FPGA	Generate bitstream
`make fpga`	FPGA	Full FPGA build pipeline
`make prog`	FPGA	Upload bitstream to ULX3S
`make fpga-tests`	FPGA	Run all ISA tests on hardware
`make clean`	Utility	Remove build artifacts

We will use many of these targets throughout the book. For now, `make test` is all we need.

3.5 Running hello.s End to End

Let's trace the complete workflow: source assembly to running hardware (or simulated hardware). This is the proof that everything is connected.

3.5.1 Step 1: Assemble

Assemble the Hello World program:

Listing 3.24: Assembling hello.s

```
cd sasm && cargo run --release -- ../examples/hello.s -o ../examples/hello.bin
```

The output confirms success:

Listing 3.25: Assembly output

```
Assembled ../examples/hello.s -> ../examples/hello.bin (301 bytes)
```

The assembler has read the source from `examples/hello.s`, resolved labels and constants, generated machine code, and written a flat binary to `hello.bin`. Those 301 bytes contain the machine instructions starting at address `0x0100`, followed by the "Hello, Sampo!\n" string data.

3.5.2 Step 2: Run in the Emulator

Test the binary in the emulator:

Listing 3.26: Running in the emulator

```
cd semu && cargo run --release -- ../examples/hello.bin
```

Listing 3.27: Emulator output

```
Sampo Emulator - Loaded 301 bytes
Starting execution at 0x0100

Hello, Sampo!

CPU halted at 0x011E
```

The emulator loaded the binary, started execution at the reset vector (`0x0100`), ran the program instruction by instruction, and captured the UART output. The CPU halted at `0x011E`, which is the address of the **`HALT`** instruction.

Try the Interactive Debugger

Add the `--interactive` flag to step through the program one instruction at a time:

```
cargo run --release -- ../examples/hello.bin -i
```

Use `s` to step, `d` to dump the full CPU state, and `r` to run to completion. This is invaluable for understanding how each instruction affects the registers and flags.

3.5.3 Step 3: Convert to Verilog Hex Format

The Verilog testbench loads programs from hex files using `$readmemh`. A Python script converts the flat binary to this format:

Listing 3.28: Converting binary to Verilog hex

```
cd verilog && python3 bin2hex.py ../examples/hello.bin programs/hello.hex
```

The script reads the binary, interprets pairs of bytes as little-endian 16-bit words, and writes them in hexadecimal:

Listing 3.29: The bin2hex conversion (`verilog/bin2hex.py`)

```
# Convert to 16-bit words (little-endian)
words = []
for i in range(0, len(data), 2):
    # Little-endian: low byte first
    word = data[i] | (data[i+1] << 8)
    words.append(word)

# Write hex file
with open(output_file, 'w') as f:
    f.write("@0000\n")
    for i, word in enumerate(words):
        f.write(f"{word:04X}\n")
```

The resulting hex file starts with an address directive (`@0000`) followed by one 16-bit word per line in uppercase hexadecimal.

3.5.4 Step 4: Simulate in Verilog

Now run the program on the simulated hardware:

Listing 3.30: Running the Verilog simulation

```
cd verilog && make test
```

The testbench instantiates the complete SoC (CPU, RAM, and UART), loads the hex file into simulated RAM, de-asserts reset, and runs until the CPU halts. The simulation output shows both debug tracing and UART output:

Listing 3.31: Verilog simulation output (abbreviated)

```
=== Sampo CPU Testbench ===
RAM init file: ../programs/hello.hex

CPU started at PC=0x0100
UART output:
----------------------------------------
Hello, Sampo!

----------------------------------------

Simulation complete:
  Final PC:    0x011E
  Cycles:      35784
  UART chars:  15
  Status:      HALTED
```

The same "Hello, Sampo!" message appears, now produced by a cycle-accurate simulation of the actual Verilog hardware. The cycle count (35784 in this example) reflects the real hardware timing, including UART transmission delays and the multi-cycle state machine processing each instruction.

3.5.5 Step 5: View the Waveforms

The testbench automatically generates a VCD (Value Change Dump) file during simulation:

Listing 3.32: VCD generation in the testbench (`verilog/tb/sampo_tb.v`)

```
initial begin
    $dumpfile("sampo.vcd");
    $dumpvars(0, sampo_tb);
end
```

Open it in GTKWave:

Listing 3.33: Viewing waveforms

```
make wave
```

In GTKWave, expand the `sampo_tb` hierarchy to see every signal in the design—the CPU state machine transitions, register file reads and writes, ALU operations, and memory bus activity. This is the most powerful debugging tool for hardware: you can zoom into any clock cycle and see exactly what every signal was doing.

Essential Signals to Watch

When first opening the waveforms, add these signals to the viewer:

- `uut.cpu_inst.state` — The FSM state
- `uut.cpu_inst.pc` — The program counter
- `uut.cpu_inst.instr` — The current instruction
- `uut.cpu_inst.flags` — The FLAGS register
- `uut.cpu_inst.mem_addr` and `uut.cpu_inst.mem_rdata` — Memory bus activity

Watch the state signal cycle through FETCH → DECODE → EXECUTE → WRITEBACK (and through MEMORY for load/store instructions). This is the heartbeat of the CPU.

3.5.6 Step 6: Run on the FPGA (Optional)

If you have a ULX3S board, you can run the full FPGA build pipeline:

Listing 3.34: Full FPGA build and program

```
cd verilog && make fpga && make prog
```

This runs three stages:

1. **Synthesis** (Yosys): Converts Verilog RTL into an ECP5 gate-level netlist
2. **Place-and-route** (nextpnr-ecp5): Maps the netlist onto physical FPGA resources and routes the connections
3. **Bitstream generation** (ecppack): Packages the configuration into a bitstream file
4. **Programming** (openFPGALoader): Uploads the bitstream to the ULX3S board via USB

Connect a serial terminal to the ULX3S's USB serial port at 115200 baud, and you will see:

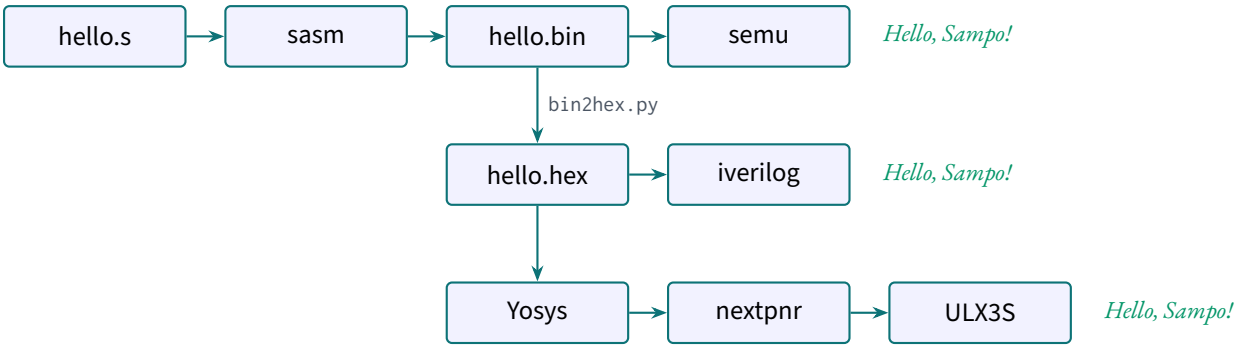
Listing 3.35: FPGA serial output

```
Hello, Sampo!
```

That is the same message, produced by the same program, running on real hardware. The electrons flowing through the ECP5 FPGA are executing the exact logic described in the Verilog files you have been reading.

3.5.7 The Complete Pipeline

Here is the entire flow, from source code to hardware execution:



Three outputs, all identical, from one source file. The assembler produces the binary. The emulator runs it in software. The Verilog simulation runs it on simulated hardware. The FPGA runs it on real hardware. If they all produce the same output, you have high confidence that the architecture specification, the assembler, the emulator, and the hardware implementation all agree.

3.6 Exercises

1. **Verify your environment.** Follow the steps in this chapter to install Rust, Icarus Verilog, and (optionally) the FPGA toolchain. Run `rustc --version`, `iverilog -V`, and (if installed) `yosys --version` to confirm everything is working. Record the version numbers.

2. **Run hello.s end-to-end.** Assemble `examples/hello.s`, run it in the emulator (both default and interactive modes), convert it to hex, and run the Verilog simulation. Confirm that all three executions produce "Hello, Sampo!" and that the final PC matches (`0x011E`).

3. **Explore the emulator TUI.** Run the emulator in TUI mode (`--tui`) with the hello binary. Use F6 to step through the first ten instructions. For each instruction, note the PC, the instruction mnemonic shown in the disassembly panel, and which registers change. Can you identify the point where the first character is written to the UART?

4. **Inspect a waveform.** After running `make test` in the `verilog/` directory, open the resulting VCD file in GTKWave. Add the `state`, `pc`, and `instr` signals. Find the first FETCH → DECODE → EXECUTE cycle and identify the instruction being executed. How many clock cycles does a basic instruction take?

5. **Run the ISA test suite.** Execute `make isa-tests` in the `verilog/` directory. This runs all ten ISA test programs in simulation. Record how many tests pass. (Spoiler: they should all pass. If they do not, check your Icarus Verilog version.)

What's next. Your workshop is ready. The tools are installed, the project builds, and "Hello, Sampo!" runs in all three environments. In Chapter 4, we will slow down and trace through that

Hello World program instruction by instruction, learning the fundamentals of SAMPO assembly along the way. It is time to start programming.

Part II

Programming the Sampo

Chapter 4

Hello, Sampo!

Every CPU needs a first program. Before we dive into the full instruction set, let's write one—a complete SAMPO assembly program that sends a message to a serial port, assemble it with `sasm`, and run it in the emulator. Along the way, we will learn the anatomy of a SAMPO source file, the directives the assembler understands, and the tools that turn text into running code.

4.1 Anatomy of a Sampo Assembly Program

A SAMPO assembly source file is a plain text file—typically with a `.s` extension—containing three kinds of lines: *directives*, *instructions*, and *labels*. Blank lines and comments are ignored. Let's start with the complete "Hello, Sampo!" program from `examples/hello.s` in the project source tree:

Listing 4.1: The complete Hello, Sampo! program (`examples/hello.s`)

```
; Hello World for Sampo CPU
; Outputs "Hello, Sampo!" to serial port

        .org 0x0100

; Serial port definitions (MC6850 ACIA style)
.equ    ACIA_STATUS 0x80
.equ    ACIA_DATA   0x81
.equ    TX_READY    0x02

start:
        LIX  R4, message        ; Load address of message

loop:
```

```
        LBU  R5, (R4)           ; Load byte from string
        CMP  R5, R0             ; Compare with zero
        BEQ  done               ; If null, we're done

wait_tx:
        INI  R6, ACIA_STATUS    ; Read serial status
        AND  R7, R6, R6         ; Copy to R7
        ADDI R7, -2             ; Subtract TX_READY (2)
        BNE  wait_tx            ; Wait if not ready

        OUTI ACIA_DATA, R5      ; Write character
        ADDI R4, 1              ; Next character
        J    loop               ; Continue

done:
        HALT

message:
        .asciz "Hello, Sampo!\n"
```

That is 15 instructions and one data declaration—fewer than 30 lines of source. Let's trace through the structure before we explain each piece.

4.1.1 The Big Picture

Every SAMPO assembly program has the same basic shape:

1. **Origin directive.** The `.org` directive tells the assembler where in the 64 KB address space the following code should be placed. Our program starts at `0x0100`, just above the interrupt vector area.

2. **Constant definitions.** The `.equ` directives name I/O port addresses so we never have to remember magic numbers.

3. **Code.** Labels mark entry points; instructions do the work.

4. **Data.** The `.asciz` directive embeds a null-terminated string directly into the binary image, right after the last instruction.

Code and Data Share the Address Space

SAMPO uses a von Neumann architecture: instructions and data live in the same 64 KB memory. The string `"Hello, Sampo!\n"` is stored in program memory immediately after the **HALT** instruction. The **LIX** at the top loads the *address* of that string into a register so the loop can read it byte by byte.

4.1.2 Registers Used in This Program

The program uses five registers:

Register	Alias	Role in this program
R0	ZERO	Hardwired zero—used as the null-terminator reference
R4	A0	Pointer into the message string
R5	A1	Current character loaded from memory
R6	A2	ACIA status register value
R7	A3	Scratch for the transmit-ready check

Table 4.1: Registers used by the Hello program

Notice that R0 is never loaded—it is always zero. This is a hardware feature of the SAMPO: writes to R0 are silently discarded, and reads always return 0x0000.

4.2 The Assembler: sasm Directives

The SAMPO assembler, sasm, is a two-pass assembler written in Rust (we will explore its internals in **??**). It reads a single source file, resolves labels and expressions, and produces a flat binary image.

Listing 4.2: Assembling a source file with sasm

```
cargo run --release -- ../examples/hello.s -o hello.bin
```

If no -o flag is given, sasm replaces the source extension with .bin: hello.s becomes hello.bin. On success, it prints a summary:

Listing 4.3: Assembler output

```
Assembled examples/hello.s -> hello.bin (46 bytes)
```

The binary is a raw memory image: the first byte corresponds to the address given by the first .org, and the file contains every byte through to the end of the last emitted data.

4.2.1 .org — Set Origin Address

```
.org 0x0100
```

The `.org` directive sets the *location counter*—the address at which the next byte will be placed. Most SAMPO programs begin with `.org 0x0100` because the first 256 bytes (`0x0000`–`0x00FF`) are reserved for the interrupt vector table and the reset entry point.

You can use multiple `.org` directives in one source file to place code or data at different addresses, though for most programs a single origin suffices.

4.2.2 .equ — Define a Constant

```
.equ    ACIA_STATUS 0x80
.equ    ACIA_DATA   0x81
.equ    TX_READY    0x02
```

The `.equ` directive binds a symbolic name to a numeric value. Wherever the assembler encounters `ACIA_STATUS` in an operand field, it substitutes the value `0x80`. Constants defined with `.equ` do *not* emit any bytes into the output—they exist only during assembly.

Name Your Constants

Hardware port addresses, buffer sizes, bit masks—anything that appears more than once should get an `.equ` name. When the hardware changes (and it will), you only need to update one line.

4.2.3 .db — Define Bytes

```
.db 0x48, 0x65, 0x6C, 0x6C, 0x6F   ; "Hello" as raw bytes
```

The `.db` directive emits one or more raw bytes into the output at the current location counter. Values are separated by commas. Each value must fit in an unsigned byte (0–255).

4.2.4 .dw — Define Words

```
.dw 0x1234, 0x5678          ; Two 16-bit words
```

The `.dw` directive emits 16-bit words in little-endian byte order. The value `0x1234` produces the byte sequence `0x34`, `0x12` in the output binary. This directive is particularly useful for embedding raw

instruction encodings when the assembler doesn't yet support a mnemonic—the test suite uses `.dw` extensively for extended instructions:

Listing 4.4: Using .dw for manual instruction encoding (`examples/tests/test_shift.s`)

```
        ; SLL4 R4, R8 = opcode A, Rd=4, Rs=8, func=8
        LIX R8, 0x1234
        .dw 0xA488          ; SLL4 R4, R8
        LIX R5, 0x2340      ; Expected result
```

Little-Endian Word Order

SAMPO is little-endian. When you write `.dw 0x1234`, the byte at the lower address is `0x34` (the least significant byte) and the byte at the higher address is `0x12`. This matches the CPU's native memory access order, but can be confusing when reading hex dumps.

4.2.5 .ascii and .asciz — Define Strings

```
.ascii "Hello"          ; 5 bytes, no terminator
.asciz "Hello, Sampo!"  ; 14 bytes, null-terminated
```

The `.ascii` directive embeds a string as raw bytes, without any terminator. The `.asciz` directive appends a zero byte (`0x00`) after the string data.

The assembler supports standard escape sequences in string literals: `\\n` (newline, `0x0A`), `\\r` (carriage return, `0x0D`), `\\t` (tab, `0x09`), `\\0` (null, `0x00`), `\\\\` (literal backslash), and `\\"` (literal double quote).

Most SAMPO programs use `.asciz` for message strings—the null terminator gives the output loop a clean stopping condition, as we will see shortly.

4.3 Labels, Comments, and Expressions

4.3.1 Labels

A label is an identifier followed by a colon. It records the current location counter so that other instructions can refer to that address by name.

```
start:
        LIX  R4, message     ; 'message' is a forward reference
        ; ...
message:
        .asciz "Hello!"      ; 'message' resolves to this address
```

Labels may appear on their own line (as shown above) or on the same line as an instruction:

```
loop:   LBU R5, (R4)
```

The assembler resolves labels in two passes. In the first pass, it collects all label addresses. In the second pass, it substitutes the numeric addresses wherever a label name appears. This means forward references—using a label before it is defined—work correctly.

Forward Reference

A *forward reference* occurs when an instruction uses a label that has not yet been defined. The assembler records a *fixup* entry during the first pass and resolves it in the second pass, once the target address is known.

Labels are used in three contexts:

1. **Branch targets.** `BEQ done` branches to the address of the `done:` label. The assembler computes the signed offset automatically.
2. **Jump targets.** `J loop` and `JX target` use labels for PC-relative or absolute addresses.
3. **Data references.** `LIX R4, message` loads the 16-bit address of `message:` into R4.

4.3.2 Comments

Comments begin with a semicolon (`;`) and extend to the end of the line. The assembler's lexer discards everything from the semicolon onward:

```
        LBU R5, (R4)        ; Load next character
        ; This entire line is a comment
```

Use comments liberally. In assembly language, the gap between what the code *says* ("subtract 2") and what it *means* ("check if TX is ready") is wider than in any high-level language.

4.3.3 Numeric Literals

The assembler accepts integers in three bases:

Negative numbers are supported for signed immediates: `ADDI R4, -3` encodes the two's complement representation of -3 in the 8-bit immediate field.

Character literals enclosed in single quotes are also accepted and convert to their ASCII value: `'A'` produces the value 65 (`0x41`).

Format	Prefix	Example
Decimal	(none)	`42`, `-128`
Hexadecimal	`0x`	`0xFF`, `0x0100`
Binary	`0b`	`0b11010010`

Table 4.2: Numeric literal formats in `sasm`

4.3.4 Register Names and Aliases

Registers can be referenced by their numeric names (`R0` through `R15`) or by their conventional aliases. The assembler accepts both forms interchangeably:

```
        MOV R4, R5          ; Numeric names
        MOV A0, A1          ; Same instruction, using aliases
        PUSH RA             ; RA is R1 (return address)
        LIX SP, 0xFFF0      ; SP is R2 (stack pointer)
```

The full alias table was introduced in **??**; here is the quick reference:

R0=ZERO	R1=RA	R2=SP	R3=GP
R4=A0	R5=A1	R6=A2	R7=A3
R8=T0	R9=T1	R10=T2	R11=T3
R12=S0	R13=S1	R14=S2	R15=S3

Table 4.3: Register numeric names and aliases

4.4 Writing "Hello, Sampo!"

Now let's walk through the Hello program instruction by instruction, understanding exactly what each one does and why it's there.

4.4.1 Setting Up: Loading the String Address

```
start:
        LIX  R4, message        ; Load address of message
```

`LIX` is a 32-bit extended instruction (opcode `0xF`, sub-opcode `0x7`) that loads a full 16-bit immediate value into a register. The assembler resolves the label `message` to its absolute address and encodes

it in the second word of the instruction. After this instruction, R4 contains the address of the first character of our string.

LIX — Load Immediate Extended

LIX Rd, *imm16* loads a full 16-bit immediate into register Rd. It is a 32-bit instruction: the first word encodes the opcode (0xF), destination register, and sub-opcode (0x7); the second word is the 16-bit immediate value. Unlike **ADDI**, which can only add a small signed offset to an existing register value, **LIX** can set a register to any 16-bit value from scratch.

Why do we need a 32-bit instruction here? Because SAMPO instructions are 16 bits wide, and the label message could be any address in the 64 KB space. There is no room in a 16-bit instruction for both an opcode, a register field, and a 16-bit address. The extended encoding solves this by using two words.

4.4.2 The Character Loop

```
loop:
        LBU  R5, (R4)           ; Load byte from string
        CMP  R5, R0             ; Compare with zero
        BEQ  done               ; If null, we're done
```

Three instructions form the heart of the loop:

1. **LBU** R5, (R4) loads a single byte from the address in R4 and *zero-extends* it to 16 bits. The "U" stands for "unsigned"—the byte 0xFF becomes 0x00FF, not 0xFFFF (which is what **LB** would produce via sign extension).

2. **CMP** R5, R0 subtracts R0 (which is always zero) from R5 and sets the flags. The subtraction result is discarded—only the flags matter. If the character is the null terminator, the result is zero and the Z (zero) flag is set.

3. **BEQ** done branches to the done label if the Z flag is set. "BEQ" stands for "Branch if Equal," but since we compared against zero, it really means "branch if the character is null."

4.4.3 The Transmit-Ready Busy Wait

```
wait_tx:
        INI  R6, ACIA_STATUS    ; Read serial status
        AND  R7, R6, R6         ; Copy to R7
        ADDI R7, -2             ; Subtract TX_READY (2)
        BNE  wait_tx            ; Wait if not ready
```

Before sending a character to the serial port, we must verify that the transmitter is ready. The MC6850 ACIA (or our emulated equivalent) presents a status register on port `0x80`. Bit 1 of this register indicates transmit readiness.

The **INI** instruction reads an 8-bit value from an I/O port into a register. We then check whether the TX_READY bit is set using a two-step trick: copy the status value with **AND** (AND-ing a value with itself is a copy that also sets flags), subtract 2 (the value of `TX_READY`), and branch back if the result is non-zero.

This Is Not How You Test a Single Bit

The busy-wait loop in the Hello program uses `ADDI R7, -2` to check the TX_READY bit. This works when bit 1 is the *only* bit set in the status register, but it is fragile—if other status bits are also set, the subtraction will give a non-zero result even when TX is ready. A robust check would use **AND** with a mask:

```
        LIX  R7, TX_READY
        AND  R7, R6, R7     ; Isolate bit 1
        BEQ  wait_tx        ; Branch if bit clear
```

The Hello program gets away with the simpler approach because the emulator's ACIA only reports TX_READY in isolation.

4.4.4 Sending the Character

```
        OUTI ACIA_DATA, R5      ; Write character
        ADDI R4, 1              ; Next character
        J    loop               ; Continue
```

OUTI writes the low byte of `R5` to the I/O port at address `0x81` (the ACIA data register). The UART hardware—or, in our case, the emulator—takes this byte and transmits it.

ADDI adds the sign-extended 8-bit immediate value $+1$ to `R4`, advancing the string pointer to the next character. Note that **ADDI** is a *two-operand* instruction: the destination and first source are the same register. The operation is $\texttt{R4} \leftarrow \texttt{R4} + \text{sign_ext}(1)$.

Finally, **J** `loop` is an unconditional PC-relative jump back to the top of the loop, where we load the next character.

4.4.5 Halting

```
done:
        HALT
```

HALT stops the processor. In hardware, HALT transitions the CPU state machine to the HALTED state, where it remains until an interrupt arrives. In the emulator, HALT causes the execution loop to exit and the final CPU state to be printed.

4.4.6 The String Data

```
message:
        .asciz "Hello, Sampo!\n"
```

The label `message` records the address of the first byte of the string. The `.asciz` directive emits 15 bytes: 14 characters (including the `\n` escape, which becomes a single `0x0A` byte) plus a terminating `0x00`.

4.5 Running in the Emulator

With the program assembled, we can run it in `semu`, the SAMPO emulator. The emulator implements the full instruction set in software, including memory, I/O ports, and a simulated MC6850 ACIA.

4.5.1 Assembling and Running

From the project root, assemble the source and run the resulting binary:

Listing 4.5: Assembling and running the Hello program

```
cd sasm && cargo run --release -- ../examples/hello.s -o hello.bin
cd ../semu && cargo run --release -- ../sasm/hello.bin
```

The emulator prints a startup banner, executes the program, and displays the serial output:

Listing 4.6: Expected emulator output

```
Sampo Emulator - Loaded 46 bytes
Starting execution at 0x0100

Hello, Sampo!

CPU halted at 0x011E
```

The emulator loads the binary into memory starting at the origin address (`0x0100`), sets the program counter to that address, and begins stepping through instructions. Each **OUTI** instruction to port `0x81`

writes a character to the emulator's terminal output. When the CPU executes **HALT**, the emulator stops and reports the final PC.

4.5.2 Trace Mode

Adding the `--trace` flag makes the emulator print every instruction as it executes—an invaluable debugging tool:

Listing 4.7: Running with instruction trace

```
cargo run --release -- ../sasm/hello.bin --trace
```

The trace output shows the PC, the raw instruction word, the disassembled mnemonic, and the register state after each step. This is the fastest way to understand what your program is actually doing, especially when a branch goes somewhere unexpected or a register contains a value you didn't anticipate.

4.5.3 Interactive Mode

For more control, the `--interactive` flag starts a command-line debugger:

Listing 4.8: Starting the interactive debugger

```
cargo run --release -- ../sasm/hello.bin --interactive
```

The interactive prompt offers five commands:

Command	Short	Description
`step`	`s`	Execute one instruction
`run`	`r`	Run until HALT
`dump`	`d`	Print full CPU state (all registers, flags)
`mem`	`m`	Hex dump of memory around the current PC
`quit`	`q`	Exit the emulator

Table 4.4: Interactive debugger commands

A typical interactive session looks like this: step through a few instructions with `s`, watch the registers change, then type `r` to run to completion:

Listing 4.9: An interactive debugging session

```
semu> s
```

```
PC=0x0104 LBU R5, (R4)  R5=0x0048
semu> s
PC=0x0106 CMP R5, R0    Z=0
semu> r
Hello, Sampo!
CPU halted at 0x011E
```

4.6 The TUI Debugger

While the interactive command-line debugger is useful, the SAMPO emulator also includes a full-screen terminal user interface (TUI) mode that shows registers, memory, disassembly, the stack, and serial output all at once.

4.6.1 Starting the TUI

Listing 4.10: Launching the TUI debugger

```
cargo run --release -- ../sasm/hello.bin --tui
```

The TUI fills the terminal with several panels:

- **Registers** (top left): The PC, SP, and FLAGS register are shown at the top, followed by all 16 general-purpose registers displayed four per row. Flag bits are displayed individually (N, Z, C, V, I) so you can see them change after each instruction.
- **Memory** (left, below registers): A hex dump with ASCII representation, showing 16 bytes per row. Use `F9`/`F10` to scroll by 16 bytes, or `PgUp`/`PgDn` to scroll by 256 bytes.
- **Disassembly** (top right): The instructions around the current PC, with the current instruction highlighted. This gives you context on where execution is and what comes next.
- **Stack** (right of disassembly): The current stack contents, showing values pushed most recently at the top.
- **Terminal** (bottom right): A VT220-compatible terminal emulator that displays serial output from the SAMPO program. Any character written to the ACIA data port appears here, complete with cursor control and escape sequence support.
- **Status bar** (bottom): Shows the run state (paused, running, halted), the effective emulation speed in MHz, host CPU usage, the cycle count, and hotkey reminders.

4.6.2 TUI Controls

The TUI is controlled entirely by keyboard:

Key	Action
`F5`	Run continuously
`F6`	Step one instruction
`F7`	Pause execution
`F8`	Reset CPU
`F9/F10`	Scroll memory view (16 bytes)
`PgUp/PgDn`	Scroll memory view (256 bytes)
`Alt+=/Alt+-`	Adjust emulation speed
`F12`	Quit

Table 4.5: TUI keyboard controls

4.6.3 Stepping Through Hello, Sampo!

Let's trace the first few instructions using the TUI. Start the program in TUI mode and press `F6` to step:

1. **First step** (`F6`): The CPU executes `LIX R4, message`. This is a 32-bit instruction, so the PC advances by 4. The Registers panel shows `R4` updated with the address of the message string.

2. **Second step**: `LBU R5, (R4)` loads the first character (`'H'`, `0x48`) into `R5`. Watch `R5` change in the register panel.

3. **Third step**: `CMP R5, R0` sets flags. The Z flag remains clear (the character is not null), so the N, Z, C, V indicators in the flags display reflect the subtraction result.

4. **Fourth step**: `BEQ done` is *not taken* because Z is clear. The PC advances past the branch to the busy-wait loop.

Continue stepping and watch the terminal panel: each time an **`OUTI`** instruction executes, the character appears in the terminal window. After 14 characters plus a newline, the program reaches **`HALT`** and the status bar changes to "HALTED."

Speed Adjustment

When running continuously with `F5`, the emulator executes a configurable number of instructions per display frame. Use `Alt+=` to increase the speed and `Alt+-` to decrease it. At maximum speed, the emulator can run thousands of cycles per second on modern hardware—fast enough to see the Hello message appear character-by-character or all at once, depending on the speed setting.

4.6.4 What the TUI Reveals

The TUI debugger makes several aspects of the SAMPO architecture directly visible:

- **The zero register.** Watch `R0`—it never changes, no matter what happens. Even if an instruction writes to `R0`, the register panel shows `0x0000`.
- **Flag effects.** After each arithmetic or logic instruction, the flag bits update in real time. Step through a **`CMP`** and watch Z set or clear; step through an **`ADD`** that overflows and watch V and C light up.
- **The stack.** If you step through the test framework's `print_str` function (which uses **`PUSH`** and **`POP`**), the stack panel shows values appearing and disappearing as the stack pointer moves.
- **32-bit instructions.** Watch the disassembly panel when the CPU fetches an **`LIX`** or **`JALX`**: the instruction spans two words, and the PC advances by 4 instead of 2.

4.7 Exercises

1. **Modify the message.** Change the string in `examples/hello.s` to print your own name instead of "Sampo." Assemble it, run it in the emulator, and verify the output. How does the binary size change?
2. **Count the characters.** Add a counter that tracks how many characters were sent. Before the **`HALT`**, use a register (say `R8`) to hold the count, incrementing it each time through the loop. Step through the program in the TUI and verify that the count matches the string length.
3. **Reverse the string.** Write a new program that prints the message backward. You will need to find the end of the string first (scan forward until you hit the null terminator), then walk backward, printing each character. Think about which load instruction to use and how to detect the beginning of the string.
4. **Uppercase converter.** Write a program that reads a string from memory and prints it with all lowercase letters converted to uppercase. Recall that in ASCII, lowercase letters are in the range `0x61`–`0x7A` and uppercase letters are `0x41`–`0x5A`—a difference of `0x20`. Use **`CMP`** and conditional branches to detect lowercase characters, and **`ADDI`** to convert them.

5. **Explore trace mode.** Run the Hello program with `--trace` and redirect the output to a file. Open the file in a text editor and identify: (a) which instruction executes most often, (b) how many total instructions execute before **`HALT`**, and (c) how many times the `wait_tx` loop iterates for each character.

What's Next. We have written our first program, but we only scratched the surface of what SAMPO instructions can do. In Chapter 5, we will take a systematic tour through the arithmetic and logic instructions—**`ADD`**, **`SUB`**, **`AND`**, **`XOR`**, and their flag-setting behavior—building the foundation you need to write programs that compute, compare, and decide.

Chapter 5

Arithmetic, Logic, and Flags

A CPU that can load data and print characters is a start, but computation demands more. In this chapter we explore the SAMPO's arithmetic and logic instructions in depth: how they encode, what flags they set, and why the carry flag behaves differently for subtraction than you might expect. By the end, you will be able to write programs that add, subtract, compare, and branch on arbitrary conditions—and you will understand every bit in the FLAGS register.

5.1 Three-Operand Arithmetic: ADD, SUB, ADC, SBC

The SAMPO's arithmetic instructions use a clean three-operand format: two source registers and one destination register. The result goes into the destination; the sources are unchanged.

5.1.1 ADD — Add Two Registers

```
ADD Rd, Rs1, Rs2        ; Rd = Rs1 + Rs2
```

The encoding uses opcode `0x0` in Format R:

Bits 15:12	Bits 11:8	Bits 7:4	Bits 3:0
`0000`	Rd	Rs1	Rs2

For example, `ADD R4, R5, R6` encodes as:

`0000 0100 0101 0110` = `0x0456`

Let's see **ADD** in action, from the ALU test suite (`examples/tests/test_alu.s`):

Listing 5.1: ADD tests from `examples/tests/test_alu.s`

```
        ; ADD basic: 0x1000 + 0x2000 = 0x3000
        LIX R8, 0x1000
        LIX R9, 0x2000
        ADD R10, R8, R9
        MOV R4, R10
        LIX R5, 0x3000
        JALX check_eq

        ; ADD carry out: 0xFFFF + 1 = 0x0000
        LIX R8, 0xFFFF
        LIX R9, 0x0001
        ADD R10, R8, R9
        MOV R4, R10
        MOV R5, R0
        JALX check_eq
```

The second test demonstrates unsigned overflow: adding 1 to `0xFFFF` wraps around to `0x0000`. The result still goes into the destination register, but the carry flag (C) is set to indicate the overflow.

ADD sets all four arithmetic flags: N, Z, C, and V.

Writing to R0 Is a No-Op

The ALU test suite also verifies that writing to `R0` is silently discarded:

```
        LIX R8, 0x1234
        LIX R9, 0x5678
        ADD R0, R8, R9      ; Result discarded
        MOV R4, R0          ; R0 is still 0x0000
```

This is a hardware guarantee, not a software convention. The register file ignores writes to register index 0.

5.1.2 SUB — Subtract Two Registers

```
SUB Rd, Rs1, Rs2        ; Rd = Rs1 - Rs2
```

Encoded with opcode `0x1` in Format R:

Bits 15:12	Bits 11:8	Bits 7:4	Bits 3:0
0001	Rd	Rs1	Rs2

Listing 5.2: SUB tests from `examples/tests/test_alu.s`

```
        ; SUB basic: 0x5000 - 0x3000 = 0x2000
        LIX R8, 0x5000
        LIX R9, 0x3000
        SUB R10, R8, R9
        MOV R4, R10
        LIX R5, 0x2000
        JALX check_eq

        ; SUB underflow: 0x1000 - 0x2000 = 0xF000
        LIX R8, 0x1000
        LIX R9, 0x2000
        SUB R10, R8, R9
        MOV R4, R10
        LIX R5, 0xF000
        JALX check_eq
```

In the "underflow" case, `0x1000` − `0x2000` produces `0xF000`: the result wraps around through zero. In signed interpretation, this is $4096 - 8192 = -4096$, and `0xF000` is indeed -4096 in 16-bit two's complement.

SUB sets the carry flag using *ARM-style inverted borrow* semantics—we will cover this in detail in Section 5.5.

5.1.3 ADC — Add with Carry

```
ADC Rd, Rs1, Rs2        ; Rd = Rs1 + Rs2 + C
```

ADC adds two registers *plus* the current carry flag. This is essential for multi-word arithmetic—adding 32-bit values on a 16-bit machine. The first **ADD** sets the carry if the low 16 bits overflow, and the subsequent **ADC** picks up that carry into the high 16 bits:

Listing 5.3: 32-bit addition using ADD and ADC

```
; Add 32-bit values: (R5:R4) = (R5:R4) + (R7:R6)
; R4/R6 = low words, R5/R7 = high words
        ADD  R4, R4, R6     ; Low word, may set C
        ADC  R5, R5, R7     ; High word, includes carry-in
```

5.1.4 SBC — Subtract with Carry

```
SBC Rd, Rs1, Rs2        ; Rd = Rs1 - Rs2 - (1 - C)
```

SBC is the subtraction counterpart of **ADC**. Because the SAMPO uses ARM-style carry (C = 1 means *no borrow*), the **SBC** operation subtracts the *inverse* of the carry flag. When a preceding **SUB** sets C = 0 (indicating a borrow occurred), the subsequent **SBC** propagates that borrow into the high word:

Listing 5.4: 32-bit subtraction using SUB and SBC

```
; Subtract 32-bit values: (R5:R4) = (R5:R4) - (R7:R6)
        SUB  R4, R4, R6     ; Low word, C=0 if borrow
        SBC  R5, R5, R7     ; High word, incorporates borrow
```

5.2 Bitwise Operations: AND, OR, XOR, NOT, NEG

The bitwise logic instructions follow the same three-operand format as **ADD** and **SUB**, with dedicated opcodes.

5.2.1 AND — Bitwise AND

```
AND Rd, Rs1, Rs2        ; Rd = Rs1 & Rs2
```

Opcode `0x2`. Sets N and Z; **clears C and V**.

Listing 5.5: AND tests from `examples/tests/test_alu.s`

```
        ; AND mask: 0xFF0F & 0x0FF0 = 0x0F00
        LIX R8, 0xFF0F
        LIX R9, 0x0FF0
        AND R10, R8, R9
        MOV R4, R10
        LIX R5, 0x0F00
        JALX check_eq

        ; AND with zero: 0x1234 & 0 = 0
        LIX R8, 0x1234
        AND R10, R8, R0
        MOV R4, R10
        MOV R5, R0
        JALX check_eq
```

AND is the workhorse for bit masking. To isolate a single bit, AND the register with a power of two. To clear specific bits, AND with the complement of the mask.

AND Clears C and V

Unlike the arithmetic instructions, **AND** (and all bitwise logic instructions) always clears the carry and overflow flags. This is an intentional design choice: bitwise operations have no concept of carry or signed overflow, so the flags are reset to avoid stale values from a prior arithmetic instruction causing confusion.

5.2.2 OR — Bitwise OR

```
OR Rd, Rs1, Rs2          ; Rd = Rs1 | Rs2
```

Opcode `0x3`. Sets N and Z.

Listing 5.6: OR tests from `examples/tests/test_alu.s`

```
        ; OR basic: 0xF000 | 0x000F = 0xF00F
        LIX R8, 0xF000
        LIX R9, 0x000F
        OR R10, R8, R9
        MOV R4, R10
        LIX R5, 0xF00F
        JALX check_eq

        ; OR all bits: 0xFF00 | 0x00FF = 0xFFFF
        LIX R8, 0xFF00
        LIX R9, 0x00FF
        OR R10, R8, R9
        MOV R4, R10
        LIX R5, 0xFFFF
        JALX check_eq
```

OR is used to set specific bits without disturbing others. For example, to set bit 3 in a register: `LIX R9, 0x0008` / `OR R4, R4, R9`.

5.2.3 XOR — Bitwise Exclusive OR

```
XOR Rd, Rs1, Rs2         ; Rd = Rs1 ^ Rs2
```

Opcode `0x4`. Sets N and Z.

Listing 5.7: XOR tests from `examples/tests/test_alu.s`

```
        ; XOR toggle: 0xFFFF ^ 0xFF00 = 0x00FF
```

```
        LIX R8, 0xFFFF
        LIX R9, 0xFF00
        XOR R10, R8, R9
        MOV R4, R10
        LIX R5, 0x00FF
        JALX check_eq

        ; XOR self = zero: 0x1234 ^ 0x1234 = 0
        LIX R8, 0x1234
        XOR R10, R8, R8
        MOV R4, R10
        MOV R5, R0
        JALX check_eq
```

XOR-ing a register with itself always produces zero—a useful idiom for clearing a register (though on the SAMPO, you can also just use `MOV Rd, R0`). XOR-ing with `0xFFFF` flips every bit, effectively computing the one's complement.

5.2.4 NOT — Bitwise Complement

```
NOT Rd, Rs1             ; Rd = ~Rs1
```

NOT is a two-operand instruction: it inverts all bits of `Rs1` and stores the result in `Rd`. It is equivalent to `XOR Rd, Rs1, 0xFFFF`, but encoded more compactly. Sets N and Z.

5.2.5 NEG — Two's Complement Negate

```
NEG Rd, Rs1             ; Rd = 0 - Rs1
```

NEG computes the two's complement negation: the result is $0 - \text{Rs1}$. This is equivalent to `NOT` followed by `ADDI Rd, 1`, but encoded as a single instruction.

Listing 5.8: NEG tests from `examples/tests/test_alu.s`

```
        ; NEG basic: -(1) = 0xFFFF
        LIX R8, 0x0001
        NEG R10, R8
        MOV R4, R10
        LIX R5, 0xFFFF
        JALX check_eq

        ; NEG zero: -(0) = 0
```

```
        NEG R10, R0
        MOV R4, R10
        MOV R5, R0
        JALX check_eq
```

NEG sets N, Z, C, and V flags—the same as **SUB**, since negation is subtraction from zero.

5.3 Immediate Arithmetic: ADDI

ADDI is the SAMPO's only 16-bit immediate arithmetic instruction. It is a *two-operand* instruction: the destination register is also the first source operand.

```
ADDI Rd, imm8           ; Rd = Rd + sign_extend(imm8)
```

ADDI Is Two-Operand, Not Three

Unlike **ADD**, which takes three registers (`ADD Rd, Rs1, Rs2`), **ADDI** reads and writes the *same* register. The operation is Rd ← Rd + sign_ext(imm8). There is no way to add an immediate to one register and store the result in a different register using a single **ADDI**.

The encoding uses opcode `0x5` in Format I:

Bits 15:12	Bits 11:8	Bits 7:0
0101	Rd	imm8

For example, `ADDI R4, 10` encodes as:

`0101 0100 0000 1010` = `0x540A`

The 8-bit immediate is *sign-extended* to 16 bits before addition. This means:

- **Positive range**: 0 to +127 (`0x00` to `0x7F`)
- **Negative range**: −128 to −1 (`0x80` to `0xFF`)

Let's look at the ADDI test suite from `examples/tests/test_addi.s`:

Listing 5.9: ADDI tests from `examples/tests/test_addi.s`

```
        ; ADDI positive: 10 + 5 = 15
        LIX R8, 0x000A
        MOV R4, R8
        ADDI R4, 5
        LIX R5, 0x000F
```

```
        JALX check_eq

        ; ADDI negative: 10 + (-3) = 7
        LIX R8, 0x000A
        MOV R4, R8
        ADDI R4, -3
        LIX R5, 0x0007
        JALX check_eq

        ; ADDI max positive: 0 + 127 = 127
        MOV R4, R0
        ADDI R4, 127
        LIX R5, 0x007F
        JALX check_eq

        ; ADDI max negative: 0 + (-128) = 0xFF80
        MOV R4, R0
        ADDI R4, -128
        LIX R5, 0xFF80
        JALX check_eq
```

The last test is revealing: adding −128 to zero produces `0xFF80`. The immediate `0x80` (which represents −128 in signed 8-bit) is sign-extended to `0xFF80` (which is −128 in signed 16-bit) before the addition.

5.3.1 Common ADDI Idioms

ADDI is one of the most frequently used instructions in SAMPO programs. Here are the common patterns:

Listing 5.10: Common ADDI usage patterns

```
        ADDI R4, 1          ; Increment: R4 = R4 + 1
        ADDI R4, -1         ; Decrement: R4 = R4 - 1
        ADDI SP, -2         ; Allocate 2 bytes on stack
        ADDI SP, 2          ; Deallocate 2 bytes from stack
```

Because **ADDI** is a 16-bit instruction (one word), it is more compact than loading a small constant with **LIX** (two words) and then using **ADD**. For values in the range −128 to +127, **ADDI** is always the right choice.

Listing 5.11: ADDI preserves upper bits

```
        ; ADDI preserves upper: 0xFF00 + 1 = 0xFF01
```

```
LIX R8, 0xFF00
MOV R4, R8
ADDI R4, 1
; R4 is now 0xFF01 (upper byte preserved)
```

The sign extension of the immediate does not affect the upper byte of the register in the way you might fear. The operation is a full 16-bit addition: `0xFF00` + `0x0001` = `0xFF01`. The upper byte is "preserved" only in the sense that adding a small positive number does not disturb it—this is ordinary arithmetic, not a byte-level operation.

5.4 The FLAGS Register in Depth

Every arithmetic and logic instruction updates some or all of the SAMPO's FLAGS register. Understanding flags is essential for writing correct conditional code.

5.4.1 Flag Layout

The FLAGS register is 8 bits wide, with six defined bits:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
N	Z	C	V	H	I	—	—

N (bit 7) — Negative. Set if the most significant bit (bit 15) of the result is 1. In two's complement representation, this indicates a negative value.

Z (bit 6) — Zero. Set if the result is exactly zero (`0x0000`).

C (bit 5) — Carry. Set if the addition produced a carry out of bit 15, or if a subtraction produced *no borrow* (ARM-style inverted carry—see Section 5.5).

V (bit 4) — Overflow. Set if signed overflow occurred: the result's sign differs from what the signs of the operands predict. This happens when adding two positive numbers and getting a negative result, or adding two negative numbers and getting a positive result.

H (bit 3) — Half-carry. Set if there was a carry from bit 3 to bit 4. Used by the **`DAA`** instruction for BCD arithmetic.

I (bit 2) — Interrupt enable. When set, maskable interrupts are enabled. Controlled by **`EI`** and **`DI`** instructions, not by arithmetic operations.

5.4.2 Which Instructions Affect Which Flags

Not all instructions update all flags. The following table summarizes flag effects by instruction class:

Instruction Class	N	Z	C	V	H
ADD, ADDI, ADC	*	*	*	*	*
SUB, SBC, CMP, NEG	*	*	*	*	*
AND, OR, XOR, NOT, TEST	*	*	0	0	0
Shift operations	*	*	*	0	0
MUL, DIV, REM	*	*	–	–	–
Load/Store	–	–	–	–	–
Branch/Jump	–	–	–	–	–

Table 5.1: Flag effects by instruction class. * = affected, 0 = cleared, – = unchanged

Logic Instructions Clear C and V

This is a deliberate design choice. After an **AND**, **OR**, or **XOR**, the carry and overflow flags are guaranteed to be zero. This means you can safely use a branch like **BCS** after a logic instruction and it will never be taken—a property that can be useful (or confusing) depending on whether you expect it.

5.4.3 Reading and Writing Flags Directly

The **GETF** and **SETF** instructions allow direct access to the FLAGS register:

```
GETF Rd          ; Rd = FLAGS (zero-extended to 16 bits)
SETF Rs          ; FLAGS = Rs[7:0]
```

This is used in the test suite to verify specific flag states after operations:

Listing 5.12: Flag verification from `examples/tests/test_alu.s`

```
        ; ADD carry flag: 0xFFFF + 1 sets C
        LIX R8, 0xFFFF
        LIX R9, 0x0001
        ADD R10, R8, R9
        GETF R4
        ; Expect: N=0, Z=1, C=1, V=0
        ; flags[7:4] = 0110 = 0x60
        LIX R8, 0x00F0
        AND R4, R4, R8      ; Mask to upper nibble
        LIX R5, 0x0060
        JALX check_eq
```

Let's trace through the flag computation for `0xFFFF + 0x0001`:

- Result = `0x0000` (wraps around)
- N = 0 (bit 15 of result is 0)
- Z = 1 (result is zero)
- C = 1 (carry out of bit 15—the addition overflowed the 16-bit range)
- V = 0 (no signed overflow: `0xFFFF` is -1 and `0x0001` is $+1$; their sum 0 is representable)

So FLAGS = `0110 0000` = `0x60`. After masking with `0xF0` we get `0x60`, confirming the test passes.

5.5 Carry Semantics: ARM-Style Inverted Carry

This is one of the most important—and most commonly misunderstood—aspects of the SAMPO architecture. The carry flag (C) behaves differently for addition and subtraction.

5.5.1 Carry on Addition

For **`ADD`** and **`ADC`**, the carry flag is set in the intuitive way:

C = 1 if the addition produced a carry out of bit 15.

For unsigned arithmetic, C = 1 means the result exceeds `0xFFFF`. This is the same convention used by virtually every processor.

5.5.2 Carry on Subtraction: Inverted Borrow

For **`SUB`**, **`SBC`**, and **`CMP`**, the SAMPO uses *ARM-style inverted borrow*:

C = 1 means *no borrow* occurred.
C = 0 means a borrow *did* occur.

This is the opposite of what the Z80 and x86 do. On those processors, the carry flag after a subtraction indicates a borrow (C = 1 means borrow). The SAMPO follows ARM's convention instead.

ARM-Style Carry for Subtraction

The SAMPO computes subtraction as $A - B = A + \overline{B} + 1$ (adding the bitwise complement of B, plus one). The carry out of this addition becomes the C flag. When $A \geq B$ (unsigned), the addition does carry out, so C = 1. When $A < B$, no carry out occurs, so C = 0. In other words: **C = 1 means $A \geq B$ (unsigned)**, and **C = 0 means $A < B$ (unsigned)**.

Let's verify with the test suite:

Listing 5.13: SUB carry flag test from `examples/tests/test_alu.s`

```
; SUB no-borrow flag: 5 - 3 sets C=1 (no borrow)
LIX R8, 0x0005
LIX R9, 0x0003
SUB R10, R8, R9
GETF R4
; Expect: C=1 (no borrow because 5 >= 3)
LIX R8, 0x0020       ; C flag is bit 5
AND R4, R4, R8
LIX R5, 0x0020
JALX check_eq
```

Here, $5 - 3 = 2$. Since $5 \geq 3$ (no borrow needed), C = 1. The test masks out just the C bit (bit 5 = `0x20`) and confirms it is set.

5.5.3 Why This Convention Matters

The inverted borrow convention affects which branch instructions you use after a comparison:

Condition	SAMPO Branch	Flag Test
$A < B$ (unsigned)	**BLTU**	C = 0
$A \geq B$ (unsigned)	**BGEU**	C = 1
$A = B$	**BEQ**	Z = 1
$A \neq B$	**BNE**	Z = 0

Table 5.2: Unsigned comparison branches with ARM-style carry

Don't Confuse BCS with "Borrow"

On the SAMPO, **BCS** (Branch if Carry Set, C = 1) after a **CMP** means "branch if the first operand is greater than or equal to the second" (unsigned). On the Z80 or x86, carry set after a comparison means the opposite. If you are coming from a Z80 background, this is the single most likely source of bugs when you start writing SAMPO code.

5.5.4 Overflow Flag (V)

The overflow flag detects *signed* arithmetic errors—when the result of an addition or subtraction falls outside the representable range of signed 16-bit integers (-32768 to $+32767$).

Listing 5.14: ADD overflow flag test from `examples/tests/test_alu.s`

```
; ADD overflow: 0x7FFF + 1 sets V
```

```
        LIX R8, 0x7FFF
        LIX R9, 0x0001
        ADD R10, R8, R9
        GETF R4
        ; Expect: N=1, Z=0, C=0, V=1
        ; flags[7:4] = 1001 = 0x90
        LIX R8, 0x00F0
        AND R4, R4, R8
        LIX R5, 0x0090
        JALX check_eq
```

Adding $+32767$ (`0x7FFF`) and $+1$ produces `0x8000`, which is -32768 in signed interpretation. Two positive numbers should not produce a negative result—the V flag catches this.

The overflow flag is computed as:

$$V = (\text{operand signs match}) \wedge (\text{result sign differs from operands})$$

Specifically:

- For addition: $V = (A[15] = B[15]) \wedge (R[15] \neq A[15])$
- For subtraction: $V = (A[15] \neq B[15]) \wedge (R[15] \neq A[15])$

Signed branches (**BLT**, **BGE**, **BGT**, **BLE**) use the relationship between N and V, not just N alone. We will cover these in detail in Chapter 7.

5.6 CMP and TEST

Two instructions exist solely to set flags without storing a result: **CMP** and **TEST**.

5.6.1 CMP — Compare

```
CMP Rd, Rs1             ; flags = Rd - Rs1 (result discarded)
```

CMP performs a subtraction (`Rd` − `Rs1`), sets all arithmetic flags (N, Z, C, V), and discards the result. Neither operand register is modified.

CMP is encoded with opcode `0xC`, func `0x2`:

Bits 15:12	Bits 11:8	Bits 7:4	Bits 3:0
1100	Rd	Rs1	0010

CMP is the standard way to set up flags before a conditional branch:

Listing 5.15: CMP flag tests from `examples/tests/test_stack.s`

```
        ; CMP equal: Z flag set
        LIX R8, 0x0042
        CMP R8, R8
        GETF R4
        LIX R8, 0x0040      ; Z flag mask (bit 6)
        AND R4, R4, R8
        LIX R5, 0x0040
        JALX check_eq       ; Z should be set

        ; CMP less (signed): 3 < 5, N!=V
        LIX R8, 0x0003
        LIX R9, 0x0005
        CMP R8, R9
        BLT cmp_lt_ok
        ; ... not taken means failure ...
```

5.6.2 TEST — Bitwise Test

```
TEST Rd, Rs1            ; flags = Rd & Rs1 (result discarded)
```

TEST performs a bitwise AND, sets N and Z (and clears C and V), and discards the result. It is used to check whether specific bits are set in a register without modifying either operand.

Listing 5.16: TEST examples from `examples/tests/test_stack.s`

```
        ; TEST: 0xFF00 & 0x00FF = 0, Z should be set
        LIX R8, 0xFF00
        LIX R9, 0x00FF
        TEST R8, R9
        GETF R4
        LIX R8, 0x0040      ; Z flag mask
        AND R4, R4, R8
        LIX R5, 0x0040
        JALX check_eq

        ; TEST non-zero: 0xFF & 0xFF != 0, Z should be clear
        LIX R8, 0x00FF
        TEST R8, R8
        GETF R4
        LIX R8, 0x0040
        AND R4, R4, R8
        MOV R5, R0          ; Z should be clear (0)
```

```
        JALX check_eq

        ; TEST clears C and V flags
        SCF                 ; Set carry first
        LIX R8, 0xFFFF
        TEST R8, R8
        GETF R4
        LIX R8, 0x0030      ; C+V mask
        AND R4, R4, R8
        MOV R5, R0          ; Both should be clear
        JALX check_eq
```

The last test confirms that **TEST** clears C and V even when they were previously set by **SCF**.

CMP vs. TEST: When to Use Which

Use **CMP** when you want to compare two values by subtraction—it answers "is A less than, equal to, or greater than B?" Use **TEST** when you want to check whether specific bits are set—it answers "is bit n of A nonzero?" The mnemonic helps: CMP for *compare*, TEST for *test bits*.

5.7 Worked Examples

Let's put everything together with some practical examples that combine arithmetic, logic, and flag testing.

5.7.1 Absolute Value

Computing the absolute value of a signed 16-bit number:

Listing 5.17: Computing absolute value

```
; abs(R4): R4 = |R4|
abs:
        CMP  R4, R0         ; Compare with zero
        BGE  abs_done       ; Already non-negative? Skip
        NEG  R4, R4         ; Negate to make positive
abs_done:
        JR   RA
```

After the **CMP**, if R4 is negative (signed), the N flag will be set and V will be clear, making $N \neq V$ true—so **BGE** (which tests $N = V$) will *not* be taken, and we fall through to the **NEG**.

5.7.2 Maximum of Two Values

Finding the larger of two signed values:

Listing 5.18: Signed maximum: R4 = max(R4, R5)

```
; max(R4, R5): result in R4
max:
        CMP  R4, R5          ; Compare R4 with R5
        BGE  max_done         ; R4 >= R5? Keep R4
        MOV  R4, R5           ; Otherwise, R4 = R5
max_done:
        JR   RA
```

5.7.3 16-Bit Population Count

Counting the number of set bits in a 16-bit value is a classic exercise in bit manipulation. Here is a loop-based approach:

Listing 5.19: Population count: R4 = popcount(R5)

```
; popcount: count set bits in R5, result in R4
popcount:
        MOV  R4, R0           ; count = 0
        CMP  R5, R0
        BEQ  pop_done
pop_loop:
        MOV  R8, R5
        ADDI R8, -1           ; R8 = R5 - 1
        AND  R5, R5, R8       ; Clear lowest set bit
        ADDI R4, 1            ; Increment count
        CMP  R5, R0           ; More bits?
        BNE  pop_loop
pop_done:
        JR   RA
```

This uses Kernighan's algorithm: `n & (n-1)` clears the lowest set bit of n. The loop runs once per set bit, not once per total bit position.

5.7.4 32-Bit Addition

As mentioned in Section 5.1.3, multi-word arithmetic chains **ADD** and **ADC**:

Listing 5.20: 32-bit addition with carry propagation

```
; 32-bit add: (R5:R4) += (R7:R6)
; R4/R6 = low 16 bits, R5/R7 = high 16 bits
add32:
        ADD  R4, R4, R6     ; Add low words (sets C if overflow)
        ADC  R5, R5, R7     ; Add high words plus carry
        JR   RA
```

If the low-word addition overflows (e.g., `0xC000` + `0x5000`), C is set to 1, and **ADC** adds that extra 1 to the high-word sum. Without **ADC**, the result would be wrong by one whenever the low words overflow.

5.7.5 Checking a Bit Field

Extracting and checking a multi-bit field from a register is a common task when dealing with hardware status registers:

Listing 5.21: Extracting a 3-bit field

```
; Extract bits [6:4] from R4 into R5
        LIX  R8, 0x0070      ; Mask: bits 6, 5, 4
        AND  R5, R4, R8      ; Isolate the field
        SRL  R5, R5          ; Shift right by 1 (bit 4 -> bit 3)
        SRL  R5, R5          ; Shift right by 1 (bit 3 -> bit 2)
        SRL  R5, R5          ; Shift right by 1 (bit 2 -> bit 1)
        SRL  R5, R5          ; Shift right by 1 (bit 1 -> bit 0)
        ; R5 now contains the 3-bit value in bits [2:0]
```

Four individual shifts are needed here because the basic shift instructions shift by only 1 bit. In Chapter 8, we will see the **SRL4** instruction that can shift by 4 in a single step, making this much more efficient.

5.8 Exercises

1. **Flag prediction.** For each of the following operations, predict the N, Z, C, and V flags *before* running the code. Then verify your predictions in the emulator using **GETF**.

   - **ADD** of `0x8000` + `0x8000`
   - **SUB** of `0x0000` − `0x0001`
   - **AND** of `0xFFFF` & `0x0000`
   - **ADDI** of `0x007F` + 1

2. **32-bit subtraction.** Write a subroutine that computes 32-bit subtraction using **SUB** and **SBC**, following the same pattern as the 32-bit addition example. Test it with values where the low-word subtraction requires a borrow.

3. **Sign extension by hand.** Write a subroutine that takes an unsigned byte value in R4 (range 0–255) and produces the sign-extended 16-bit result. Use **AND** to check bit 7, and **OR** with 0xFF00 if the sign bit is set. Verify that your routine agrees with **LB** for all values from 0 to 255.

4. **Division by constant.** Write a subroutine that divides a 16-bit unsigned value in R4 by 10 using only shifts, additions, and subtractions (no **DIV** instruction). Hint: multiplying by the reciprocal $\lfloor 2^{16}/10 \rfloor = 6553$ and taking the upper 16 bits gives an approximation.

5. **BCD addition.** Write a program that adds two BCD-encoded bytes. For example, adding the BCD values 0x19 and 0x23 should produce 0x42 (19 + 23 = 42 in decimal). Use **ADD** followed by **DAA** to correct the result. We will cover **DAA** in detail in Chapter 8.

What's Next. We can now compute with numbers and make decisions based on flags. But our programs so far work entirely in registers—16 of them are not enough for real work. In Chapter 6, we will explore the SAMPO's memory system: load and store instructions, offset addressing, and the stack that makes function calls and local variables possible.

Chapter 6

Memory and the Stack

Sixteen registers are not enough. Any program beyond a toy needs to store data in memory—arrays, strings, structures, return addresses. The SAMPO is a *load/store* architecture: arithmetic instructions operate only on registers, and separate load and store instructions move data between registers and memory. This chapter covers those memory instructions, the offset addressing modes that make them flexible, and the stack mechanism that ties everything together for function calls and local variables.

6.1 Load/Store Architecture

Load/Store Architecture

In a *load/store architecture*, memory is accessed only through dedicated load (read) and store (write) instructions. Arithmetic and logic instructions operate exclusively on registers. This is the design used by RISC-V, ARM, and MIPS—and by the SAMPO.

The SAMPO provides three types of load and two types of store in its base (16-bit) instruction set, plus extended (32-bit) variants for larger offsets:

All load and store instructions leave the FLAGS register unchanged.

6.1.1 LW — Load Word

```
LW Rd, (Rs1)            ; Rd = mem16[Rs1]
```

Instruction	Width	Description
LW Rd, (Rs1)	16-bit word	Load word
LB Rd, (Rs1)	8-bit byte	Load byte, sign-extend to 16 bits
LBU Rd, (Rs1)	8-bit byte	Load byte, zero-extend to 16 bits
SW (Rs1), Rs2	16-bit word	Store word
SB (Rs1), Rs2	8-bit byte	Store low byte of Rs2

Table 6.1: Base load and store instructions

LW reads a 16-bit word from the memory address contained in `Rs1` and places it in `Rd`. The SAMPO is little-endian: the byte at address `Rs1` becomes the low byte of `Rd`, and the byte at address `Rs1` + 1 becomes the high byte.

The encoding uses opcode `0x6` with func = `0x0`:

Bits 15:12	Bits 11:8	Bits 7:4	Bits 3:0
`0110`	Rd	Rs1	`0000`

6.1.2 LB and LBU — Load Byte

```
LB  Rd, (Rs1)          ; Rd = sign_extend(mem8[Rs1])
LBU Rd, (Rs1)          ; Rd = zero_extend(mem8[Rs1])
```

Both instructions read a single byte from memory. The difference is how the byte is extended to fill the 16-bit register:

- **LB** *sign-extends*: if bit 7 of the byte is 1, the upper byte of the register is filled with `0xFF`. For example, loading the byte `0x80` produces `0xFF80`.
- **LBU** *zero-extends*: the upper byte is always zero. Loading `0x80` produces `0x0080`.

Listing 6.1: LB vs. LBU from `examples/tests/test_loadstore.s`

```
        ; LB sign extension: store 0x80, LB gives 0xFF80
        LIX R13, 0x8000
        SW (R13), R0           ; Clear the word first
        LIX R9, 0x0080
        SB (R13), R9           ; Store byte 0x80
        LB R4, (R13)           ; Sign-extend: 0xFF80
        LIX R5, 0xFF80
        JALX check_eq
```

```
        ; LBU zero extension: store 0xFF, LBU gives 0x00FF
        SW (R13), R0
        LIX R9, 0x00FF
        SB (R13), R9
        LBU R4, (R13)          ; Zero-extend: 0x00FF
        LIX R5, 0x00FF
        JALX check_eq
```

Use LBU for ASCII Characters

When processing text (as in the Hello program), always use **LBU**. ASCII characters have values 0–127 and the sign bit is never set, so **LB** and **LBU** produce the same result. But if your data includes bytes above 127 (extended ASCII, binary data), **LB** would sign-extend them to negative 16-bit values, which is rarely what you want for character processing.

6.1.3 SW — Store Word

```
SW (Rs1), Rs2            ; mem16[Rs1] = Rs2
```

SW writes the full 16-bit value of Rs2 to the memory address in Rs1. The low byte goes to address Rs1, the high byte to address Rs1 + 1.

Note the operand order: the *address* comes first (in parentheses), the *data* comes second. This is consistent with the store instruction conventions in MIPS and RISC-V.

The encoding uses opcode 0x7 with func = 0x0:

Bits 15:12	Bits 11:8	Bits 7:4	Bits 3:0
0111	Rs2	Rs1	0000

Store Encoding: Rs2 in the Rd Field

In store instructions, the register field at bits 11:8 contains the *data source* register (Rs2), not a destination. This is different from load instructions, where that field contains the destination register. The encoding reuses the same bit position for consistency, but the semantics differ.

6.1.4 SB — Store Byte

```
SB (Rs1), Rs2            ; mem8[Rs1] = Rs2[7:0]
```

SB writes only the low byte of `Rs2` to memory. The upper byte of the register is ignored, and only one byte in memory is modified.

6.1.5 A Complete Load/Store Round-Trip

Here is a complete test demonstrating a store-then-load round-trip:

Listing 6.2: SW/LW round-trip from `examples/tests/test_loadstore.s`

```
; Use 0x8000 as test data area
LIX R13, 0x8000

; SW/LW round-trip: store 0x1234, load it back
LIX R9, 0x1234
SW (R13), R9          ; Store 0x1234 at address 0x8000
LW R4, (R13)          ; Load it back
LIX R5, 0x1234
JALX check_eq          ; Should match

; SB/LBU round-trip: store byte 0x56, load back
SW (R13), R0           ; Clear the word first
LIX R9, 0x0056
SB (R13), R9           ; Store just byte 0x56
LBU R4, (R13)          ; Load byte, zero-extend
LIX R5, 0x0056
JALX check_eq
```

6.2 Offset Addressing

The base load and store instructions use register-indirect addressing with no offset: the address is exactly the value in the base register. But the SAMPO also provides several *fixed-offset* variants that add a small constant to the base address.

6.2.1 Load with Offset

The load instruction's func field (bits 3:0) selects the offset:

The offsets are hard-coded in the instruction encoding—you cannot use an arbitrary offset in a 16-bit load instruction. The available offsets ($+2$, $+4$, $+6$, -2, -4) are chosen to efficiently access small structures and stack frames where fields are separated by the 16-bit word size.

Listing 6.3: Offset load tests from `examples/tests/test_loadstore.s`

```
; LW offset +2: store at addr+2, load with 2(addr)
```

func	Syntax	Operation
0x0	`LW Rd, (Rs1)`	$Rd = mem16[Rs1]$
0x3	`LW Rd, 2(Rs1)`	$Rd = mem16[Rs1 + 2]$
0x4	`LW Rd, 4(Rs1)`	$Rd = mem16[Rs1 + 4]$
0x5	`LW Rd, 6(Rs1)`	$Rd = mem16[Rs1 + 6]$
0x6	`LW Rd, -2(Rs1)`	$Rd = mem16[Rs1 - 2]$
0x7	`LW Rd, -4(Rs1)`	$Rd = mem16[Rs1 - 4]$

Table 6.2: Load word offset variants

```
        LIX R8, 0x8010
        LIX R9, 0x5678
        LIX R10, 0x8012       ; addr+2
        SW (R10), R9           ; Store at 0x8012
        LW R4, 2(R8)          ; Load from R8+2 = 0x8012
        LIX R5, 0x5678
        JALX check_eq

        ; LW offset -2: load from addr-2
        LIX R8, 0x8022
        LIX R9, 0xDEF0
        LIX R10, 0x8020        ; addr-2
        SW (R10), R9
        LW R4, -2(R8)         ; Load from R8-2 = 0x8020
        LIX R5, 0xDEF0
        JALX check_eq
```

6.2.2 Store with Offset

Store instructions have matching offset variants:

These offsets are ideal for accessing fields of a structure when a register holds the base address:

Listing 6.4: Accessing a structure with offset stores

```
; A 4-field structure at address in R8:
; field0 at offset 0, field1 at +2, field2 at +4, field3 at +6
        SW   (R8), R4          ; Store field0
        SW   2(R8), R5         ; Store field1
        SW   4(R8), R6         ; Store field2
        SW   6(R8), R7         ; Store field3
```

func	Syntax	Operation
0x0	`SW (Rs1), Rs2`	mem16[Rs1] = Rs2
0x2	`SW 2(Rs1), Rs2`	mem16[Rs1 + 2] = Rs2
0x3	`SW 4(Rs1), Rs2`	mem16[Rs1 + 4] = Rs2
0x4	`SW 6(Rs1), Rs2`	mem16[Rs1 + 6] = Rs2
0x5	`SW -2(Rs1), Rs2`	mem16[Rs1 − 2] = Rs2
0x6	`SW -4(Rs1), Rs2`	mem16[Rs1 − 4] = Rs2

Table 6.3: Store word offset variants

6.3 Extended Load/Store: LWX and SWX

When the fixed offsets (±2, ±4, +6) are not sufficient, the SAMPO provides 32-bit *extended* load and store instructions that accept a full 16-bit offset.

6.3.1 LWX — Load Word Extended

```
LWX Rd, imm16(Rs)       ; Rd = mem16[Rs + imm16]
```

LWX is a 32-bit instruction: the first word encodes the opcode (`0xF`), destination register, base register, and sub-opcode (`0x5`); the second word is the 16-bit offset.

Word 0			
Bits 15:12	**Bits 11:8**	**Bits 7:4**	**Bits 3:0**
1111	Rd	Rs	0101

Word 1
imm16 (16-bit signed offset)

LWX can access any location in the 64 KB address space relative to a base register, making it suitable for large arrays, lookup tables, and data structures that exceed the reach of the fixed-offset variants.

6.3.2 SWX — Store Word Extended

```
SWX imm16(Rs), Rd       ; mem16[Rs + imm16] = Rd
```

SWX is the store counterpart, using sub-opcode `0x6`. Same 32-bit format, with the 16-bit offset in the second word.

Extended Instructions Cost Two Words

LWX and **SWX** are each 4 bytes (32 bits). Use the base **LW**/**SW** with fixed offsets whenever possible—they are half the size and execute in the same number of cycles. Reserve the extended forms for offsets that do not match the fixed set.

6.4 Loading 16-Bit Immediates: LIX

We have used **LIX** throughout the previous chapters without fully explaining it. Now is the time.

```
LIX Rd, imm16           ; Rd = imm16
```

LIX loads a full 16-bit immediate value into a register. It is a 32-bit instruction using opcode `0xF` with sub-opcode `0x7`:

Word 0			
Bits 15:12	**Bits 11:8**	**Bits 7:4**	**Bits 3:0**
1111	Rd	`0000`	`0111`

Word 1
imm16

For example, `LIX R4, 0x1234` encodes as:

Word 0: `0xF407` Word 1: `0x1234`

Listing 6.5: LIX tests from `examples/tests/test_loadstore.s`

```
        ; LIX loads 16-bit immediate
        LIX R4, 0xBEEF
        LIX R5, 0xBEEF
        JALX check_eq

        ; LIX zero
        LIX R4, 0x0000
        MOV R5, R0
        JALX check_eq
```

6.4.1 LIX vs. ADDI for Small Constants

For values in the range -128 to $+127$, you can initialize a register using **MOV** from `R0` followed by **ADDI**:

Listing 6.6: Loading small constants: LIX vs. MOV+ADDI

```
; Using LIX (4 bytes):
LIX  R4, 42

; Using MOV + ADDI (4 bytes total):
MOV  R4, R0          ; R4 = 0
ADDI R4, 42          ; R4 = 42
```

Both approaches use the same number of bytes (4). However, **LIX** is a single instruction (one fetch-decode-execute cycle in the pipeline), while **MOV** + **ADDI** is two. For values outside the -128 to $+127$ range, **LIX** is the only option.

When to Use LIX

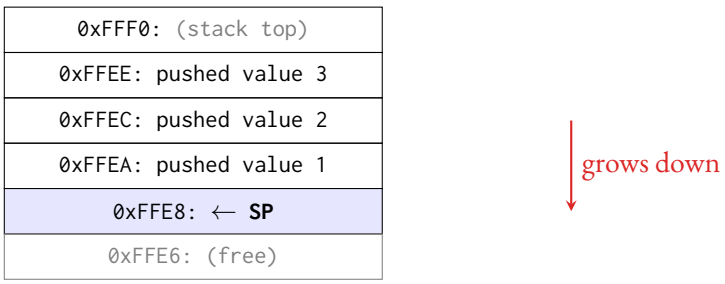
Use **LIX** whenever you need a 16-bit value in a register and do not already have a useful starting value. Common uses: loading memory addresses (pointers), I/O port addresses, bitmasks, and test constants. It is the SAMPO's equivalent of ARM's `LDR Rd, =value` pseudo-instruction.

6.5 The Stack

The SAMPO uses register R2 (alias SP) as the stack pointer. The stack grows *downward*—toward lower memory addresses—following the convention used by ARM, x86, and RISC-V.

6.5.1 Stack Organization

After reset, SP is initialized to 0xFFFF (actually, programs typically set it explicitly near the top of RAM). As values are pushed onto the stack, SP decreases. As values are popped, SP increases.



Initialize SP Early

The SAMPO reference manual specifies that SP defaults to `0xFFFF` after reset, but most test programs explicitly set it with `LIX R2, 0xFFF0` to ensure a known, word-aligned stack base. Always initialize SP before any **PUSH** or function call.

6.6 PUSH and POP

The **PUSH** and **POP** instructions provide atomic stack operations:

6.6.1 PUSH

```
PUSH Rs1                ; SP = SP - 2; mem16[SP] = Rs1
```

PUSH decrements SP by 2 (making room for one 16-bit word), then stores the register value at the new SP address. This is a "pre-decrement" operation.

Encoded with opcode `0xC`, func `0x0`:

Bits 15:12	Bits 11:8	Bits 7:4	Bits 3:0
1100	0000	Rs1	0000

6.6.2 POP

```
POP Rd                  ; Rd = mem16[SP]; SP = SP + 2
```

POP reads the 16-bit word at the current SP address into `Rd`, then increments SP by 2. This is a "post-increment" operation.

Encoded with opcode `0xC`, Rd in bits 11:8, func `0x1`:

Bits 15:12	Bits 11:8	Bits 7:4	Bits 3:0
1100	Rd	0000	0001

6.6.3 PUSH/POP Round-Trip

The most fundamental property of the stack: what you push, you can pop back:

Listing 6.7: PUSH/POP round-trip from `examples/tests/test_stack.s`

```
        ; PUSH/POP single register round-trip
        LIX R8, 0x1234
```

```
        PUSH R8
        POP R4
        LIX R5, 0x1234
        JALX check_eq          ; R4 should be 0x1234
```

6.6.4 LIFO Ordering

The stack is Last-In, First-Out. If you push two values, the first pop returns the *second* pushed value:

Listing 6.8: LIFO order verification from `examples/tests/test_stack.s`

```
        ; PUSH/POP LIFO order
        LIX R8, 0xAAAA
        LIX R9, 0xBBBB
        PUSH R8                ; Push first
        PUSH R9                ; Push second
        POP R4                 ; Pop: should be 0xBBBB (second)
        LIX R5, 0xBBBB
        JALX check_eq

        ; Second pop gives first pushed value
        POP R4                 ; Pop: should be 0xAAAA (first)
        LIX R5, 0xAAAA
        JALX check_eq
```

6.6.5 SP Preservation

A matched pair of **PUSH** and **POP** leaves SP unchanged:

Listing 6.9: SP preservation test from `examples/tests/test_stack.s`

```
        ; SP preserved after push/pop pair
        MOV R8, R2             ; Save original SP
        LIX R9, 0x5555
        PUSH R9
        POP R10
        ; SP should be back to original
        MOV R4, R2
        MOV R5, R8
        JALX check_eq          ; SP should match saved value
```

Mismatched PUSH/POP Corrupts the Stack

Every **PUSH** must eventually be balanced by a **POP** (or equivalent SP adjustment). If you push three registers but only pop two, the stack pointer is off by 2 bytes—and the next **POP** or function return will read the wrong value. Stack imbalance bugs are among the most common and hardest-to-debug errors in assembly programming.

6.7 Stack Frames and Local Variables

The stack is not just for saving registers—it is the mechanism that makes function calls possible. Each function call creates a *stack frame*: a region of the stack that holds the function's return address, saved registers, and local variables.

6.7.1 The Calling Convention

The SAMPO calling convention (detailed in the reference manual) defines how functions communicate:

Role	Registers	Responsibility
Arguments	R4–R7 (A0–A3)	Caller fills before call
Return values	R4–R5 (A0–A1)	Callee fills before return
Caller-saved	R4–R11	May be clobbered by callee
Callee-saved	R12–R15 (S0–S3)	Must be preserved by callee
Return address	R1 (RA)	Set by JALX/JALR
Stack pointer	R2 (SP)	Must be preserved by callee

Table 6.4: The SAMPO calling convention

The key principle: if a function uses any callee-saved register (S0–S3), it must push that register's value on entry and pop it before returning. The caller-saved registers (A0–A3, T0–T3) are "free" for the callee to use without saving.

6.7.2 Function Prologue

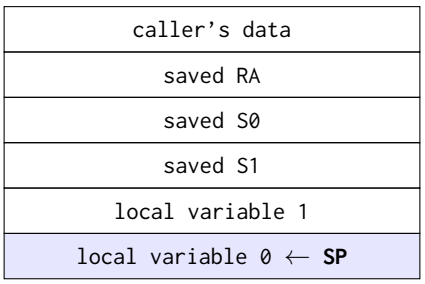
A function prologue saves the return address and any callee-saved registers the function will use, then allocates space for local variables:

Listing 6.10: Standard function prologue

```
my_function:
        PUSH RA                 ; Save return address
```

```
        PUSH S0             ; Save callee-saved registers
        PUSH S1             ;   (only those we will use)
        ADDI SP, -4         ; Allocate 4 bytes for locals
```

After the prologue, the stack looks like this:



6.7.3 Function Epilogue

The epilogue undoes the prologue in reverse order:

Listing 6.11: Standard function epilogue

```
        ADDI SP, 4          ; Deallocate locals
        POP  S1             ; Restore callee-saved registers
        POP  S0             ;   (reverse order of prologue)
        POP  RA             ; Restore return address
        JR   RA             ; Return to caller
```

The LIFO nature of the stack means the pops must occur in the *reverse order* of the pushes. Push RA first, pop it last.

6.7.4 Accessing Local Variables

Local variables allocated on the stack are accessed using offset loads and stores relative to SP. With the stack frame shown above:

Listing 6.12: Accessing stack-allocated local variables

```
        ; Store a value to local variable 0 (at SP)
        SW   (SP), R4

        ; Store a value to local variable 1 (at SP+2)
        SW   2(SP), R5

        ; Load local variable 0 back
```

```
        LW   R4, (SP)

        ; Load local variable 1 back
        LW   R5, 2(SP)
```

The fixed-offset load and store instructions (`LW Rd, 2(Rs1)`, `SW 2(Rs1), Rs2`, etc.) are designed exactly for this purpose. For stack frames with more than three words of local storage, use the extended **LWX** and **SWX** instructions.

6.7.5 A Complete Function Example

Let's look at the test framework's `print_str` function, which is used throughout the test suite. It demonstrates a real function with prologue, epilogue, and callee-saved register usage:

Listing 6.13: The print_str function from the test framework

```
print_str:
        PUSH R12            ; Save callee-saved register
        MOV R12, R4         ; Copy string pointer to S0
ps_loop:
        LBU R5, (R12)       ; Load next character
        CMP R5, R0          ; Null terminator?
        BEQ ps_done
ps_wait:
        INI R6, ACIA_STATUS ; Read serial status
        AND R7, R6, R6      ; Copy to R7
        ADDI R7, -2         ; Check TX ready
        BNE ps_wait
        OUTI ACIA_DATA, R5  ; Send character
        ADDI R12, 1         ; Advance pointer
        J ps_loop
ps_done:
        POP R12             ; Restore callee-saved register
        JR RA               ; Return to caller
```

Notice the pattern:

- **PUSH** R12 at the start saves the callee-saved register (the function uses R12 to hold the string pointer).
- R4 (the argument) is moved to R12 because R4 is caller-saved and may be clobbered if this function called another function.
- **POP** R12 at the end restores the register.

- **JR** RA returns to the caller.

This function does not push RA because it is a *leaf function*—it never calls another function (the busy-wait and OUTI are instructions, not function calls). If `print_str` called another function, it would need to push RA to avoid losing the return address.

The `check_eq` function in the test framework demonstrates the non-leaf case:

Listing 6.14: The check_eq function — a non-leaf function

```
check_eq:
        PUSH RA             ; Save RA (we will call print_str)
        CMP R4, R5
        BNE ce_fail
        LIX R4, str_pass
        JALX print_str      ; This call overwrites RA
        POP RA              ; Restore our RA
        JR RA
ce_fail:
        LIX R4, str_fail
        JALX print_str
        POP RA
        JR RA
```

The **PUSH** RA is essential: the **JALX** `print_str` will overwrite RA with `check_eq`'s own return address, destroying the address we need to return to *our* caller. By saving RA on the stack, we can restore it after the call.

6.8 Exercises

1. **Array sum.** Write a program that stores the values 1, 2, 3, 4, 5 in consecutive memory locations starting at `0x8000`, then loops through the array to compute the sum. Store the sum at `0x8100` and verify it equals 15 using **CMP** and a conditional branch to a pass/fail label.

2. **Memory copy.** Write a subroutine that copies n words from a source address to a destination address, using a loop with **LW**, **SW**, and **ADDI** to advance the pointers. Arguments: R4 = source, R5 = destination, R6 = word count.

3. **Nested function calls.** Write three functions: `outer`, `middle`, and `inner`. `outer` calls `middle`, which calls `inner`. Each function should push RA and any callee-saved registers it uses. Verify that all three return correctly by checking a value that `inner` places in R4.

4. **Stack-based swap.** Write a subroutine that swaps the values of R4 and R5 using the stack (PUSH and POP) without using any other registers. Verify the swap works.

5. **Byte vs. word alignment.** The SAMPO supports unaligned word access. Write a test that stores a word at an odd address (e.g., `0x8001`) and loads it back. Does the result match? Try the same experiment on the emulator and note any differences from aligned access.

What's Next. We can now store data in memory, build stack frames, and call functions. But so far our programs flow in straight lines or loop forever. In Chapter 7, we will master the SAMPO's rich set of branch and jump instructions—all 16 conditional branches, PC-relative and absolute jumps, and the subroutine calling mechanisms that make structured programs possible.

Chapter 7

Control Flow

A program that executes every instruction in sequence is not very useful. Control flow—branches, loops, function calls, and returns—is what transforms a list of instructions into a program that makes decisions and solves problems.

The SAMPO provides a rich set of control flow mechanisms: 16 conditional branch instructions, PC-relative and register jumps, extended absolute jumps, and a clean subroutine calling convention. In this chapter, we will explore them all, culminating in a recursive factorial implementation.

7.1 Branch Instructions: All 16 Conditions

Branch instructions change the PC conditionally, based on the flags set by a previous arithmetic or comparison instruction. The SAMPO provides 16 branch conditions—enough to cover every useful comparison between signed numbers, unsigned numbers, and individual flag bits.

7.1.1 Branch Encoding

All branch instructions share the same encoding: opcode `0x8` in Format B:

Bits 15:12	Bits 11:8	Bits 7:0
`1000`	cond	offset8

The 4-bit *cond* field selects which of the 16 conditions to test. The 8-bit *offset8* is a signed value that specifies the branch target relative to the next instruction.

cond	Mnemonic	Condition	Flag Test	Typical Use After CMP
0x0	**BEQ**	Equal	Z = 1	$A = B$
0x1	**BNE**	Not equal	Z = 0	$A \neq B$
0x2	**BLT**	Less than (signed)	N ≠ V	$A < B$ (signed)
0x3	**BGE**	Greater or equal (signed)	N = V	$A \geq B$ (signed)
0x4	**BLTU**	Less than (unsigned)	C = 0	$A < B$ (unsigned)
0x5	**BGEU**	Greater or equal (unsigned)	C = 1	$A \geq B$ (unsigned)
0x6	**BMI**	Minus	N = 1	Result negative
0x7	**BPL**	Plus	N = 0	Result positive or zero
0x8	**BVS**	Overflow set	V = 1	Signed overflow
0x9	**BVC**	Overflow clear	V = 0	No signed overflow
0xA	**BCS**	Carry set	C = 1	Carry/no borrow
0xB	**BCC**	Carry clear	C = 0	No carry/borrow
0xC	**BGT**	Greater than (signed)	Z=0 ∧ N=V	$A > B$ (signed)
0xD	**BLE**	Less or equal (signed)	Z=1 ∨ N≠V	$A \leq B$ (signed)
0xE	**BHI**	Higher (unsigned)	C=1 ∧ Z=0	$A > B$ (unsigned)
0xF	**BLS**	Lower or same (unsigned)	C=0 ∨ Z=1	$A \leq B$ (unsigned)

Table 7.1: All 16 SAMPO branch conditions

7.1.2 The Complete Condition Table

Signed vs. Unsigned Branches

The first six comparison conditions come in signed/unsigned pairs:

- Signed: **BLT**/**BGE**/**BGT**/**BLE** (use N and V flags)
- Unsigned: **BLTU**/**BGEU**/**BHI**/**BLS** (use C and Z flags)

Choose the right variant for your data. Comparing pointers or array indices? Use unsigned. Comparing temperatures or deltas? Use signed.

7.1.3 Testing the Branches

The test suite in `examples/tests/test_branch.s` verifies every branch condition with both taken and not-taken cases. Here is the pattern for **BEQ**:

Listing 7.1: BEQ taken and not-taken tests from `examples/tests/test_branch.s`

```
; BEQ taken when equal
LIX R8, 42
CMP R8, R8          ; Z = 1 (42 - 42 = 0)
BEQ beq_t_ok
JALX fail_and_next  ; Should not reach here
J beq_t_end
```

```
beq_t_ok:
        JALX pass_and_next
beq_t_end:

        ; BEQ not taken when not equal
        LIX R8, 42
        LIX R9, 43
        CMP R8, R9          ; Z = 0 (42 - 43 != 0)
        BEQ beq_nt_fail
        JALX pass_and_next  ; Should reach here (fall through)
        J beq_nt_end
beq_nt_fail:
        JALX fail_and_next
beq_nt_end:
```

Every branch test follows this template: set up flags with a **CMP** (or arithmetic instruction), execute the branch, and verify whether execution went to the expected target.

7.1.4 Signed Comparison Example: BLT and BGE

Signed comparisons use the relationship between the N and V flags:

Listing 7.2: BLT and BGE tests from `examples/tests/test_branch.s`

```
        ; BLT: 3 < 5 (signed)
        LIX R8, 0x0003
        LIX R9, 0x0005
        CMP R8, R9          ; 3 - 5: result negative, N!=V
        BLT blt_t_ok
        JALX fail_and_next
        J blt_t_end
blt_t_ok:
        JALX pass_and_next
blt_t_end:

        ; BGE: 5 >= 3 (signed)
        LIX R8, 0x0005
        LIX R9, 0x0003
        CMP R8, R9          ; 5 - 3 = 2: positive, N=V=0
        BGE bge_t_ok
        JALX fail_and_next
        J bge_t_end
bge_t_ok:
        JALX pass_and_next
bge_t_end:
```

For **BLT** (cond = 0x2), the test is $N \neq V$. When we compute $3 - 5 = -2$, the result is negative (N = 1) and there is no signed overflow (V = 0), so $N \neq V$ is true and the branch is taken.

For **BGE** (cond = 0x3), the test is $N = V$. When we compute $5 - 3 = 2$, both N and V are 0, so $N = V$ and the branch is taken.

7.1.5 Unsigned Comparison Example: BLTU and BGEU

Unsigned comparisons use the carry flag. Remember: the SAMPO uses ARM-style carry where $C = 1$ means *no borrow*:

Listing 7.3: BLTU and BGEU tests from `examples/tests/test_branch.s`

```
        ; BLTU: 3 < 5 (unsigned), C = 0 after CMP
        LIX R8, 0x0003
        LIX R9, 0x0005
        CMP R8, R9          ; 3 - 5 borrows, so C = 0
        BLTU bltu_t_ok
        JALX fail_and_next
        J bltu_t_end
bltu_t_ok:
        JALX pass_and_next
bltu_t_end:

        ; BGEU: 5 >= 3 (unsigned), C = 1 after CMP
        LIX R8, 0x0005
        LIX R9, 0x0003
        CMP R8, R9          ; 5 - 3 no borrow, so C = 1
        BGEU bgeu_t_ok
        JALX fail_and_next
        J bgeu_t_end
bgeu_t_ok:
        JALX pass_and_next
bgeu_t_end:
```

Remember: BLTU Tests C = 0

Because the SAMPO uses ARM-style inverted carry, **BLTU** (unsigned less than) tests $C = 0$, not $C = 1$. This is the opposite of what you might expect if you are coming from the Z80 or x86. See Section 5.5 for a detailed explanation.

7.1.6 The BMI/BPL Pair

BMI (Branch if Minus) and **BPL** (Branch if Plus) test the N flag directly. They are often used after arithmetic operations to check the sign of a result:

Listing 7.4: BMI and BPL tests from `examples/tests/test_branch.s`

```
        ; BMI: result is negative
        LIX R8, 0x0001
        LIX R9, 0x0005
        SUB R10, R8, R9      ; 1 - 5 = -4, N = 1
        BMI bmi_t_ok
        JALX fail_and_next
        J bmi_t_end
bmi_t_ok:
        JALX pass_and_next
bmi_t_end:

        ; BPL: result is positive
        LIX R8, 0x0005
        LIX R9, 0x0001
        SUB R10, R8, R9      ; 5 - 1 = 4, N = 0
        BPL bpl_t_ok
        JALX fail_and_next
        J bpl_t_end
bpl_t_ok:
        JALX pass_and_next
bpl_t_end:
```

7.1.7 Compound Conditions: BGT, BLE, BHI, BLS

The last four branches test compound flag conditions:

- **BGT** (signed greater than): $Z = 0 \wedge N = V$. The result is both non-zero *and* has the correct sign for "greater than."
- **BLE** (signed less or equal): $Z = 1 \vee N \neq V$. Either the values are equal, or the first is less.
- **BHI** (unsigned higher): $C = 1 \wedge Z = 0$. No borrow (first $\geq$ second) and not equal.
- **BLS** (unsigned lower or same): $C = 0 \vee Z = 1$. Either borrow occurred or the values are equal.

Listing 7.5: BGT and BLE from `examples/tests/test_branch.s`

```
        ; BGT: 5 > 3 (signed)
```

```
        LIX R8, 0x0005
        LIX R9, 0x0003
        CMP R8, R9
        BGT bgt_t_ok
        JALX fail_and_next
        J bgt_t_end
bgt_t_ok:
        JALX pass_and_next
bgt_t_end:

        ; BGT not taken when equal
        LIX R8, 0x0005
        CMP R8, R8          ; Z = 1, so Z=0 fails
        BGT bgt_nt_fail
        JALX pass_and_next
        J bgt_nt_end
bgt_nt_fail:
        JALX fail_and_next
bgt_nt_end:
```

7.2 Branch Encoding

Understanding the branch offset encoding is crucial for debugging and for understanding the assembler's code generation (covered in Chapter 10).

7.2.1 The Offset Calculation

When a branch is taken, the new PC is computed as:

$$\text{PC}_{\text{new}} = \text{PC} + 2 + \text{sign_extend}(\text{offset8}) \times 2$$

- The PC already points to the branch instruction itself.
- $+2$ advances past the branch instruction (which is 2 bytes long).
- The offset is multiplied by 2 because all instructions are word-aligned (2-byte boundaries).
- The offset is an 8-bit *signed* value, giving a range of -128 to $+127$ words, or -256 to $+254$ bytes relative to the instruction following the branch.

Branch Range

The maximum branch range is:

- **Forward**: up to +254 bytes (127 words) past the next instruction.
- **Backward**: up to −256 bytes (128 words) before the next instruction.

For targets beyond this range, use an unconditional jump (**J**, **JX**) combined with a reversed condition branch:

```
        BNE skip            ; Reverse: branch over the jump
        JX  far_target      ; Jump to distant target
skip:
```

7.2.2 The Assembler Computes Offsets Automatically

You never need to calculate branch offsets by hand. The assembler resolves label references during code generation and computes the correct signed offset:

```
        BEQ my_label        ; Assembler computes offset to my_label
```

If the target is out of range, the assembler reports an error.

7.3 Jump Instructions: J, JR, JALR

While branches are *conditional* and *short-range*, jumps are *unconditional* and can reach farther.

7.3.1 J — PC-Relative Jump

```
J offset12              ; PC = PC + 2 + sign_extend(offset12) * 2
```

Opcode `0x9`, Format J:

Bits 15:12	Bits 11:0
1001	offset12

The 12-bit signed offset gives a range of −4096 to +4094 bytes from the next instruction—much farther than a branch. **J** is the standard unconditional jump, used for loops, if/else fallthrough avoidance, and forward skips.

Listing 7.6: J forward test from `examples/tests/test_jump.s`

```
        ; J forward: jump over some instructions
```

```
        LIX R8, 0x0001
        J j_fwd_target       ; Jump forward
        ; These should be skipped
        MOV R8, R0
        MOV R8, R0
j_fwd_target:
        MOV R4, R8           ; R8 should still be 0x0001
        LIX R5, 0x0001
        JALX check_eq
```

7.3.2 JR — Jump to Register

```
JR Rs                   ; PC = Rs
```

JR sets the PC to the value in a register, providing an *absolute* jump to any address in the 64 KB space. Its primary use is returning from subroutines:

```
        JR RA               ; Return to caller (RA = R1)
```

But it can also be used for computed jumps—jumping to an address that is determined at runtime, such as entries in a function pointer table:

Listing 7.7: JR test from `examples/tests/test_jump.s`

```
        ; JR: jump to register value
        LIX R8, 0x0001
        LIX R9, jr_target    ; Load target address into R9
        JR R9                ; Jump to it
        ; Should be skipped
        MOV R8, R0
jr_target:
        MOV R4, R8
        LIX R5, 0x0001
        JALX check_eq
```

The encoding uses opcode `0x9` with a special bit pattern:

Bits 15:12	Bits 11:8	Bits 7:4	Bits 3:0
1001	1111	Rs	0000

7.3.3 JALR — Jump and Link Register

```
JALR Rd, Rs             ; Rd = PC + 2; PC = Rs
```

JALR saves the return address (the address of the next instruction) in Rd and then jumps to the address in Rs. This is the register-indirect subroutine call:

Listing 7.8: JALR test from `examples/tests/test_jump.s`

```
        ; JALR: jump and link via register
        LIX R9, jalr_sub     ; Load subroutine address
        JALR R3, R9           ; Call: R3 = return addr, PC = R9
        ; Returns here; R8 should be 0x42
        MOV R4, R8
        LIX R5, 0x0042
        JALX check_eq
        J jalr_done
jalr_sub:
        LIX R8, 0x0042
        JR R3                 ; Return via R3
jalr_done:
```

In this example, **JALR** uses R3 (the GP register) as the link register instead of the usual RA. This is legal—any register can hold the return address—but the calling convention recommends using RA (R1) for consistency.

7.4 Extended Jumps: JX and JALX

The base **J** instruction has a 12-bit offset, which can reach ±4 KB. For programs larger than 4 KB, or when you need to call a function at a known absolute address, the SAMPO provides 32-bit extended jumps.

7.4.1 JX — Jump Extended (Absolute)

```
JX addr16               ; PC = addr16
```

JX is a 32-bit instruction that jumps to an absolute 16-bit address. Encoded with opcode 0xF, sub-opcode 0x8:

Word 0			
1111	0000	0000	1000

Word 1: addr16

Listing 7.9: JX test from `examples/tests/test_jump.s`

```
        ; JX: absolute jump
        LIX R8, 0x0001
        JX jx_target          ; Jump to absolute address
        ; Should be skipped
        MOV R8, R0
jx_target:
        MOV R4, R8
        LIX R5, 0x0001
        JALX check_eq
```

7.4.2 JALX — Jump and Link Extended

```
JALX addr16             ; RA = PC + 4; PC = addr16
```

JALX saves the return address in RA (R1) and jumps to the absolute address. Since **JALX** is 4 bytes long, the saved return address is PC + 4 (pointing to the instruction after the **JALX**).

Encoded with opcode `0xF`, sub-opcode `0x9`:

Word 0			
1111	0000	0000	1001

Word 1: addr16

JALX is the standard function call instruction for the SAMPO. It appears throughout every test program:

Listing 7.10: JALX and return pattern from `examples/tests/test_jump.s`

```
        ; JALX: jump and link, verify RA is set correctly
        JALX jalx_sub        ; Call subroutine
        ; Returns here
        MOV R4, R8
        LIX R5, 0x0099
        JALX check_eq
        J jalx_done
jalx_sub:
        LIX R8, 0x0099       ; Set a return value
        JR RA                ; Return via RA
jalx_done:
```

The test suite also verifies that **JALX** correctly sets the return address:

Listing 7.11: Verifying JALX saves the return address

```
        ; JALX saves correct return address
        JALX jalx_ra_sub      ; Call subroutine
        J jalx_ra_check       ; (returns here)
jalx_ra_sub:
        MOV R8, RA            ; Save RA for inspection
        JR RA                 ; Return
jalx_ra_check:
        ; R8 should be non-zero (the return address)
        CMP R8, R0
        BEQ jalx_ra_bad
        JALX pass_print
        J jalx_ra_end
jalx_ra_bad:
        JALX fail_print
jalx_ra_end:
```

J vs. JX vs. JALX

- Use **J** for loops and local forward/backward jumps (no link, 12-bit range).
- Use **JX** for long unconditional jumps when no return is needed (absolute, 16-bit address).
- Use **JALX** for function calls (absolute, sets RA).
- Use **JR RA** to return from any function.

7.5 Subroutines and the Calling Convention

With branches and jumps understood, we can now formalize the subroutine calling convention introduced in Section 6.7.1.

7.5.1 Calling a Subroutine

The caller:

1. Places arguments in R4–R7 (A0–A3).
2. Saves any caller-saved registers it still needs after the call (optional).
3. Executes **JALX** *target* (or **JALR** for indirect calls).
4. On return, reads the result from R4 (and R5 for 32-bit results).

Listing 7.12: Calling convention example: caller side

```
        ; Call add_values(10, 20)
```

```
        LIX  R4, 10          ; First argument in A0
        LIX  R5, 20          ; Second argument in A1
        JALX add_values       ; Call
        ; Result is now in R4
        MOV  R8, R4          ; Save result
```

7.5.2 Inside the Subroutine

The callee:

1. Pushes RA if it will make further calls (non-leaf function).
2. Pushes any callee-saved registers (S0–S3) it will use.
3. Allocates stack space for local variables if needed.
4. Does its work.
5. Deallocates local variables.
6. Pops callee-saved registers in reverse order.
7. Pops RA (if saved).
8. Returns with **JR RA**.

Listing 7.13: Calling convention example: callee side

```
add_values:
        ; Leaf function: no need to push RA
        ADD  R4, R4, R5      ; R4 = arg0 + arg1
        JR   RA              ; Return with result in R4
```

This is a leaf function—it calls no other function—so it does not need to save RA. It also uses no callee-saved registers, so the prologue and epilogue are trivially empty.

7.6 Function Prologue and Epilogue

Non-leaf functions and functions that need callee-saved registers follow a standard prologue/epilogue pattern.

7.6.1 General Template

Listing 7.14: General function prologue and epilogue

```
my_function:
        ; ---- Prologue ----
        PUSH RA             ; Save return address
        PUSH S0             ; Save callee-saved regs we use
        PUSH S1
        ADDI SP, -N         ; Allocate N bytes for locals

        ; ---- Function body ----
        ; Use S0, S1 freely
        ; Access locals with LW Rd, offset(SP)
        ; Call other functions with JALX

        ; ---- Epilogue ----
        ADDI SP, N          ; Deallocate locals
        POP  S1             ; Restore callee-saved regs
        POP  S0             ;   (reverse order!)
        POP  RA             ; Restore return address
        JR   RA             ; Return
```

7.6.2 Stack Frame Layout

With the prologue above, the stack frame looks like this (SP grows downward):

Offset from SP	Contents	Access
$N+4$	Saved RA	`LW Rd, (N+4)(SP)`
$N+2$	Saved S0	`LW Rd, (N+2)(SP)`
$N+0$	Saved S1	`LW Rd, N(SP)`
2	Local 1	`LW Rd, 2(SP)`
0	Local 0	`LW Rd, (SP)`

The callee accesses local variables at low offsets from SP, and saved registers at higher offsets. The caller's stack frame lies above the saved RA.

7.7 Worked Example: Recursive Factorial

The classic test of a subroutine mechanism is recursive factorial: $n! = n \times (n-1)!$, with $0! = 1$. This exercises every aspect of the calling convention: argument passing, RA preservation, stack frame management, and return values.

Listing 7.15: Recursive factorial on the SAMPO

```
; factorial(n): R4 = n, returns n! in R4
```

```
; Assumes n >= 0 and result fits in 16 bits (n <= 7)

        .org 0x0100

        LIX  SP, 0xFFF0       ; Initialize stack pointer

        ; Compute 5! = 120
        LIX  R4, 5            ; Argument: n = 5
        JALX factorial
        ; R4 now holds 120 (0x0078)
        HALT

factorial:
        ; ---- Prologue ----
        PUSH RA               ; Save return address
        PUSH S0               ; Save S0 (we'll use it for n)

        MOV  S0, R4           ; S0 = n (callee-saved copy)

        ; ---- Base case: n == 0? ----
        CMP  R4, R0
        BNE  fact_recurse

        ; n == 0: return 1
        LIX  R4, 1
        J    fact_return

fact_recurse:
        ; ---- Recursive case: n * factorial(n-1) ----
        MOV  R4, S0           ; R4 = n
        ADDI R4, -1           ; R4 = n - 1
        JALX factorial         ; R4 = factorial(n - 1)

        ; R4 = (n-1)!, S0 still holds n
        MUL  R4, S0           ; R4 = n * factorial(n-1)

fact_return:
        ; ---- Epilogue ----
        POP  S0               ; Restore S0
        POP  RA               ; Restore return address
        JR   RA               ; Return with result in R4
```

7.7.1 Tracing the Execution

Let's trace the call for $n = 3$:

1. **Call**: `factorial(3)`. Push RA, push S0. S0 = 3. $3 \neq 0$, so recurse with $n = 2$.
2. **Call**: `factorial(2)`. Push RA, push S0. S0 = 2. $2 \neq 0$, so recurse with $n = 1$.
3. **Call**: `factorial(1)`. Push RA, push S0. S0 = 1. $1 \neq 0$, so recurse with $n = 0$.
4. **Call**: `factorial(0)`. Push RA, push S0. S0 = 0. $0 = 0$: base case! R4 = 1. Pop S0, pop RA, return.
5. **Return to** `factorial(1)`. R4 = 1 (returned value). S0 = 1 (restored from stack). MUL R4, S0: R4 = $1 \times 1 = 1$. Pop S0, pop RA, return.
6. **Return to** `factorial(2)`. R4 = 1. S0 = 2. MUL R4, S0: R4 = $2 \times 1 = 2$. Pop, return.
7. **Return to** `factorial(3)`. R4 = 2. S0 = 3. MUL R4, S0: R4 = $3 \times 2 = 6$. Pop, return.

The final result is 6 ($3! = 6$).

At the deepest recursion level (the call for $n = 0$), the stack holds four saved RA/S0 pairs—8 words, or 16 bytes. With our stack starting at `0xFFF0`, the stack pointer reaches `0xFFE0` at maximum depth. For $n = 7$ (the largest factorial that fits in 16 bits: $7! = 5040$), the stack goes 8 levels deep, using 32 bytes.

Factorial Overflow

On a 16-bit CPU, $8! = 40320$ fits in 16 bits (maximum unsigned value is 65535), but $9! = 362880$ does not. The **`MUL`** instruction returns only the low 16 bits of the product, so computing 9! with this code would silently produce the wrong answer. For larger factorials, you would need a multi-word multiplication routine.

7.7.2 Why Recursion Works

Recursion works on the SAMPO because each function call gets its own copy of:

- **The return address** (saved on the stack via **`PUSH RA`**).
- **The local state** (saved in callee-saved register S0, which is pushed/popped per call).

The stack provides the implicit "memory" that keeps each call level's data separate. Without PUSH/POP and the stack, recursion would be impossible because each recursive call would overwrite the caller's RA and register values.

7.7.3 Iterative Alternative

For comparison, here is the iterative version:

Listing 7.16: Iterative factorial

```
; factorial_iter(n): R4 = n, returns n! in R4
factorial_iter:
        MOV  R5, R4             ; R5 = counter = n
        LIX  R4, 1              ; R4 = accumulator = 1
fact_loop:
        CMP  R5, R0             ; counter == 0?
        BEQ  fact_done
        MUL  R4, R5             ; accumulator *= counter
        ADDI R5, -1             ; counter--
        J    fact_loop
fact_done:
        JR   RA
```

The iterative version uses no stack at all (it is a leaf function that doesn't need to save RA), runs faster, and cannot overflow the stack regardless of n. In real SAMPO programs, prefer iteration when it is straightforward. Use recursion when the problem structure demands it (tree traversal, divide-and-conquer algorithms).

7.8 Exercises

1. **Branch condition quiz.** For each of the following situations, identify which branch(es) would be taken:

   - After `CMP R4, R5` where R4 = `0x0003` and R5 = `0xFFFE` (which is -2 signed or 65534 unsigned).
   - After `SUB R4, R4, R5` where both are `0x8000`.
   - After `ADDI R4, 1` where R4 was `0x7FFF`.

2. **Countdown loop.** Write a program that counts down from 10 to 0, printing each number as an ASCII digit (for 10, print "10"; for single digits, print one character). Use **`CMP`**, **`BGT`**, and **`ADDI`** for the loop control.

3. **Linear search.** Write a subroutine that searches an array of words for a given value. Arguments: `R4` = array base address, `R5` = array length, `R6` = search value. Return the index in `R4` (or -1 if not found). Use **`BEQ`** after **`CMP`** to detect a match.

4. **Fibonacci.** Implement the Fibonacci sequence both recursively and iteratively. Compare the number of instructions executed for $n = 10$ using the emulator's trace mode. Which is faster? Why?

5. **Jump table.** Implement a 4-way dispatch using a jump table: given a value 0–3 in R4, compute the table address (using shifts and adds), load the target address from the table, and jump to it with **JR**. Store the four target addresses in memory using `.dw`.

What's Next. We have now covered the core of SAMPO programming: arithmetic, memory access, and control flow. In Chapter 8, we will survey the remaining instructions: shifts and rotates, multiply and divide, block memory operations, I/O instructions, and the system control instructions that give programs control over the CPU itself.

Chapter 8

The Full Instruction Set

The previous chapters covered the SAMPO's core: arithmetic, logic, memory access, and control flow. But a practical CPU needs more. In this chapter, we survey the remaining instructions—shifts and rotates, multiply and divide, block operations for bulk memory moves, I/O instructions for communicating with peripherals, and system instructions for controlling the CPU itself. By the end, you will know every instruction the SAMPO can execute.

8.1 Shift and Rotate Instructions

Shift and rotate operations are essential for bit manipulation, fast multiplication and division by powers of two, and field extraction. The SAMPO provides a comprehensive set of shift operations with fixed shift amounts of 1, 4, and 8 bits.

8.1.1 Shift-by-One Instructions

All shift instructions use opcode `0xA` in Format R, with the func field selecting the operation and shift amount:

Bits 15:12	Bits 11:8	Bits 7:4	Bits 3:0
`1010`	Rd	Rs1	func

The basic shift-by-one operations:

Let's see each in action, from `examples/tests/test_shift.s`:

Listing 8.1: Shift tests from `examples/tests/test_shift.s`

```
        ; SLL by 1: 0x1234 << 1 = 0x2468
        LIX R8, 0x1234
```

func	Mnemonic	Operation	Carry
0x0	**SLL** Rd, Rs1	Shift left by 1	C = old bit 15
0x1	**SRL** Rd, Rs1	Shift right by 1 (logical)	C = old bit 0
0x2	**SRA** Rd, Rs1	Shift right by 1 (arithmetic)	C = old bit 0
0x3	**ROL** Rd, Rs1	Rotate left by 1	C = old bit 15
0x4	**ROR** Rd, Rs1	Rotate right by 1	C = old bit 0
0x5	**RCL** Rd, Rs1	Rotate left through carry	17-bit rotate
0x6	**RCR** Rd, Rs1	Rotate right through carry	17-bit rotate
0x7	**SWAP** Rd, Rs1	Swap high and low bytes	—

Table 8.1: Shift and rotate instructions (shift amount = 1)

```
        SLL R4, R8
        LIX R5, 0x2468
        JALX check_eq

        ; SRL by 1: 0x1234 >> 1 = 0x091A
        LIX R8, 0x1234
        SRL R4, R8
        LIX R5, 0x091A
        JALX check_eq

        ; SRA positive: 0x1234 >>> 1 = 0x091A (bit 15 was 0)
        LIX R8, 0x1234
        SRA R4, R8
        LIX R5, 0x091A
        JALX check_eq

        ; SRA negative: 0x8234 >>> 1 = 0xC11A (bit 15 preserved)
        LIX R8, 0x8234
        SRA R4, R8
        LIX R5, 0xC11A
        JALX check_eq
```

SRL vs. SRA

SRL (Shift Right Logical) fills the vacated bit 15 with zero. **SRA** (Shift Right Arithmetic) fills it with a copy of the original bit 15—the sign bit. This means **SRA** preserves the sign of negative numbers, making it equivalent to signed division by 2 (rounding toward negative infinity). For unsigned values, use **SRL**.

8.1.2 Rotate Operations

Rotates wrap bits around instead of discarding them:

Listing 8.2: Rotate tests from `examples/tests/test_shift.s`

```
; ROL by 1: 0x8001 ROL = 0x0003
; Bit 15 wraps to bit 0: 1000_0000_0000_0001 -> 0000_0000_0000_0011
LIX R8, 0x8001
ROL R4, R8
LIX R5, 0x0003
JALX check_eq

; ROR by 1: 0x8001 ROR = 0xC000
; Bit 0 wraps to bit 15: 1000_0000_0000_0001 -> 1100_0000_0000_0000
LIX R8, 0x8001
ROR R4, R8
LIX R5, 0xC000
JALX check_eq
```

RCL and **RCR** rotate through the carry flag, treating the carry as an extra bit in a 17-bit ring. This is useful for multi-word shifts, where the bit shifted out of one word must be shifted into the next.

8.1.3 SWAP — Byte Swap

Listing 8.3: SWAP test from `examples/tests/test_shift.s`

```
; SWAP bytes: 0x1234 -> 0x3412
LIX R8, 0x1234
SWAP R4, R8
LIX R5, 0x3412
JALX check_eq
```

SWAP exchanges the high and low bytes of a 16-bit register. This is equivalent to **ROL8** (rotate left by 8) and is useful for byte-order manipulation and for accessing individual bytes of a 16-bit value.

8.1.4 Shift Carry Behavior

Shift instructions set the carry flag to the bit shifted out:

Listing 8.4: Shift carry flag tests from `examples/tests/test_shift.s`

```
; SLL carry out: SLL 0x8000, carry should be 1
LIX R8, 0x8000
SLL R10, R8
```

```
        GETF R4
        LIX R8, 0x0020       ; C flag = bit 5
        AND R4, R4, R8
        LIX R5, 0x0020
        JALX check_eq

        ; SRL carry out: SRL 0x0001, carry should be 1
        LIX R8, 0x0001
        SRL R10, R8
        GETF R4
        LIX R8, 0x0020
        AND R4, R4, R8
        LIX R5, 0x0020
        JALX check_eq
```

For **SLL**, the carry receives the old bit 15 (the bit shifted out of the most significant position). For **SRL** and **SRA**, the carry receives the old bit 0 (the bit shifted out of the least significant position).

8.2 Extended Shifts

Shifting by a single bit at a time would require four instructions to shift by 4. The SAMPO provides multi-bit shift variants that shift by 4 or 8 bits in a single instruction.

8.2.1 Shift-by-4 and Shift-by-8

These use the same opcode `0xA` with higher func values:

func	Mnemonic	Operation
`0x8`	SLL4 Rd, Rs1	Rd = Rs1 $\ll$ 4
`0x9`	SRL4 Rd, Rs1	Rd = Rs1 $\gg$ 4 (logical)
`0xA`	SRA4 Rd, Rs1	Rd = Rs1 $\ggg$ 4 (arithmetic)
`0xB`	ROL4 Rd, Rs1	rotate left 4
`0xC`	SLL8 Rd, Rs1	Rd = Rs1 $\ll$ 8
`0xD`	SRL8 Rd, Rs1	Rd = Rs1 $\gg$ 8 (logical)
`0xE`	SRA8 Rd, Rs1	Rd = Rs1 $\ggg$ 8 (arithmetic)
`0xF`	ROL8 Rd, Rs1	rotate left 8 (same as SWAP)

Table 8.2: Multi-bit shift and rotate instructions

Assembler Support for Multi-Bit Shifts

At the time of writing, the assembler (`sasm`) does not have dedicated mnemonics for all multi-bit shift variants. The test suite encodes them manually using `.dw`. For example:

```
; SLL4 R4, R8: opcode=0xA, Rd=4, Rs=8, func=0x8
.dw 0xA488
```

This is a temporary limitation of the assembler, not the CPU. See Chapter 10 for details on adding new mnemonics.

Listing 8.5: Multi-bit shift tests from `examples/tests/test_shift.s`

```
; SLL4: 0x1234 << 4 = 0x2340
LIX R8, 0x1234
.dw 0xA488          ; SLL4 R4, R8
LIX R5, 0x2340
JALX check_eq

; SRL4: 0x1234 >> 4 = 0x0123
LIX R8, 0x1234
.dw 0xA489          ; SRL4 R4, R8
LIX R5, 0x0123
JALX check_eq

; SRA4: 0x8234 >>> 4 = 0xF823 (sign preserved)
LIX R8, 0x8234
.dw 0xA48A          ; SRA4 R4, R8
LIX R5, 0xF823
JALX check_eq

; SLL8: 0x1234 << 8 = 0x3400
LIX R8, 0x1234
.dw 0xA48C          ; SLL8 R4, R8
LIX R5, 0x3400
JALX check_eq

; SRL8: 0x1234 >> 8 = 0x0012
LIX R8, 0x1234
.dw 0xA48D          ; SRL8 R4, R8
LIX R5, 0x0012
JALX check_eq

; SRA8: 0x8234 >>> 8 = 0xFF82 (sign fill)
LIX R8, 0x8234
```

```
        .dw 0xA48E          ; SRA8 R4, R8
        LIX R5, 0xFF82
        JALX check_eq
```

8.2.2 Extended Shifts with Arbitrary Amounts: SLLX, SRLX, SRAX

For shift amounts other than 1, 4, or 8, the extended instruction format provides **SLLX**, **SRLX**, and **SRAX**. These are 32-bit instructions where the shift amount is encoded in the lower bits of the second word:

Listing 8.6: Extended shift tests from `examples/tests/test_extended.s`

```
        ; SLLX R4, R8, 4: R4 = R8 << 4
        ; Encoding: 0xF48D 0x0004
        LIX R8, 0x1234
        .dw 0xF48D, 0x0004
        LIX R5, 0x2340
        JALX check_eq

        ; SRLX R4, R8, 4: R4 = R8 >> 4
        ; Encoding: 0xF48E 0x0004
        LIX R8, 0x1234
        .dw 0xF48E, 0x0004
        LIX R5, 0x0123
        JALX check_eq

        ; SRAX R4, R8, 4: R4 = R8 >>> 4 (arithmetic)
        ; Encoding: 0xF48F 0x0004
        LIX R8, 0x8234
        .dw 0xF48F, 0x0004
        LIX R5, 0xF823
        JALX check_eq
```

These are encoded with opcode `0xF` (extended), sub-opcodes `0xD` (SLLX), `0xE` (SRLX), and `0xF` (SRAX). The shift amount comes from the low 4 bits of the 16-bit immediate in the second word, allowing shifts from 0 to 15 bits.

8.3 Multiply and Divide

The SAMPO includes hardware multiply and divide instructions. Unlike the three-operand ALU instructions, multiply and divide are *two-operand*: the destination register is also the first source operand.

8.3.1 MUL and MULH

```
MUL  Rd, Rs1            ; Rd = (Rd * Rs1)[15:0]  (low 16 bits)
MULH Rd, Rs1            ; Rd = (Rd * Rs1)[31:16] (high 16 bits, signed)
```

MUL computes the full 32-bit product of two 16-bit values and returns the low 16 bits. **MULH** returns the high 16 bits of the signed product. Together, they give you the complete 32-bit result.

MUL Is Two-Operand

MUL Rd, Rs1 computes Rd = (Rd × Rs1)[15:0]. The destination register is also one of the source operands. This means you must set up Rd with the first multiplicand *before* executing MUL. There is no three-operand `MUL Rd, Rs1, Rs2` form.

Encoded with opcode `0xB`:

Bits 15:12	Bits 11:8	Bits 7:4	Bits 3:0
1011	Rd	Rs1	func

Listing 8.7: MUL tests from `examples/tests/test_muldiv.s`

```
        ; MUL basic: 3 * 4 = 12
        LIX R8, 0x0004
        LIX R4, 0x0003          ; R4 = first operand
        MUL R4, R8              ; R4 = 3 * 4 = 12
        LIX R5, 0x000C
        JALX check_eq

        ; MUL by zero: 5 * 0 = 0
        LIX R4, 0x0005
        MUL R4, R0              ; R4 = 5 * 0 = 0
        MOV R5, R0
        JALX check_eq

        ; MUL large: 100 * 200 = 20000 (0x4E20)
        LIX R8, 0x00C8
        LIX R4, 0x0064
        MUL R4, R8
        LIX R5, 0x4E20
        JALX check_eq

        ; MUL overflow: 256 * 256 = 65536, low word = 0x0000
        LIX R8, 0x0100
        LIX R4, 0x0100
```

```
        MUL R4, R8
        MOV R5, R0              ; Low 16 bits are 0
        JALX check_eq
```

The overflow case is instructive: $256 \times 256 = 65536 = \texttt{0x10000}$. The low 16 bits are `0x0000`, so **`MUL`** returns zero. To get the full result, use **`MULH`**:

Listing 8.8: MULH for the high word

```
        ; MULH: 0x100 * 0x100 = 0x10000, high word = 0x0001
        LIX R8, 0x0100
        LIX R4, 0x0100
        MULH R4, R8             ; R4 = high 16 bits = 0x0001
        LIX R5, 0x0001
        JALX check_eq
```

8.3.2 DIV and DIVU

```
DIV  Rd, Rs1            ; Rd = Rd / Rs1 (signed)
DIVU Rd, Rs1            ; Rd = Rd / Rs1 (unsigned)
```

Like **`MUL`**, these are two-operand: the dividend is in `Rd`, the divisor in `Rs1`, and the quotient replaces the dividend in `Rd`.

Listing 8.9: DIV tests from `examples/tests/test_muldiv.s`

```
        ; DIV basic: 12 / 4 = 3
        LIX R8, 0x0004
        LIX R4, 0x000C
        DIV R4, R8
        LIX R5, 0x0003
        JALX check_eq

        ; DIV by zero: 5 / 0 = 0xFFFF (hardware defined)
        LIX R4, 0x0005
        DIV R4, R0
        LIX R5, 0xFFFF
        JALX check_eq

        ; DIVU basic: 0x7FF0 / 16 = 0x07FF
        LIX R8, 0x0010
        LIX R4, 0x7FF0
        DIVU R4, R8
```

```
        LIX R5, 0x07FF
        JALX check_eq
```

Division by Zero

Division by zero does not cause a hardware exception on the SAMPO. Instead, **DIV** and **DIVU** return `0xFFFF` when the divisor is zero. This is a defined behavior, not undefined—but it is still a bug in your program if you divide by zero unintentionally. Always check the divisor before dividing.

8.3.3 REM and REMU

```
REM  Rd, Rs1            ; Rd = Rd % Rs1 (signed remainder)
REMU Rd, Rs1            ; Rd = Rd % Rs1 (unsigned remainder)
```

Listing 8.10: REM tests from `examples/tests/test_muldiv.s`

```
        ; REM basic: 13 % 4 = 1
        LIX R8, 0x0004
        LIX R4, 0x000D
        REM R4, R8
        LIX R5, 0x0001
        JALX check_eq

        ; REM zero division: 5 % 0 = 5 (returns dividend)
        LIX R4, 0x0005
        REM R4, R0
        LIX R5, 0x0005
        JALX check_eq

        ; REMU basic: 17 % 5 = 2
        LIX R8, 0x0005
        LIX R4, 0x0011
        REMU R4, R8
        LIX R5, 0x0002
        JALX check_eq
```

When dividing by zero, **REM** returns the original dividend—a reasonable fallback that avoids crashing the CPU.

8.3.4 DAA — Decimal Adjust

```
DAA Rd                  ; Decimal-adjust Rd after BCD addition
```

DAA adjusts the result of a binary addition to produce a valid BCD (Binary-Coded Decimal) result. It uses the H (half-carry) flag to determine whether the low nibble needs correction. BCD arithmetic is rarely used in modern code, but it is handy for applications that display decimal numbers directly.

8.4 Block Operations

The SAMPO borrows a page from the Z80's playbook with its block transfer and search instructions. These use three registers as implicit operands:

Register	Role in Block Operations
R4 (A0)	Source address pointer
R5 (A1)	Destination address pointer
R6 (A2)	Byte count / search value

Table 8.3: Implicit registers for block operations

8.4.1 LDI and LDD — Block Transfer (Single Step)

```
LDI                     ; mem[R5] = mem[R4]; R4++; R5++; R6--
LDD                     ; mem[R5] = mem[R4]; R4--; R5--; R6--
```

LDI (Load and Increment) copies one byte from the source to the destination, increments both pointers, and decrements the count. **LDD** (Load and Decrement) does the same but decrements the pointers—useful for backward copies.

8.4.2 LDIR and LDDR — Block Transfer (Repeat)

```
LDIR                    ; Repeat LDI until R6 = 0
LDDR                    ; Repeat LDD until R6 = 0
```

LDIR and **LDDR** are the repeating versions. They execute the single-step operation repeatedly until the byte count in R6 reaches zero. A single **LDIR** instruction can copy an entire buffer:

Listing 8.11: Memory copy using LDIR

```
; Copy 100 bytes from 0x8000 to 0x9000
        LIX R4, 0x8000      ; Source
```

```
        LIX R5, 0x9000       ; Destination
        LIX R6, 100          ; Count
        LDIR                  ; Copy all bytes
```

This replaces what would otherwise be a 6-instruction loop.

LDIR Is Interruptible

Unlike a software loop, the hardware implementation of **LDIR** can be interrupted mid-transfer. When an interrupt fires, the CPU saves the state (including the current pointer and count values) and resumes the block operation after the interrupt handler returns. This ensures that long transfers do not lock out interrupts.

8.4.3 CPIR — Compare and Search

```
CPIR                  ; Search for R6 value starting at R4
```

CPIR (Compare, Increment, Repeat) searches memory starting at the address in R4 for the byte value in R6. After execution, R4 points to the byte after the match (or past the end if not found), and the Z flag indicates whether a match was found.

8.4.4 FILL — Fill Memory

```
FILL                  ; Fill R6 bytes at R5 with value from R4
```

FILL writes the low byte of R4 to R6 consecutive memory locations starting at the address in R5. This is the fastest way to clear a buffer or initialize memory to a known value:

Listing 8.12: Zeroing a buffer with FILL

```
; Clear 256 bytes at 0x8000
        MOV  R4, R0          ; Fill value = 0
        LIX  R5, 0x8000      ; Start address
        LIX  R6, 256         ; Count
        FILL
```

8.5 I/O Instructions

The SAMPO uses a separate I/O address space with 256 ports (0–255). Peripheral devices—serial ports, timers, GPIO—are accessed through dedicated I/O instructions rather than memory-mapped registers.

8.5.1 Immediate Port I/O: INI and OUTI

```
INI  Rd, port          ; Rd = port_read(port)
OUTI port, Rs          ; port_write(port, Rs)
```

These use a 4-bit port address encoded directly in the instruction, giving access to ports 0–15. For the ACIA serial port (ports `0x80`–`0x81`), the extended forms must be used.

Encoding uses opcode `0xD`:

Bits 15:12	Bits 11:8	Bits 7:4	Bits 3:0
1101	Rd/Rs	port	func

8.5.2 Register Port I/O: IN and OUT

```
IN  Rd, (Rs1)          ; Rd = port_read(Rs1[7:0])
OUT (Rs1), Rd           ; port_write(Rs1[7:0], Rd)
```

When the port address is known only at runtime (stored in a register), use the register-indirect forms.

8.5.3 Extended I/O: INX and OUTX

```
INX  Rd, port8         ; Rd = port_read(port8)
OUTX port8, Rs         ; port_write(port8, Rs)
```

For ports above 15 (including the ACIA at ports `0x80`–`0x81`), 32-bit extended instructions provide an 8-bit port address in the second word. In practice, most SAMPO programs use the **`INI`** and **`OUTI`** mnemonics with the port constant defined via `.equ`, and the assembler selects the correct encoding automatically:

Listing 8.13: I/O with ACIA ports

```
.equ ACIA_STATUS 0x80
.equ ACIA_DATA   0x81

        INI  R6, ACIA_STATUS  ; Read status register
```

```
        OUTI ACIA_DATA, R5    ; Write data to transmit
```

8.6 System Instructions

System instructions control the CPU's operating mode and flag state. They are encoded with opcode `0xE`:

Bits 15:12	Bits 11:8	Bits 7:0
1110	func	operand

func	Mnemonic	Description
0x0	**NOP**	No operation
0x1	**HALT**	Halt the processor
0x2	**DI**	Disable interrupts
0x3	**EI**	Enable interrupts
0x4	**RETI**	Return from interrupt
0x5	**SWI**	Software interrupt
0x6	**SCF**	Set carry flag
0x7	**CCF**	Complement carry flag

Table 8.4: System instructions

8.6.1 NOP

```
NOP                     ; Do nothing for one cycle
```

NOP does not modify any register or flag. It advances the PC by 2 and consumes one instruction cycle. Uses include timing delays, alignment padding, and reserving space for patches.

Listing 8.14: NOP test from `examples/tests/test_misc.s`

```
        ; NOP: verify execution continues unchanged
        LIX R8, 0x1234
        NOP
        NOP
        NOP
        MOV R4, R8
        LIX R5, 0x1234
        JALX check_eq           ; R8 should be unchanged
```

8.6.2 HALT

```
HALT                    ; Stop executing
```

HALT transitions the CPU state machine from FETCH to HALTED. No further instructions execute until an interrupt arrives. In the emulator, HALT terminates the execution loop.

8.6.3 DI and EI

```
DI                      ; Disable maskable interrupts
EI                      ; Enable maskable interrupts
```

These instructions modify the I (interrupt enable) flag in the FLAGS register. **DI** clears it, preventing the CPU from responding to interrupt requests. **EI** sets it. Use **DI** during critical sections where interrupt handling would cause problems (such as when updating multi-word data structures), and **EI** afterward.

8.6.4 RETI

```
RETI                    ; Return from interrupt
```

RETI pops the saved PC and FLAGS from the stack and re-enables interrupts. It is used at the end of interrupt service routines. Unlike a normal **JR RA**, **RETI** restores the full CPU state (including flags) to what it was before the interrupt occurred.

8.6.5 SWI — Software Interrupt

```
SWI imm8                ; Software interrupt with vector imm8
```

SWI triggers a software-initiated interrupt. The 8-bit immediate selects the interrupt vector. This is used for system calls—a user program can request services from a monitor or operating system without knowing the service routine's address.

8.6.6 SCF and CCF

```
SCF                     ; Set carry flag (C = 1)
CCF                     ; Complement carry flag (C = ~C)
```

These provide direct control over the carry flag. **SCF** sets it; **CCF** toggles it.

Listing 8.15: SCF and CCF tests from `examples/tests/test_misc.s`

```
        ; SCF: set carry flag
        MOV R8, R0
        SETF R8              ; Clear all flags
        SCF                  ; Set carry
        GETF R4
        LIX R8, 0x0020       ; C flag mask
        AND R4, R4, R8
        LIX R5, 0x0020
        JALX check_eq         ; C should be set

        ; CCF: complement carry (toggle from 1 to 0)
        SCF                  ; C = 1
        CCF                  ; C = 0
        GETF R4
        LIX R8, 0x0020
        AND R4, R4, R8
        MOV R5, R0
        JALX check_eq         ; C should be clear

        ; CCF: toggle from 0 to 1
        MOV R8, R0
        SETF R8              ; C = 0
        CCF                  ; C = 1
        GETF R4
        LIX R8, 0x0020
        AND R4, R4, R8
        LIX R5, 0x0020
        JALX check_eq
```

8.6.7 EXX — Exchange Register Sets

```
EXX                     ; Swap R4-R11 with R4'-R11'
```

EXX swaps the primary registers R4–R11 with an alternate (shadow) register set R4'–R11'. After **EXX**, the values that were in R4–R11 are now in the shadow registers, and the former shadow values are now in R4–R11.

EXX Swaps Only R4–R11

EXX does *not* swap R0–R3 or R12–R15. The zero register, RA, SP, and GP are always available. The callee-saved registers (S0–S3) are also unaffected, so the calling convention remains consistent across the swap.

Listing 8.16: EXX test from `examples/tests/test_misc.s`

```
        ; EXX: swap R4-R11, verify R8 changes and restores
        LIX R8, 0xAAAA          ; Set R8 to a known value
        EXX                     ; Swap to alternate set
        LIX R8, 0xBBBB          ; Set R8' (alternate) to 0xBBBB
        EXX                     ; Swap back
        MOV R4, R8              ; R8 should be 0xAAAA again
        LIX R5, 0xAAAA
        JALX check_eq

        ; Verify alternate set was preserved
        EXX                     ; Swap to alternates
        MOV R13, R8             ; R13 outside swap range; R8' = 0xBBBB
        EXX                     ; Swap back
        MOV R4, R13
        LIX R5, 0xBBBB
        JALX check_eq

        ; EXX preserves R0-R3 and R12-R15
        LIX R13, 0xDEAD
        EXX
        EXX                     ; Double swap = identity
        MOV R4, R13             ; R13 should be unchanged
        LIX R5, 0xDEAD
        JALX check_eq
```

EXX is inspired by the Z80's instruction of the same name. It is useful for interrupt handlers (save the working registers in one **EXX**, use the alternates freely, restore with another **EXX**) and for algorithms that need to switch between two sets of working variables quickly.

8.7 The MC6850 ACIA

The SAMPO's primary I/O peripheral is a Motorola MC6850-compatible ACIA (Asynchronous Communications Interface Adapter). This classic UART chip provides serial communication and is emulated in both the software emulator and the FPGA implementation.

8.7.1 ACIA Register Map

The ACIA presents two I/O ports:

Port	Read	Write
`0x80`	Status register	Control register
`0x81`	Receive data	Transmit data

Table 8.5: MC6850 ACIA port assignments

8.7.2 Status Register Bits

The status register (`0x80`, read-only) contains the following bits:

Bit	Name	Description
0	RDRF	Receive Data Register Full
1	TDRE	Transmit Data Register Empty
2	DCD	Data Carrier Detect
3	CTS	Clear to Send
4	FE	Framing Error
5	OVRN	Overrun
6	PE	Parity Error
7	IRQ	Interrupt Request

Table 8.6: MC6850 status register bits

The two most important bits for basic serial communication:

- **Bit 0 (RDRF)**: Set when a received byte is available for reading.
- **Bit 1 (TDRE)**: Set when the transmitter is ready to accept a new byte for sending.

8.8 Busy-Wait UART Output

The standard pattern for sending a character over the serial port uses a busy-wait loop to check the transmit-ready status:

Listing 8.17: Standard busy-wait serial output pattern

```
.equ ACIA_STATUS 0x80
.equ ACIA_DATA   0x81
```

```
; send_char: send byte in R4 to serial port
send_char:
sc_wait:
        INI  R5, ACIA_STATUS  ; Read status register
        LIX  R6, 0x0002       ; TDRE mask (bit 1)
        AND  R5, R5, R6       ; Isolate TDRE bit
        BEQ  sc_wait           ; Loop until TDRE = 1
        OUTI ACIA_DATA, R4    ; Send the character
        JR   RA               ; Return
```

This version correctly uses **AND** with a mask to isolate the TDRE bit, rather than the fragile **ADDI** trick used in the Hello program.

The complete `print_str` routine used throughout the test suite wraps this pattern into a string output function:

Listing 8.18: Complete print_str with busy-wait UART

```
print_str:
        PUSH R12
        MOV R12, R4           ; R12 = string pointer
ps_loop:
        LBU R5, (R12)         ; Load next character
        CMP R5, R0            ; Null terminator?
        BEQ ps_done
ps_wait:
        INI R6, ACIA_STATUS   ; Read status
        AND R7, R6, R6        ; Copy for flag test
        ADDI R7, -2           ; Check TDRE
        BNE ps_wait
        OUTI ACIA_DATA, R5    ; Send character
        ADDI R12, 1           ; Next character
        J ps_loop
ps_done:
        POP R12
        JR RA
```

From Busy-Wait to Interrupt-Driven

Busy-wait I/O wastes CPU cycles spinning in a loop. In Chapter 8, we showed the basic pattern; in real applications, you would use interrupt-driven I/O instead. The ACIA's IRQ bit (status bit 7) and the SAMPO's **EI**/**DI**/**RETI** instructions provide the foundation for interrupt-driven serial communication. We will cover interrupts in more detail in **??**.

8.9 Exercises

1. **Multiply by shift.** Write a subroutine that multiplies a value in `R4` by 10 without using the **`MUL`** instruction. Hint: $10x = 8x + 2x$, and both $8x$ and $2x$ can be computed with shifts and an add. Compare the cycle count with the **`MUL`** instruction.

2. **Block memory compare.** Write a subroutine that compares two blocks of memory byte-by-byte. Arguments: `R4` = block 1 address, `R5` = block 2 address, `R6` = length. Return 0 in `R4` if the blocks are equal, or the offset of the first differing byte if not.

3. **Hex digit output.** Write a subroutine that converts the low nibble (bits 3:0) of `R4` to an ASCII hex character ('0'–'9' or 'A'–'F') and sends it to the serial port. Then extend it to print a full 16-bit register as four hex digits. Use **`SRL4`** and **`AND`** for nibble extraction.

4. **EXX context switch.** Write a program that demonstrates using **`EXX`** as a lightweight context switch. Set up two different "tasks" with different register values. Use **`EXX`** to switch between them, and verify that each task's registers are preserved across the switch.

5. **Echo program.** Write a program that reads characters from the serial port (check RDRF, then read from `0x81`) and echoes each character back to the transmitter. Add a special case: if the received character is a lowercase letter, convert it to uppercase before echoing. Use the TUI debugger to test it—characters you type should appear on the terminal.

What's Next. You now know every instruction the SAMPO can execute. Part II is complete—you can write, assemble, and run any program the hardware supports. In Part III, we will turn the tables and look *inside* the assembler itself: how `sasm` lexes, parses, and generates the machine code that brings your programs to life.

Part III

Inside the Assembler

Chapter 9

Lexing and Parsing

Every assembler begins with text and ends with bytes. Between those two endpoints lies a pipeline of transformations—each one narrowing the gap between the programmer's intent and the machine's appetite for binary. In this chapter we open the hood on `sasm`, the SAMPO assembler, and trace how raw source text becomes a structured representation that the code generator can consume.

We will start with the overall architecture of `sasm`, walk through the lexer token by token, then step into the parser to see how it builds an abstract syntax tree. By the end, you will be able to predict exactly what data structures exist in memory after assembling any SAMPO program—and understand the design decisions that shaped them.

9.1 Overview of `sasm`

9.1.1 A Zero-Dependency Rust Assembler

The SAMPO assembler lives in the `sasm/` directory of the project. Here is its `Cargo.toml`:

Listing 9.1: `sasm/Cargo.toml`: the assembler's manifest

```
[package]
name = "sasm"
version = "0.1.0"
edition = "2021"
description = "Sampo CPU Assembler"

[dependencies]
```

Notice the empty `[dependencies]` section. The assembler has *no external dependencies*—not even a command-line parser library. Every line of code uses the Rust standard library alone. This is a deliberate choice: the assembler is a teaching tool, and every piece of it should be readable without chasing down crate documentation.

Build It Yourself

Build the assembler from the project root:

```
cd sasm && cargo build --release
```

The resulting binary lands in `sasm/target/release/sasm`.

9.1.2 The Three-Stage Pipeline

The entry point in `sasm/src/main.rs` lays out the assembler's pipeline in three clear stages:

Listing 9.2: `sasm/src/main.rs`: the three-stage pipeline

```
// Lexical analysis
let mut lexer = Lexer::new(&source);
let tokens = match lexer.tokenize() {
    Ok(t) => t,
    Err(e) => {
        eprintln!("Lexer error: {}", e);
        std::process::exit(1);
    }
};

// Parsing
let mut parser = Parser::new(tokens);
let program = match parser.parse() {
    Ok(p) => p,
    Err(e) => {
        eprintln!("Parser error: {}", e);
        std::process::exit(1);
    }
};

// Code generation
let mut codegen = CodeGen::new();
let binary = match codegen.generate(&program) {
    Ok(b) => b,
    Err(e) => {
        eprintln!("Code generation error: {}", e);
```

```
        std::process::exit(1);
    }
};
```

Each stage takes the output of the previous one and transforms it into a richer representation:

1. **Lexer**: Raw source text → a flat vector of `Token` values.
2. **Parser**: Token stream → a `Program` containing `Statement` nodes.
3. **Code generator**: Abstract syntax → binary machine code (covered in Chapter 10).

Pipeline Architecture

A *pipeline architecture* processes input through sequential stages, where each stage transforms the data into a form closer to the final output. Pipelines make compilers and assemblers easier to reason about because each stage has a single, well-defined responsibility.

The pipeline terminates with a file write:

Listing 9.3: Writing the binary output

```
match fs::write(&output_file, &binary) {
    Ok(_) => {
        println!("Assembled {} -> {} ({} bytes)",
                 input_file, output_file, binary.len());
    }
    Err(e) => {
        eprintln!("Error writing {}: {}", output_file, e);
        std::process::exit(1);
    }
}
```

The assembler reports the input file, output file, and byte count—enough to confirm success at a glance.

9.1.3 Command-Line Interface

The CLI is intentionally minimal:

Listing 9.4: Running the assembler

```
# Assemble with automatic output name
cargo run --release -- ../examples/hello.s
```

```
# Specify output file explicitly
cargo run --release -- ../examples/hello.s -o hello.bin
```

If no `-o` flag is given, the assembler replaces the `.s` or `.asm` extension with `.bin`:

Listing 9.5: Output file name derivation

```
let output_file = if args.len() >= 4 && args[2] == "-o" {
    args[3].clone()
} else {
    input_file.replace(".s", ".bin").replace(".asm", ".bin")
};
```

9.1.4 Module Structure

The codebase is split into three modules, each in its own file:

Listing 9.6: Module declarations in `main.rs`

```
mod lexer;
mod parser;
mod codegen;

use lexer::Lexer;
use parser::Parser;
use codegen::CodeGen;
```

File	Module	Responsibility
`lexer.rs`	`lexer`	Tokenization
`parser.rs`	`parser`	Syntax analysis
`codegen.rs`	`codegen`	Code generation and fixup

Table 9.1: The `sasm` module layout

This chapter covers the first two modules. Code generation gets its own chapter (Chapter 10) because the encoding logic is intricate enough to deserve focused attention.

9.2 The Lexer

The lexer's job is to turn a stream of characters into a stream of *tokens*—classified chunks of text that the parser can consume without worrying about whitespace, comments, or character-level details.

9.2.1 Token Types

Every distinct syntactic element in SAMPO assembly maps to a variant of the `Token` enum in `sasm/src/lexer.rs`:

Listing 9.7: The `Token` enum

```rust
#[derive(Debug, Clone, PartialEq)]
pub enum Token {
    // Registers
    Register(u8),
    // Numbers
    Number(i32),
    // Identifiers (labels, symbols)
    Ident(String),
    // String literals
    StringLit(String),
    // Punctuation
    Comma,
    Colon,
    LParen,
    RParen,
    Plus,
    Minus,
    // Directives
    Directive(String),
    // End of line
    Newline,
    // End of file
    Eof,
}
```

Let us walk through each variant and what it represents:

`Register(u8)`
: A register name that has already been resolved to its 4-bit encoding (0–15). Both numeric names like `R4` and aliases like `A0` produce the same `Register(4)` token.

`Number(i32)`
: A numeric literal. The lexer stores it as a signed 32-bit integer to accommodate negative values and to defer range checking to the code generator.

`Ident(String)`
: Any identifier that is *not* a register name—instruction mnemonics, label references, and symbol names all land here. The distinction between an instruction and a label reference is deferred to the parser.

`StringLit(String)`
: A double-quoted string with escape sequences already processed.

`Comma`, `Colon`, `LParen`, `RParen`, `Plus`, `Minus`
: Single-character punctuation tokens. These drive the parser's operand recognition.

`Directive(String)`
: An assembler directive (the dot-prefixed keywords like `.org`, `.equ`, `.db`). The directive name is stored lowercase, so both `.ORG` and `.org` parse identically.

`Newline`
: Assembly is a line-oriented language. Newlines are significant because they terminate instructions—unlike C, there are no semicolons.

`Eof` Signals the end of the source file. This token is always the last one in the stream.

Why `i32` for Numbers?

The SAMPO is a 16-bit machine, so you might expect `i16` or `u16`. But the lexer uses `i32` because intermediate values during assembly—addresses, offsets, arithmetic in `.equ` definitions—can temporarily exceed 16 bits. Range checking happens later, in the code generator, where the context determines whether a value must fit in 8, 12, or 16 bits.

9.2.2 The Lexer Structure

The lexer maintains three pieces of state:

Listing 9.8: Lexer state

```
pub struct Lexer<'a> {
    input: Peekable<Chars<'a>>,
    line: usize,
    col: usize,
}
```

The `input` field is a `Peekable<Chars>` iterator over the source string. The `Peekable` wrapper lets the lexer look ahead one character without consuming it—essential for distinguishing between, say, a minus sign and a negative number.

The `line` and `col` fields track the current position for error messages. They start at `line: 1, col: 1`.

Listing 9.9: Lexer constructor

```
pub fn new(input: &'a str) -> Self {
    Lexer {
```

```
        input: input.chars().peekable(),
        line: 1,
        col: 1,
    }
}
```

9.2.3 The Main Tokenization Loop

The public entry point collects all tokens into a vector:

Listing 9.10: The `tokenize` method

```
pub fn tokenize(&mut self) -> Result<Vec<Token>, String> {
    let mut tokens = Vec::new();

    loop {
        match self.next_token()? {
            Token::Eof => {
                tokens.push(Token::Eof);
                break;
            }
            token => tokens.push(token),
        }
    }

    Ok(tokens)
}
```

This design—collecting all tokens into a `Vec` before parsing—is deliberate. It makes the parser's lookahead and position tracking straightforward: the parser can index freely into a flat array rather than pulling tokens one at a time from the lexer. For a 64 KB address space worth of assembly, the token vector fits comfortably in memory.

9.2.4 Whitespace and Comment Handling

Each call to `next_token` begins by stripping whitespace and comments:

Listing 9.11: Pre-token cleanup

```
fn next_token(&mut self) -> Result<Token, String> {
    self.skip_whitespace();
    self.skip_comment();
    self.skip_whitespace();
    // ...
```

```
}
```

The double whitespace skip matters: a comment line like `;; debug note` has whitespace before the comment, and there may be whitespace between the comment and the newline.

Whitespace skipping is conservative—it only consumes spaces and tabs, not newlines:

Listing 9.12: Whitespace skipping

```
fn skip_whitespace(&mut self) {
    while let Some(c) = self.peek() {
        if c == ' ' || c == '\t' {
            self.advance();
        } else {
            break;
        }
    }
}
```

Comments run from a semicolon to the end of the line:

Listing 9.13: Comment skipping

```
fn skip_comment(&mut self) {
    if self.peek() == Some(';') {
        while let Some(c) = self.peek() {
            if c == '\n' || c == '\r' {
                break;
            }
            self.advance();
        }
    }
}
```

The comment consumer stops at (but does not consume) the newline character, leaving it for the main dispatcher to emit as a `Token::Newline`.

9.2.5 The Token Dispatcher

After whitespace and comments are cleared, the lexer examines the next character and dispatches to the appropriate handler. The logic follows a `match` on the peeked character:

Listing 9.14: Token dispatch (abbreviated)

```
match self.peek() {
    None => Ok(Token::Eof),
    Some('\n') => { /* emit Newline, update line counter */ }
    Some('\r') => { /* handle CR+LF, emit Newline */ }
    Some(',') => { self.advance(); Ok(Token::Comma) }
    Some(':') => { self.advance(); Ok(Token::Colon) }
    Some('(') => { self.advance(); Ok(Token::LParen) }
    Some(')') => { self.advance(); Ok(Token::RParen) }
    Some('+') => { self.advance(); Ok(Token::Plus) }
    Some('-') => { /* negative number or Minus token */ }
    Some('.') => { /* directive */ }
    Some('"') => { /* string literal */ }
    Some('\'') => { /* character literal */ }
    Some(c) if c.is_ascii_digit() => { /* number */ }
    Some(c) if c.is_alphabetic() || c == '_' => { /* ident or register */ }
    Some(c) => Err(format!("Unexpected character '{}' ...", c)),
}
```

The single-character tokens (`Comma`, `Colon`, etc.) are consumed in one step. More complex tokens—numbers, identifiers, strings—delegate to dedicated reader functions.

9.2.6 Number Formats

The SAMPO assembler supports three number bases:

Base	Prefix	Example	Value
Decimal	(none)	`42`	42
Hexadecimal	`0x` or `0X`	`0xFF`	255
Binary	`0b` or `0B`	`0b1010`	10

Table 9.2: Supported number formats

The `read_number` method detects the base by inspecting the first two characters:

Listing 9.15: Number parsing with base detection

```
fn read_number(&mut self) -> Result<i32, String> {
    let mut num_str = String::new();
    let mut base = 10;

    // Check for hex prefix
    if self.peek() == Some('0') {
        num_str.push(self.advance().unwrap());
```

```
        if let Some(c) = self.peek() {
            if c == 'x' || c == 'X' {
                self.advance();
                base = 16;
                num_str.clear();
            } else if c == 'b' || c == 'B' {
                self.advance();
                base = 2;
                num_str.clear();
            }
        }
    }

    while let Some(c) = self.peek() {
        if c.is_ascii_hexdigit() || c == '_' {
            if c != '_' {
                num_str.push(c);
            }
            self.advance();
        } else {
            break;
        }
    }

    i32::from_str_radix(&num_str, base)
        .map_err(|e| format!("Invalid number at line {}: {}", self.line, e))
}
```

Several design details are worth noting:

- **Underscore separators**: The lexer silently skips underscores within numbers, so you can write `0b1111_0000` or `0xFF_FF` for readability.

- **Hex-digit greediness**: The digit loop accepts `is_ascii_hexdigit()`, which matches `0`–`9` and `a`–`f`/`A`–`F`. For decimal and binary, this means the lexer will consume characters like `A` even in a decimal context—but the `from_str_radix` call will reject them with an error.

- **Negative numbers**: The minus sign is handled in the main dispatcher, not in `read_number`. When the lexer sees a `-` followed by a digit, it reads the number and negates it.

Underscore Separators

When writing large immediates in assembly, use underscores for clarity:

```
    LIX R4, 0xFF_00       ; Much easier to read than 0xFF00
    LIX R5, 0b1010_0101   ; Binary is especially hard without separators
```

9.2.7 The Minus Sign Ambiguity

One of the trickiest parts of the lexer is the minus sign. The character `-` can mean two things:

1. The start of a negative number: `ADDI R4, -10`
2. A stand-alone minus token: `LW R4, (R5 - 2)`

The lexer resolves this with a one-character lookahead:

Listing 9.16: Minus sign disambiguation

```
Some('-') => {
    self.advance();
    // Check if it's a negative number
    if let Some(c) = self.peek() {
        if c.is_ascii_digit() {
            let num = self.read_number()?;
            return Ok(Token::Number(-num));
        }
    }
    Ok(Token::Minus)
}
```

If the character after the minus is a digit, the lexer reads a full number and negates it. Otherwise, it emits a standalone `Token::Minus`. This means the expression `(R5 - 2)` produces the token sequence `[LParen, Register(5), Minus, Number(2), RParen]`, while `ADDI R4, -10` produces `[Ident("ADDI"), Register(4), Comma, Number(-10)]`.

9.2.8 Register Names and Aliases

When the lexer encounters an alphabetic identifier, it checks whether it matches a register name before emitting an `Ident` token:

Listing 9.17: Register recognition in the lexer

```
Some(c) if c.is_alphabetic() || c == '_' => {
```

```
    let ident = self.read_identifier();
    // Check if it's a register
    if let Some(reg) = parse_register(&ident) {
        Ok(Token::Register(reg))
    } else {
        Ok(Token::Ident(ident))
    }
}
```

The `parse_register` function maps both numeric and ABI alias names to register numbers:

Listing 9.18: The `parse_register` function

```
fn parse_register(name: &str) -> Option<u8> {
    let upper = name.to_uppercase();
    match upper.as_str() {
        "R0" | "ZERO" => Some(0),
        "R1" | "RA" => Some(1),
        "R2" | "SP" => Some(2),
        "R3" | "GP" => Some(3),
        "R4" | "A0" => Some(4),
        "R5" | "A1" => Some(5),
        "R6" | "A2" => Some(6),
        "R7" | "A3" => Some(7),
        "R8" | "T0" => Some(8),
        "R9" | "T1" => Some(9),
        "R10" | "T2" => Some(10),
        "R11" | "T3" => Some(11),
        "R12" | "S0" => Some(12),
        "R13" | "S1" => Some(13),
        "R14" | "S2" => Some(14),
        "R15" | "S3" => Some(15),
        _ => None,
    }
}
```

The comparison is case-insensitive (via `to_uppercase()`), so `sp`, `Sp`, and `SP` all resolve to register 2. This happens entirely at lex time: by the time the parser sees the token stream, every register reference is already a bare `Register(u8)` value.

9.2.9 String Literals and Escape Sequences

String literals are enclosed in double quotes. The lexer processes escape sequences at lex time, converting them to their byte values:

Code	Register	Alias	Convention
0x0	R0	ZERO	Hardwired zero
0x1	R1	RA	Return address
0x2	R2	SP	Stack pointer
0x3	R3	GP	Global pointer / frame pointer
0x4–0x7	R4–R7	A0–A3	Arguments / return values
0x8–0xB	R8–R11	T0–T3	Temporaries
0xC–0xF	R12–R15	S0–S3	Saved registers

Table 9.3: Register encoding and aliases

Listing 9.19: String literal parsing with escape handling

```
fn read_string(&mut self) -> Result<String, String> {
    let mut s = String::new();
    loop {
        match self.advance() {
            None => return Err(format!(
                "Unterminated string at line {}", self.line)),
            Some('"') => break,
            Some('\\') => {
                match self.advance() {
                    Some('n') => s.push('\n'),
                    Some('r') => s.push('\r'),
                    Some('t') => s.push('\t'),
                    Some('0') => s.push('\0'),
                    Some('\\') => s.push('\\'),
                    Some('"') => s.push('"'),
                    Some(c) => s.push(c),
                    None => return Err(format!(
                        "Unterminated escape at line {}", self.line)),
                }
            }
            Some(c) => s.push(c),
        }
    }
    Ok(s)
}
```

The supported escape sequences are:

Escape	Byte	Description
`\\n`	`0x0A`	Newline (line feed)
`\\r`	`0x0D`	Carriage return
`\\t`	`0x09`	Horizontal tab
`\\0`	`0x00`	Null byte
`\\\\`	`0x5C`	Literal backslash
`\\"`	`0x22`	Literal double quote

Table 9.4: Supported string escape sequences

Any other character after a backslash is passed through unchanged. This is forgiving by design—an unrecognized escape like `\\q` silently becomes the character `q`.

9.2.10 Character Literals

Single-quoted characters are converted to their ASCII value at lex time:

Listing 9.20: Character literal handling

```
Some('\'') => {
    self.advance();
    let c = self.advance()
        .ok_or("Unexpected end of character literal")?;
    if self.advance() != Some('\'') {
        return Err(format!(
            "Expected closing quote at line {}", self.line));
    }
    Ok(Token::Number(c as i32))
}
```

A character literal like `'A'` produces `Token::Number(65)`—the ASCII code for 'A'. This lets you write readable byte data:

Listing 9.21: Using character literals

```
    .db 'H', 'i', '!', 0    ; Same as .db 0x48, 0x69, 0x21, 0x00
```

9.2.11 Directives

When the lexer sees a dot, it reads the following identifier and produces a `Directive` token:

Listing 9.22: Directive tokenization

```
Some('.') => {
    self.advance();
    let name = self.read_identifier();
    Ok(Token::Directive(name.to_lowercase()))
}
```

The directive name is lowercased immediately, so the parser never needs to worry about case. The assembler supports six directives: `.org`, `.equ`, `.db`, `.dw`, `.ascii`, and `.asciz`. We will see how the parser and code generator handle each one in the following sections.

9.3 The Parser

The parser transforms the flat token vector into a structured `Program`—a list of `Statement` nodes that the code generator can iterate over. Unlike parsing a general-purpose programming language, parsing assembly is relatively straightforward: each line is either a label, an instruction, or a directive.

9.3.1 Parser State

The parser holds only two fields:

Listing 9.23: Parser state

```
pub struct Parser {
    tokens: Vec<Token>,
    pos: usize,
}
```

The `pos` cursor starts at zero and advances through the token vector. All parsing decisions are made by looking at the token at `pos` (via `peek()`) and optionally the tokens that follow it.

Listing 9.24: Core navigation methods

```
fn peek(&self) -> &Token {
    self.tokens.get(self.pos).unwrap_or(&Token::Eof)
}

fn advance(&mut self) -> &Token {
    if !self.is_at_end() {
        self.pos += 1;
    }
    self.tokens.get(self.pos - 1).unwrap_or(&Token::Eof)
}
```

```
fn is_at_end(&self) -> bool {
    matches!(self.peek(), Token::Eof)
}
```

The `peek` method returns an `Eof` token if the position is past the end, ensuring the parser never panics on out-of-bounds access.

9.3.2 Two Utility Methods: `check` and `expect`

The `check` method tests whether the current token has the same variant as a given token, ignoring the inner value:

Listing 9.25: Token variant comparison

```
fn check(&self, token: &Token) -> bool {
    std::mem::discriminant(self.peek())
        == std::mem::discriminant(token)
}
```

Using `std::mem::discriminant` compares enum variants without caring about the data they carry. This lets the parser ask "Is the next token *some* register?" without specifying *which* register.

The `expect` method advances past the expected token or produces an error:

Listing 9.26: The `expect` method

```
fn expect(&mut self, expected: &Token) -> Result<(), String> {
    if self.check(expected) {
        self.advance();
        Ok(())
    } else {
        Err(format!("Expected {:?}, got {:?}",
            expected, self.peek()))
    }
}
```

9.3.3 The Parse Loop

The top-level parse method loops over the token stream, skipping newlines and delegating to `parse_statement`:

Listing 9.27: Top-level parse loop

```
pub fn parse(&mut self) -> Result<Program, String> {
    let mut statements = Vec::new();

    while !self.is_at_end() {
        self.skip_newlines();
        if self.is_at_end() {
            break;
        }

        if let Some(stmt) = self.parse_statement()? {
            statements.push(stmt);
        }
    }

    Ok(Program { statements })
}
```

The `parse_statement` method returns `Option<Statement>`: it returns `None` for blank lines and `Some(...)` for actual content.

9.3.4 Statement Parsing

The parser determines what kind of statement it is looking at by examining the first non-newline token:

Listing 9.28: Statement dispatch

```
fn parse_statement(&mut self) -> Result<Option<Statement>, String> {
    match self.peek() {
        Token::Eof => Ok(None),
        Token::Newline => {
            self.advance();
            Ok(None)
        }
        Token::Directive(name) => {
            let name = name.clone();
            self.advance();
            let args = self.parse_directive_args()?;
            Ok(Some(Statement::Directive { name, args }))
        }
        Token::Ident(name) => {
            let name = name.clone();
            self.advance();
```

```
            // Check if it's a label
            if self.check(&Token::Colon) {
                self.advance();
                Ok(Some(Statement::Label(name)))
            } else {
                // It's an instruction
                let operands = self.parse_operands()?;
                Ok(Some(Statement::Instruction {
                    mnemonic: name.to_uppercase(),
                    operands,
                }))
            }
        }
        _ => Err(format!("Unexpected token: {:?}", self.peek())),
    }
}
```

The key insight is the **label versus instruction ambiguity**. Both labels and instructions begin with an `Ident` token. The parser resolves this by checking whether the next token is a colon:

- `start:` → `Ident("start")` followed by `Colon` → it is a label.
- `ADD R4, R5, R6` → `Ident("ADD")` followed by `Register(4)` → it is an instruction.

Notice that the mnemonic is uppercased at parse time (`name.to_uppercase()`), so the code generator can use a simple string match regardless of how the programmer capitalized the instruction.

Labels Must Use a Colon

In `sasm`, the colon after a label name is mandatory. Writing `start` on its own line (without a colon) would be interpreted as an instruction mnemonic, and the assembler would report "Unknown instruction: start" during code generation.

9.3.5 Parsing Directive Arguments

Directives can have a variable number of arguments: numbers, strings, or identifiers, separated by commas. The parser collects them into a `Vec<DirectiveArg>`:

Listing 9.29: Directive argument parsing

```
fn parse_directive_args(&mut self) -> Result<Vec<DirectiveArg>, String> {
    let mut args = Vec::new();

    loop {
        match self.peek() {
```

```
            Token::Newline | Token::Eof => break,
            Token::Number(n) => {
                let n = *n;
                self.advance();
                args.push(DirectiveArg::Number(n));
            }
            Token::StringLit(s) => {
                let s = s.clone();
                self.advance();
                args.push(DirectiveArg::String(s));
            }
            Token::Ident(s) => {
                let s = s.clone();
                self.advance();
                args.push(DirectiveArg::Ident(s));
            }
            Token::Comma => {
                self.advance();
            }
            _ => break,
        }
    }

    Ok(args)
}
```

The loop consumes arguments until it hits a newline, EOF, or any unrecognized token. Commas are consumed and discarded—they serve as visual separators for the programmer but carry no semantic weight.

9.3.6 Parsing Operands

Instruction operands are where the real complexity lives. The parser must handle registers, immediates, labels, and—most tricky—indirect addressing modes.

Listing 9.30: Operand parsing (core loop)

```
fn parse_operands(&mut self) -> Result<Vec<Operand>, String> {
    let mut operands = Vec::new();

    loop {
        match self.peek() {
            Token::Newline | Token::Eof => break,
            Token::Comma => {
```

```
                self.advance();
                continue;
            }
            Token::Register(r) => {
                let r = *r;
                self.advance();
                operands.push(Operand::Register(r));
            }
            Token::Number(n) => {
                let n = *n;
                self.advance();
                // Check for indirect addressing: imm(Rs)
                if self.check(&Token::LParen) {
                    self.advance();
                    if let Token::Register(r) = self.peek() {
                        let r = *r;
                        self.advance();
                        self.expect(&Token::RParen)?;
                        operands.push(Operand::Indirect(r, n));
                    } else {
                        return Err(
                            "Expected register in indirect addressing"
                                .to_string());
                    }
                } else {
                    operands.push(Operand::Immediate(n));
                }
            }
            Token::Ident(name) => {
                let name = name.clone();
                self.advance();
                operands.push(Operand::Label(name));
            }
            // ... (LParen and Minus cases follow)
            _ => break,
        }
    }

    Ok(operands)
}
```

Let us trace through the operand forms the parser handles:

Registers

A bare register token is pushed directly:

```
    ADD R4, R5, R6    ; Three Operand::Register values
```

Immediates and Indirect Addressing

When the parser sees a number, it must decide: is this a plain immediate, or the offset part of an indirect address? The answer depends on whether a left parenthesis follows:

```
    ADDI R4, 10        ; Number alone -> Operand::Immediate(10)
    LW   R4, 2(R5)     ; Number + '(' + register + ')' -> Operand::Indirect(5, 2)
```

Zero-Offset Indirect Addressing

When the first token of an operand is a left parenthesis (not preceded by a number), the offset defaults to zero:

Listing 9.31: Parsing zero-offset indirect addressing

```
Token::LParen => {
    self.advance();
    if let Token::Register(r) = self.peek() {
        let r = *r;
        self.advance();

        let offset = if self.check(&Token::Plus) {
            self.advance();
            if let Token::Number(n) = self.peek() {
                let n = *n;
                self.advance();
                n
            } else { 0 }
        } else if self.check(&Token::Minus) {
            self.advance();
            if let Token::Number(n) = self.peek() {
                let n = *n;
                self.advance();
                -n
            } else { 0 }
        } else { 0 };
```

```
        self.expect(&Token::RParen)?;
        operands.push(Operand::Indirect(r, offset));
    } else {
        return Err(
            "Expected register in indirect addressing".to_string());
    }
}
```

This handles several syntactic forms:

Listing 9.32: Indirect addressing variants

```
    LW R4, (R5)         ; Operand::Indirect(5, 0)
    LW R4, (R5 + 2)     ; Operand::Indirect(5, 2)
    LW R4, (R5 - 4)     ; Operand::Indirect(5, -4)
```

Label References

Any identifier that the lexer did *not* recognize as a register becomes a label operand:

```
    BEQ done            ; Operand::Label("done")
    LIX R4, message     ; Operand::Label("message")
```

The parser does not resolve labels—it records the name as a string. Resolution happens during code generation, when the symbol table is available.

9.4 Abstract Syntax: Operand Types

The parser's output is a `Program` structure containing a vector of `Statement` nodes. Let us examine the complete type hierarchy.

9.4.1 The `Operand` Enum

Listing 9.33: The `Operand` type

```
#[derive(Debug, Clone)]
pub enum Operand {
    Register(u8),
    Immediate(i32),
    Label(String),
    Indirect(u8, i32),  // Register + offset: (Rs + imm)
}
```

Each variant captures the information the code generator needs:

`Register(u8)`
: A register number (0–15), already resolved from the textual name.

`Immediate(i32)`
: A numeric constant. May be positive or negative; range checking is deferred to code generation.

`Label(String)`
: A reference to a named symbol. Could be a branch target, a data address, or a constant defined with `.equ`.

`Indirect(u8, i32)`
: A memory address formed from a base register and a constant offset. The first field is the register number; the second is the signed offset.

No Dedicated Instruction Enum

The parser does not validate instruction mnemonics. An `Instruction` statement stores the mnemonic as a raw string. The code generator's massive `match` block is where unknown mnemonics are caught. This keeps the parser lean: it only cares about syntax, not semantics.

9.4.2 The `Statement` Enum

Listing 9.34: The `Statement` type

```
#[derive(Debug, Clone)]
pub enum Statement {
    Label(String),
    Instruction {
        mnemonic: String,
        operands: Vec<Operand>,
    },
    Directive {
        name: String,
        args: Vec<DirectiveArg>,
    },
}
```

Labels carry no data beyond their name. Instructions carry a mnemonic and a vector of operands. Directives carry a name and a vector of arguments.

9.4.3 The `DirectiveArg` Enum

Listing 9.35: Directive argument types

```
#[derive(Debug, Clone)]
pub enum DirectiveArg {
    Number(i32),
    String(String),
    Ident(String),
}
```

Directive arguments are more heterogeneous than instruction operands: they can include string data (for `.ascii` and `.asciz`) and identifier references (for `.equ` symbol definitions and `.dw` label references).

9.4.4 The `Program` Structure

Listing 9.36: The top-level `Program` type

```
pub struct Program {
    pub statements: Vec<Statement>,
}
```

A `Program` is a flat list of statements. There is no nesting, no scope, no hierarchy—assembly is a linear language, and the data structure reflects that. The code generator walks this list exactly twice: once to collect labels, once to emit bytes.

9.4.5 Putting It Together

Consider this assembly fragment:

Listing 9.37: A short assembly fragment

```
        .org 0x0100
start:
        LIX  R4, message
        ADDI R4, 1
        HALT
message:
        .asciz "Hi"
```

After lexing and parsing, the `Program` structure contains:

Listing 9.38: Abstract syntax for the fragment above

```
Program {
    statements: [
```

```
        Directive { name: "org", args: [Number(256)] },
        Label("start"),
        Instruction {
            mnemonic: "LIX",
            operands: [Register(4), Label("message")]
        },
        Instruction {
            mnemonic: "ADDI",
            operands: [Register(4), Immediate(1)]
        },
        Instruction {
            mnemonic: "HALT",
            operands: []
        },
        Label("message"),
        Directive {
            name: "asciz",
            args: [String("Hi")]
        },
    ]
}
```

Notice how labels appear as separate statements in the list, not attached to the instructions they precede. The code generator uses the *position* of labels relative to other statements to compute their addresses.

9.5 Error Handling and Line Tracking

Every stage of the pipeline uses Rust's `Result<T, String>` type for error propagation. Errors are strings containing a human-readable message with enough context to locate the problem.

9.5.1 Lexer Error Reporting

The lexer includes line numbers in its error messages:

Listing 9.39: Lexer errors with line context

```
// Character literal
return Err(format!("Expected closing quote at line {}", self.line));

// Number parsing
i32::from_str_radix(&num_str, base)
    .map_err(|e| format!("Invalid number at line {}: {}", self.line, e))
```

```
// Unexpected character
Some(c) => Err(format!("Unexpected character '{}' at line {}", c, self.line)),
```

The `line` counter is updated whenever the lexer encounters a newline:

Listing 9.40: Line counter update

```
Some('\n') => {
    self.advance();
    self.line += 1;
    self.col = 1;
    Ok(Token::Newline)
}
```

9.5.2 Parser Error Reporting

The parser's errors tend to be more structural—unexpected token types, missing operands:

Listing 9.41: Parser error examples

```
Err(format!("Expected {:?}, got {:?}", expected, self.peek()))

Err("Expected register in indirect addressing".to_string())

Err(format!("Unexpected token: {:?}", self.peek()))
```

Because the parser operates on tokens (not characters), its errors reference token types rather than line numbers. The Rust `Debug` formatting of token variants provides diagnostic detail—an error might read: "Expected Register(0), got Number(42)".

Improving Error Messages

The current error messages from the parser lack line numbers because the `Token` enum does not carry source location information. A production assembler would wrap each token with a `Span` indicating its source line and column. For our purposes, the lexer's line-number errors catch most typos, and the parser's structural errors provide enough context to find the problem.

9.5.3 Error Propagation Through the Pipeline

Back in `main`, each stage is wrapped in a `match` that converts errors into diagnostic output and an exit code:

Listing 9.42: Error propagation in the pipeline

```
let tokens = match lexer.tokenize() {
    Ok(t) => t,
    Err(e) => {
        eprintln!("Lexer error: {}", e);
        std::process::exit(1);
    }
};
```

The error prefix (“Lexer error”, “Parser error”, “Code generation error”) tells the user which stage failed. Combined with the error message itself, this is usually enough to diagnose the problem.

9.5.4 A Taxonomy of Errors

Table 9.5 summarizes the error conditions across the assembler:

Stage	Error	Example Cause
Lexer	Unexpected character	`@` in source
Lexer	Invalid number	`0xGG`
Lexer	Unterminated string	Missing closing quote
Lexer	Unterminated escape	`"\` at end of file
Parser	Unexpected token	Number where register expected
Parser	Missing register	`LW (42)`
Parser	Missing closing paren	`LW R4, (R5`
Codegen	Unknown instruction	Misspelled mnemonic
Codegen	Operand mismatch	Too few or too many operands
Codegen	Range overflow	Immediate too large
Codegen	Undefined symbol	Forward ref not resolved

Table 9.5: Error taxonomy across assembler stages

9.6 Hands-On: Trace Through Assembling a Program

Let us trace the full lexer and parser pipeline on a real program from the SAMPO project. We will use the first few lines of `examples/hello.s`:

Listing 9.43: Opening lines of `examples/hello.s`

```
; Hello World for Sampo CPU
; Outputs "Hello, Sampo!" to serial port
```

```
        .org 0x0100

.equ    ACIA_STATUS 0x80
.equ    ACIA_DATA   0x81

start:
        LIX  R4, message

loop:
        LBU  R5, (R4)
        CMP  R5, R0
        BEQ  done
```

9.6.1 Step 1: Lexing

The lexer processes the source character by character. Here is the token stream for the fragment above (blank lines and comments stripped):

1. **Line 1–2**: The `;` comment lines are consumed entirely by `skip_comment`. The `\textbackslash n` at the end of each line produces `Token::Newline`.
2. **Line 4**: `.org 0x0100`
   - The `.` triggers directive reading: `Directive("org")`
   - Whitespace is skipped
   - `0x0100` is recognized as hexadecimal: `Number(256)`
   - The line ends: `Newline`
3. **Line 6**: `.equ ACIA_STATUS 0x80`
   - `Directive("equ")`
   - `ACIA_STATUS` is not a register name: `Ident("ACIA_STATUS")`
   - `0x80`: `Number(128)`
   - `Newline`
4. **Line 7**: `.equ ACIA_DATA 0x81`
   - `Directive("equ")`, `Ident("ACIA_DATA")`, `Number(129)`, `Newline`
5. **Line 9**: `start:`
   - `start` is not a register: `Ident("start")`

- `Colon`
- `Newline`

6. **Line 10**: `LIX R4, message`
   - `LIX` is not a register: `Ident("LIX")`
   - `R4` matches `parse_register`: `Register(4)`
   - `Comma`
   - `message` is not a register: `Ident("message")`
   - `Newline`
7. **Line 12**: `loop:`
   - `Ident("loop")`, `Colon`, `Newline`
8. **Line 13**: `LBU R5, (R4)`
   - `Ident("LBU")`
   - `Register(5)`
   - `Comma`
   - `LParen`
   - `Register(4)`
   - `RParen`
   - `Newline`
9. **Line 14**: `CMP R5, R0`
   - `Ident("CMP")`, `Register(5)`, `Comma`, `Register(0)`, `Newline`
10. **Line 15**: `BEQ done`
    - `Ident("BEQ")`, `Ident("done")`, `Newline`

9.6.2 Step 2: Parsing

The parser walks the token stream and produces statements. Here is the resulting `Program`:

Listing 9.44: Parsed statements for the hello.s fragment

```
[
  Directive { name: "org", args: [Number(256)] },
```

```
  Directive { name: "equ",
              args: [Ident("ACIA_STATUS"), Number(128)] },
  Directive { name: "equ",
              args: [Ident("ACIA_DATA"), Number(129)] },
  Label("start"),
  Instruction { mnemonic: "LIX",
                operands: [Register(4), Label("message")] },
  Label("loop"),
  Instruction { mnemonic: "LBU",
                operands: [Register(5), Indirect(4, 0)] },
  Instruction { mnemonic: "CMP",
                operands: [Register(5), Register(0)] },
  Instruction { mnemonic: "BEQ",
                operands: [Label("done")] },
]
```

Pay attention to a few key transformations:

- The `.equ` arguments `ACIA_STATUS` and `0x80` are stored as `DirectiveArg` values (not `Operand` values). The code generator will insert them into the symbol table during pass 1.
- The `LBU R5, (R4)` instruction has its second operand parsed as `Indirect(4, 0)`. The parser saw `LParen`, then `Register(4)`, then `RParen`—with no `Plus` or `Minus` token inside the parentheses, the offset defaults to zero.
- The `BEQ done` branch target is stored as `Label("done")`. The assembler does not yet know where `done` is. That resolution happens during code generation (see Chapter 10).
- Register aliases (`R0` for `ZERO`, `R4` for `A0`) have already been resolved to numbers by the lexer. The parser—and by extension the code generator—never see alias names.

9.6.3 What Happens Next

The `Program` is now ready for the code generator. In Chapter 10, we will see how the three-pass code generator:

1. Walks the statement list to compute label addresses (pass 1).
2. Walks again to emit binary, recording fixups for unresolved symbols (pass 2).
3. Applies those fixups to produce the final binary (pass 3).

But the lexer and parser have already done the hard work of turning text into structure.

Exercises

1. **Token counting.** Take the `examples/simple.s` file from the SAMPO project and manually trace the lexer, listing every token that would be produced. How many `Register` tokens appear? How many `Newline` tokens?

2. **Negative offset parsing.** Write a short assembly fragment that uses the syntax `LW R4, (R5 -4)` and trace it through both the lexer and parser. What tokens does the lexer produce? What `Operand` does the parser construct? How does this differ from `LW R4, -4(R5)`?

3. **Add an octal number format.** The lexer supports decimal, hexadecimal, and binary. Sketch the changes to `read_number` that would add octal support with a `0o` prefix (e.g., `0o777` = 511 decimal). What happens to the underscore separator logic?

4. **Extend the `Token` with spans.** Currently the `Token` enum does not carry source location information. Design a wrapper type `SpannedToken` that pairs a `Token` with its source line and column. What changes would be needed in the parser to thread this information through to error messages?

5. **FP alias.** The `print_help` function in `main.rs` lists `R3` as `GP`, but some calling conventions use `R3` as a frame pointer (`FP`). Add support for an `FP` alias in `parse_register` and explain how the change propagates (or doesn't) through the parser and code generator.

What's Next With the abstract syntax in hand, we are ready to tackle the hardest part of the assembler: turning statements and operands into binary machine code. Chapter 10 dives into the three-pass code generation strategy, instruction encoding, branch offset calculation, and the fixup mechanism that resolves forward references.

Chapter 10

Code Generation

The parser gave us structure; now we need bytes. The code generator is where abstract syntax becomes machine code—where `Instruction { mnemonic: "ADD", operands: [Register(4), Register(5), Register(6)] }` becomes the two bytes `0x56 0x04`, ready to be loaded into the SAMPO's memory.

This is also where the assembler's real complexity lives. Instruction encoding must respect the SAMPO's six instruction formats, each with its own field layout. Forward references to labels that haven't been defined yet require a multi-pass strategy. Branch offsets must be computed relative to the program counter. And the little-endian byte order means the "obvious" byte ordering is always backwards.

Let us trace through all of it.

10.1 Three-Pass Assembly

The code generator in `sasm/src/codegen.rs` uses a three-pass strategy to transform the parsed `Program` into a binary image:

Listing 10.1: The three-pass entry point

```
pub fn generate(&mut self, program: &Program) -> Result<Vec<u8>, String> {
    // Pass 1: Collect labels
    self.pass1(program)?;

    // Pass 2: Generate code
    self.pass2(program)?;

    // Pass 3: Apply fixups
```

```
    self.apply_fixups()?;

    Ok(self.output.clone())
}
```

10.1.1 Why Three Passes?

The fundamental problem is *forward references*. When the assembler encounters `BEQ done`, the label `done` might not have been defined yet—it could appear fifty lines later. The assembler cannot compute the branch offset without knowing where `done` is, and it cannot know where `done` is without knowing how many bytes lie between the branch and the label.

A single-pass assembler would have to backtrack. A two-pass assembler could collect labels first, then generate code—but that still leaves the problem of encoding branches and jumps to forward labels during pass 2, when the offset must be computed from a target address that was collected during pass 1.

The SAMPO assembler solves this with three passes:

Pass 1: Label Collection
: Walk the statement list, tracking the program counter. Whenever a label is encountered, record its address in the symbol table. Do *not* emit any bytes.

Pass 2: Code Generation
: Walk the statement list again, this time emitting bytes for each instruction and directive. When a label reference cannot be resolved (typically branches and jumps to forward labels), emit placeholder bytes and record a *fixup*.

Pass 3: Fixup Application
: Walk the fixup list and patch the placeholder bytes with the correct values, now that all labels are known.

10.1.2 Code Generator State

The `CodeGen` struct holds everything the three passes need:

Listing 10.2: Code generator state

```
pub struct CodeGen {
    origin: u16,
    pc: u16,
    symbols: HashMap<String, u16>,
    output: Vec<u8>,
    fixups: Vec<Fixup>,
```

```
}
```

origin
: The starting address set by `.org`. Defaults to zero.

pc
: The current program counter. Advances as instructions and data are emitted.

symbols
: The symbol table: a map from label names to 16-bit addresses.

output
: The binary output buffer. Bytes are appended here during pass 2.

fixups
: A list of deferred patches that need to be applied in pass 3.

10.1.3 Pass 1: Label Collection

Pass 1 walks the statement list and simulates the program counter without emitting any bytes:

Listing 10.3: Pass 1: label collection

```
fn pass1(&mut self, program: &Program) -> Result<(), String> {
    self.pc = self.origin;

    for stmt in &program.statements {
        match stmt {
            Statement::Label(name) => {
                self.symbols.insert(name.clone(), self.pc);
            }
            Statement::Directive { name, args } => {
                match name.as_str() {
                    "org" => {
                        if let Some(DirectiveArg::Number(addr)) = args.first() {
                            self.origin = *addr as u16;
                            self.pc = self.origin;
                        }
                    }
                    "equ" => {
                        if args.len() >= 2 {
                            if let (DirectiveArg::Ident(sym),
                                    DirectiveArg::Number(val)) =
                                (&args[0], &args[1])
                            {
                                self.symbols.insert(sym.clone(), *val as u16);
```

```
                            }
                        }
                    }
                    "db" => { self.pc += args.len() as u16; }
                    "dw" => { self.pc += (args.len() * 2) as u16; }
                    "ascii" | "asciz" => {
                        for arg in args {
                            if let DirectiveArg::String(s) = arg {
                                self.pc += s.len() as u16;
                                if name == "asciz" {
                                    self.pc += 1; // null terminator
                                }
                            }
                        }
                    }
                    _ => {}
                }
            }
            Statement::Instruction { mnemonic, operands } => {
                self.pc += self.instruction_size(mnemonic, operands)?;
            }
        }
    }

    Ok(())
}
```

Key points:

- **Labels** are inserted into the symbol table at the current `pc` value.
- `.org` sets both `origin` and `pc`. This means labels defined after an `.org` are correctly placed at the new origin.
- `.equ` inserts a constant value into the symbol table. These are not addresses—they are arbitrary 16-bit values that can be used as immediates.
- **Data directives** advance `pc` by the number of bytes they will produce. The `.asciz` directive adds one extra byte for the null terminator.
- **Instructions** advance `pc` by their encoded size—2 bytes for standard instructions, 4 bytes for extended (32-bit) instructions.

10.1.4 Instruction Size Determination

The `instruction_size` method determines whether an instruction is 16-bit (2 bytes) or 32-bit (4 bytes):

Listing 10.4: Instruction size classification

```
fn instruction_size(&self, mnemonic: &str, _operands: &[Operand])
    -> Result<u16, String>
{
    match mnemonic.to_uppercase().as_str() {
        // Extended 32-bit instructions
        "LIX" | "ADDIX" | "SUBIX" | "ANDIX" | "ORIX" | "XORIX" |
        "LWX" | "SWX" | "JX" | "JALX" | "CMPIX" | "INX" | "OUTX" |
        "SLLX" | "SRLX" | "SRAX" |
        "INI" | "OUTI" => Ok(4),
        // All others are 16-bit
        _ => Ok(2),
    }
}
```

Extended Instructions Include INI and OUTI

The **INI** and **OUTI** instructions use the extended (32-bit) format even though they appear to have small operands. This is because the 8-bit port number does not fit in the 4-bit field available in the base 16-bit I/O encoding (opcode `0xD`). The assembler encodes them as extended instructions with a 16-bit second word carrying the port number.

10.1.5 Pass 2: Code Emission

Pass 2 resets the program counter and walks the statement list again, this time emitting actual bytes:

Listing 10.5: Pass 2: code emission (structure)

```
fn pass2(&mut self, program: &Program) -> Result<(), String> {
    self.pc = self.origin;

    // Pad output to origin if needed
    while self.output.len() < self.origin as usize {
        self.output.push(0);
    }

    for stmt in &program.statements {
        match stmt {
            Statement::Label(_) => {}
            Statement::Directive { name, args } => {
```

```
                self.emit_directive(name, args)?;
            }
            Statement::Instruction { mnemonic, operands } => {
                self.emit_instruction(mnemonic, operands)?;
            }
        }
    }

    Ok(())
}
```

The initial zero-padding is important: if the program starts at `0x0100` (the standard SAMPO reset vector), the output buffer is padded with 256 zero bytes before any code is emitted. This ensures that byte offsets in the output buffer match the corresponding memory addresses.

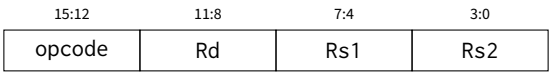
Labels are skipped in pass 2—they were already processed in pass 1.

10.2 Instruction Encoding

The heart of the code generator is the `emit_instruction` method: a large `match` block that encodes each instruction mnemonic into binary. Let us examine the encoding strategies for each instruction format.

10.2.1 R-Format: Three-Register ALU Operations

The core ALU operations (**ADD**, **SUB**, **AND**, **OR**, **XOR**) use the R-format:



The opcode occupies bits 15:12 and identifies the operation. Three 4-bit register fields follow. For example, **ADD** uses opcode `0x0`:

Listing 10.6: Encoding **ADD**: three-register R-format

```
"ADD" => {
    let (rd, rs1, rs2) = self.get_three_regs(operands)?;
    self.emit_word(0x0000 | ((rd as u16) << 8)
                          | ((rs1 as u16) << 4)
                          | (rs2 as u16));
}
```

The helper `get_three_regs` extracts three register numbers from the operand vector, returning an error if the operands are wrong:

Listing 10.7: Extracting three register operands

```
fn get_three_regs(&self, operands: &[Operand])
    -> Result<(u8, u8, u8), String>
{
    if operands.len() < 3 {
        return Err("Expected three registers".to_string());
    }
    match (&operands[0], &operands[1], &operands[2]) {
        (Operand::Register(r1), Operand::Register(r2),
         Operand::Register(r3)) => Ok((*r1, *r2, *r3)),
        _ => Err("Expected three registers".to_string()),
    }
}
```

Let us encode `ADD R4, R5, R6` by hand:

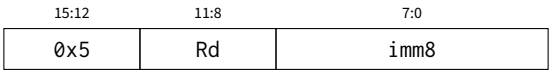
Field	Value	Bits	Binary
opcode	`0x0` (ADD)	15:12	`0000`
Rd	4 (R4)	11:8	`0100`
Rs1	5 (R5)	7:4	`0101`
Rs2	6 (R6)	3:0	`0110`

Assembled: `0000 0100 0101 0110` = `0x0456`. In little-endian memory: byte at PC is `0x56`, byte at PC+1 is `0x04`.

All five ALU operations (**ADD**, **SUB**, **AND**, **OR**, **XOR**) use this same layout, differing only in the opcode nibble (`0x0`–`0x4`).

10.2.2 I-Format: Add Immediate

The **ADDI** instruction uses the I-format with an 8-bit signed immediate:



Listing 10.8: Encoding **ADDI**

```
"ADDI" => {
    let (rd, imm) = self.get_reg_imm(operands)?;
    if imm < -128 || imm > 127 {
```

```
        return Err(format!("Immediate {} out of range for ADDI", imm));
    }
    self.emit_word(0x5000 | ((rd as u16) << 8) | ((imm as u8) as u16));
}
```

ADDI Is Two-Operand

Unlike **ADD**, which writes to a separate destination register, **ADDI** is a two-operand instruction: it adds the immediate to the destination register *in place*. That is, `ADDI R4, 10` means R4 = R4 + 10, not Rd = Rs + 10. The 8 bits freed by eliminating the second source register field are given entirely to the immediate, providing a -128 to $+127$ range.

The range check `imm < -128 | imm > 127` ensures the immediate fits in 8 signed bits. The cast chain `(imm as u8) as u16` performs the sign-extension truncation: for negative values, this preserves the two's complement bit pattern. For instance, -1 as `i32` is cast to `u8` value `0xFF`, then zero-extended to `u16` value `0x00FF`—which, ORed into bits 7:0 of the instruction word, produces the correct encoding.

10.2.3 Load and Store Format

Load and store instructions encode a base register, a destination/source register, and a "function" field that indicates the operation type and offset:

Listing 10.9: Encoding **LW** and **SW**

```
"LW" => {
    let (rd, rs, offset) = self.get_load_store_ops(operands)?;
    let func = self.offset_to_func(offset, true)?;
    self.emit_word(0x6000 | ((rd as u16) << 8)
                          | ((rs as u16) << 4) | func);
}
"SW" => {
    let (rs2, rs1, offset) = self.get_store_ops(operands)?;
    let func = self.offset_to_func(offset, false)?;
    self.emit_word(0x7000 | ((rs2 as u16) << 8)
                          | ((rs1 as u16) << 4) | func);
}
```

The `offset_to_func` method maps common offsets to the 4-bit function code:

Listing 10.10: Offset-to-function mapping

```
fn offset_to_func(&self, offset: i32, is_load: bool) -> Result<u16, String> {
    match offset {
```

```
            0 => Ok(0x0),
            2 => Ok(0x3),
            4 => Ok(0x4),
            6 => Ok(0x5),
            -2 => Ok(0x6),
            -4 => Ok(0x7),
            _ => Err(format!("Unsupported offset {} for short load/store",
                             offset)),
        }
    }
```

Limited Offset Set

The base 16-bit load/store encoding only supports offsets of 0, $+2$, $+4$, $+6$, -2, and -4—a set tailored for struct field access and stack operations. For arbitrary offsets, use the extended **LWX** or **SWX** instructions, which carry a full 16-bit offset in the second word.

The **LB**, **LBU**, and **SB** variants use function codes `0x1`, `0x2`, and `0x1` respectively, with the offset hard-coded to zero in the current implementation.

10.2.4 Two-Register Operations

Shift, multiply/divide, and comparison instructions use a two-register format with a function code in bits 3:0:

Listing 10.11: Encoding shift instructions

```
"SLL" => {
    let (rd, rs) = self.get_two_regs(operands)?;
    self.emit_word(0xA000 | ((rd as u16) << 8)
                          | ((rs as u16) << 4) | 0x0);
}
"SRL" => {
    let (rd, rs) = self.get_two_regs(operands)?;
    self.emit_word(0xA000 | ((rd as u16) << 8)
                          | ((rs as u16) << 4) | 0x1);
}
```

The pattern is identical across the shift family (**SLL**, **SRL**, **SRA**, **ROL**, **ROR**, **SWAP**) and the multiply/divide family (**MUL**, **MULH**, **MULHU**, **DIV**, **DIVU**, **REM**, **REMU**). Only the opcode and function code differ:

Instruction	Opcode	func
SLL	0xA	0x0
SRL	0xA	0x1
SRA	0xA	0x2
ROL	0xA	0x3
ROR	0xA	0x4
SWAP	0xA	0x7
MUL	0xB	0x0
MULH	0xB	0x1
DIV	0xB	0x3
DIVU	0xB	0x4
REM	0xB	0x5
REMU	0xB	0x6
DAA	0xB	0x7

Table 10.1: Function codes for shift and multiply/divide operations

10.2.5 Stack and Miscellaneous Operations

Opcode `0xC` covers a mixed bag: stack operations, comparison, data movement, block transfers, and the register-swap instruction. The encoding follows the same R-format layout, with the function code distinguishing each operation:

Listing 10.12: Stack and misc instruction encoding

```
"PUSH" => {
    let rs = self.get_one_reg(operands)?;
    self.emit_word(0xC000 | ((rs as u16) << 4) | 0x0);
}
"POP" => {
    let rd = self.get_one_reg(operands)?;
    self.emit_word(0xC000 | ((rd as u16) << 8) | 0x1);
}
"CMP" => {
    let (rd, rs) = self.get_two_regs(operands)?;
    self.emit_word(0xC000 | ((rd as u16) << 8)
                          | ((rs as u16) << 4) | 0x2);
}
"EXX" => self.emit_word(0xC00B),
```

Note how **PUSH** places the source register in the Rs1 field (bits 7:4), while **POP** places the destination in the Rd field (bits 11:8). The zero-operand instructions (**EXX**, **LDI**, **LDD**, etc.) are encoded as fixed 16-bit constants.

10.2.6 System Instructions

System instructions use opcode `0xE` with a function nibble in bits 11:8:

Listing 10.13: System instruction encoding

```
"NOP"  => self.emit_word(0xE000),
"HALT" => self.emit_word(0xE100),
"DI"   => self.emit_word(0xE200),
"EI"   => self.emit_word(0xE300),
"RETI" => self.emit_word(0xE400),
"SWI"  => {
    let imm = self.get_imm(operands)?;
    self.emit_word(0xE500 | ((imm as u8) as u16));
}
```

Most system instructions are fully determined by the opcode—they have no register or immediate operands. **SWI** is the exception: it carries an 8-bit interrupt vector number in bits 7:0.

10.2.7 Pseudo-Instructions: NEG and NOT

Some instructions are *pseudo-instructions*—they do not have their own opcode but are instead translated into other encodings:

Listing 10.14: Pseudo-instruction encoding

```
"NEG" => {
    let (rd, rs) = self.get_two_regs(operands)?;
    // NEG is SUB Rd, R0, Rs
    self.emit_word(0x1000 | ((rd as u16) << 8) | ((rs as u16)));
}
"NOT" => {
    let (rd, rs) = self.get_two_regs(operands)?;
    // NOT is XOR Rd, Rs, 0xFFFF - use extended
    self.emit_word(0xF000 | ((rd as u16) << 8)
                          | ((rs as u16) << 4) | 0x04);
    self.emit_word(0xFFFF);
}
```

NEG Rd, Rs is encoded as **SUB** Rd, **R0**, Rs—subtracting the source from the hardwired-zero register produces the two's complement negation.

NOT Rd, Rs uses an extended **XORIX** with the immediate 0xFFFF. XOR with all-ones flips every bit, producing the bitwise complement. Because the immediate does not fit in any base format, this pseudo-instruction always produces a 32-bit encoding.

10.3 Immediate Encoding and Sign Extension

Immediates in the SAMPO ISA come in several widths:

Width	Range	Used By
8-bit signed	-128 to $+127$	**ADDI**, branch offset, **SWI**
12-bit signed	-2048 to $+2047$	**J** (jump offset)
16-bit unsigned	0 to 65535	Extended instructions (**LIX**, **JX**, etc.)

Table 10.2: Immediate widths in the SAMPO ISA

10.3.1 Sign Extension in Hardware

When the CPU executes an **ADDI** with an 8-bit immediate, the hardware *sign-extends* the value to 16 bits before performing the addition. Sign extension copies the most significant bit of the narrow value into all the upper bits:

$$\text{sign_ext}(\texttt{0000 1010}) = \texttt{0x000A} \quad (+10)$$
$$\text{sign_ext}(\texttt{1111 0110}) = \texttt{0xFFF6} \quad (-10)$$

The assembler's job is to ensure the programmer's intended value fits in the available bits. For **ADDI**, the check is explicit:

Listing 10.15: Range checking for **ADDI**

```
if imm < -128 || imm > 127 {
    return Err(format!("Immediate {} out of range for ADDI", imm));
}
```

10.3.2 Truncation for Encoding

After range checking, the assembler must truncate the 32-bit `i32` value down to the appropriate width while preserving the two's complement bit pattern. The critical cast sequence is:

```
((imm as u8) as u16)
```

This first truncates to 8 bits (`as u8`), discarding the upper 24 bits. The resulting `u8` is then zero-extended to `u16`, placing the 8-bit value in the low byte. For negative numbers, the two's complement representation is preserved: -1 becomes `0xFF`, which is correct because the hardware will sign-extend `0xFF` back to `0xFFFF` at execution time.

Two's Complement Truncation

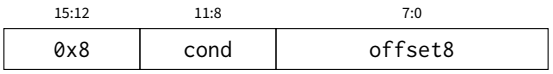
When truncating a signed value to fewer bits, the two's complement bit pattern is preserved if the value fits in the target width. For a value v with target width n bits: if $-2^{n-1} \leq v \leq 2^{n-1} - 1$, then the low n bits of v are its correct n-bit two's complement representation.

10.4 Branch Offset Calculation

Branches are PC-relative: the target address is computed from the current program counter plus a signed offset. This section details exactly how the assembler computes those offsets.

10.4.1 Branch Encoding Format

The branch instruction format uses opcode `0x8`:



The hardware computes the target as:

$$\text{target} = \text{PC} + 2 + \text{sign_ext}(\text{offset8}) \times 2$$

The "$+2$" accounts for the fact that the PC has already advanced past the branch instruction itself (each instruction is 2 bytes). The "$\times 2$" converts the offset from *word* units to *byte* units—since all instructions are at least 2-byte aligned, this effectively doubles the branch range.

10.4.2 The `emit_branch` Method

Listing 10.16: Branch emission with fixup recording

```
fn emit_branch(&mut self, cond: u16, operands: &[Operand])
    -> Result<(), String>
{
    match operands.first() {
```

```
        Some(Operand::Label(label)) => {
            self.emit_word(0x8000 | (cond << 8));
            let fixup_addr = self.pc - 2;
            self.fixups.push(Fixup {
                address: fixup_addr,
                symbol: label.clone(),
                kind: FixupKind::Relative8,
            });
        }
        Some(Operand::Immediate(offset)) => {
            let off = *offset / 2; // Convert to words
            if off < -128 || off > 127 {
                return Err("Branch offset out of range".to_string());
            }
            self.emit_word(0x8000 | (cond << 8)
                           | ((off as u8) as u16));
        }
        _ => return Err("Branch requires target".to_string()),
    }
    Ok(())
}
```

When the target is a label, the assembler emits the instruction with a zero offset and records a `Fixup` to be patched in pass 3. The `fixup_addr` is `self.pc - 2` because `emit_word` has already advanced the program counter past the instruction.

When the target is a raw numeric offset, the assembler divides by 2 (converting from byte offset to word offset) and encodes it directly.

10.4.3 The 16 Branch Conditions

The condition code occupies bits 11:8, giving 16 possible conditions:

10.4.4 Branch Range

With an 8-bit signed offset measured in words, a branch can reach:

$$-128 \times 2 = -256 \text{ bytes} \quad \text{to} \quad +127 \times 2 = +254 \text{ bytes}$$

relative to the instruction following the branch. That is a range of roughly ± 128 instructions in each direction. For targets beyond this range, the programmer must use an unconditional **JX** (absolute jump) with a conditional branch around it.

cond	Mnemonic	Condition	Flags
0x0	**BEQ**	Equal	Z = 1
0x1	**BNE**	Not equal	Z = 0
0x2	**BLT**	Signed less than	N ≠ V
0x3	**BGE**	Signed greater/equal	N = V
0x4	**BLTU**	Unsigned less than	C = 0
0x5	**BGEU**	Unsigned greater/eq	C = 1
0x6	**BMI**	Minus (negative)	N = 1
0x7	**BPL**	Plus (positive)	N = 0
0x8	**BVS**	Overflow set	V = 1
0x9	**BVC**	Overflow clear	V = 0
0xA	**BCS**	Carry set	C = 1
0xB	**BCC**	Carry clear	C = 0
0xC	**BGT**	Signed greater	Z=0 and N=V
0xD	**BLE**	Signed less/equal	Z=1 or N≠V
0xE	**BHI**	Unsigned higher	C=1 and Z=0
0xF	**BLS**	Unsigned lower/same	C=0 or Z=1

Table 10.3: Branch condition encoding

10.5 Jump Encoding

The **J** instruction (unconditional relative jump) uses a 12-bit signed offset:

Listing 10.17: Jump emission with fixup recording

```
fn emit_jump(&mut self, operands: &[Operand]) -> Result<(), String> {
    match operands.first() {
        Some(Operand::Label(label)) => {
            self.emit_word(0x9000);
            let fixup_addr = self.pc - 2;
            self.fixups.push(Fixup {
                address: fixup_addr,
                symbol: label.clone(),
                kind: FixupKind::Relative12,
            });
        }
        Some(Operand::Immediate(offset)) => {
            let off = *offset / 2;
            if off < -2048 || off > 2047 {
                return Err("Jump offset out of range".to_string());
            }
```

```
            self.emit_word(0x9000 | ((off as u16) & 0x0FFF));
        }
        _ => return Err("Jump requires target".to_string()),
    }
    Ok(())
}
```

The 12-bit offset gives a range of ± 4094 bytes—substantially more than a branch. The mask `& 0x0FFF` ensures only the low 12 bits of the offset are placed into the instruction word.

10.5.1 Register Jumps: JR and JALR

The **JR** and **JALR** instructions share opcode `0x9` with the relative **J** but use register fields instead of an offset:

Listing 10.18: Register jump encoding

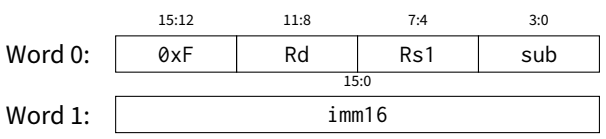
```
"JR" => {
    let rs = self.get_one_reg(operands)?;
    self.emit_word(0x9F00 | ((rs as u16) << 4));
}
"JALR" => {
    let (rd, rs) = self.get_two_regs(operands)?;
    self.emit_word(0x9000 | ((rd as u16) << 8)
                          | ((rs as u16) << 4) | 0x1);
}
```

JR sets bits 11:8 to `0xF` (the pattern `1001 1111 SSSS 0000`), which the hardware uses to distinguish it from a relative jump. **JALR** uses function code `0x1` in bits 3:0.

10.6 Extended Instruction Encoding

When a 16-bit encoding cannot carry enough information—a full 16-bit immediate, an absolute address, or an 8-bit port number—the assembler uses the extended format with opcode `0xF`. Extended instructions are 32 bits (two words).

10.6.1 Extended Format Layout



The sub field in bits 3:0 of the first word selects the extended operation:

sub	Mnemonic	Operation
0x0	**ADDIX**	Rd = Rs1 + imm16
0x1	**SUBIX**	Rd = Rs1 − imm16
0x2	**ANDIX**	Rd = Rs1 & imm16
0x3	**ORIX**	Rd = Rs1 | imm16
0x4	**XORIX**	Rd = Rs1 ⊕ imm16
0x5	**LWX**	Rd = mem16[Rs1 + imm16]
0x6	**SWX**	mem16[Rs1 + imm16] = Rd
0x7	**LIX**	Rd = imm16
0x8	**JX**	PC = imm16
0x9	**JALX**	RA = PC + 4; PC = imm16
0xA	**CMPIX**	flags = Rd − imm16
0xB	**INX**	Rd = port_read(imm16[7:0])
0xC	**OUTX**	port_write(imm16[7:0], Rs1)
0xD	**SLLX**	Rd = Rs1 ≪ imm16[3:0]
0xE	**SRLX**	Rd = Rs1 ≫ imm16[3:0]
0xF	**SRAX**	Rd = Rs1 ⋙ imm16[3:0]

Table 10.4: Extended instruction sub-opcodes

10.6.2 Encoding `LIX`: Load Immediate Extended

`LIX` is the workhorse for loading any 16-bit value into a register:

Listing 10.19: Encoding **`LIX`** with label support

```
"LIX" => {
    let (rd, imm) = self.get_reg_imm_or_label(operands)?;
    self.emit_word(0xF000 | ((rd as u16) << 8) | 0x07);
    match imm {
        Either::Imm(v) => self.emit_word(v as u16),
        Either::Label(l) => {
            self.fixups.push(Fixup {
                address: self.pc,
                symbol: l,
                kind: FixupKind::Absolute16,
            });
            self.emit_word(0);
        }
    }
```

```
}
```

The first word is `0xF`R`07`, where R is the destination register. The second word is the 16-bit immediate value. When the value is a label, the assembler emits a zero placeholder and records an `Absolute16` fixup.

The `Either` Helper

The code generator uses a private `Either` enum to handle operands that can be either an immediate or a label:

```
enum Either {
    Imm(i32),
    Label(String),
}
```

This avoids duplicating the emission logic for each case.

Let us encode `LIX R4, 0x1234`:

Word	Value	Meaning
0	`0xF407`	opcode=0xF, Rd=4, Rs1=0, sub=0x7
1	`0x1234`	16-bit immediate

In memory (little-endian): `0x07`, `0xF4`, `0x34`, `0x12`—four bytes.

10.6.3 Encoding I/O Instructions

The **`INI`** and **`OUTI`** instructions use the extended format to carry the 8-bit port number:

Listing 10.20: Encoding **`INI`** and **`OUTI`**

```
"INI" => {
    let (rd, port) = self.get_reg_imm(operands)?;
    if port < 0 || port > 255 {
        return Err("Port number out of range".to_string());
    }
    self.emit_word(0xF000 | ((rd as u16) << 8) | 0x0B);
    self.emit_word(port as u16);
}
"OUTI" => {
    let (port, rs) = self.get_imm_reg(operands)?;
    if port < 0 || port > 255 {
        return Err("Port number out of range".to_string());
```

```
    }
    self.emit_word(0xF000 | ((rs as u16) << 4) | 0x0C);
    self.emit_word(port as u16);
}
```

Notice the operand order difference: **INI** takes `Rd, port` while **OUTI** takes `port, Rs`. This mirrors the data flow direction: "read *into* register" versus "write *from* register to port". The assembler handles this asymmetry with different helper methods (`get_reg_imm` vs. `get_imm_reg`).

10.7 Symbol Table and Forward References

10.7.1 The Fixup Mechanism

When pass 2 encounters a reference to a label—whether in a branch, jump, or **LIX**—it may not be able to compute the final value yet. The assembler records a *fixup*: a note that says "at byte address A in the output, patch in the value for symbol S using method K."

Listing 10.21: The `Fixup` and `FixupKind` types

```
struct Fixup {
    address: u16,
    symbol: String,
    kind: FixupKind,
}

#[derive(Clone, Copy)]
enum FixupKind {
    Absolute16,
    Relative8,
    Relative12,
}
```

Three fixup kinds cover all the cases:

Absolute16
: The full 16-bit address of the symbol replaces the two bytes at the fixup address. Used by **LIX**, **JX**, **JALX**, and `.dw` with label arguments.

Relative8
: A signed 8-bit word offset from the instruction following the branch. Used by all conditional branch instructions (**BEQ**, **BNE**, etc.).

`Relative12`
: A signed 12-bit word offset from the instruction following the jump. Used by the **`J`** instruction.

10.7.2 Pass 3: Applying Fixups

The `apply_fixups` method iterates over all recorded fixups and patches the output buffer:

Listing 10.22: The fixup application pass

```
fn apply_fixups(&mut self) -> Result<(), String> {
    for fixup in &self.fixups {
        let target = *self.symbols.get(&fixup.symbol)
            .ok_or_else(|| format!("Undefined symbol: {}",
                                   fixup.symbol))?;

        let addr = fixup.address as usize;

        match fixup.kind {
            FixupKind::Absolute16 => {
                self.output[addr] = (target & 0xFF) as u8;
                self.output[addr + 1] = (target >> 8) as u8;
            }
            FixupKind::Relative8 => {
                let pc_after = fixup.address + 2;
                let offset = (target as i32 - pc_after as i32) / 2;
                if offset < -128 || offset > 127 {
                    return Err(format!(
                        "Branch to {} out of range", fixup.symbol));
                }
                self.output[addr] = (offset as i8) as u8;
            }
            FixupKind::Relative12 => {
                let pc_after = fixup.address + 2;
                let offset = (target as i32 - pc_after as i32) / 2;
                if offset < -2048 || offset > 2047 {
                    return Err(format!(
                        "Jump to {} out of range", fixup.symbol));
                }
                let existing = u16::from_le_bytes([
                    self.output[addr],
                    self.output[addr + 1]]);
                let new_word = (existing & 0xF000)
                    | ((offset as u16) & 0x0FFF);
                self.output[addr] = (new_word & 0xFF) as u8;
                self.output[addr + 1] = (new_word >> 8) as u8;
```

```
            }
        }
    }
    Ok(())
}
```

Each fixup kind uses a different patching strategy:

- **Absolute16**: Write the 16-bit target address in little-endian order. This completely replaces the two placeholder bytes.
- **Relative8**: Compute the offset as $(\text{target} - \text{PC}_{\text{after}})/2$, where PC_{after} is the address of the instruction *following* the branch. The division by 2 converts from bytes to words. Only the low byte (the offset field, bits 7:0) is patched; the high byte (opcode and condition) was already correct.
- **Relative12**: Similar to Relative8, but the offset occupies bits 11:0 of a 16-bit word. The fixup reads the existing word, preserves the opcode nibble (bits 15:12), and OR-s in the 12-bit offset.

Range Checking Happens Twice

For forward references, the range check happens during fixup application, not during emission. This means an out-of-range branch is only detected in pass 3. For backward references (where the target is already known), the check could happen in pass 2. In practice, the current implementation always defers label-based offset computation to pass 3.

10.7.3 The Offset Formula in Detail

Let us derive the offset calculation for a concrete example. Consider a program where **BEQ** is at address `0x0108` and the target label done is at address `0x0112`:

$$\text{PC}_{\text{after}} = \texttt{0x0108} + 2 = \texttt{0x010A}$$

$$\text{offset} = \frac{\texttt{0x0112} - \texttt{0x010A}}{2} = \frac{8}{2} = 4$$

The hardware will compute:

$$\text{target} = \text{PC} + 2 + 4 \times 2 = \texttt{0x0108} + 2 + 8 = \texttt{0x0112} \checkmark$$

For a backward branch (target before the branch), the offset is negative. If **BNE** is at `0x0112` targeting a label at `0x0108`:

$$\text{offset} = \frac{\texttt{0x0108} - \texttt{0x0114}}{2} = \frac{-12}{2} = -6$$

As an 8-bit signed value: $-6 = \texttt{0xFA}$. The hardware decodes this correctly via sign extension.

10.7.4 Little-Endian Byte Emission

Every 16-bit word is emitted in little-endian order:

Listing 10.23: Little-endian word emission

```
fn emit_word(&mut self, w: u16) {
    // Little-endian
    self.output.push((w & 0xFF) as u8);
    self.output.push((w >> 8) as u8);
    self.pc += 2;
}
```

The low byte goes first, the high byte second. This matches the SAMPO's memory organization: the byte at the lower address holds bits 7:0, and the byte at the higher address holds bits 15:8.

10.7.5 Directive Emission

The code generator also handles emitting data from directives:

Listing 10.24: Directive emission (selected cases)

```
"db" => {
    for arg in args {
        match arg {
            DirectiveArg::Number(n) => { self.emit_byte(*n as u8); }
            DirectiveArg::String(s) => {
                for b in s.bytes() { self.emit_byte(b); }
            }
            DirectiveArg::Ident(sym) => {
                if let Some(&val) = self.symbols.get(sym) {
                    self.emit_byte(val as u8);
                } else {
                    return Err(format!("Undefined symbol: {}", sym));
                }
            }
        }
    }
}
```

```
"asciz" => {
    for arg in args {
        if let DirectiveArg::String(s) = arg {
            for b in s.bytes() { self.emit_byte(b); }
            self.emit_byte(0); // null terminator
        }
    }
}
```

The `.asciz` directive emits the string bytes followed by a null byte. The `.db` directive handles a mix of numbers, strings, and symbol references—this flexibility lets you build lookup tables and mixed data blocks in assembly.

10.8 Hands-On: Encode Instructions by Hand

Let us work through several encoding examples from the SAMPO project's test suite, computing the binary output by hand and verifying against the assembler's behavior.

10.8.1 Example 1: Simple ALU (from `examples/simple.s`)

```
        .org 0x0100
start:
        ADD  R4, R5, R6         ; 0x0456
        SUB  R4, R5, R6         ; 0x1456
        ADDI R4, 10             ; 0x540A
```

The comments in `simple.s` helpfully include the expected encoding. Let us verify:

ADD R4, R5, R6:

$$\texttt{opcode=0000}, \text{Rd}=\texttt{0100}, \text{Rs1}=\texttt{0101}, \text{Rs2}=\texttt{0110}$$

$$= \texttt{0000 0100 0101 0110} = \texttt{0x0456} \checkmark$$

SUB R4, R5, R6:

$$\texttt{opcode=0001}, \text{Rd}=\texttt{0100}, \text{Rs1}=\texttt{0101}, \text{Rs2}=\texttt{0110}$$

$$= \texttt{0001 0100 0101 0110} = \texttt{0x1456} \checkmark$$

ADDI R4, 10:

$$\texttt{opcode=0101}, \text{Rd}=\texttt{0100}, \text{imm8}=10=\texttt{0000 1010}$$

$$= \texttt{0101 0100 0000 1010} = \texttt{0x540A} \checkmark$$

Memory layout starting at `0x0100` (little-endian):

Address	Byte	Note
`0x0100`	`0x56`	ADD low byte
`0x0101`	`0x04`	ADD high byte
`0x0102`	`0x56`	SUB low byte
`0x0103`	`0x14`	SUB high byte
`0x0104`	`0x0A`	ADDI low byte
`0x0105`	`0x54`	ADDI high byte

10.8.2 Example 2: Branch Offset (Forward Reference)

```
        .org 0x0100
start:                          ; address 0x0100
        ADD  R4, R5, R6         ; 0x0100 (2 bytes)
        SUB  R4, R5, R6         ; 0x0102 (2 bytes)
        BEQ  next               ; 0x0104 (2 bytes)
        J    start              ; 0x0106 (2 bytes)
next:                           ; address 0x0108
        PUSH R4                 ; 0x0108
```

`BEQ next`: The branch is at `0x0104`, and `next` is at `0x0108`.

$$\text{offset} = \frac{\texttt{0x0108} - (\texttt{0x0104} + 2)}{2} = \frac{\texttt{0x0108} - \texttt{0x0106}}{2} = \frac{2}{2} = 1$$

Encoding: opcode=`0x8`, cond=`0x0` (BEQ), offset8=1=`0x01`

$=$ `0x8001`

`J start`: The jump is at `0x0106`, and `start` is at `0x0100`.

$$\text{offset} = \frac{\texttt{0x0100} - (\texttt{0x0106} + 2)}{2} = \frac{\texttt{0x0100} - \texttt{0x0108}}{2} = \frac{-8}{2} = -4$$

As a 12-bit signed value: -4 = `0xFFC`.

Encoding: opcode=`0x9`, offset12=`0xFFC`

$=$ `0x9FFC`

10.8.3 Example 3: Extended Instruction (from `examples/tests/test_extended.s`)

The test suite verifies extended instruction encoding with manual `.dw` directives:

Listing 10.25: ADDIX encoding from the test suite

```
; ADDIX R4, R8, 0x1234: R4 = R8 + 0x1234
; Encoding: 0xF480 0x1234
LIX R8, 0x1000
.dw 0xF480, 0x1234
```

Let us verify the `0xF480` encoding:

Field	Value	Meaning
opcode	`0xF`	Extended
Rd	`0x4` (`R4`)	Destination
Rs1	`0x8` (`R8`)	Source
sub	`0x0`	ADDIX

= `1111 0100 1000 0000` = `0xF480` ✓

The second word `0x1234` is the 16-bit immediate. At execution time, the CPU computes `R4` = `R8` + `0x1234` = `0x1000` + `0x1234` = `0x2234`.

10.8.4 Example 4: HALT and PUSH

From the encoding specification:

`HALT`: opcode=`0xE`, func=`0x1`, operand=`0x00`

= `0xE100`

`PUSH R4`: opcode=`0xC`, Rd=`0x0`, Rs1=`0x4` (the register being pushed), func=`0x0`

= `0xC040`

Note that **`PUSH`** places the source register in the Rs1 field, not the Rd field. This makes sense from the hardware perspective: **`PUSH`** reads a register (Rs1) and writes to memory, while **`POP`** reads from memory and writes to a register (Rd).

Exercises

1. **Encode a loop by hand.** Given this fragment starting at `0x0100`, compute the binary encoding of every instruction, including all branch and jump offsets:

```
        .org 0x0100
start:
        LIX  R4, 0x0010
loop:
        ADDI R4, -1
        BNE  loop
        HALT
```

Write out the complete byte sequence at addresses `0x0100` through `0x010B`.

2. **Maximum branch range.** What is the farthest forward a **`BEQ`** can branch, in bytes? What about the farthest backward? Write a test program that places a branch at the exact limit of its range and verify it assembles without error.

3. **Fixup archaeology.** Modify the assembler (or add a `--verbose` flag) to print each fixup as it is applied during pass 3. For each fixup, print the address, symbol name, kind, and the computed value. Run it on `examples/hello.s` and verify the fixup values match your hand calculations.

4. **The `NOT` pseudo-instruction.** The assembler encodes `NOT R4, R5` as an extended **`XORIX`** with `0xFFFF`. Could **`NOT`** be encoded more efficiently? Consider using the base-format **`XOR`** with a register holding `0xFFFF`, or the ALU's reserved function codes. What are the trade-offs?

5. **Label shadowing.** What happens if two labels have the same name in the current assembler? Trace through pass 1 and explain the behavior. Is this desirable? How would you detect and report duplicate labels?

What's Next We now have a complete picture of how `sasm` turns assembly source into binary machine code. But the story does not end at the `.bin` file. In Chapter 11, we will follow the binary through the rest of the toolchain: converting it to Verilog's hex format for simulation, patching it for FPGA continuous execution, and building a disassembler that can reverse the entire process.

Chapter 11

The Binary Toolchain

The assembler produces a `.bin` file—a flat image of the SAMPO's memory. But that binary is not the end of the road. Before it can run in Verilog simulation, it must be converted to a hex format that `$readmemh` can ingest. Before it can run on an FPGA, the **`HALT`** instruction at the end must be patched into a delay loop, because an FPGA does not "stop" the way a simulator does. And somewhere along the way, we discovered a bug in the address translation that taught us more about the CPU's memory model than weeks of careful design ever could.

This chapter follows the SAMPO binary from the assembler's output to the simulator's input, through the FPGA patching tool, and finally into a disassembler that can reverse the entire journey.

11.1 Binary Output Format

11.1.1 What the Assembler Produces

The `sasm` assembler writes a flat binary file: a byte-for-byte image of the SAMPO's 64 KB address space, starting from address zero and extending through the last byte emitted. There is no header, no metadata, no symbol table—just raw bytes.

Consider the `examples/simple.s` test program:

Listing 11.1: `examples/simple.s`: a minimal test program

```
; Simple test for Sampo assembler
        .org 0x0100

start:
        ADD  R4, R5, R6         ; 0x0456
```

```
        SUB  R4, R5, R6         ; 0x1456
        ADDI R4, 10             ; 0x540A
        LW   R4, (R5)           ; 0x6450
        SW   (R5), R4           ; 0x7450
        BEQ  next               ; branch forward
        J    start              ; jump back
next:
        PUSH R4                 ; push
        POP  R5                 ; pop
        MOV  R4, R5             ; move
        NOP                     ; no-op
        HALT                    ; halt
```

The `.org 0x0100` directive places the code at byte address `0x0100` (256 decimal). The assembler pads the output with 256 zero bytes, then emits the instructions. The result is a binary file of approximately 278 bytes: 256 bytes of padding plus the encoded instructions.

11.1.2 The Origin Padding

As we saw in Section 10.1.5, the code generator pads the output buffer to the origin address before emitting any code:

Listing 11.2: Origin padding in the code generator

```
// Pad output to origin if needed
while self.output.len() < self.origin as usize {
    self.output.push(0);
}
```

For the standard SAMPO reset vector at `0x0100`, this produces 256 bytes of zeros. Those zeros occupy addresses `0x0000` through `0x00FF`—an area that is available for data or, as we will see in Section 31.5.3, for patching in a delay loop.

11.1.3 Little-Endian Byte Order

Every 16-bit instruction is stored in little-endian order: the low byte first, the high byte second. The **ADD** R4, R5, R6 instruction encodes as `0x0456`, but appears in the binary file as bytes `0x56`, `0x04`.

This matches the SAMPO's memory architecture (see **??**). The RAM module in `verilog/rtl/ram.v` reads 16-bit words from adjacent byte pairs:

Listing 11.3: RAM word addressing in `rtl/ram.v`

```
// Word-aligned address (ignore LSB)
```

```
wire [14:0] word_addr = addr[15:1];
```

The address bus drops its least significant bit, converting a 16-bit byte address into a 15-bit word address. This means addresses `0x0100` and `0x0101` both refer to the same 16-bit word in memory.

11.2 bin2hex.py: Binary to Verilog Hex

11.2.1 Why Convert?

Verilog's `$readmemh` system task reads hexadecimal text files to initialize memory arrays. It cannot read raw binary. The `verilog/bin2hex.py` script bridges this gap, converting the assembler's `.bin` output into a hex file suitable for simulation.

Listing 11.4: The conversion step in the build chain

```
# Assemble the source
cd sasm && cargo run --release -- ../examples/hello.s -o ../examples/hello.bin

# Convert to hex
cd ../verilog && python3 bin2hex.py ../examples/hello.bin programs/hello.hex
```

11.2.2 The Complete Script

The script is short enough to show in its entirety:

Listing 11.5: `verilog/bin2hex.py`: binary to Verilog hex conversion

```
#!/usr/bin/env python3
"""Convert binary file to Verilog hex format for $readmemh"""

import sys

def main():
    if len(sys.argv) < 3:
        print("Usage: bin2hex.py <input.bin> <output.hex>")
        sys.exit(1)

    input_file = sys.argv[1]
    output_file = sys.argv[2]

    with open(input_file, 'rb') as f:
        data = f.read()
```

```
    # Pad to even length if necessary
    if len(data) % 2:
        data += b'\x00'

    # Convert to 16-bit words (little-endian)
    words = []
    for i in range(0, len(data), 2):
        # Little-endian: low byte first
        word = data[i] | (data[i+1] << 8)
        words.append(word)

    # Write hex file
    with open(output_file, 'w') as f:
        f.write("@0000\n")
        for i, word in enumerate(words):
            f.write(f"{word:04X}\n")

    print(f"Converted {len(data)} bytes to {len(words)} words")

if __name__ == '__main__':
    main()
```

11.2.3 How It Works

The conversion proceeds in three steps:

1. **Read the binary**: The entire `.bin` file is loaded into memory.
2. **Pad to even length**: If the file has an odd number of bytes, a zero byte is appended. This ensures every pair of bytes forms a complete 16-bit word.
3. **Convert to words**: Each pair of bytes is combined into a 16-bit word using little-endian ordering: the first byte (at the lower address) becomes the low byte, and the second byte becomes the high byte.
4. **Write the hex file**: The output starts with `@0000`, which tells `$readmemh` to begin loading at word address 0. Each subsequent line contains one word in uppercase hexadecimal.

The @ Address Directive

In Verilog's hex file format, a line starting with `@` followed by a hexadecimal number sets the memory address for the following data. The address is in *word* units, not bytes. So `@0000` means "start loading at word 0 (byte address 0x0000)" and `@0080` would mean "start loading at word 128 (byte address 0x0100)."

11.2.4 Output Format

For the hello.s program, the hex file looks like this:

Listing 11.6: Excerpt from `programs/hello.hex` (first code words at word 128+)

```
@0000
0000
0000
...
0000
F407
011E
6542
C502
8009
F60B
0080
```

The first 128 words (addresses `0x0000`–`0x00FF`) are zeros—the padding before the `.org 0x0100` origin. Starting at word 129 (address `0x0100`), we see the actual code: `0xF407` is the first word of the `LIX R4`, `message` instruction.

11.2.5 Integration with the RAM Module

The hex file is loaded into the simulated RAM via the `INIT_FILE` parameter:

Listing 11.7: RAM initialization in `rtl/ram.v`

```
// 64KB = 32K x 16-bit words
reg [15:0] mem [0:32767];

// Initialize memory from hex file if specified
initial begin
    if (INIT_FILE != "") begin
        $readmemh(INIT_FILE, mem);
    end
end
```

The `$readmemh` call reads the hex file and populates the `mem` array. Each line in the hex file fills one entry in the 32K-word array. The simulation testbench passes the file path as a parameter:

Listing 11.8: Running simulation with a specific program

```
iverilog -Psampo_tb.RAM_INIT_FILE=\"../programs/hello.hex\" -o build/sampo_tb \
```

```
    rtl/alu.v rtl/shifter.v rtl/regfile.v rtl/decode.v \
    rtl/cpu.v rtl/ram.v rtl/uart.v rtl/soc.v tb/sampo_tb.v
```

11.2.6 The Makefile Pipeline

The complete build chain is automated in `verilog/Makefile`:

Listing 11.9: Makefile rules for assembly and conversion

```
# Assemble source to binary
../examples/tests/%.bin: ../examples/tests/%.s
  cd ../sasm && cargo run --release -- $< -o $@

# Convert test binary to hex
programs/%.hex: ../examples/tests/%.bin
  python3 bin2hex.py $< $@
```

These pattern rules let you build any test program with a single command:

Listing 11.10: Building and running a test

```
make isa-test_alu
```

Make traces the dependencies: the hex file depends on the binary, which depends on the assembly source. If the source changes, Make rebuilds everything automatically.

11.3 `hex_loop_patch.py`: Patching for FPGA

11.3.1 The Problem: HALT on an FPGA

In simulation, **`HALT`** works beautifully—the testbench detects the halted state and terminates. On an FPGA, **`HALT`** is a problem.

When the CPU halts on the FPGA, it simply stops executing. The UART transmit buffer may still contain characters that have not been shifted out. There is no mechanism to "end" the program and report results. Worse, if you connect a serial terminal after programming the FPGA, you might miss the entire output—the program ran and halted before you opened the connection.

The solution: replace the **`HALT`** with a delay loop that waits about half a second, then jumps back to the reset vector. The program runs continuously, printing its output every half second. You can connect your serial terminal at any time and catch the next iteration.

11.3.2 The Patching Strategy

The `verilog/hex_loop_patch.py` script implements this strategy:

1. Find the **HALT** instruction in the hex file.
2. Place a delay loop in the unused memory before the reset vector (addresses `0x0000`–`0x00FF`).
3. Replace the **HALT** with a **J** instruction that jumps to the delay loop.

This approach avoids shifting any program data, preserving all address references that were computed by the assembler.

Why Not Just Modify the Source?

We could add the delay loop to the assembly source, but that would change every program's label addresses and binary output. By patching the hex file *after* assembly, we keep the original binary identical for simulation while producing a modified version for FPGA use. The Makefile automates this with a separate `%_loop.hex` target.

11.3.3 The Delay Loop

The delay loop is placed at address `0x0000` and consists of a nested counter:

Listing 11.11: The delay loop (placed at address `0x0000`)

```
        ; At address 0x0000:
        LIX R8, 0x0008       ; Outer counter = 8
outer:
        LIX R9, 0xFFFF       ; Inner counter = 65535
inner:
        ADDI R9, -1           ; R9 = R9 - 1
        BNE inner             ; Loop while R9 != 0
        ADDI R8, -1           ; R8 = R8 - 1
        BNE outer             ; Loop while R8 != 0
        JX 0x0100             ; Restart at reset vector
```

The total delay is approximately $8 \times 65535 \times 9 \approx 4.7$ million cycles. At the SAMPO's 12.5 MHz clock rate on the ULX3S board, this is roughly 0.38 seconds—long enough for the UART to flush its buffer, short enough that the serial output is responsive.

In the script, the loop is encoded as a list of raw instruction words:

Listing 11.12: Delay loop encoding in `hex_loop_patch.py`

```
LOOP_PATCH = [
```

```
    0xF807, 0x0008,  # LIX R8, 0x0008
    0xF907, 0xFFFF,  # LIX R9, 0xFFFF
    0x59FF,          # ADDI R9, -1
    0x81FE,          # BNE -2 (inner)
    0x58FF,          # ADDI R8, -1
    0x81FA,          # BNE -6 (outer)
    0xF008, 0x0100,  # JX 0x0100
]
```

11.3.4 The Jump Encoding Challenge

The patch replaces **HALT** with a relative **J** instruction that jumps to the delay loop at address 0. But there is a subtlety: the **J** instruction shares opcode `0x9` with **JR** and **JALR**. If the 12-bit offset field happens to match the bit patterns that the hardware decodes as **JR** or **JALR**, the CPU will interpret the instruction incorrectly.

The script checks for these conflicts:

Listing 11.13: Jump encoding with decode conflict detection

```
def encode_j_offset(halt_addr, target_addr):
    """Encode a J (relative jump) instruction."""
    raw_offset = (target_addr - halt_addr - 2) // 2

    # Check range: 12-bit signed = [-2048, 2047]
    if raw_offset < -2048 or raw_offset > 2047:
        return None

    offset_12 = raw_offset & 0xFFF

    # Check for JR/JALR decode conflicts:
    # JR:   bits[11:8] == 0xF AND bits[3:0] == 0x0
    # JALR: bits[3:0] == 0x1 AND bits[11:8] != 0xF
    rd_bits = (offset_12 >> 8) & 0xF
    func_bits = offset_12 & 0xF

    if rd_bits == 0xF and func_bits == 0x0:
        return None  # Would decode as JR
    if func_bits == 0x1 and rd_bits != 0xF:
        return None  # Would decode as JALR

    return 0x9000 | offset_12
```

Shared Opcode Hazard

The **J**, **JR**, and **JALR** instructions all share opcode `0x9`. The hardware distinguishes them by bit patterns within the instruction:

- **JR**: bits 11:8 = `0xF` and bits 3:0 = `0x0`
- **JALR**: bits 3:0 = `0x1` and bits 11:8 ≠ `0xF`
- **J**: everything else

The patch script's offset-scanning loop ensures the generated **J** instruction never accidentally matches a **JR** or **JALR** pattern.

If the offset to address 0 happens to conflict, the script tries placing the delay loop at word offsets 1, 2, 3, … until it finds one that produces a valid **J** encoding.

11.3.5 The Patching Process

The main patching logic:

Listing 11.14: Core patching logic

```
# Find HALT instruction
halt_idx = None
for i, w in enumerate(words):
    if w == HALT_WORD:
        halt_idx = i
        break

if halt_idx is None:
    print("WARNING: No HALT instruction found", file=sys.stderr)
    # Write unchanged
    ...
    return

halt_addr = halt_idx * 2  # byte address

# Find a valid loop placement
j_word = None
for loop_base in range(0, 64):
    target_addr = loop_base * 2
    j_word = encode_j_offset(halt_addr, target_addr)
    if j_word is not None:
        break

# Place the delay loop
for i, w in enumerate(LOOP_PATCH):
```

```
    words[loop_base + i] = w

# Replace HALT with J instruction
words[halt_idx] = j_word
```

The script looks for the **HALT** instruction encoded as `0xE100` (opcode `0xE`, func `0x1`). Once found, it converts the word index to a byte address, computes the **J** offset, places the delay loop, and patches the **HALT**.

11.3.6 Verification Output

The script prints a diagnostic message showing exactly what it changed:

Listing 11.15: Typical patch output

```
python3 hex_loop_patch.py programs/hello.hex programs/hello_loop.hex
Patched: HALT at 0x011E -> J 0x0000 (loop at word 0, 0x9F71)
```

This tells us:

- The **HALT** was at byte address `0x011E`.
- It was replaced with a **J** instruction targeting address `0x0000`.
- The delay loop starts at word 0.
- The encoded jump word is `0x9F71`.

11.3.7 Build Integration

The Makefile creates loop-patched hex files with a pattern rule:

Listing 11.16: Loop-patching Makefile rule

```
# Create looped hex file
programs/%_loop.hex: programs/%.hex
  python3 hex_loop_patch.py $< $@
```

The FPGA test targets depend on the loop-patched version:

Listing 11.17: Running a test on the FPGA

```
make fpga-test_alu
```

This chain of dependencies—`.s` → `.bin` → `.hex` → `_loop.hex` → synthesis → bitstream → upload—is fully automated.

11.4 The Address-Shift Bug: A Cautionary Tale

11.4.1 The Symptom

During early testing of the binary toolchain, programs that worked in simulation produced garbled output on the FPGA. Branch targets were off by one instruction. String data appeared shifted. The UART printed characters from the wrong addresses.

11.4.2 The Root Cause

The problem lay in the relationship between byte addresses and word addresses. The SAMPO CPU uses *byte* addressing: the program counter counts in bytes, and each instruction is two bytes wide. But the RAM module indexes memory in *words*: each entry in the 32K-entry memory array holds one 16-bit word.

The conversion happens in the RAM module's address line:

Listing 11.18: Byte-to-word address conversion in RAM

```
wire [14:0] word_addr = addr[15:1];
```

This drops the least significant bit, dividing the byte address by 2. Byte address `0x0100` becomes word address `0x0080` (word 128).

The original `bin2hex.py` script was emitting words addressed by their *position in the file*—which happened to be a word index—but the `@0000` address was being interpreted as byte address 0 by an early version of the RAM initialization code. The off-by-factor-of-two meant every address above `0x0000` was wrong.

11.4.3 The Fix

The fix was straightforward once diagnosed: ensure the hex file's `@` address uses *word* indices (matching the RAM array's native addressing), and ensure the RAM module consistently converts CPU byte addresses to word addresses by dropping bit 0.

The current `bin2hex.py` writes `@0000` to indicate word address 0. Since `$readmemh` interprets addresses as array indices, and the RAM array is indexed by word address, the mapping is correct: word 0 corresponds to byte addresses `0x0000`–`0x0001`, word 128 corresponds to byte addresses `0x0100`–`0x0101`, and so on.

Byte vs. Word Addressing

When working with the SAMPO toolchain, always be aware of which address space you are in:

- The **CPU** uses byte addresses. PC, label values, and branch targets are all in bytes.
- The **RAM array** uses word addresses. Index 0 is the first 16-bit word.
- The **hex file** uses word indices as `@` addresses.
- The **binary file** is a flat byte sequence.

Confusing these will produce bugs that are subtle in simulation and catastrophic on hardware.

11.4.4 Lessons Learned

This bug taught several lessons:

1. **Test the toolchain end-to-end early.** The assembler, the hex converter, and the RAM module each worked correctly in isolation. The bug only appeared when they were connected.
2. **Address spaces are interfaces.** Every boundary between byte addressing and word addressing is a potential bug. Document which space each tool operates in.
3. **Start with known-good programs.** The `examples/simple.s` program, with its hand-computed encodings in the comments, made it possible to compare expected bytes against actual memory contents. Without that ground truth, the bug would have been much harder to find.

11.5 Building a Disassembler

A disassembler reverses the assembler's work: it reads binary machine code and produces human-readable assembly. Building one is an excellent exercise because it forces you to internalize the instruction encoding from the hardware's perspective.

11.5.1 The Decoding Algorithm

A SAMPO disassembler needs to:

1. Read two bytes (one word) from the binary.
2. Extract the opcode from bits 15:12.
3. Depending on the opcode, extract register numbers, immediates, function codes, or offsets from the remaining bits.
4. If the opcode is `0xF` (extended), read two more bytes for the second word.
5. Format the result as an assembly instruction.

Here is a sketch of the core logic:

Listing 11.19: Disassembler core (Python sketch)

```
def disassemble_word(word, pc, next_word=None):
    opcode = (word >> 12) & 0xF
    rd = (word >> 8) & 0xF
    rs1 = (word >> 4) & 0xF
    rs2 = word & 0xF

    if opcode == 0x0:
        return f"ADD R{rd}, R{rs1}, R{rs2}"
    elif opcode == 0x1:
        return f"SUB R{rd}, R{rs1}, R{rs2}"
    elif opcode == 0x2:
        return f"AND R{rd}, R{rs1}, R{rs2}"
    elif opcode == 0x3:
        return f"OR R{rd}, R{rs1}, R{rs2}"
    elif opcode == 0x4:
        return f"XOR R{rd}, R{rs1}, R{rs2}"
    elif opcode == 0x5:
        imm8 = sign_extend_8(word & 0xFF)
        return f"ADDI R{rd}, {imm8}"
    elif opcode == 0x8:
        cond = rd
        offset = sign_extend_8(word & 0xFF)
        target = pc + 2 + (offset * 2)
        mnemonic = BRANCH_NAMES[cond]
        return f"{mnemonic} 0x{target:04X}"
    elif opcode == 0x9:
        # Check for JR/JALR patterns
        if rd == 0xF and rs2 == 0x0:
            return f"JR R{rs1}"
        elif rs2 == 0x1:
            return f"JALR R{rd}, R{rs1}"
        else:
            offset = sign_extend_12(word & 0xFFF)
            target = pc + 2 + (offset * 2)
            return f"J 0x{target:04X}"
    elif opcode == 0xF:
        sub = word & 0xF
        imm16 = next_word if next_word else 0
        return decode_extended(rd, rs1, sub, imm16)
    # ... remaining opcodes
```

11.5.2 Sign Extension Helpers

The disassembler needs sign extension functions for 8-bit and 12-bit values:

Listing 11.20: Sign extension for disassembly

```
def sign_extend_8(val):
    """Sign-extend an 8-bit value to Python int."""
    if val & 0x80:
        return val - 0x100
    return val

def sign_extend_12(val):
    """Sign-extend a 12-bit value to Python int."""
    if val & 0x800:
        return val - 0x1000
    return val
```

11.5.3 The Shared-Opcode Trap

The hardest part of writing a SAMPO disassembler is the shared opcode `0x9`. The same 4-bit opcode encodes three different instructions (**J**, **JR**, **JALR**), and the disassembler must check the same bit patterns that the hardware checks:

- If bits 11:8 = `0xF` and bits 3:0 = `0x0`: it is **JR**.
- If bits 3:0 = `0x1` and bits 11:8 $\neq$ `0xF`: it is **JALR**.
- Otherwise: it is a relative **J** with a 12-bit offset.

Getting this wrong produces disassembly that looks plausible but is incorrect—the same class of bug that the `hex_loop_patch.py` script guards against.

11.5.4 Extended Instruction Handling

When the disassembler sees opcode `0xF`, it must consume the *next* word as the 16-bit immediate. This means the disassembler's main loop cannot blindly step two bytes at a time—it must advance by two or four bytes depending on the opcode:

Listing 11.21: Variable-width instruction stepping

```
def disassemble_binary(data, base_addr=0):
    pc = base_addr
    i = 0
    while i < len(data) - 1:
        word = data[i] | (data[i+1] << 8)
```

```
        opcode = (word >> 12) & 0xF

        if opcode == 0xF and i + 3 < len(data):
            next_word = data[i+2] | (data[i+3] << 8)
            text = disassemble_word(word, pc, next_word)
            print(f"0x{pc:04X}: {word:04X} {next_word:04X}  {text}")
            i += 4
            pc += 4
        else:
            text = disassemble_word(word, pc)
            print(f"0x{pc:04X}: {word:04X}        {text}")
            i += 2
            pc += 2
```

11.5.5 Output Format

A well-formatted disassembly listing shows the address, the hex encoding, and the assembly text:

Listing 11.22: Sample disassembly output

```
0x0100: F407 011E  LIX R4, 0x011E
0x0104: 6542       LBU R5, (R4)
0x0106: C502       CMP R5, R0
0x0108: 8009       BEQ 0x011C
0x010A: F60B 0080  INI R6, 0x80
0x010E: 2766       AND R7, R6, R6
0x0110: 57FE       ADDI R7, -2
0x0112: 81FB       BNE 0x010A
0x0114: F05C 0081  OUTI 0x81, R5
0x0118: 5401       ADDI R4, 1
0x011A: 9FF4       J 0x0104
0x011C: E100       HALT
```

Disassembly as a Debugging Tool

When a program misbehaves on the FPGA, disassembling the hex file is one of the fastest ways to check that the assembler produced what you expected. Compare the disassembly against the original source, instruction by instruction. Mismatches reveal encoding bugs, off-by-one errors in branch offsets, or accidental label collisions.

11.5.6 Round-Trip Verification

The ultimate test of both the assembler and disassembler is a *round-trip*: assemble a program, disassemble the binary, reassemble the disassembly, and compare the two binaries byte for byte. If they match, both tools are consistent.

This is not the same as proving either tool is correct—they could be consistently wrong in the same way—but it is a powerful smoke test. Label names are lost during assembly (the binary does not contain them), so the disassembly will use absolute addresses instead of symbolic names. A round-trip test must account for this.

Exercises

1. **Hex file anatomy.** Assemble `examples/simple.s`, convert it with `bin2hex.py`, and open the resulting hex file in a text editor. Identify each instruction by matching the hex words against the expected encodings in the source comments. Where does the zero padding end and the code begin?

2. **Verify the patch.** Run `hex_loop_patch.py` on `programs/hello.hex` and compare the resulting `hello_loop.hex` against the original. Find the word that changed from **`HALT`** to a **`J`** instruction. Manually decode the **`J`** instruction to verify it targets address `0x0000`. Then examine the delay loop at the start of the file and decode each instruction.

3. **Build a disassembler.** Implement a disassembler in the language of your choice (Python recommended for speed of development). Start with opcodes `0x0`–`0x5` and `0xE`, then add branches and jumps. Test it against `examples/simple.s` by comparing your disassembly to the original source.

4. **Round-trip test.** Using your disassembler from the previous exercise, disassemble the `hello.bin` binary. Feed the disassembly back through the assembler (you will need to replace absolute branch targets with labels or use raw offsets). Compare the reassembled binary against the original. Document any differences and explain why they occur.

What's Next We have now traced the complete journey of a SAMPO program from source text through the assembler's lexer, parser, and code generator, through the binary-to-hex converter, and into the FPGA's memory. With the toolchain understood, we are ready to see what happens when those bytes come to life. In **??**, we will build an emulator that executes SAMPO machine code instruction by instruction, giving us a software model of the CPU that we can test, debug, and compare against the hardware implementation.

Part IV

The Emulator

Chapter 12

The CPU Emulation Loop

Every CPU—whether etched in silicon or described in Verilog—executes the same fundamental loop: fetch an instruction from memory, decode it, and execute it. When we build that loop in software instead of hardware, we gain something powerful: a controllable, inspectable reference implementation that can run SAMPO programs long before we have a working FPGA bitstream.

The SAMPO emulator, `semu`, is a Rust program that implements the entire SAMPO ISA in roughly a thousand lines of code. In this chapter we will walk through every piece of it: the data structures that model the CPU state, the fetch–decode–execute loop, ALU flag computation, memory and I/O handling, and cycle counting. By the end, you will be able to trace a complete program—`hello.s`—instruction by instruction through the emulator and predict exactly what it will do.

12.1 Fetch–Decode–Execute in Software

The hardware SAMPO CPU moves through a multi-cycle state machine: RESET, FETCH, FETCH_EXT, DECODE, EXECUTE, MEMORY, WRITEBACK, and HALTED (we will study this FSM in detail in **??**). An emulator does not need to model every hardware cycle. Instead, it collapses the entire pipeline into a single `step()` call:

Listing 12.1: The emulator's step function

```
pub fn step(&mut self) -> Result<bool, String> {
    if self.halted {
        return Ok(false);
    }

    // Fetch instruction
    let instr = self.fetch_word()?;
```

```
    if self.trace {
        self.trace_instruction(instr);
    }

    // Decode and execute
    self.execute(instr)?;

    self.cycles += 1;
    Ok(!self.halted)
}
```

Three things happen inside each step:

1. **Fetch.** Read a 16-bit word from `memory[pc]` and advance the program counter by two bytes.
2. **Decode and Execute.** Extract the 4-bit opcode from bits 15–12 of the instruction word, then dispatch to the appropriate handler.
3. **Bookkeeping.** Increment the cycle counter and return a boolean indicating whether the CPU is still running.

The return type `Result<bool, String>` lets us propagate errors—an out-of-bounds PC, an unrecognized opcode—without panicking. The outer `bool` tells the caller whether to keep stepping: `true` means the CPU is still alive, `false` means it has executed a **`HALT`**.

Why Not Model Every Pipeline Stage?

A cycle-accurate emulator would track each FSM state separately, which is valuable for verifying timing-sensitive behavior. Our emulator deliberately trades cycle accuracy for clarity and speed: one call to `step()` retires exactly one instruction. When you need cycle-level accuracy, use the Verilog testbench from Chapter 24 instead.

12.1.1 Fetching a Word

The SAMPO is little-endian: the low byte of each 16-bit word comes first in memory. The `fetch_word` method reads two bytes and assembles them:

Listing 12.2: Little-endian word fetch

```
fn fetch_word(&mut self) -> Result<u16, String> {
    if self.pc as usize + 1 >= MEM_SIZE {
        return Err("PC out of bounds".to_string());
    }
    let lo = self.memory[self.pc as usize];
```

```
    let hi = self.memory[self.pc as usize + 1];
    self.pc = self.pc.wrapping_add(2);
    Ok(u16::from_le_bytes([lo, hi]))
}
```

Note the use of `wrapping_add`. The SAMPO has a 16-bit address space, so the PC wraps around at `0xFFFF`—an unlikely scenario in practice, but one that the emulator handles gracefully rather than panicking.

Extended (32-bit) instructions use this same method twice: once for the opcode word and once for the 16-bit immediate that follows. We will see that in Section 12.3.3.

12.1.2 The Run Loop

The emulator provides two execution modes. In *batch* mode, the `run()` function calls `step()` in a tight loop until the CPU halts or an error occurs:

Listing 12.3: Batch execution loop in main.rs

```
fn run(cpu: &mut Cpu) {
    loop {
        match cpu.step() {
            Ok(true) => {}  // Continue
            Ok(false) => {
                println!("\nCPU halted at 0x{:04X}", cpu.get_pc());
                break;
            }
            Err(e) => {
                eprintln!("\nError at 0x{:04X}: {}", cpu.get_pc(), e);
                cpu.dump_state();
                std::process::exit(1);
            }
        }
    }
    cpu.dump_state();
}
```

In *interactive* mode, the emulator presents a command prompt where the user can single-step, inspect registers, or dump memory—a text-based precursor to the full TUI debugger we will build in Chapter 13.

Listing 12.4: Launching the emulator in different modes

```
# Batch mode (run to completion)
```

```
cargo run --release -- hello.bin

# Interactive debugger
cargo run --release -- hello.bin --interactive

# Trace mode (print every instruction)
cargo run --release -- hello.bin --trace

# TUI mode (graphical debugger)
cargo run --release -- hello.bin --tui
```

12.2 The CPU Struct

The Cpu struct in `semu/src/cpu.rs` captures the complete architectural state of the SAMPO processor:

Listing 12.5: The Cpu struct—complete SAMPO state

```
pub struct Cpu {
    // Registers
    regs: [u16; 16],
    regs_alt: [u16; 8],   // Alternate R4-R11
    pc: u16,
    flags: u8,

    // Memory
    memory: Vec<u8>,

    // I/O ports
    ports: [u8; 256],

    // State
    halted: bool,
    trace: bool,
    cycles: u64,
    quiet: bool,

    // Serial output buffer
    serial_out: Vec<u8>,
}
```

Let's examine each field group in turn.

12.2.1 Register File

The SAMPO has 16 general-purpose 16-bit registers stored in the `regs` array. Register zero is hardwired to zero, enforced in software by the accessor methods:

Listing 12.6: R0 is always zero

```
fn get_reg(&self, r: usize) -> u16 {
    if r == 0 { 0 } else { self.regs[r] }
}

fn set_reg(&mut self, r: usize, val: u16) {
    if r != 0 { self.regs[r] = val; }
}
```

Writes to `R0` are silently discarded—exactly as the hardware does. This convention lets the ISA use `R0` as a "discard" destination: `CMP R5, R0` compares `R5` against zero without needing a separate **`CMPI`** instruction.

The `regs_alt` array holds the alternate register bank for `R4`–`R11`. The **`EXX`** instruction swaps the two banks:

Listing 12.7: EXX swaps the alternate register bank

```
0xB => {
    // EXX - swap alternate registers
    for i in 0..8 {
        std::mem::swap(&mut self.regs[4 + i],
                       &mut self.regs_alt[i]);
    }
}
```

Why Only R4–R11?

Registers `R0`–`R3` have architectural roles (zero, return address, stack pointer, frame/global pointer) that must remain stable across bank switches. `R12`–`R15` are callee-saved registers that also stay put. Only the argument and temporary registers `R4`–`R11` participate in the exchange, following a design pattern inspired by the Z80's **`EXX`** instruction.

12.2.2 Flags Register

The flags register is an 8-bit value with six defined bits:

Listing 12.8: Flag bit constants

```
pub const FLAG_N: u8 = 0x80; // Negative   (bit 7)
pub const FLAG_Z: u8 = 0x40; // Zero       (bit 6)
pub const FLAG_C: u8 = 0x20; // Carry      (bit 5)
pub const FLAG_V: u8 = 0x10; // Overflow   (bit 4)
pub const FLAG_H: u8 = 0x08; // Half-carry (bit 3)
pub const FLAG_I: u8 = 0x04; // Interrupt  (bit 2)
```

Table 12.1: The SAMPO flags register layout

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
N	Z	C	V	H	I	—	—

The N, Z, C, and V flags drive conditional branches. The H (half-carry) flag supports the **DAA** instruction for BCD arithmetic. The I flag enables or disables interrupts via **EI** and **DI**.

12.2.3 Memory

The SAMPO addresses 64 KB of memory—the full range of a 16-bit address bus. In the emulator, this is a flat `Vec<u8>` of 65,536 bytes:

Listing 12.9: Memory constant and initialization

```
const MEM_SIZE: usize = 65536; // 64KB
// In Cpu::new():
memory: vec![0; MEM_SIZE],
```

Programs are loaded directly into this array by `load_program`, which also searches for the first non-zero word to determine the entry point:

Listing 12.10: Loading a binary into the emulator

```
pub fn load_program(&mut self, program: &[u8]) {
    for (i, &byte) in program.iter().enumerate() {
        if i < MEM_SIZE {
            self.memory[i] = byte;
        }
    }

    // Find first non-zero word as entry point
    for i in (0..program.len()).step_by(2) {
        if i + 1 < program.len() {
```

```
            let word = u16::from_le_bytes([program[i],
                                           program[i + 1]]);
            if word != 0 {
                self.pc = i as u16;
                break;
            }
        }
    }
}
```

Why Search for the Entry Point?

The assembler's `.org 0x0100` directive places code at offset `0x0100`, leaving the first 256 bytes as zero padding. Rather than hardcoding a start address, the emulator scans forward to the first non-zero instruction word. For most programs assembled with `.org 0x0100`, this finds `0x0100` automatically.

12.2.4 Initialization and Reset

The `Cpu::new()` constructor sets up the initial machine state, and `reset()` restores it:

Listing 12.11: CPU initialization

```
pub fn new() -> Self {
    let mut cpu = Cpu {
        regs: [0; 16],
        regs_alt: [0; 8],
        pc: 0x0100,         // Default start address
        flags: 0,
        memory: vec![0; MEM_SIZE],
        ports: [0; 256],
        halted: false,
        trace: false,
        cycles: 0,
        quiet: false,
        serial_out: Vec::new(),
    };

    // Initialize SP to top of RAM
    cpu.regs[2] = 0xFFFE;

    // Set serial TX ready bit
    cpu.ports[0x80] = 0x02;
```

```
    cpu
}
```

Two initialization details are worth noting. First, the stack pointer (R2/SP) starts at `0xFFFE`—two bytes below the top of the 64 KB address space, so the first **PUSH** writes to `0xFFFC`. Second, port `0x80` (the UART status register) is pre-loaded with `0x02` (TX ready), so the first serial write will succeed immediately without the program needing to wait.

12.3 Instruction Decoding: Opcode Dispatch

The `execute()` method is the heart of the emulator. It extracts the standard fields from the 16-bit instruction word, then dispatches on the 4-bit opcode:

Listing 12.12: Top-level instruction dispatch

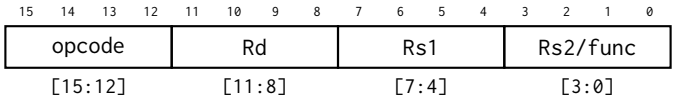
```
fn execute(&mut self, instr: u16) -> Result<(), String> {
    let opcode = (instr >> 12) & 0xF;
    let rd = ((instr >> 8) & 0xF) as usize;
    let rs1 = ((instr >> 4) & 0xF) as usize;
    let rs2 = (instr & 0xF) as usize;
    let imm8 = (instr & 0xFF) as i8 as i16 as u16;
    let func = instr & 0xF;

    match opcode {
        0x0 => { /* ADD  */ }
        0x1 => { /* SUB  */ }
        0x2 => { /* AND  */ }
        0x3 => { /* OR   */ }
        0x4 => { /* XOR  */ }
        0x5 => { /* ADDI */ }
        0x6 => { self.execute_load(rd, rs1, func)?; }
        0x7 => { self.execute_store(rd, rs1, func)?; }
        0x8 => { /* Branch */ }
        0x9 => { /* Jump   */ }
        0xA => { self.execute_shift(rd, rs1, func)?; }
        0xB => { self.execute_muldiv(rd, rs1, func)?; }
        0xC => { self.execute_misc(rd, rs1, func)?; }
        0xD => { self.execute_io(rd, rs1, func)?; }
        0xE => { self.execute_system(rd, imm8 as u8)?; }
        0xF => { /* Extended 32-bit */ }
        _ => { return Err(format!(
                  "Unknown opcode: 0x{:X}", opcode)); }
    }
    Ok(())
```

```
}
```

12.3.1 Field Extraction

Every SAMPO instruction packs its fields into the same 16-bit layout:



The extraction code pre-computes all possible field interpretations before the `match`. The `imm8` field deserves special attention: it is sign-extended from 8 bits to 16 bits through the chain `as i8 as i16 as u16`. This cast sequence first truncates to a signed byte, then sign-extends to a signed 16-bit integer, then reinterprets as unsigned—matching the hardware's sign-extension logic exactly.

Sign Extension in the Emulator

The cast chain `as i8 as i16 as u16` converts an 8-bit unsigned value to a sign-extended 16-bit value. For example, `0xFE` (decimal −2 as a signed byte) becomes `0xFFFE` (decimal −2 as a signed 16-bit value), reinterpreted as the unsigned value 65534. This is the software equivalent of the hardware's sign-extension wiring, where bit 7 is replicated across bits 8–15.

12.3.2 The Opcode Map

Opcodes `0x0`–`0x5` are the "fast path": common ALU operations that decode entirely from the top-level match. Opcodes `0x6`–`0xF` delegate to secondary dispatch methods that further decode the `func` field.

12.3.3 Extended Instructions

When the opcode is `0xF`, the instruction is 32 bits wide. The emulator fetches a second word for the 16-bit immediate:

Listing 12.13: Extended instruction handling

```
0xF => {
    // Extended (32-bit) instructions
    let imm16 = self.fetch_word()?;
    self.execute_extended(rd, rs1, func, imm16)?;
}
```

Table 12.2: Primary opcode map

Opcode	Category	Instruction(s)
`0x0`	ALU	**ADD** Rd, Rs1, Rs2
`0x1`	ALU	**SUB** Rd, Rs1, Rs2
`0x2`	ALU	**AND** Rd, Rs1, Rs2
`0x3`	ALU	**OR** Rd, Rs1, Rs2
`0x4`	ALU	**XOR** Rd, Rs1, Rs2
`0x5`	Immediate	**ADDI** Rd, imm8
`0x6`	Memory	Load variants (func field)
`0x7`	Memory	Store variants (func field)
`0x8`	Branch	Conditional branches (16 conditions)
`0x9`	Jump	**J**, **JR**, **JALR**
`0xA`	Shift	Shift/rotate variants (func field)
`0xB`	Multiply	**MUL**, **DIV**, **DAA**
`0xC`	Misc	**PUSH**, **POP**, **CMP**, **MOV**, etc.
`0xD`	I/O	**IN**, **OUT**, **INI**, **OUTI**
`0xE`	System	**NOP**, **HALT**, **EI**, **DI**, etc.
`0xF`	Extended	32-bit instructions with imm16

The `execute_extended` method then dispatches on the `func` field to handle operations like **LIX** (load 16-bit immediate), **JX** (jump to absolute address), **ADDIX** (add with 16-bit immediate), and many others:

Listing 12.14: Selected extended instruction handlers

```
fn execute_extended(&mut self, rd: usize, rs1: usize,
                    sub: u16, imm16: u16)
                    -> Result<(), String> {
    match sub {
        0x7 => {
            // LIX Rd, imm16
            self.set_reg(rd, imm16);
        }
        0x8 => {
            // JX addr16
            self.pc = imm16;
        }
        0x9 => {
            // JALX addr16
            self.set_reg(rd, self.pc);
```

```
            self.pc = imm16;
        }
        // ... 13 more sub-opcodes
        _ => return Err(format!(
               "Unknown extended sub: 0x{:X}", sub)),
    }
    Ok(())
}
```

12.4 ALU Operations with Flag Computation

The core ALU operations—**ADD**, **SUB**, **AND**, **OR**, **XOR**—are three-operand: they read two source registers and write the result to a destination. Let's trace through the **ADD** handler:

Listing 12.15: ADD with flag generation

```
0x0 => {
    // ADD Rd, Rs1, Rs2
    let a = self.get_reg(rs1);
    let b = self.get_reg(rs2);
    let (result, carry) = a.overflowing_add(b);
    self.set_reg(rd, result);
    self.set_flags_add(a, b, result, carry);
}
```

Rust's `overflowing_add` returns both the truncated result and a boolean indicating whether an unsigned overflow (carry out of bit 15) occurred. This maps directly to the hardware ALU's carry output.

12.4.1 Addition Flags

The `set_flags_add` method computes all four arithmetic flags:

Listing 12.16: Flag computation for addition

```
fn set_flags_add(&mut self, a: u16, b: u16,
                 result: u16, carry: bool) {
    self.flags = 0;
    if result == 0 {
        self.flags |= FLAG_Z;
    }
    if (result & 0x8000) != 0 {
        self.flags |= FLAG_N;
```

```
    }
    if carry {
        self.flags |= FLAG_C;
    }
    // Overflow: sign of result differs from both operands
    let a_neg = (a & 0x8000) != 0;
    let b_neg = (b & 0x8000) != 0;
    let r_neg = (result & 0x8000) != 0;
    if a_neg == b_neg && a_neg != r_neg {
        self.flags |= FLAG_V;
    }
}
```

The flag semantics:

- **Z (Zero):** Set when the result is exactly zero.
- **N (Negative):** Copies bit 15 of the result—the sign bit in two's complement.
- **C (Carry):** Set when the unsigned addition overflows past 16 bits.
- **V (Overflow):** Set when a signed overflow occurs. This happens when both operands have the same sign but the result has a different sign—for example, adding two large positive numbers to get a negative result.

Flags Are Replaced, Not Merged

Notice that `set_flags_add` starts by clearing *all* flags to zero. This matches the hardware behavior: every arithmetic instruction replaces the entire flags register. The only flags that survive across instructions are the I and H flags, which are preserved by the logic flag setter (discussed next).

12.4.2 Subtraction Flags and ARM-Style Carry

Subtraction flags follow the same pattern, with one critical difference in the carry flag:

Listing 12.17: Flag computation for subtraction

```
fn set_flags_sub(&mut self, a: u16, b: u16,
                 result: u16, borrow: bool) {
    self.flags = 0;
    if result == 0 {
        self.flags |= FLAG_Z;
    }
    if (result & 0x8000) != 0 {
```

```
        self.flags |= FLAG_N;
    }
    if !borrow {
        self.flags |= FLAG_C; // Carry set if NO borrow
    }
    // Overflow for subtraction
    let a_neg = (a & 0x8000) != 0;
    let b_neg = (b & 0x8000) != 0;
    let r_neg = (result & 0x8000) != 0;
    if a_neg != b_neg && b_neg == r_neg {
        self.flags |= FLAG_V;
    }
}
```

The SAMPO uses *ARM-style carry* for subtraction: the C flag is set when there is *no* borrow, and cleared when there is a borrow. This is the opposite of what you might expect if you're used to x86 or Z80, where the carry flag directly represents the borrow.

ARM-Style Subtraction Carry

In ARM-style carry convention, `SUB a, b` computes $a + \overline{b} + 1$ (the two's complement subtraction). The carry output of this addition is 1 when $a \geq b$ (unsigned), meaning no borrow occurred. The carry is 0 when $a < b$ (unsigned), meaning a borrow was needed. This convention simplifies chained multi-word subtraction using **SBC**.

The overflow detection for subtraction also differs subtly: overflow occurs when the operands have *different* signs and the result's sign matches the subtrahend. For example, subtracting a large negative number from a positive number can produce a negative result—a signed overflow.

12.4.3 Logic Flags

Logical operations (**AND**, **OR**, **XOR**) and shift instructions use a lighter-weight flag setter that only updates N and Z, clearing C and V:

Listing 12.18: Flag computation for logical operations

```
fn set_flags_logic(&mut self, result: u16) {
    self.flags &= !(FLAG_Z | FLAG_N | FLAG_C | FLAG_V);
    if result == 0 {
        self.flags |= FLAG_Z;
    }
    if (result & 0x8000) != 0 {
        self.flags |= FLAG_N;
```

```
    }
}
```

Unlike the arithmetic flag setters, this method uses a mask (`&= !(...)`) instead of clearing all flags. This preserves the H and I flags across logical operations—important because the interrupt enable state should not change due to an **AND** instruction.

12.4.4 The ADDI Quirk

The **ADDI** instruction is notable because it is *two-operand*: the destination register is also the first source. The instruction adds a sign-extended 8-bit immediate to the register's current value:

Listing 12.19: ADDI is two-operand: Rd = Rd + sign_ext(imm8)

```
0x5 => {
    // ADDI Rd, imm8
    let a = self.get_reg(rd);
    let b = imm8;
    let (result, carry) = a.overflowing_add(b);
    self.set_reg(rd, result);
    self.set_flags_add(a, b, result, carry);
}
```

This design choice gives **ADDI** an 8-bit immediate range (-128 to $+127$) while keeping the instruction at 16 bits. A three-operand encoding would only leave 4 bits for the immediate—too narrow to be useful.

12.5 Memory Operations

The SAMPO provides word and byte loads and stores, with several addressing modes encoded in the `func` field.

12.5.1 Load Instructions

The `execute_load` method handles opcode `0x6`:

Listing 12.20: Load instruction dispatch

```
fn execute_load(&mut self, rd: usize, rs1: usize,
                func: u16) -> Result<(), String> {
    let base = self.get_reg(rs1);
    let offset: i16 = match func {
        0x0 => 0,           // LW
```

```
        0x1 => 0,           // LB (signed)
        0x2 => 0,           // LBU (unsigned)
        0x3 => 2,           // LW +2
        0x4 => 4,           // LW +4
        0x5 => 6,           // LW +6
        0x6 => -2,          // LW -2
        0x7 => -4,          // LW -4
        0x8 => {            // LUI
            let val = (rs1 as u16) << 8;
            self.set_reg(rd, val);
            return Ok(());
        }
        _ => return Err(format!(
               "Unknown load func: 0x{:X}", func)),
    };

    let addr = (base as i16).wrapping_add(offset) as u16;

    match func {
        0x1 => {
            // LB (sign extend)
            let val = self.read_byte(addr)? as i8
                      as i16 as u16;
            self.set_reg(rd, val);
        }
        0x2 => {
            // LBU (zero extend)
            let val = self.read_byte(addr)? as u16;
            self.set_reg(rd, val);
        }
        _ => {
            // LW
            let val = self.read_word(addr)?;
            self.set_reg(rd, val);
        }
    }
    Ok(())
}
```

The built-in offsets (+2, +4, +6, −2, −4) enable efficient access to structure fields and stack frames without requiring a separate **ADDI** to compute the address. The extended instruction **LWX** (opcode 0xF, sub 0x5) provides an arbitrary 16-bit offset for cases where the built-in offsets are insufficient.

LB vs. LBU

LB sign-extends the loaded byte: value `0xFF` becomes `0xFFFF` (−1). **LBU** zero-extends: value `0xFF` becomes `0x00FF` (255). Choose **LBU** for ASCII characters and unsigned byte data, **LB** for signed byte values.

12.5.2 Store Instructions

Stores mirror loads, writing register values back to memory:

Listing 12.21: Store instruction dispatch

```
fn execute_store(&mut self, rs2: usize, rs1: usize,
                 func: u16) -> Result<(), String> {
    let base = self.get_reg(rs1);
    let value = self.get_reg(rs2);

    let offset: i16 = match func {
        0x0 => 0,       // SW
        0x1 => 0,       // SB
        0x2 => 2,
        0x3 => 4,
        0x4 => 6,
        0x5 => -2,
        0x6 => -4,
        _ => return Err(format!(
               "Unknown store func: 0x{:X}", func)),
    };

    let addr = (base as i16).wrapping_add(offset) as u16;

    if func == 0x1 {
        self.write_byte(addr, value as u8)?;
    } else {
        self.write_word(addr, value)?;
    }
    Ok(())
}
```

12.5.3 Byte-Level Memory Access

The underlying memory access methods enforce little-endian byte ordering:

Listing 12.22: Word-level memory access

```
fn read_word(&self, addr: u16) -> Result<u16, String> {
    let lo = self.memory[addr as usize];
    let hi = self.memory[addr.wrapping_add(1) as usize];
    Ok(u16::from_le_bytes([lo, hi]))
}

fn write_word(&mut self, addr: u16, val: u16)
              -> Result<(), String> {
    let bytes = val.to_le_bytes();
    self.memory[addr as usize] = bytes[0];
    self.memory[addr.wrapping_add(1) as usize] = bytes[1];
    Ok(())
}
```

Note that the emulator does *not* enforce word alignment. An unaligned word read at an odd address will work in the emulator but may behave differently in hardware, depending on the memory controller implementation.

12.5.4 Block Transfer Instructions

The SAMPO includes Z80-inspired block transfer instructions that move or fill memory regions efficiently. These use fixed register conventions: `R4` holds the byte count, `R5` the source address, and `R6` the destination.

Listing 12.23: LDIR—repeating block copy

```
0x7 => {
    // LDIR - block load repeat increment
    loop {
        let src = self.get_reg(5);
        let dst = self.get_reg(6);
        let count = self.get_reg(4);

        if count == 0 { break; }

        let byte = self.read_byte(src)?;
        self.write_byte(dst, byte)?;

        self.set_reg(5, src.wrapping_add(1));
        self.set_reg(6, dst.wrapping_add(1));
        self.set_reg(4, count.wrapping_sub(1));
    }
    self.flags |= FLAG_Z;
}
```

In the emulator, **LDIR** completes in a single `step()` call—the entire loop runs atomically. In hardware, each byte transfer would be a separate memory cycle. This is one of the deliberate simplifications we make for emulation speed.

12.6 Branch and Jump Logic

12.6.1 Conditional Branches

Branch instructions (opcode `0x8`) use the Rd field as a 4-bit condition code and the low byte as a signed offset:

Listing 12.24: Branch execution

```
0x8 => {
    let cond = rd as u16;
    let offset = (instr & 0xFF) as i8 as i16;
    if self.check_condition(cond) {
        self.pc = (self.pc as i16)
                  .wrapping_add(offset * 2) as u16;
    }
}
```

The offset is multiplied by 2 because instructions are word-aligned. This gives branches a range of -256 to $+254$ bytes from the instruction following the branch.

The `check_condition` method implements all 16 branch conditions:

Listing 12.25: Condition code evaluation (selected)

```
fn check_condition(&self, cond: u16) -> bool {
    match cond {
        0x0 => (self.flags & FLAG_Z) != 0,  // BEQ
        0x1 => (self.flags & FLAG_Z) == 0,  // BNE
        0x2 => {                              // BLT (signed)
            let n = (self.flags & FLAG_N) != 0;
            let v = (self.flags & FLAG_V) != 0;
            n != v
        }
        0x3 => {                              // BGE (signed)
            let n = (self.flags & FLAG_N) != 0;
            let v = (self.flags & FLAG_V) != 0;
            n == v
```

```
        }
        0x4 => (self.flags & FLAG_C) == 0,  // BLTU
        0x5 => (self.flags & FLAG_C) != 0,  // BGEU
        // ...
        _ => false,
    }
}
```

Unsigned Comparisons and ARM-Style Carry

Because the SAMPO uses ARM-style carry (C=1 means no borrow), the unsigned comparison conditions are "inverted" compared to x86. **BLTU** (branch if less than, unsigned) checks `C == 0`—the carry is clear when a borrow occurred, meaning $a < b$. **BGEU** checks `C == 1`. If you're porting code from an x86-style ISA, watch out for this difference.

12.6.2 Jump Instructions

Opcode `0x9` handles three jump variants, distinguished by bit patterns within the instruction word:

Listing 12.26: Jump instruction decoding

```
0x9 => {
    if (instr & 0x0F0F) == 0x0F00 {
        // JR Rs (register indirect)
        let rs = rs1;
        self.pc = self.get_reg(rs);
    } else if func == 0x1 && rd != 0 {
        // JALR Rd, Rs (jump and link register)
        let ret_addr = self.pc;
        self.pc = self.get_reg(rs1);
        self.set_reg(rd, ret_addr);
    } else {
        // J offset12 (PC-relative)
        let offset = (instr & 0x0FFF) as i16;
        let offset = if offset & 0x800 != 0 {
            offset | 0xF000u16 as i16
        } else { offset };
        self.pc = (self.pc as i16)
                  .wrapping_add(offset * 2) as u16;
    }
}
```

The **JR** encoding uses a sentinel pattern: the magic value `0x0F0R` where R is the source register. The **JALR** encoding stores the return address in Rd before jumping, enabling subroutine calls. The plain **J** provides a 12-bit signed PC-relative offset for unconditional local jumps.

12.7 I/O Simulation

The SAMPO uses port-mapped I/O with a 256-port address space. The emulator stores port values in a flat array and handles reads and writes through two methods:

Listing 12.27: I/O instruction dispatch

```
fn execute_io(&mut self, rd: usize, rs1: usize,
              func: u16) -> Result<(), String> {
    match func {
        0x0 => {
            // INI Rd, port (immediate port address)
            let port = rs1 as u8;
            let val = self.port_read(port);
            self.set_reg(rd, val as u16);
        }
        0x1 => {
            // OUTI port, Rd
            let port = rs1 as u8;
            let val = self.get_reg(rd) as u8;
            self.port_write(port, val);
        }
        0x2 => {
            // IN Rd, (Rs1)  -- register-indirect port
            let port = self.get_reg(rs1) as u8;
            let val = self.port_read(port);
            self.set_reg(rd, val as u16);
        }
        0x3 => {
            // OUT (Rd), Rs1 -- register-indirect port
            let port = self.get_reg(rd) as u8;
            let val = self.get_reg(rs1) as u8;
            self.port_write(port, val);
        }
        _ => return Err(format!(
               "Unknown I/O func: 0x{:X}", func)),
    }
    Ok(())
}
```

The **INI** and **OUTI** variants encode a 4-bit port number directly in the instruction, giving fast access to the first 16 ports. The register-indirect **IN** and **OUT** variants can reach all 256 ports.

12.7.1 The UART Ports

The emulator simulates a UART peripheral at ports `0x80` (status) and `0x81` (data), modeled after the MC6850 ACIA:

Listing 12.28: UART port simulation

```
fn port_read(&mut self, port: u8) -> u8 {
    match port {
        0x80 => {
            // ACIA status - always TX ready
            0x02
        }
        0x81 => {
            // ACIA data - nothing to read
            0
        }
        _ => self.ports[port as usize],
    }
}

fn port_write(&mut self, port: u8, val: u8) {
    match port {
        0x81 => {
            // ACIA data - output character
            self.serial_out.push(val);
            if !self.quiet {
                print!("{}", val as char);
                io::stdout().flush().ok();
            }
        }
        _ => {
            self.ports[port as usize] = val;
        }
    }
}
```

Port `0x80` always returns `0x02` (bit 1 set), indicating that the transmitter is ready. In hardware, the program would need to poll this status bit and wait; in the emulator, writes are instantaneous, so we always report ready.

Writing to port `0x81` captures the byte in the `serial_out` buffer (for the TUI debugger) and optionally prints it to stdout. The `quiet` flag suppresses direct output when the TUI is active—we don't want raw characters appearing behind the ratatui interface.

Keyboard Input Simulation

The emulator also supports serial input through the `send_key` method, which places a byte in the data port and sets the RX-ready bit in the status register:

```
pub fn send_key(&mut self, key: u8) {
    self.ports[0x81] = key;
    self.ports[0x80] |= 0x01; // RX ready
}
```

This enables interactive programs in the TUI debugger to receive keyboard input as if it arrived over a serial line.

12.7.2 Cycle Counting

The emulator maintains a 64-bit cycle counter that increments by one for each instruction executed:

Listing 12.29: Cycle counting in step()

```
self.cycles += 1;
```

This is a simplification—in hardware, different instructions take different numbers of clock cycles (a **LDIR** block transfer, for instance, takes many more cycles than a register **ADD**). The emulator's single-increment approach is sufficient for performance profiling at the instruction level, but should not be used for precise timing calculations. The TUI debugger uses the cycle counter to compute an "effective MHz" metric, giving a rough sense of emulation throughput.

12.8 Shift and Miscellaneous Operations

The SAMPO provides a rich set of shift and rotate operations, all dispatched through opcode `0xA` with the `func` field selecting the specific operation:

Listing 12.30: Shift dispatch (selected operations)

```
fn execute_shift(&mut self, rd: usize, rs1: usize,
                 func: u16) -> Result<(), String> {
    let val = self.get_reg(rs1);
    let result = match func {
        0x0 => val << 1,                        // SLL 1
```

```
        0x1 => val >> 1,                          // SRL 1
        0x2 => ((val as i16) >> 1) as u16,        // SRA 1
        0x3 => val.rotate_left(1),                // ROL 1
        0x4 => val.rotate_right(1),               // ROR 1
        0x7 => ((val & 0xFF) << 8)
             | ((val >> 8) & 0xFF),               // SWAP
        0x8 => val << 4,                          // SLL 4
        0x9 => val >> 4,                          // SRL 4
        0xA => ((val as i16) >> 4) as u16,        // SRA 4
        0xC => val << 8,                          // SLL 8
        0xD => val >> 8,                          // SRL 8
        // ... more variants
        _ => return Err(format!(
               "Unknown shift func: 0x{:X}", func)),
    };
    self.set_reg(rd, result);
    self.set_flags_logic(result);
    Ok(())
}
```

The shift amounts are fixed at compile time (1, 4, or 8 positions), not taken from a register. This mirrors the hardware barrel shifter design from Chapter 20, where fixed shifts are single-cycle combinational paths. For variable-amount shifts, the extended instructions **SLLX**, **SRLX**, and **SRAX** accept an immediate shift count.

The **SWAP** instruction (func `0x7`) exchanges the high and low bytes of a register, which is invaluable for byte-order manipulation on a little-endian machine.

12.8.1 Multiply and Divide

Multiply and divide operations (opcode `0xB`) are two-operand: the destination register is also the first source operand. This matches the hardware, where a 16×16 multiply is multi-cycle and uses a single accumulator:

Listing 12.31: MUL and DIV handlers

```
fn execute_muldiv(&mut self, rd: usize, rs1: usize,
                  func: u16) -> Result<(), String> {
    let a = self.get_reg(rd);
    let b = self.get_reg(rs1);

    match func {
        0x0 => {
            // MUL (low 16 bits)
```

```
            let result = (a as u32).wrapping_mul(b as u32);
            self.set_reg(rd, result as u16);
        }
        0x1 => {
            // MULH (high 16 bits, signed)
            let result = (a as i16 as i32)
                        .wrapping_mul(b as i16 as i32);
            self.set_reg(rd, (result >> 16) as u16);
        }
        0x3 => {
            // DIV (signed)
            if b == 0 {
                self.set_reg(rd, 0xFFFF);
            } else {
                let result = (a as i16)
                            .wrapping_div(b as i16);
                self.set_reg(rd, result as u16);
            }
        }
        // ... MULHU, DIVU, REM, REMU, DAA
        _ => return Err(format!(
               "Unknown muldiv func: 0x{:X}", func)),
    }
    Ok(())
}
```

Note the division-by-zero behavior: rather than causing an error, dividing by zero yields `0xFFFF`. The **REM** variants return the dividend unchanged when dividing by zero. These are defined behaviors in the SAMPO ISA, matching what the hardware division circuit produces.

12.8.2 System Instructions

Opcode `0xE` handles system-level operations. The `func` field (encoded in the Rd position) selects the operation:

Listing 12.32: System instruction handling

```
fn execute_system(&mut self, func: usize, imm: u8)
                  -> Result<(), String> {
    match func {
        0x0 => {}           // NOP
        0x1 => {
            self.halted = true;  // HALT
        }
```

```
            0x2 => {
                self.flags &= !FLAG_I;  // DI
            }
            0x3 => {
                self.flags |= FLAG_I;   // EI
            }
            0x4 => {
                // RETI - return from interrupt
                let sp = self.get_reg(2);
                let pc = self.read_word(sp)?;
                self.set_reg(2, sp.wrapping_add(2));
                self.pc = pc;
                self.flags |= FLAG_I;
            }
            0x5 => {
                // SWI imm - software interrupt
                let sp = self.get_reg(2).wrapping_sub(2);
                self.set_reg(2, sp);
                self.write_word(sp, self.pc)?;
                self.pc = (imm as u16) * 2;
            }
            _ => return Err(format!(
                    "Unknown system func: 0x{:X}", func)),
        }
        Ok(())
    }
```

The **SWI** (software interrupt) instruction pushes the return address onto the stack and jumps to a vector table entry, enabling system calls in a future operating system. The **RETI** instruction reverses this process and re-enables interrupts.

12.9 Hands-on: Trace Execution of hello.s

Let's put everything together by tracing the emulator through the classic "Hello, Sampo!" program. Here is the source:

Listing 12.33: hello.s—our test program

```
        .org 0x0100

.equ    ACIA_STATUS 0x80
.equ    ACIA_DATA   0x81
.equ    TX_READY    0x02
```

```
start:
        LIX  R4, message        ; Load address of message

loop:
        LBU  R5, (R4)           ; Load byte from string
        CMP  R5, R0             ; Compare with zero
        BEQ  done               ; If null, we're done

wait_tx:
        INI  R6, ACIA_STATUS    ; Read serial status
        AND  R7, R6, R6         ; Copy to R7
        ADDI R7, -2             ; Subtract TX_READY (2)
        BNE  wait_tx            ; Wait if not ready

        OUTI ACIA_DATA, R5      ; Write character
        ADDI R4, 1              ; Next character
        J    loop               ; Continue

done:
        HALT

message:
        .asciz "Hello, Sampo!\n"
```

12.9.1 Running with Trace Output

First, assemble and run with tracing enabled:

Listing 12.34: Assembling and running hello.s with trace

```
cd sasm && cargo run --release -- ../examples/hello.s -o hello.bin
cd ../semu && cargo run --release -- ../sasm/hello.bin --trace
```

The trace output shows each instruction as it executes:

Listing 12.35: Trace output for the first few instructions

```
Sampo Emulator - Loaded 288 bytes
Starting execution at 0x0100

0100: F700  EXTENDED R7, R0, sub=7
0104: 6540  LOAD R5, (R4) func=2
0106: C502  MISC R5, R0, func=2
0108: 800C  BRANCH cond=0, offset=12
```

```
010A: D680  I/O R6, port=8, func=0
010C: 2766  AND R7, R6, R6
010E: 57FE  ADDI R7, -2
0110: 81FC  BRANCH cond=1, offset=-4
0112: D518  I/O R5, port=1, func=1
H0114: 5401  ADDI R4, 1
```

12.9.2 Stepping Through the First Iteration

Let's trace the first character output manually, following the emulator's logic:

1. **LIX R4, message** (at `0x0100`): This is a 32-bit extended instruction (opcode `0xF`, sub `0x7`). The emulator fetches two words: the opcode word and the 16-bit address of the `message` label. After execution, `R4` holds the address of the string "Hello, Sampo!\n".
2. **LBU R5, (R4)** (at `0x0104`): Opcode `0x6`, func `0x2`. The emulator reads one byte from the address in `R4` and zero-extends it to 16 bits. The first character is `'H'` (`0x48`), so `R5` = `0x0048`.
3. **CMP R5, R0** (at `0x0106`): Opcode `0xC`, func `0x2`. Computes $R5 - R0 = 0x48 - 0 = 0x48$. The Z flag is clear (result is non-zero), C flag is set (no borrow, since $0x48 \geq 0$).
4. **BEQ done** (at `0x0108`): Opcode `0x8`, condition 0 (BEQ). The Z flag is clear, so the branch is *not* taken. Execution falls through to the TX-ready poll.
5. **INI R6, ACIA_STATUS** (at `0x010A`): Opcode `0xD`, func `0x0`. Reads port `0x80`, which always returns `0x02` (TX ready). `R6` = `0x0002`.
6. **AND R7, R6, R6** (at `0x010C`): Copies `R6` to `R7`. `R7` = `0x0002`.
7. **ADDI R7, -2** (at `0x010E`): Subtracts 2 from `R7`. $0x0002 + 0xFFFE = 0x0000$. The Z flag is now set.
8. **BNE wait_tx** (at `0x0110`): Condition 1 (BNE) checks Z=0. Since Z is set, the branch is not taken—the transmitter is ready.
9. **OUTI ACIA_DATA, R5** (at `0x0112`): Opcode `0xD`, func `0x1`. Writes the value in `R5` (`0x48`, the character 'H') to port `0x81`. The emulator's `port_write` method pushes this byte to `serial_out` and prints 'H' to stdout.
10. **ADDI R4, 1** (at `0x0114`): Advances the string pointer. `R4` now points to 'e'.
11. **J loop** (at `0x0116`): Jumps back to the **`LBU`** instruction to process the next character.

This loop repeats for each character in "Hello, Sampo!\n" until the null terminator is reached. At that point, **`CMP`** `R5, R0` sets the Z flag, **`BEQ`** is taken, and execution reaches **`HALT`**.

12.9.3 Final State Dump

After halting, the emulator prints the final CPU state:

Listing 12.36: Expected CPU state after hello.s completes

```
=== CPU State ===
PC: 011A  Flags: 60 [NZ--]

Registers:
  R 0=0000  R 1=0000  R 2=FFFE  R 3=0000
  R 4=0126  R 5=0000  R 6=0002  R 7=0000
  R 8=0000  R 9=0000  R10=0000  R11=0000
  R12=0000  R13=0000  R14=0000  R15=0000

Cycles: 231

Serial output:
  "Hello, Sampo!\n"
```

The key things to verify: `R4` points one past the null terminator, `R5` is zero (the null byte that ended the loop), the Z flag is set (from the final **`CMP`**), and the serial output buffer contains the complete message.

Using the Emulator as a Reference

When debugging your Verilog implementation, run the same binary through `semu` with `--trace` and compare the instruction-by-instruction output against your waveform. Any divergence points directly to the bug—either in the emulator or in the hardware. This comparative debugging technique saved countless hours during SAMPO development.

Exercises

1. **Add an ADC Instruction.** The SAMPO ISA includes an **`ADC`** (add with carry) instruction that adds two registers plus the current carry flag: $\text{Rd} = \text{Rs1} + \text{Rs2} + C$. Add a handler for this instruction to the emulator. Choose an unused opcode or func slot, implement the operation, and write a test program that chains two 16-bit additions into a 32-bit addition using **`ADD`** followed by **`ADC`**.

2. **Alignment Checking.** Modify the `read_word` and `write_word` methods to detect unaligned accesses (odd addresses) and print a warning. Run the test suite and observe whether any programs trigger the warning. Consider whether unaligned access should be a hard error or a performance warning.

3. **Instruction Histogram.** Add a 16-element array to the `Cpu` struct that counts how many times each primary opcode is executed. After the program halts, print the histogram. Run several test programs and compare their opcode distributions. Which opcodes dominate in typical SAMPO code?

4. **Memory-Mapped I/O.** Instead of using **`IN`**/**`OUT`** for peripheral access, some architectures map peripherals into the memory address space. Modify the emulator to intercept reads and writes to addresses `0xFF00`–`0xFFFF` and route them through the port I/O handlers. Write a test program that accesses the UART through memory-mapped addresses instead of port instructions.

5. **Breakpoint Support.** Add a `breakpoints: Vec<u16>` field to the `Cpu` struct. In the `step()` method, check whether the current PC matches any breakpoint address. If it does, return a special value that causes the run loop to pause. Test your implementation in the interactive debugger mode.

What's Next. We now have a fully functional SAMPO emulator that can run any program the assembler produces. But staring at trace output scrolling past on a terminal is hardly the best debugging experience. In the next chapter, we will build a rich terminal user interface around the emulator—with live register displays, a scrollable memory viewer, an integrated disassembler, and a virtual terminal—turning `semu` from a command-line tool into a proper development environment.

Chapter 13

The TUI Debugger

Trace output tells you *what* the CPU did, but not *why* it did it. When a branch goes the wrong way or a register holds an unexpected value, you need to see the entire machine state at once: registers, flags, memory, the instruction stream, and the program's output, all updating in real time as you step through instructions. That is exactly what the `semu` TUI debugger provides.

Built on the ratatui framework, the TUI transforms the terminal into a multi-panel development environment: registers in the top left, a hex memory dump below, disassembly and stack on the right, a VT220 terminal emulator at the bottom, and a status bar showing execution speed and controls. In this chapter we will walk through every panel, the keyboard control system, and the integration between the debugger and the emulation core from Chapter 12.

13.1 The Ratatui TUI Framework

The `semu` TUI is built with three crates, declared in `semu/Cargo.toml`:

Listing 13.1: TUI dependencies in Cargo.toml

```
[dependencies]
ratatui = "0.29"
crossterm = "0.28"
sysinfo = "0.32"
```

Each crate plays a distinct role:

- **ratatui** provides the widget system: layouts, styled text, borders, and the rendering pipeline. It operates on an *immediate-mode* principle—every frame, the entire UI is redrawn from scratch.

There is no persistent widget tree to update; you simply describe what the screen should look like right now.

- **crossterm** handles the platform-specific terminal operations: entering and leaving the alternate screen buffer, enabling raw mode (so keystrokes are delivered without waiting for Enter), and reading input events.
- **sysinfo** provides host system metrics (CPU usage, memory consumption) displayed in the status bar.

Immediate-Mode Rendering

In an immediate-mode UI, the application redraws the entire interface every frame. The framework computes the minimal set of terminal escape sequences needed to transition from the previous frame to the current one, so only changed cells are actually written to the terminal. This model is conceptually the same as a hardware frame buffer: you write pixels (or characters) every refresh cycle, and the display controller handles the output.

13.1.1 Terminal Setup and Teardown

The `run_tui` function in `semu/src/tui.rs` manages the terminal lifecycle:

Listing 13.2: Terminal initialization and cleanup

```
pub fn run_tui(cpu: &mut Cpu) -> io::Result<()> {
    // Suppress direct stdout output in TUI mode
    cpu.set_quiet(true);

    // Setup terminal
    enable_raw_mode()?;
    stdout().execute(EnterAlternateScreen)?;
    let backend = CrosstermBackend::new(stdout());
    let mut terminal = Terminal::new(backend)?;

    let start_pc = cpu.get_pc();
    let mut app = App::new(start_pc);
    let tick_rate = Duration::from_millis(TICK_RATE_MS);

    loop {
        terminal.draw(|f| ui(f, &app, cpu))?;

        if event::poll(tick_rate)? {
            if let Event::Key(key) = event::read()? {
                match handle_key(&mut app, cpu, key) {
                    Ok((false, _)) => break,
```

```
                    Ok((true, needs_clear)) => {
                        if needs_clear {
                            terminal.clear()?;
                        }
                    }
                    Err(_) => {}
                }
            }
        }

        // Execute CPU cycles if running
        if app.run_state == RunState::Running
           && !cpu.is_halted() {
            for _ in 0..app.cycles_per_frame {
                match cpu.step() {
                    Ok(true) => {
                        let output = cpu.get_serial_output();
                        if !output.is_empty() {
                            for &b in output {
                                app.output_buffer
                                   .push_back(b);
                            }
                            cpu.clear_serial_output();
                        }
                    }
                    Ok(false) | Err(_) => {
                        app.run_state = RunState::Halted;
                        break;
                    }
                }
            }
        }

        app.flush_output();
        app.update_metrics(cpu);
    }

    // Restore terminal
    disable_raw_mode()?;
    stdout().execute(LeaveAlternateScreen)?;
    Ok(())
}
```

The sequence is important: `enable_raw_mode` switches the terminal to character-at-a-time input (no line buffering), and `EnterAlternateScreen` saves the user's existing terminal content so it can be restored when the debugger exits. If we forget the cleanup—or if the program panics—the user's terminal will be left in raw mode, requiring a manual `reset` command.

Always Restore the Terminal

If you modify the TUI code, ensure that *every* exit path (including error paths) calls `disable_raw_mode()` and `LeaveAlternateScreen`. A panic handler or `Drop` implementation can serve as a safety net.

13.1.2 The Main Loop Structure

The TUI main loop runs at approximately 60 frames per second, governed by the constant `TICK_RATE_MS = 16`. Each iteration:

1. **Renders the UI** by calling `terminal.draw()`.
2. **Polls for input** with a timeout of one tick.
3. **Executes CPU cycles** if the run state is `Running`.
4. **Flushes serial output** from the CPU to the terminal emulator.
5. **Updates metrics** (effective MHz, host CPU/memory usage).

When running, the emulator executes up to `cycles_per_frame` instructions per frame—defaulting to 50,000. This is adjustable at runtime with Alt+= and Alt+−, letting you trade responsiveness for emulation speed.

13.1.3 Application State

The `App` struct holds all TUI-specific state, separate from the CPU state:

Listing 13.3: The App struct

```
pub struct App {
    pub run_state: RunState,
    pub cycles_per_frame: usize,
    pub memory_view_addr: u16,
    pub terminal: TerminalEmulator,
    pub output_buffer: VecDeque<u8>,
    pub input_buffer: VecDeque<u8>,
    pub cursor_blink: bool,
    pub last_blink: Instant,
    pub effective_mhz: f64,
```

```
    pub host_cpu_percent: f32,
    pub host_memory_mb: u64,
    pub last_metrics_update: Instant,
    pub cycles_this_second: u64,
    pub last_cycle_count: u64,
    pub system: System,
    pub start_pc: u16,
}
```

The `RunState` enum captures the three possible execution states:

Listing 13.4: Execution states

```
pub enum RunState {
    Paused,
    Running,
    Halted,
}
```

Separating app state from CPU state keeps the concerns clean: the CPU doesn't know about the TUI, and the TUI doesn't modify CPU internals except through the public API (`step()`, `reset()`, `send_key()`).

13.2 The Layout System

The TUI divides the terminal into six panels using ratatui's constraint-based layout system:

Listing 13.5: Top-level layout construction

```
fn ui(f: &mut Frame, app: &App, cpu: &Cpu) {
    // Main layout: content + status bar
    let main_chunks = Layout::default()
        .direction(Direction::Vertical)
        .constraints([
            Constraint::Min(10),
            Constraint::Length(1),
        ])
        .split(f.area());

    // Content: left (40%) | right (60%)
    let content_chunks = Layout::default()
        .direction(Direction::Horizontal)
        .constraints([
            Constraint::Percentage(40),
```

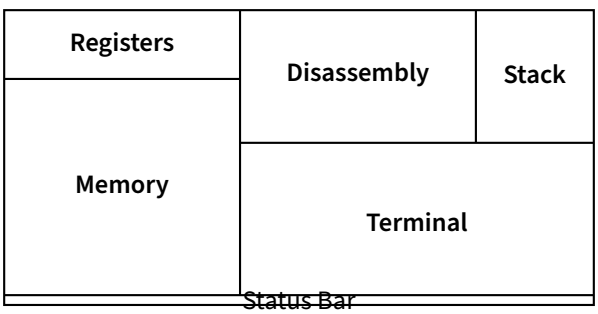
```
            Constraint::Percentage(60),
        ])
        .split(main_chunks[0]);

    // Left: registers (7 lines) | memory (rest)
    let left_chunks = Layout::default()
        .direction(Direction::Vertical)
        .constraints([
            Constraint::Length(7),
            Constraint::Min(10),
        ])
        .split(content_chunks[0]);

    // Right: disasm+stack (45%) | terminal (55%)
    let right_chunks = Layout::default()
        .direction(Direction::Vertical)
        .constraints([
            Constraint::Percentage(45),
            Constraint::Percentage(55),
        ])
        .split(content_chunks[1]);

    // Upper right: disassembly (70%) | stack (30%)
    let upper_right_chunks = Layout::default()
        .direction(Direction::Horizontal)
        .constraints([
            Constraint::Percentage(70),
            Constraint::Percentage(30),
        ])
        .split(right_chunks[0]);
    // ...
}
```

The layout nests vertically and horizontally to produce the final panel arrangement:



This layout adapts to the terminal size. The `Constraint::Min(10)` on the memory panel ensures it gets at least 10 rows, while `Constraint::Length(7)` on the register panel gives it exactly the space it needs (one header line plus four rows of four registers).

13.3 Register Display Panel

The register panel shows the PC, SP, flags, and all 16 registers, with conventional aliases displayed alongside the register numbers:

Listing 13.6: Register panel rendering

```
fn render_registers(f: &mut Frame, area: Rect,
                    cpu: &Cpu) {
    let flags = cpu.get_flags();

    let mut lines = vec![
        Line::from(vec![
            Span::styled("PC ",
                Style::default().fg(Color::DarkGray)),
            Span::styled(format!("{:04X}", cpu.get_pc()),
                Style::default().fg(Color::Green)
                    .add_modifier(Modifier::BOLD)),
            Span::raw("  "),
            Span::styled("SP ",
                Style::default().fg(Color::DarkGray)),
            Span::styled(format!("{:04X}", cpu.get_sp()),
                Style::default().fg(Color::Yellow)),
            Span::raw("  "),
            Span::styled("Flags ",
                Style::default().fg(Color::DarkGray)),
            Span::styled(
                format!("{}{}{}{}{}",
                    if flags & FLAG_N != 0 {'N'} else {'-'},
                    if flags & FLAG_Z != 0 {'Z'} else {'-'},
                    if flags & FLAG_C != 0 {'C'} else {'-'},
                    if flags & FLAG_V != 0 {'V'} else {'-'},
                    if flags & FLAG_I != 0 {'I'} else {'-'},
                ),
                Style::default().fg(Color::Cyan)),
        ]),
    ];
    // ... register rows follow
}
```

The header line displays the program counter in green (the most important value when debugging), the stack pointer in yellow, and the flags as individual letters with dashes for unset flags. This format—`NZC---`—lets you read the flag state at a glance.

13.3.1 Register Naming

Each register is displayed with both its number and its conventional alias:

Listing 13.7: Register name mapping

```
let name = match r {
    0 => "R0/ZR",
    1 => "R1/RA",
    2 => "R2/SP",
    3 => "R3/GP",
    4 => "R4/A0",   5 => "R5/A1",
    6 => "R6/A2",   7 => "R7/A3",
    8 => "R8/T0",   9 => "R9/T1",
    10 => "R10/T2", 11 => "R11/T3",
    12 => "R12/S0", 13 => "R13/S1",
    14 => "R14/S2", 15 => "R15/S3",
    _ => "???",
};
```

The naming conventions follow the SAMPO ABI: `R0`–`R3` are special-purpose (zero, return address, stack pointer, global pointer), `R4`–`R7` are argument registers, `R8`–`R11` are temporaries, and `R12`–`R15` are callee-saved. Displaying both names helps you quickly identify which registers correspond to function arguments versus saved values.

The registers are arranged four per row, producing a compact 4×4 grid:

Listing 13.8: Register grid rendering

```
for row in 0..4 {
    let mut spans = vec![];
    for col in 0..4 {
        let r = row * 4 + col;
        if col > 0 {
            spans.push(Span::raw(" "));
        }
        spans.push(Span::styled(
            format!("{:7}", name),
            Style::default().fg(Color::DarkGray)));
        spans.push(Span::styled(
            format!("{:04X}", cpu.get_register(r)),
```

```
            Style::default().fg(Color::White)));
    }
    lines.push(Line::from(spans));
}
```

Reading the Register Panel

When debugging, focus on the register colors. The PC is always bright green so your eye finds it immediately. Register values are white against the dark border. If a register holds zero, its `0000` display fades into the background, drawing your attention to the non-zero (interesting) values.

13.4 Memory View with Hex Dump

The memory panel renders a traditional hex dump: address, 16 hex bytes, and an ASCII sidebar. It occupies the lower-left portion of the screen and is scrollable with the keyboard.

Listing 13.9: Memory panel rendering

```
fn render_memory(f: &mut Frame, area: Rect,
                 cpu: &Cpu, view_addr: u16) {
    let mut lines = vec![];
    let visible_lines = area.height.saturating_sub(2)
                        as usize;

    for row in 0..visible_lines {
        let addr = view_addr
                   .wrapping_add((row * 16) as u16);

        let mut hex_spans = vec![
            Span::styled(
                format!("{:04X}: ", addr),
                Style::default().fg(Color::DarkGray)),
        ];

        let mut ascii = String::new();
        for col in 0..16 {
            let byte = cpu.read_memory(
                           addr.wrapping_add(col));
            hex_spans.push(Span::styled(
                format!("{:02X} ", byte),
                Style::default().fg(Color::White)));
            ascii.push(if byte >= 0x20 && byte < 0x7F
                       { byte as char } else { '.' });
```

```
        }

        hex_spans.push(Span::styled(ascii,
            Style::default().fg(Color::Yellow)));
        lines.push(Line::from(hex_spans));
    }

    let block = Block::default()
        .title(format!(" Memory @ {:04X} ", view_addr))
        .borders(Borders::ALL)
        .border_style(Style::default().fg(Color::Cyan));

    let paragraph = Paragraph::new(lines).block(block);
    f.render_widget(paragraph, area);
}
```

Each row shows 16 bytes, which is the standard hex dump width and lines up cleanly with 16-byte aligned addresses. The ASCII column on the right replaces non-printable bytes with dots, letting you spot string data embedded in memory—like the "Hello, Sampo!" message in our test program.

The panel title dynamically shows the current view address, updating as you scroll. The view address is stored in `app.memory_view_addr` and adjusted by the keyboard handler.

Wrapping at Address Boundaries

The memory viewer uses `wrapping_add` for address arithmetic, so scrolling past `0xFFFF` wraps to `0x0000`. The full 64 KB address space is accessible without boundary errors.

13.5 Disassembly View

The disassembly panel shows a live disassembly of memory around the current program counter. The current instruction is highlighted with a green arrow marker:

Listing 13.10: Disassembly panel rendering

```
fn render_disassembly(f: &mut Frame, area: Rect,
                      cpu: &Cpu) {
    let pc = cpu.get_pc();
    let mut lines = vec![];

    let mut addr = pc.saturating_sub(8);
    let visible_lines = area.height.saturating_sub(2)
                        as usize;
```

```
    for _ in 0..visible_lines {
        let (mnemonic, size) = disassemble(cpu, addr);

        let mut bytes = String::new();
        for i in 0..size {
            bytes.push_str(&format!("{:02X}",
                cpu.read_memory(addr.wrapping_add(i))));
        }

        let is_current = addr == pc;
        let marker = if is_current {">"} else {" "};

        let style = if is_current {
            Style::default().fg(Color::White)
                .add_modifier(Modifier::BOLD)
        } else {
            Style::default().fg(Color::Gray)
        };

        lines.push(Line::from(vec![
            Span::styled(marker,
                Style::default().fg(Color::Green)),
            Span::styled(format!("{:04X} ", addr),
                Style::default().fg(Color::DarkGray)),
            Span::styled(format!("{:8} ", bytes),
                Style::default().fg(Color::Blue)),
            Span::styled(mnemonic, style),
        ]));

        addr = addr.wrapping_add(size);
    }
    // ... wrapped in a bordered block
}
```

The panel starts disassembly a few bytes before the PC, providing context for what has already executed. Each line shows:

1. A > marker for the current instruction (blank otherwise).
2. The memory address in gray.
3. The raw instruction bytes in blue (2 bytes for standard instructions, 4 for extended).
4. The disassembled mnemonic and operands.

13.5.1 The Disassembler

The `disassemble` function in `semu/src/tui.rs` mirrors the structure of the emulator's `execute` method, but produces text instead of modifying state. It returns a mnemonic string and the instruction size in bytes:

Listing 13.11: Disassembler—selected cases

```
pub fn disassemble(cpu: &Cpu, addr: u16)
                   -> (String, u16) {
    let lo = cpu.read_memory(addr);
    let hi = cpu.read_memory(addr.wrapping_add(1));
    let instr = u16::from_le_bytes([lo, hi]);

    let opcode = (instr >> 12) & 0xF;
    let rd = (instr >> 8) & 0xF;
    let rs1 = (instr >> 4) & 0xF;
    let rs2 = instr & 0xF;
    let imm8 = (instr & 0xFF) as i8;
    let func = instr & 0xF;

    let (mnemonic, size) = match opcode {
        0x0 => (format!("ADD R{}, R{}, R{}",
                        rd, rs1, rs2), 2),
        0x5 => (format!("ADDI R{}, {}", rd, imm8), 2),
        0xC => {
            let op = match func {
                0x0 => format!("PUSH R{}", rs1),
                0x1 => format!("POP R{}", rd),
                0x2 => format!("CMP R{}, R{}", rd, rs1),
                0x4 => format!("MOV R{}, R{}", rd, rs1),
                0xB => "EXX".to_string(),
                _ => format!("MISC f={}", func),
            };
            (op, 2)
        }
        0xE => {
            let op = match rd {
                0x0 => "NOP".to_string(),
                0x1 => "HALT".to_string(),
                _ => format!("SYS f={}", rd),
            };
            (op, 2)
        }
        0xF => {
            // Extended: read second word
```

```
            let lo2 = cpu.read_memory(
                        addr.wrapping_add(2));
            let hi2 = cpu.read_memory(
                        addr.wrapping_add(3));
            let imm16 = u16::from_le_bytes([lo2, hi2]);
            let op = match func {
                0x7 => format!("LIX R{}, 0x{:04X}",
                               rd, imm16),
                0x8 => format!("JX 0x{:04X}", imm16),
                // ... more variants
                _ => format!("EXT sub={}", func),
            };
            (op, 4)
        }
        _ => (format!("??? {:04X}", instr), 2),
    };

    (mnemonic, size)
}
```

The disassembler handles all 16 primary opcodes and their sub-opcodes, producing human-readable mnemonics that match the assembly syntax from Chapter 8. Extended instructions correctly consume 4 bytes and display the 16-bit immediate.

Disassembler vs. Emulator Symmetry

The disassembler's match structure mirrors the emulator's `execute` method almost exactly. When adding a new instruction to the emulator, remember to add the corresponding case to the disassembler—otherwise the TUI will display raw hex instead of a mnemonic.

13.6 The Stack Panel

The stack panel provides a focused view of memory starting at the current stack pointer. Each line shows one 16-bit word, with the top of stack marked:

Listing 13.12: Stack panel rendering

```
fn render_stack(f: &mut Frame, area: Rect, cpu: &Cpu) {
    let sp = cpu.get_sp();
    let mut lines = vec![];
    let visible_lines = area.height.saturating_sub(2)
                        as usize;
```

```
    for i in 0..visible_lines {
        let addr = sp.wrapping_add((i * 2) as u16);
        let lo = cpu.read_memory(addr);
        let hi = cpu.read_memory(addr.wrapping_add(1));
        let val = u16::from_le_bytes([lo, hi]);

        let marker = if i == 0 { ">" } else { " " };

        lines.push(Line::from(vec![
            Span::styled(marker,
                Style::default().fg(Color::Green)),
            Span::styled(format!("{:04X}: ", addr),
                Style::default().fg(Color::DarkGray)),
            Span::styled(format!("{:04X}", val),
                Style::default().fg(Color::White)),
        ]));
    }

    let block = Block::default()
        .title(" Stack ")
        .borders(Borders::ALL)
        .border_style(Style::default().fg(Color::Cyan));

    let paragraph = Paragraph::new(lines).block(block);
    f.render_widget(paragraph, area);
}
```

The stack grows downward on the SAMPO (each **PUSH** decrements SP by 2), so the panel shows addresses increasing *upward* from SP. The first entry (marked with >) is the value that the next **POP** would retrieve.

This panel is invaluable when debugging function calls: you can see return addresses, saved registers, and local variables laid out on the stack exactly as your code placed them.

13.7 Keyboard Controls

The TUI's keyboard handler maps function keys and modifiers to debugger commands:

Listing 13.13: Keyboard handler (selected keys)

```
fn handle_key(app: &mut App, cpu: &mut Cpu,
              key: KeyEvent)
              -> io::Result<(bool, bool)> {
    let mut needs_clear = false;
```

```
    match key.code {
        KeyCode::F(5) => {
            if !cpu.is_halted() {
                app.run_state = RunState::Running;
            }
        }
        KeyCode::F(6) => {
            if !cpu.is_halted() {
                match cpu.step() {
                    Ok(true) => {
                        let output =
                            cpu.get_serial_output();
                        if !output.is_empty() {
                            for &b in output {
                                app.output_buffer
                                   .push_back(b);
                            }
                            cpu.clear_serial_output();
                        }
                    }
                    Ok(false) | Err(_) => {
                        app.run_state = RunState::Halted;
                    }
                }
                app.run_state = RunState::Paused;
            }
        }
        KeyCode::F(7) => {
            app.run_state = RunState::Paused;
        }
        KeyCode::F(8) => {
            cpu.reset();
            cpu.set_pc(app.start_pc);
            app.run_state = RunState::Paused;
            app.terminal = TerminalEmulator::new();
            app.output_buffer.clear();
            needs_clear = true;
        }
        KeyCode::F(12) => {
            return Ok((false, false)); // Quit
        }
        // ...
        _ => {}
    }
    Ok((true, needs_clear))
```

```
}
```

Table 13.1: TUI keyboard controls

Key	Action	Notes
F5	Run continuously	Executes at `cycles_per_frame` rate
F6	Step one instruction	Pauses after single step
F7	Pause execution	Stops continuous run
F8	Reset CPU	Restores initial PC and clears state
F9	Memory view up	Scrolls 16 bytes
F10	Memory view down	Scrolls 16 bytes
PgUp	Memory view up	Scrolls 256 bytes
PgDn	Memory view down	Scrolls 256 bytes
Alt+=	Increase speed	+10,000 cycles per frame (max 500K)
Alt+−	Decrease speed	−10,000 cycles per frame (min 1K)
F12	Quit	Exits TUI and restores terminal

13.7.1 Single-Step Mechanics

The F6 (step) handler deserves closer inspection. After calling `cpu.step()`, it immediately checks for serial output and transfers it to the output buffer. This ensures that the terminal panel updates even during single-step debugging—you see each character appear as the **`OUTI`** instruction executes.

The handler always sets the run state to `Paused` after stepping, even if the step completed successfully. This prevents a single F6 press from accidentally running multiple instructions.

13.7.2 Reset Behavior

The F8 (reset) handler restores the CPU to its initial state by calling `cpu.reset()` and then setting the PC back to the original entry point stored in `app.start_pc`. It also clears the terminal emulator and output buffer, giving you a completely fresh start. The `needs_clear` flag triggers a full terminal redraw to remove any visual artifacts.

13.7.3 Keyboard Input Forwarding

When the emulator is running, printable characters and special keys are forwarded to the CPU as serial input:

Listing 13.14: Character forwarding to the CPU

```
KeyCode::Char(c) => {
```

```
    if app.run_state == RunState::Running {
        if key.modifiers.contains(KeyModifiers::CONTROL) {
            let ctrl_char =
                (c as u8).wrapping_sub(b'a' - 1);
            cpu.send_key(ctrl_char);
        } else {
            cpu.send_key(c as u8);
        }
    }
}
KeyCode::Enter => {
    if app.run_state == RunState::Running {
        cpu.send_key(0x0D);
    }
}
```

Control characters are computed by subtracting the ASCII offset: Ctrl+C becomes `0x03`, Ctrl+D becomes `0x04`, and so on. Arrow keys are translated into VT220 escape sequences (`ESC [ A` through `ESC [ D`) by sending three bytes in succession through `send_key`.

This forwarding only happens when the run state is `Running`. In `Paused` mode, keystrokes go to the debugger controls instead, preventing accidental input to the emulated program while you're stepping through code.

13.8 Serial Output Capture

The TUI includes a VT220-compatible terminal emulator that renders serial output from the SAMPO CPU, turning the debugger into a self-contained development environment.

13.8.1 The Terminal Emulator

The `TerminalEmulator` struct maintains an 80×24 character buffer, a cursor position, and an escape sequence parser:

Listing 13.15: Terminal emulator state

```
pub struct TerminalEmulator {
    buffer: [[char; TERM_COLS]; TERM_ROWS],
    cursor_row: usize,
    cursor_col: usize,
    cursor_visible: bool,
    escape_state: EscapeState,
    escape_buffer: String,
}
```

```
#[derive(Clone, Copy, PartialEq)]
enum EscapeState {
    Normal,
    Escape,
    Csi,
}
```

The constants `TERM_COLS = 80` and `TERM_ROWS = 24` match the classic VT220 screen dimensions. The escape state machine tracks whether we're in the middle of parsing an ANSI escape sequence.

13.8.2 Character Processing

Each byte from the serial output flows through `putchar`, which handles control characters and printable characters:

Listing 13.16: Normal character handling

```
fn handle_normal(&mut self, c: u8) {
    match c {
        0x1B => {
            self.escape_state = EscapeState::Escape;
            self.escape_buffer.clear();
        }
        0x0D => { self.cursor_col = 0; }        // CR
        0x0A => {                                 // LF
            self.cursor_row += 1;
            if self.cursor_row >= TERM_ROWS {
                self.scroll_up();
                self.cursor_row = TERM_ROWS - 1;
            }
        }
        0x08 => {                                 // BS
            if self.cursor_col > 0 {
                self.cursor_col -= 1;
            }
        }
        0x09 => {                                 // TAB
            self.cursor_col =
                (self.cursor_col + 8) & !7;
            if self.cursor_col >= TERM_COLS {
                self.cursor_col = TERM_COLS - 1;
            }
        }
        0x20..=0x7E => {                          // Print
```

```
            if self.cursor_col < TERM_COLS
               && self.cursor_row < TERM_ROWS {
                self.buffer[self.cursor_row]
                           [self.cursor_col] = c as char;
                self.cursor_col += 1;
                if self.cursor_col >= TERM_COLS {
                    self.cursor_col = 0;
                    self.cursor_row += 1;
                    if self.cursor_row >= TERM_ROWS {
                        self.scroll_up();
                        self.cursor_row = TERM_ROWS - 1;
                    }
                }
            }
        }
        _ => {}
    }
}
```

The terminal emulator supports the essential control characters: carriage return (`0x0D`), line feed (`0x0A`), backspace (`0x08`), and horizontal tab (`0x09`). When the cursor reaches the bottom of the screen, `scroll_up` shifts all rows up by one and clears the last line.

13.8.3 ANSI Escape Sequence Handling

When the emulator encounters an ESC character (`0x1B`), it enters a state machine that parses CSI (Control Sequence Introducer) sequences—the `ESC [` prefix followed by parameters and a final command byte:

Listing 13.17: CSI sequence execution (selected commands)

```
fn execute_csi(&mut self) {
    let seq = &self.escape_buffer;

    if seq.ends_with('H') || seq.ends_with('f') {
        // Cursor position: ESC[row;colH
        let params: Vec<usize> = seq[..seq.len()-1]
            .split(';')
            .filter_map(|s| s.parse().ok())
            .collect();
        self.cursor_row = params.get(0).copied()
            .unwrap_or(1).saturating_sub(1)
            .min(TERM_ROWS - 1);
        self.cursor_col = params.get(1).copied()
```

```
            .unwrap_or(1).saturating_sub(1)
            .min(TERM_COLS - 1);
    } else if seq.ends_with('J') {
        // Erase display
        let param: usize = seq[..seq.len()-1]
            .parse().unwrap_or(0);
        match param {
            0 => self.clear_to_end(),
            1 => self.clear_to_start(),
            2 => self.clear_screen(),
            _ => {}
        }
    } else if seq.ends_with('K') {
        // Erase line
        // ...
    }
    // Cursor movement: A(up) B(down) C(right) D(left)
    // Cursor visibility: ?25h (show) ?25l (hide)
}
```

This subset of VT220 escape sequences covers the most common operations: cursor positioning, screen and line clearing, and cursor visibility. It is sufficient for SAMPO programs that produce formatted terminal output, such as a monitor program or a text-mode game.

13.8.4 Output Buffering and Flow Control

Serial output doesn't go directly from the CPU to the terminal emulator. Instead, it passes through an output buffer (`VecDeque<u8>`) that is drained at a controlled rate:

Listing 13.18: Output buffer flushing

```
pub fn flush_output(&mut self) {
    let mut count = 0;
    while count < OUTPUT_CHARS_PER_FRAME
          && !self.output_buffer.is_empty() {
        if let Some(c) = self.output_buffer.pop_front() {
            self.terminal.putchar(c);
            count += 1;
        }
    }
}
```

The constant `OUTPUT_CHARS_PER_FRAME = 120` limits how many characters are rendered per frame. This prevents a program that outputs thousands of characters from freezing the TUI while the terminal

emulator processes them all. The visual effect is smooth scrolling output, similar to watching text appear on a real serial terminal.

13.8.5 Rendering the Terminal Panel

The terminal panel renders the 80×24 character buffer, with a blinking cursor:

Listing 13.19: Terminal panel with cursor rendering

```
fn render_terminal(f: &mut Frame, area: Rect,
                   app: &App) {
    let term_lines = app.terminal.get_lines();
    let (cursor_row, cursor_col) =
        app.terminal.cursor_position();

    let mut lines: Vec<Line> = vec![];

    for (row_idx, row) in term_lines.iter().enumerate() {
        if row_idx == cursor_row
           && app.terminal.is_cursor_visible()
           && app.cursor_blink {
            let chars: Vec<char> =
                row.chars().collect();
            if cursor_col < chars.len() {
                let mut spans = vec![];
                spans.push(Span::raw(
                    chars[..cursor_col].iter()
                         .collect::<String>()));
                spans.push(Span::styled(
                    chars[cursor_col].to_string(),
                    Style::default()
                        .bg(Color::White)
                        .fg(Color::Black)));
                if cursor_col + 1 < chars.len() {
                    spans.push(Span::raw(
                        chars[cursor_col + 1..].iter()
                             .collect::<String>()));
                }
                lines.push(Line::from(spans));
            } else {
                lines.push(Line::from(row.clone()));
            }
        } else {
            lines.push(Line::from(row.clone()));
        }
```

```
    }
    // ... wrapped in bordered block
}
```

The cursor blink toggles every 500 milliseconds, controlled by `app.cursor_blink` and `app.last_blink`. When visible, the cursor is rendered as an inverted character (white background, black text)—matching the appearance of a hardware block cursor on a real terminal.

13.9 The Status Bar

The bottom row of the screen shows execution state, performance metrics, and a keyboard shortcut reminder:

Listing 13.20: Status bar rendering

```
fn render_status(f: &mut Frame, area: Rect,
                 app: &App, cpu: &Cpu) {
    let state_span = match app.run_state {
        RunState::Running => Span::styled("[RUNNING]",
            Style::default().fg(Color::Green)),
        RunState::Paused => Span::styled("[PAUSED]",
            Style::default().fg(Color::Yellow)),
        RunState::Halted => Span::styled("[HALTED]",
            Style::default().fg(Color::Red)),
    };

    let line = Line::from(vec![
        state_span,
        Span::raw(" "),
        Span::styled(
            format!("{:.2} MHz", app.effective_mhz),
            Style::default().fg(Color::Cyan)),
        Span::raw("  "),
        Span::styled(
            format!("CPU:{:.0}%", app.host_cpu_percent),
            Style::default().fg(Color::Gray)),
        Span::raw("  "),
        Span::styled(
            format!("Cycles:{}", cpu.get_cycles()),
            Style::default().fg(Color::DarkGray)),
        Span::raw("  "),
        Span::styled(
            "F5:Run F6:Step F7:Pause F8:Reset F12:Quit",
            Style::default().fg(Color::DarkGray)),
```

```
    ]);
    // ...
}
```

13.9.1 Performance Metrics

The "effective MHz" metric is computed every 500 milliseconds by comparing the cycle counter against the previous sample:

Listing 13.21: Effective MHz calculation

```
pub fn update_metrics(&mut self, cpu: &Cpu) {
    let now = Instant::now();
    if now.duration_since(self.last_metrics_update)
       >= Duration::from_millis(500) {
        let cycles_now = cpu.get_cycles();
        let delta = cycles_now - self.last_cycle_count;
        self.effective_mhz =
            (delta as f64) / 500_000.0;
        self.last_cycle_count = cycles_now;

        self.system.refresh_cpu_all();
        self.system.refresh_memory();

        let cpus = self.system.cpus();
        if !cpus.is_empty() {
            self.host_cpu_percent = cpus.iter()
                .map(|c| c.cpu_usage())
                .sum::<f32>() / cpus.len() as f32;
        }
        self.host_memory_mb =
            self.system.used_memory() / 1024 / 1024;

        self.last_metrics_update = now;
    }
}
```

On a modern machine, the emulator typically achieves 50–100 effective MHz—far faster than the hardware SAMPO would run, but useful for getting a sense of throughput. The host CPU and memory metrics come from the `sysinfo` crate, confirming that the emulator is lightweight even during continuous execution.

The run state indicator uses color-coding for instant recognition: green for running, yellow for paused, and red for halted. This is one of those small design choices that compounds over hours of debugging—you never have to wonder whether the CPU is still executing.

Exercises

1. **Breakpoint Panel.** Add a breakpoint list to the TUI. When the user presses a key (perhaps F4), prompt for an address (or use the current memory view address) and add it to a breakpoint set. Modify the run loop to pause when the PC hits a breakpoint. Display active breakpoints in a new panel or as markers in the disassembly view.
2. **Register Change Highlighting.** Modify the register panel to highlight registers that changed during the last instruction step. Store the previous register values and compare them after each step. Display changed registers in a contrasting color (perhaps bright yellow) for one frame, then return to normal.
3. **Memory Write Highlighting.** Track the most recent memory write address and size. In the memory viewer, highlight the bytes that were modified by the last instruction. This makes it much easier to see the effects of `SW`, `SB`, and `PUSH` instructions.
4. **Watchpoint Support.** Implement data watchpoints: the user specifies a memory address, and the emulator pauses whenever that address is written to. This requires checking every memory write against the watchpoint list—consider how to do this without significant performance impact.

What's Next. The TUI gives us a window into the CPU's soul, but so far the only peripheral we've simulated is a UART. In the next chapter, we will examine the I/O port abstraction in detail, build a proper UART simulation with TX/RX buffers and status registers, and explore how to add new peripherals—timers, GPIO, and more—to the emulator.

Chapter 14

Peripheral Simulation

A CPU without peripherals is a calculator. The programs we've been running—`hello.s`, the ALU test suite, the branch tests—all rely on a serial port to report their results. Without the UART simulation in the emulator, we'd have no way to observe what the CPU is doing short of inspecting registers one at a time.

In this chapter we take a closer look at how the SAMPO emulator models peripheral hardware. We'll start with the I/O port abstraction that underlies all peripheral access, then study the UART simulation in detail—its TX and RX buffers, its status register, and how it bridges between the emulated CPU and the host terminal. Finally, we'll walk through the process of adding new peripherals, giving you the tools to extend the emulator with timers, GPIO, and whatever else your SAMPO programs need.

14.1 I/O Port Abstraction

The SAMPO uses *port-mapped* I/O—a separate address space from memory, accessed through dedicated **`IN`** and **`OUT`** instructions rather than through **`LW`** and **`SW`**. This is the same model used by the Z80, x86, and many microcontrollers, in contrast to the memory-mapped I/O favored by ARM and RISC-V.

Port-Mapped vs. Memory-Mapped I/O

In *port-mapped* I/O, peripheral registers occupy a separate address space from main memory. The CPU uses dedicated instructions (**IN**, **OUT**) to access them. In *memory-mapped* I/O, peripheral registers share the memory address space and are accessed through regular load/store instructions. The SAMPO uses port-mapped I/O with a 256-port address space (`0x00`–`0xFF`), keeping the full 64 KB memory space available for code and data.

14.1.1 The Port Array

In the emulator, the 256-port address space is represented by a flat array inside the `Cpu` struct:

Listing 14.1: Port storage in the CPU struct

```
pub struct Cpu {
    // ...
    ports: [u8; 256],
    // ...
}
```

Each port holds an 8-bit value, matching the SAMPO hardware where port data is always byte-width. The array is initialized to all zeros, with one exception: port `0x80` is pre-set to `0x02` to indicate that the UART transmitter is ready:

Listing 14.2: Port initialization

```
// In Cpu::new():
cpu.ports[0x80] = 0x02; // Set serial TX ready bit
```

14.1.2 Port Access Instructions

The SAMPO ISA provides four I/O instructions, all under opcode `0xD`:

The immediate variants (**INI**, **OUTI**) encode the port address in the 4-bit Rs1 field, limiting them to ports `0x0`–`0xF`. This is a deliberate trade-off: the most frequently accessed ports (like the UART at `0x80`) are reached through the extended **INX**/**OUTX** instructions, while the first 16 ports can be accessed with a single 16-bit instruction.

The register-indirect variants (**IN**, **OUT**) take the port address from a register, providing access to the full 256-port space. In the emulator, all four variants route through the same `port_read` and `port_write` methods:

Listing 14.3: I/O instruction dispatch

Table 14.1: I/O instruction variants

Func	Mnemonic	Syntax	Description
0x0	**INI**	`INI Rd, port`	Read from immediate port (4-bit address)
0x1	**OUTI**	`OUTI port, Rs`	Write to immediate port (4-bit address)
0x2	**IN**	`IN Rd, (Rs1)`	Read from register-indirect port
0x3	**OUT**	`OUT (Rd), Rs1`	Write to register-indirect port

```
fn execute_io(&mut self, rd: usize, rs1: usize,
              func: u16) -> Result<(), String> {
    match func {
        0x0 => {
            // INI Rd, port
            let port = rs1 as u8;
            let val = self.port_read(port);
            self.set_reg(rd, val as u16);
        }
        0x1 => {
            // OUTI port, Rd
            let port = rs1 as u8;
            let val = self.get_reg(rd) as u8;
            self.port_write(port, val);
        }
        0x2 => {
            // IN Rd, (Rs1)
            let port = self.get_reg(rs1) as u8;
            let val = self.port_read(port);
            self.set_reg(rd, val as u16);
        }
        0x3 => {
            // OUT (Rd), Rs1
            let port = self.get_reg(rd) as u8;
            let val = self.get_reg(rs1) as u8;
            self.port_write(port, val);
        }
        _ => return Err(format!(
               "Unknown I/O func: 0x{:X}", func)),
    }
    Ok(())
}
```

14.1.3 Extended Port Instructions

For ports outside the 4-bit immediate range, the extended instructions **INX** (sub `0xB`) and **OUTX** (sub `0xC`) provide an 8-bit port address in the 16-bit immediate word:

Listing 14.4: Extended I/O instructions

```
0xB => {
    // INX Rd, port8
    let val = self.port_read(imm16 as u8);
    self.set_reg(rd, val as u16);
}
0xC => {
    // OUTX port8, Rs1
    let val = self.get_reg(rs1) as u8;
    self.port_write(imm16 as u8, val);
}
```

These are 32-bit instructions (4 bytes each), making them larger than the immediate variants, but they can address any of the 256 ports. In practice, the UART ports at `0x80` and `0x81` are accessed almost exclusively through **INX** and **OUTX**—or, as we see in the test programs, through the **INI**/**OUTI** forms with the assembler handling the encoding details.

14.2 UART Simulation

The SAMPO emulator simulates a serial port modeled after the Motorola MC6850 ACIA (Asynchronous Communications Interface Adapter), a classic peripheral found in many 8-bit computers of the 1970s and 1980s. The simulation uses two ports:

Table 14.2: UART port map

Port	Name	Description
`0x80`	Status	Transmit/receive status bits
`0x81`	Data	Transmit/receive data register

14.2.1 The Status Register

The status register at port `0x80` contains two flags that control data flow:

In the emulator, reading the status port always returns `0x02` (TX ready), with the RX ready bit set dynamically when keyboard input arrives:

Table 14.3: UART status register bits

Bit	Name	Meaning
0	RX Ready	Data available to read from port `0x81`
1	TX Ready	Transmitter ready to accept data

Listing 14.5: UART status port read

```
fn port_read(&mut self, port: u8) -> u8 {
    match port {
        0x80 => {
            // ACIA status - always ready
            0x02 // TX ready
        }
        0x81 => {
            // ACIA data - nothing to read
            0
        }
        _ => self.ports[port as usize],
    }
}
```

This simplification reflects the reality of emulation: writing to a software buffer is instantaneous, so there is never a reason for the transmitter to be "not ready." A real UART shifts bits out serially at a baud rate (say, 115200 bps), which takes time. The emulator collapses that latency to zero.

When TX Ready Matters

Even though the emulator always reports TX ready, the SAMPO programs still poll the status register in a loop. This is good practice—the same binary must run correctly on hardware, where the UART is genuinely busy during transmission. The `hello.s` program includes exactly this pattern:

```
wait_tx:
        INI  R6, ACIA_STATUS    ; Read serial status
        AND  R7, R6, R6         ; Copy to R7
        ADDI R7, -2             ; Subtract TX_READY (2)
        BNE  wait_tx            ; Wait if not ready
```

In the emulator, this loop executes exactly once per character. On real hardware, it might spin for several microseconds.

14.2.2 The TX Data Path

When the CPU writes to port `0x81`, the emulator captures the byte in two ways:

Listing 14.6: UART data port write

```
fn port_write(&mut self, port: u8, val: u8) {
    match port {
        0x81 => {
            // ACIA data - output character
            self.serial_out.push(val);
            if !self.quiet {
                print!("{}", val as char);
                io::stdout().flush().ok();
            }
        }
        _ => {
            self.ports[port as usize] = val;
        }
    }
}
```

The byte is:

1. **Appended to `serial_out`**: This buffer is the primary output mechanism, read by the TUI debugger and the final state dump.
2. **Printed to stdout** (unless `quiet` is set): In batch mode, characters appear on the host terminal as they would on a real serial monitor. The `flush()` call ensures immediate display rather than waiting for a newline.

The `quiet` flag is set when the TUI debugger is active. In that mode, serial output routes exclusively through the output buffer to the terminal emulator panel, preventing raw characters from corrupting the ratatui display.

14.2.3 The RX Data Path

Receiving data—keyboard input from the host flowing to the emulated CPU—works through the `send_key` method:

Listing 14.7: Serial input injection

```
pub fn send_key(&mut self, key: u8) {
    // Queue input for serial port
    self.ports[0x81] = key;
    self.ports[0x80] |= 0x01; // RX ready
```

```
}
```

This method places the byte directly in the data register and sets the RX-ready bit in the status register. The next time the emulated program reads port `0x80`, it will see bit 0 set and know that data is available at port `0x81`.

There is a limitation here: the emulator can only buffer one byte at a time. If `send_key` is called twice before the CPU reads the data port, the first byte is overwritten. For the simple programs we run on the SAMPO, this is acceptable—keyboard input arrives at human typing speed, which is glacially slow compared to the emulated CPU.

A typical receive loop in SAMPO assembly looks like this:

Listing 14.8: Polling loop for serial input

```
wait_rx:
        INX  R4, 0x80           ; Read ACIA status
        ADDI R4, -1             ; Check RX ready (bit 0)
        BNE  wait_rx            ; Loop until data ready
        INX  R4, 0x81           ; Read received byte
```

14.2.4 TUI Integration

The TUI debugger bridges the gap between the CPU's serial output and the host terminal through a buffering mechanism. As we saw in Chapter 13, the main loop transfers bytes from `cpu.serial_out` to `app.output_buffer`, which the terminal emulator then processes at a controlled rate:

Listing 14.9: Serial output flow in the TUI loop

```
// In the main TUI loop:
match cpu.step() {
    Ok(true) => {
        let output = cpu.get_serial_output();
        if !output.is_empty() {
            for &b in output {
                app.output_buffer.push_back(b);
            }
            cpu.clear_serial_output();
        }
    }
    // ...
}

// Later in the same loop iteration:
```

```
app.flush_output();  // Process up to 120 chars/frame
```

This three-stage pipeline—CPU serial buffer → app output buffer → terminal emulator—decouples the emulation speed from the rendering speed. The CPU can produce thousands of characters per frame, but the terminal emulator only processes 120 of them, preventing the UI from freezing on programs that generate rapid output.

Adjusting Output Speed

The constant `OUTPUT_CHARS_PER_FRAME = 120` in `semu/src/tui.rs` controls how quickly serial output appears on screen. For programs that produce very little output, you might want to increase this value to reduce perceived latency. For programs that flood the serial port (like a memory dump), the current limit provides smoother scrolling.

14.3 Adding New Peripherals

The SAMPO emulator's port I/O architecture makes it straightforward to add new peripherals. The process follows a consistent pattern: choose a port range, handle reads and writes in the `port_read` and `port_write` methods, and add any internal state to the `Cpu` struct (or, for cleaner separation, to a new peripheral struct).

14.3.1 The Peripheral Pattern

Every peripheral follows the same general structure:

1. **Define a port range.** Choose addresses that don't conflict with existing peripherals. The UART occupies `0x80`–`0x81`; the rest of the 256-port space is available.

2. **Implement read behavior.** Add a `match` arm to `port_read` for each port that returns data. Status registers typically report peripheral state; data registers return buffered values.

3. **Implement write behavior.** Add a `match` arm to `port_write` for each port that accepts commands or data. Control registers configure the peripheral; data registers accept values for transmission or processing.

4. **Add internal state.** Peripheral logic may require counters, buffers, state machines, or timers. Add these as fields on the `Cpu` struct or as a separate struct owned by `Cpu`.

5. **Handle timing.** If the peripheral has time-dependent behavior (like a timer), update its state in the `step()` method or in a dedicated `tick()` method called each cycle.

14.3.2 Example: A Simple Timer

Let's design a timer peripheral that counts emulator cycles and generates an interrupt when the count reaches a programmed threshold. We'll use ports `0x90`–`0x93`:

Table 14.4: Proposed timer port map

Port	Name	Description
`0x90`	TIMER_CTRL	Control register (bit 0 = enable, bit 1 = interrupt enable)
`0x91`	TIMER_STATUS	Status register (bit 0 = overflow flag)
`0x92`	TIMER_COUNT_LO	Current count, low byte (read-only)
`0x93`	TIMER_COUNT_HI	Current count, high byte (read-only)
`0x94`	TIMER_THRESH_LO	Threshold, low byte (write-only)
`0x95`	TIMER_THRESH_HI	Threshold, high byte (write-only)

The implementation would add timer state to the CPU:

Listing 14.10: Timer state fields (proposed)

```
// Added to the Cpu struct:
timer_enabled: bool,
timer_int_enable: bool,
timer_count: u16,
timer_threshold: u16,
timer_overflow: bool,
```

The port handlers would look like this:

Listing 14.11: Timer port handlers (proposed)

```
// In port_read:
0x90 => {
    let mut val = 0u8;
    if self.timer_enabled { val |= 0x01; }
    if self.timer_int_enable { val |= 0x02; }
    val
}
0x91 => {
    if self.timer_overflow { 0x01 } else { 0x00 }
}
0x92 => (self.timer_count & 0xFF) as u8,
0x93 => (self.timer_count >> 8) as u8,

// In port_write:
```

```
0x90 => {
    self.timer_enabled = (val & 0x01) != 0;
    self.timer_int_enable = (val & 0x02) != 0;
    if (val & 0x80) != 0 {
        // Reset count on control write with bit 7
        self.timer_count = 0;
        self.timer_overflow = false;
    }
}
0x94 => {
    self.timer_threshold =
        (self.timer_threshold & 0xFF00) | (val as u16);
}
0x95 => {
    self.timer_threshold =
        (self.timer_threshold & 0x00FF)
        | ((val as u16) << 8);
}
```

And in the `step()` method, after executing the instruction:

Listing 14.12: Timer tick logic (proposed)

```
// In step(), after execute():
if self.timer_enabled {
    self.timer_count = self.timer_count.wrapping_add(1);
    if self.timer_count >= self.timer_threshold {
        self.timer_overflow = true;
        self.timer_count = 0;
        // Could trigger interrupt here if enabled
    }
}
```

A SAMPO program could then use this timer to implement periodic tasks:

Listing 14.13: Using the timer peripheral

```
; Configure timer for 1000-cycle interval
        LIX  R4, 1000
        OUTX 0x94, R4          ; Set threshold low byte
        SRL8 R4, R4            ; Shift high byte down
        OUTX 0x95, R4          ; Set threshold high byte

; Enable timer
        LIX  R4, 0x81          ; Enable + reset
```

```
        OUTX 0x90, R4

; Poll for overflow
wait_timer:
        INX  R4, 0x91          ; Read status
        ADDI R4, -1            ; Check overflow bit
        BNE  wait_timer

; Timer fired -- do something
        ; ...
```

14.3.3 Example: GPIO Port

A GPIO (General Purpose Input/Output) peripheral is even more straightforward. At its core, it's a register that can be read and written, with the bits corresponding to physical pins:

Table 14.5: Proposed GPIO port map

Port	Name	Description
0xA0	GPIO_DIR	Direction register (0 = input, 1 = output)
0xA1	GPIO_OUT	Output register (write value to pins)
0xA2	GPIO_IN	Input register (read current pin state)

In the emulator, the GPIO could be backed by simple port storage:

Listing 14.14: GPIO port handlers (proposed)

```
// In port_read:
0xA0 => self.ports[0xA0],       // Direction
0xA1 => self.ports[0xA1],       // Output (readback)
0xA2 => {
    // Input pins - could read from host
    // For now, reflect output for loopback testing
    self.ports[0xA1] & !self.ports[0xA0]
}

// In port_write:
0xA0 => { self.ports[0xA0] = val; }
0xA1 => { self.ports[0xA1] = val; }
```

The loopback behavior—reading back the output value through the input register—lets you test GPIO-dependent code without actual hardware. For more realistic simulation, you could connect the GPIO to host-side resources like named pipes or sockets.

14.4 Connecting to Host I/O

The emulator already connects the UART to the host terminal (stdout in batch mode, the TUI terminal emulator in graphical mode). The same principle can extend to other peripherals.

14.4.1 Stdout and Stdin

The simplest host connection is the one we already have: mapping UART TX to stdout and UART RX to stdin. In batch mode, the emulator writes characters directly:

Listing 14.15: Direct stdout output in batch mode

```
0x81 => {
    self.serial_out.push(val);
    if !self.quiet {
        print!("{}", val as char);
        io::stdout().flush().ok();
    }
}
```

The `flush()` call after every character is important. Without it, output would be buffered by the Rust standard library until a newline appears, making interactive programs appear unresponsive.

14.4.2 File-Backed Peripherals

For peripherals that need to persist data—like a disk controller or an EEPROM—you could map port writes to a file on the host filesystem. The emulator already accepts a binary file as input through `load_program`; a file-backed peripheral would reverse this flow, writing emulated device output to a host file.

Listing 14.16: Conceptual file-backed storage peripheral

```
// Hypothetical block device write handler:
0xB1 => {
    // BLOCK_DATA port: write byte to sector buffer
    self.block_buffer.push(val);
    if self.block_buffer.len() >= 512 {
        // Full sector -- flush to host file
        let offset = (self.block_sector as u64) * 512;
```

```
            self.block_file.seek(
                SeekFrom::Start(offset)).ok();
            self.block_file.write_all(
                &self.block_buffer).ok();
            self.block_buffer.clear();
        }
    }
```

14.4.3 The Default Port Handler

Ports that don't have special behavior fall through to the default handler, which simply reads from or writes to the `ports` array:

Listing 14.17: Default port handling

```
// In port_read:
_ => self.ports[port as usize],

// In port_write:
_ => {
    self.ports[port as usize] = val;
}
```

This means any port can be used as a simple register even without explicit peripheral logic. Programs can store temporary values in unused ports, and the emulator will faithfully preserve them. However, these "unconnected" ports won't do anything visible—they're just storage.

14.4.4 Port Address Map Summary

Table 14.6: Current and reserved port address ranges

Range	Peripheral	Status
`0x00–0x0F`	Fast I/O	Available (4-bit immediate access)
`0x10–0x7F`	Unassigned	Available for user peripherals
`0x80–0x81`	UART (ACIA)	Implemented
`0x82–0x8F`	Reserved	For additional serial ports
`0x90–0x9F`	Timer	Proposed (see Section 14.3.2)
`0xA0–0xAF`	GPIO	Proposed (see Section 14.3.3)
`0xB0–0xBF`	Block device	Future
`0xC0–0xFF`	Expansion	Available

Port Conflicts

If you add a new peripheral, check that its port range doesn't overlap with existing peripherals. The emulator's `match` statements in `port_read` and `port_write` will silently use whichever arm matches first, which can lead to subtle bugs if two peripherals share an address.

14.5 Peripheral Design Principles

Before adding peripherals to the emulator, it helps to establish some design principles that will keep the simulation consistent with the eventual hardware implementation.

14.5.1 Match the Hardware Interface

Every peripheral you simulate in the emulator should have a corresponding hardware design in mind. Define the port map, the register bit assignments, and the behavioral contract before writing code. This ensures that assembly programs written against the emulator will work unchanged on the FPGA implementation.

14.5.2 Keep Reads Side-Effect-Free (Mostly)

Port reads should generally not change peripheral state. The one common exception is a data register that auto-clears a "ready" flag when read—for example, reading UART data port `0x81` might clear the RX-ready bit in the status register. Our current emulator doesn't implement this behavior, but a more faithful simulation would.

14.5.3 Document the Register Map

For each peripheral, document:

- The port addresses and their names.
- The bit assignments within each register.
- The read vs. write behavior (some registers behave differently when read vs. written).
- Reset values.
- Any timing constraints (e.g., "must wait 3 cycles after writing the control register before reading status").

This documentation becomes the contract between your software (assembly programs and emulator) and your hardware (Verilog modules).

14.5.4 Test with Real Programs

The best way to validate a peripheral simulation is to write an assembly program that exercises it, then run the same program on both the emulator and the Verilog simulation. If the outputs match, you have confidence that the emulator is faithful. If they don't, the mismatch tells you exactly where to look.

The test suite in `examples/tests/` follows this principle: each test program uses the serial port to report pass/fail results, and the same binaries run on both the emulator and the Verilog testbench.

Exercises

1. **Implement RX-Ready Auto-Clear.** Modify the UART simulation so that reading port `0x81` (the data register) automatically clears the RX-ready bit (bit 0) in port `0x80` (the status register). Write an assembly program that reads a character, echoes it back, and loops. Verify that the program correctly waits for new input after each echo.

2. **Build the Timer Peripheral.** Implement the timer peripheral described in Section 14.3.2. Write an assembly program that configures the timer to fire every 100 cycles, then counts the number of overflows in a loop. Print the count to the serial port and verify it matches expectations.

3. **SPI Peripheral Simulation.** Design and implement a simple SPI (Serial Peripheral Interface) controller peripheral using ports `0xB0`–`0xB3` for control, status, TX data, and RX data. The controller should simulate a shift register that exchanges one byte per "transfer." Write an assembly program that sends and receives data.

4. **Interrupt-Driven UART.** Modify the UART simulation to trigger a software interrupt (via the **`SWI`** mechanism) when a character is received, instead of requiring the CPU to poll the status register. This requires integrating the interrupt logic with the `step()` method: after each instruction, check if the UART has pending data and the interrupt-enable flag is set, then simulate an interrupt by pushing PC and jumping to a vector address.

5. **Multi-Byte UART Buffer.** Replace the single-byte RX mechanism in `send_key` with a proper FIFO buffer (a `VecDeque<u8>`). Modify `port_read` for port `0x81` to pop from the FIFO, and `port_read` for port `0x80` to report RX-ready only when the FIFO is non-empty. Test with rapid keystroke input in the TUI to verify that no characters are lost.

What's Next. With the emulator complete—CPU core, debugger, and peripheral simulation—we have a fully functional software model of the SAMPO system. Every program we write can be tested and debugged before we touch any hardware. But the goal of this project has always been to build a *real* CPU, not just simulate one. In Part V, we shift gears from software to hardware, starting

with the digital logic foundations that underlie every circuit in the SAMPO: gates, flip-flops, and the combinational and sequential building blocks from which CPUs are made.

Part V

Digital Logic Foundations

Chapter 15

Combinational Logic

Every computation that the SAMPO processor performs—every addition, every comparison, every bitwise mask—reduces to a network of logic gates whose outputs depend only on their current inputs. There is no memory here, no notion of "before" or "after." Change an input, and the output changes. This is *combinational logic*, the foundation on which we will build everything else.

In this chapter, we start from individual logic gates and work our way up through Boolean algebra, multiplexers, and adders until we arrive at the concept of an arithmetic logic unit. By the end, you will understand how a handful of gate types can be combined to perform any operation the SAMPO instruction set demands.

15.1 Logic Gates: The Primitive Building Blocks

Digital circuits operate on two voltage levels that we represent as `0` (low, false) and `1` (high, true). A *logic gate* is a circuit element that takes one or more binary inputs and produces a single binary output according to a fixed rule. Every digital system—from a wristwatch to a supercomputer—is built from these primitives.

15.1.1 The NOT Gate (Inverter)

The NOT gate is the simplest possible gate: it has one input and one output. The output is the logical complement of the input. If the input is `1`, the output is `0`, and vice versa.

In Boolean algebra, we write the NOT of A as $\overline{A}$ or A'. In Verilog, we write `~a`. The SAMPO ALU uses the NOT operation directly—look at the `ALU_NOT` case in `verilog/rtl/alu.v`:

Listing 15.1: The NOT operation in the SAMPO ALU

Table 15.1: NOT gate truth table

A	$\overline{A}$
0	1
1	0

```
`ALU_NOT: begin
    result = ~a;
end
```

The ~ operator performs a bitwise NOT across all 16 bits simultaneously. Each bit of the input a passes through its own independent inverter, producing 16 inverted output bits in parallel.

15.1.2 The AND Gate

The AND gate outputs 1 only when *all* of its inputs are 1. For a two-input AND gate:

Table 15.2: AND gate truth table

A	**B**	**A · B**
0	0	0
0	1	0
1	0	0
1	1	1

We write the AND of A and B as $A \cdot B$ (or sometimes AB). In Verilog: `a & b`.

The AND operation appears directly in the SAMPO instruction set as the **AND** instruction. In the ALU implementation:

Listing 15.2: The AND operation in the SAMPO ALU

```
`ALU_AND: begin
    result = a & b;
    // C and V cleared for logical ops
end
```

Notice that logical operations like AND clear the carry and overflow flags. This is a deliberate design choice: logical operations have no arithmetic meaning for carry or overflow, so the SAMPO zeroes them to give the programmer a clean slate.

15.1.3 The OR Gate

The OR gate outputs 1 when *at least one* of its inputs is 1:

Table 15.3: OR gate truth table

A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	1

We write OR as $A + B$ in Boolean algebra (not to be confused with arithmetic addition). In Verilog: `a | b`. The SAMPO's **OR** instruction maps directly:

Listing 15.3: The OR operation in the SAMPO ALU

```
`ALU_OR: begin
    result = a | b;
end
```

15.1.4 The XOR Gate

The XOR (exclusive OR) gate outputs 1 when its inputs *differ*:

Table 15.4: XOR gate truth table

A	B	A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0

We write XOR as $A \oplus B$. In Verilog: `a ^ b`. The XOR gate is one of the most useful gates in digital design. It detects differences between bits, which makes it essential for comparison, parity checking, and—as we will see shortly—arithmetic.

Listing 15.4: The XOR operation in the SAMPO ALU

```
`ALU_XOR: begin
    result = a ^ b;
end
```

XOR for Zeroing a Register

A register XORed with itself always produces zero. Many assembly programmers use `XOR R4, R4` as a compact way to clear a register. The SAMPO has `R0` hardwired to zero, so this trick is less necessary—but understanding *why* it works comes down to the XOR truth table: every bit compared with itself produces `0`.

15.1.5 The NAND and NOR Gates

A NAND gate is an AND gate followed by an inverter: its output is `0` only when all inputs are `1`. A NOR gate is an OR gate followed by an inverter: its output is `1` only when all inputs are `0`.

Table 15.5: NAND and NOR truth tables

A	B	$\overline{A \cdot B}$ (NAND)	$\overline{A + B}$ (NOR)
0	0	1	1
0	1	1	0
1	0	1	0
1	1	0	0

Functional Completeness

A set of logic gates is *functionally complete* if every possible Boolean function can be built from those gates alone. Both the NAND gate and the NOR gate are individually functionally complete. Any digital circuit—including the entire SAMPO CPU—could, in principle, be constructed using nothing but NAND gates. In practice, synthesis tools for FPGAs map designs to lookup tables (LUTs), which can implement any Boolean function of their inputs directly.

While the SAMPO ALU does not have explicit NAND or NOR instructions, these gates are fundamental inside the FPGA fabric. When the synthesis tool maps our Verilog code to the ECP5 FPGA, every operation ultimately reduces to 4-input lookup tables—each of which can implement any Boolean function of four variables, including NAND and NOR.

15.2 Boolean Algebra and Truth Tables

Logic gates are the hardware; Boolean algebra is the mathematics that describes their behavior. Developed by George Boole in the 1840s—over a century before the first digital computer—Boolean algebra gives us a formal language for reasoning about logic circuits.

15.2.1 Fundamental Laws

Boolean algebra obeys a set of identities that let us simplify expressions. Many of these mirror the laws of ordinary algebra, but a few are unique to the binary domain.

Table 15.6: Laws of Boolean algebra

Law	AND Form	OR Form
Identity	$A \cdot 1 = A$	$A + 0 = A$
Null	$A \cdot 0 = 0$	$A + 1 = 1$
Idempotent	$A \cdot A = A$	$A + A = A$
Complement	$A \cdot \overline{A} = 0$	$A + \overline{A} = 1$
Commutative	$A \cdot B = B \cdot A$	$A + B = B + A$
Associative	$(A \cdot B) \cdot C = A \cdot (B \cdot C)$	$(A + B) + C = A + (B + C)$
Distributive	$A \cdot (B + C) = AB + AC$	$A + BC = (A + B)(A + C)$
Absorption	$A \cdot (A + B) = A$	$A + AB = A$

15.2.2 De Morgan's Theorems

Two identities deserve special attention because they let us convert between AND and OR operations—or between NAND and NOR:

$$\overline{A \cdot B} = \overline{A} + \overline{B} \tag{15.1}$$

$$\overline{A + B} = \overline{A} \cdot \overline{B} \tag{15.2}$$

De Morgan's theorems are not merely theoretical. When the SAMPO ALU needs to compute the `NOT` of a 16-bit value, each bit is independently inverted. But when the synthesis tool optimizes our Verilog into FPGA logic, it regularly applies De Morgan's theorems to rearrange expressions so they fit more efficiently into the available lookup tables.

15.2.3 Truth Tables and Minterm Expansion

A truth table completely specifies a Boolean function by listing the output for every possible combination of inputs. For n inputs, the table has 2^n rows. Each row where the output is 1 corresponds to a *minterm*—a product (AND) of all the input variables, where each variable appears in either true or complemented form.

Let us work through a concrete example. Suppose we want a function $F(A, B, C)$ that outputs 1 when exactly two of the three inputs are 1:

Table 15.7: Truth table for "exactly two of three" function

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Reading the 1 rows, we get the sum-of-products (SOP) expression:

$$F = \overline{A} \cdot B \cdot C + A \cdot \overline{B} \cdot C + A \cdot B \cdot \overline{C} \tag{15.3}$$

This function could be implemented directly in Verilog:

Listing 15.5: "Exactly two of three" in Verilog

```
wire f = (~a & b & c) | (a & ~b & c) | (a & b & ~c);
```

15.2.4 Karnaugh Maps

For functions of up to four or five variables, a Karnaugh map (K-map) provides a visual method for finding minimal expressions. Adjacent cells in a K-map differ by exactly one input variable, so grouping adjacent 1 cells reveals terms that can be simplified by the complement law ($A + \overline{A} = 1$).

K-maps are less relevant when writing Verilog for an FPGA—the synthesis tool handles optimization automatically—but understanding them builds intuition for how complex expressions simplify. When you look at the synthesis report and see that a 6-input function was packed into a single LUT, the underlying mathematics is the same minimization technique.

From Gates to LUTs

On the ECP5 FPGA that hosts the SAMPO, the fundamental logic element is a 4-input lookup table (LUT4). A LUT4 stores a 16-bit truth table in a small SRAM. Any Boolean function of four variables—no matter how complex—fits in a single LUT. The synthesis tool's job is to decompose larger functions into a network of LUT4s and route the connections between them.

15.3 Multiplexers and Decoders

Armed with logic gates and Boolean algebra, we can now build two of the most important combinational building blocks: multiplexers and decoders. These components appear throughout the SAMPO design—from the register file's read ports to the ALU's operation selector.

15.3.1 The Multiplexer

A *multiplexer* (mux) selects one of several inputs and routes it to a single output. A 2-to-1 multiplexer has two data inputs (D_0 and D_1), one select input (S), and one output (Y):

$$Y = \overline{S} \cdot D_0 + S \cdot D_1 \tag{15.4}$$

When $S = 0$, the output follows D_0. When $S = 1$, the output follows D_1. This is the hardware equivalent of an `if-else` statement or a ternary operator.

Table 15.8: 2-to-1 multiplexer truth table

S	Y	Source
0	D_0	Input 0 selected
1	D_1	Input 1 selected

In Verilog, the conditional operator `? :` maps directly to a multiplexer. The SAMPO register file in `verilog/rtl/regfile.v` uses exactly this pattern to enforce the hardwired-zero behavior of `R0`:

Listing 15.6: Multiplexer enforcing `R0` = 0 in the register file

```
// Read ports - R0 always returns 0
assign rd_data1 = (rd_addr1 == 4'h0) ? 16'h0000 : regs[rd_addr1];
assign rd_data2 = (rd_addr2 == 4'h0) ? 16'h0000 : regs[rd_addr2];
```

This is a 2-to-1 multiplexer: when the address equals zero, the output is hardwired to `0x0000`; otherwise, it passes through the value stored in the register array. The synthesis tool implements this as a bank of 16 parallel 2-to-1 multiplexers, one per bit of the 16-bit data path.

Wider Multiplexers

A 4-to-1 multiplexer uses a 2-bit select signal to choose among four inputs. An 8-to-1 mux uses a 3-bit select. In general, a 2^n-to-1 multiplexer requires an n-bit select.

Wider multiplexers can be built from a tree of 2-to-1 multiplexers, or they can be expressed directly using Verilog's `case` statement. The SAMPO ALU is, at its core, a large multiplexer: the 4-bit `op` signal selects among 16 possible operations, and the corresponding result is routed to the output.

Listing 15.7: The ALU as a 16-way multiplexer (simplified)

```
always @(*) begin
    case (op)
        `ALU_ADD:    result = add_result[15:0];
        `ALU_SUB:    result = sub_result[15:0];
        `ALU_AND:    result = a & b;
        `ALU_OR:     result = a | b;
        `ALU_XOR:    result = a ^ b;
        // ... 11 more operations
        default:     result = 16'h0000;
    endcase
end
```

The synthesis tool converts this `case` statement into a tree of multiplexers. Each of the 16 possible results is computed in parallel by dedicated combinational logic, and then the `op` signal selects which result reaches the output.

15.3.2 The Decoder

A *decoder* is the inverse of a multiplexer: it takes an n-bit input and activates exactly one of 2^n output lines. A 2-to-4 decoder, for example, has 2 input bits and 4 outputs. If the input is `10` (decimal 2), then output line 2 goes high and all others stay low.

Table 15.9: 2-to-4 decoder truth table

A_1	A_0	Y_0	Y_1	Y_2	Y_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Each output is a minterm of the inputs:

$$Y_0 = \overline{A_1} \cdot \overline{A_0} \tag{15.5}$$
$$Y_1 = \overline{A_1} \cdot A_0 \tag{15.6}$$
$$Y_2 = A_1 \cdot \overline{A_0} \tag{15.7}$$
$$Y_3 = A_1 \cdot A_0 \tag{15.8}$$

Decoders appear wherever we need to convert a binary-encoded signal into a one-hot representation. In the SAMPO, the instruction decoder converts the 4-bit primary opcode field (bits 15:12 of the instruction word) into control signals that activate different parts of the data path. Each opcode value "selects" a unique combination of control signals, much like a decoder activating a specific output line.

In the RAM module (`verilog/rtl/ram.v`), the word address performs a similar decoding function—selecting one of 32,768 memory locations:

Listing 15.8: Address decoding in the SAMPO RAM module

```
// Word-aligned address (ignore LSB)
wire [14:0] word_addr = addr[15:1];

// Synchronous read and write
always @(posedge clk) begin
    if (en) begin
        if (we) begin
            // Write with byte enables
            if (be[0]) mem[word_addr][7:0]  <= wdata[7:0];
            if (be[1]) mem[word_addr][15:8] <= wdata[15:8];
        end
        rdata <= mem[word_addr];
    end
end
```

The `word_addr` signal acts as a 15-bit decoder input: it selects exactly one of the $2^{15} = 32{,}768$ memory locations in the `mem` array. The `be` (byte enable) signals provide an additional level of decoding, selecting which byte within the 16-bit word gets written.

Little-Endian Addressing

The SAMPO uses little-endian byte ordering: the least-significant byte of a 16-bit word is stored at the lower address. The RAM module's `addr[15:1]` extracts the word address by discarding the least-significant bit, and the byte enable signal `be[0]` controls the low byte while `be[1]` controls the high byte.

15.4 Adders: From Half to Ripple-Carry

Arithmetic is the bread and butter of a CPU. The **ADD** instruction, address calculations for load and store, branch target computation, stack pointer adjustment—all of these require binary addition. Let us build an adder from first principles.

15.4.1 The Half Adder

A *half adder* adds two single-bit inputs and produces a sum bit and a carry bit. It is called "half" because it has no input for a carry from a previous stage.

Table 15.10: Half adder truth table

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Look carefully at the Sum column: it matches the XOR truth table exactly. The Carry column matches the AND truth table. So a half adder is nothing more than an XOR gate and an AND gate:

$$\text{Sum} = A \oplus B \tag{15.9}$$

$$\text{Carry} = A \cdot B \tag{15.10}$$

Listing 15.9: A half adder in Verilog

```
wire sum   = a ^ b;
wire carry = a & b;
```

Two gates. That is all it takes to add two single-bit numbers. The XOR gate computes the sum (because $0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1$, and $1 + 1 = 0$ with a carry), and the AND gate detects when both inputs are 1—the only case where a carry is generated.

15.4.2 The Full Adder

A *full adder* extends the half adder by accepting a carry input (C_{in}) from the previous bit position. It adds three single-bit values: A, B, and C_{in}.

Table 15.11: Full adder truth table

A	**B**	C_{in}	**Sum**	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

The equations for a full adder:

$$\text{Sum} = A \oplus B \oplus C_{\text{in}} \tag{15.11}$$

$$C_{\text{out}} = (A \cdot B) + (C_{\text{in}} \cdot (A \oplus B)) \tag{15.12}$$

A full adder can be constructed from two half adders and an OR gate. The first half adder computes the partial sum $A \oplus B$ and partial carry $A \cdot B$. The second half adder adds C_{in} to the partial sum, producing the final sum and another carry. The OR gate combines both partial carries into the final C_{out}.

Listing 15.10: A full adder in Verilog

```
wire sum   = a ^ b ^ c_in;
wire c_out = (a & b) | (c_in & (a ^ b));
```

15.4.3 The Ripple-Carry Adder

To add multi-bit numbers, we chain full adders together. The carry output of bit position i feeds into the carry input of bit position $i + 1$. This is a *ripple-carry adder*—the carry signal "ripples" through the chain from the least significant bit to the most significant.

For a 16-bit adder like the one inside the SAMPO, we need 16 full adders connected in series. The carry output of bit 0 feeds into the carry input of bit 1, which feeds into bit 2, and so on through bit 15. The final carry output from bit 15 becomes the carry flag.

Ripple-Carry Adder

A *ripple-carry adder* is a chain of n full adders where the carry output of each stage feeds the carry input of the next. For an n-bit ripple-carry adder, the worst-case delay is $O(n)$ gate delays, because the carry must propagate through every stage before the final result is valid.

In Verilog, we do not need to instantiate individual full adders. The `+` operator tells the synthesis tool to generate an adder, and it will choose the most efficient architecture for the target device. The SAMPO ALU uses 17-bit addition to capture the carry:

Listing 15.11: 17-bit addition with carry capture in the SAMPO ALU

```
// Internal signals for 17-bit arithmetic (to capture carry)
wire [16:0] add_result;
wire [16:0] sub_result;

// Calculate extended results
assign add_result = {1'b0, a} + {1'b0, b};
assign sub_result = {1'b0, a} - {1'b0, b};
```

The expression `{1'b0, a}` zero-extends the 16-bit input `a` to 17 bits by concatenating a zero bit at the top. When two 17-bit values are added, the 17th bit of the result is the carry output. This is a standard Verilog idiom for capturing carry without manually instantiating adder cells.

Propagation Delay in Ripple-Carry Adders

In a 16-bit ripple-carry adder, the carry must propagate through all 16 stages. If each full adder has a gate delay of t, the worst-case delay is $16t$—this occurs when adding values like `0xFFFF + 0x0001`, where the carry propagates all the way from bit 0 to bit 15. Faster adder architectures like carry-lookahead and carry-select adders exist to reduce this delay, but for the SAMPO running at 25 MHz on the ECP5 FPGA, a ripple-carry adder is more than fast enough.

15.4.4 Subtraction via Two's Complement

Subtraction does not require a separate circuit. In two's complement arithmetic, subtracting B from A is equivalent to adding A and the bitwise complement of B, then adding 1:

$$A - B = A + \overline{B} + 1 \tag{15.13}$$

The "+1" is accomplished by feeding a 1 into the carry input of the least-significant full adder. So a single adder, an inverter bank, and a multiplexer on the B input can perform both addition and subtraction.

The SAMPO ALU treats subtraction separately in Verilog for clarity, using the - operator:

Listing 15.12: Subtraction with ARM-style carry semantics in the SAMPO ALU

```
`ALU_SUB: begin
    result = sub_result[15:0];
    // Carry is set when there's NO borrow (a >= b for unsigned)
    flag_c = ~sub_result[16];
    flag_v = sub_overflow;
end
```

ARM-Style Carry for Subtraction

The SAMPO uses ARM-style carry semantics for subtraction: the carry flag is set to 1 when there is *no borrow* ($A \geq B$ for unsigned values). This is the complement of the raw borrow bit from the subtractor. The expression `flag_c = ~sub_result[16]` inverts the borrow bit to produce the ARM-style carry. This convention means that after **SUB**, the carry flag can be tested directly for unsigned comparisons: $C = 1$ means $A \geq B$.

15.5 The Arithmetic Logic Unit Concept

We have now seen all the ingredients: logic gates for Boolean operations, adders for arithmetic, and multiplexers for selection. An *arithmetic logic unit* combines all of these into a single module that can perform any of several operations on its inputs, controlled by an operation select signal.

15.5.1 ALU Architecture

At its core, an ALU is a collection of parallel computation circuits—one for each supported operation—followed by a large multiplexer that selects the appropriate result. The inputs flow simultaneously into the adder, the subtractor, the AND gates, the OR gates, and every other functional unit. All results are computed at the same time. The operation select signal then picks which result to pass through to the output.

The SAMPO ALU in `verilog/rtl/alu.v` follows this pattern exactly. Its interface consists of:

Listing 15.13: The SAMPO ALU port declaration

```
module alu (
    input  wire [15:0] a,
```

```
    input  wire [15:0] b,
    input  wire [3:0]  op,
    input  wire        carry_in,
    output reg  [15:0] result,
    output wire        flag_n,
    output wire        flag_z,
    output reg         flag_c,
    output reg         flag_v
);
```

The 16-bit inputs `a` and `b` are the two operands. The 4-bit `op` signal selects one of up to 16 operations. The `carry_in` signal supports multi-precision arithmetic (for the **ADC** and **SBC** instructions). The outputs include the 16-bit `result` and four status flags.

15.5.2 Operations and the Select Signal

The 4-bit `op` signal is defined in `verilog/rtl/sampo_pkg.vh`. Each value corresponds to a distinct ALU operation:

Table 15.12: ALU operation codes from `sampo_pkg.vh`

Code	Name	Operation
`0x0`	`ALU_ADD`	$A + B$
`0x1`	`ALU_SUB`	$A - B$
`0x2`	`ALU_AND`	$A \wedge B$ (bitwise AND)
`0x3`	`ALU_OR`	$A \vee B$ (bitwise OR)
`0x4`	`ALU_XOR`	$A \oplus B$ (bitwise XOR)
`0x5`	`ALU_SLL`	$A \ll B[3{:}0]$
`0x6`	`ALU_SRL`	$A \gg B[3{:}0]$ (logical)
`0x7`	`ALU_SRA`	$A \ggg B[3{:}0]$ (arithmetic)
`0x8`	`ALU_MUL`	$(A \times B)[15{:}0]$ (signed)
`0x9`	`ALU_MULH`	$(A \times B)[31{:}16]$ (signed)
`0xA`	`ALU_DIV`	$A \div B$ (signed)
`0xB`	`ALU_REM`	$A \bmod B$ (signed)
`0xC`	`ALU_PASS_A`	Pass A through
`0xD`	`ALU_PASS_B`	Pass B through
`0xE`	`ALU_NOT`	$\sim A$ (bitwise NOT)
`0xF`	`ALU_NEG`	$0 - A$ (two's complement negate)

This is a richer ALU than many educational designs, which typically support only addition, subtraction, and a few logical operations. The SAMPO's ALU handles multiplication, division, remainder, shift operations, and pass-through modes—all as combinational functions.

Pass-Through Operations

The `ALU_PASS_A` and `ALU_PASS_B` operations route an input directly to the output without modification. These are used by instructions that need to move data through the ALU without performing computation—for example, the **`MOV`** instruction uses `ALU_PASS_B` to copy one register's value to another.

15.5.3 Flag Generation

An ALU does not merely compute results—it also produces status flags that describe properties of the result. The SAMPO generates four flags on every ALU operation:

N (Negative):
Set to the most significant bit of the result. In two's complement, this indicates a negative value.

Z (Zero):
Set when the result is exactly zero.

C (Carry):
Set when the operation produces a carry out (addition) or no borrow (subtraction).

V (Overflow):
Set when the result overflows the signed range—when the sign of the result is inconsistent with the signs of the operands.

The N and Z flags are derived purely from the result and are the same for every operation:

Listing 15.14: N and Z flag generation in the SAMPO ALU

```
// N flag: negative (MSB of result)
assign flag_n = result[15];

// Z flag: zero
assign flag_z = (result == 16'h0000);
```

These are **`assign`** statements—continuous assignments that update whenever the result changes. The N flag is a single wire tap on bit 15. The Z flag is a 16-input NOR: the result is zero when *none* of its bits are set. (Recall that NOR outputs 1 only when all inputs are `0`—this is exactly the zero-detection condition.)

The C and V flags are operation-specific and are set within each `case` branch. For addition:

Listing 15.15: Carry and overflow detection for addition

```
// Overflow detection for addition:
// when signs of operands match but result sign differs
wire add_overflow = (a[15] == b[15]) && (add_result[15] != a[15]);
```

Signed overflow occurs when adding two positive numbers produces a negative result, or adding two negative numbers produces a positive result. The expression checks whether the operand signs agree (`a[15] == b[15]`) and the result sign differs (`add_result[15] != a[15]`).

For subtraction, overflow occurs when the operand signs *differ* and the result sign matches the subtrahend:

Listing 15.16: Overflow detection for subtraction

```
// Overflow detection for subtraction:
// when signs of operands differ and result sign matches b
wire sub_overflow = (a[15] != b[15]) && (sub_result[15] == b[15]);
```

15.5.4 A Complete Combinational Pipeline

The entire SAMPO ALU is a purely combinational circuit. It contains no flip-flops, no clock input, no state. Change any input (`a`, `b`, `op`, or `carry_in`), and the `result` and flag outputs settle to their new values after the propagation delay of the longest path through the circuit.

This is the hallmark of combinational logic: the outputs are a pure function of the current inputs. There is no internal state to remember, no history to track. The ALU computes fresh answers every time the inputs change.

In Chapter 18, we will examine the SAMPO ALU implementation in exhaustive detail—every operation, every edge case, every flag. But the concept is what matters here: an ALU is a combinational circuit built from the gates, adders, and multiplexers we have studied in this chapter, combined into a single module that serves as the computational heart of the processor.

Everything Happens at Once

In the SAMPO ALU, the adder, subtractor, AND, OR, XOR, multiplier, and all other functional units compute their results *simultaneously*. All 16 operations produce outputs in parallel. The 4-bit `op` signal merely selects which of these pre-computed results reaches the `result` output. This is fundamentally different from software, where operations execute one at a time. In hardware, parallelism is the default.

15.6 Exercises

1. **Gate-level XOR.** Using only AND, OR, and NOT gates, derive a gate-level implementation of the XOR function $A \oplus B$. Write the Boolean expression, draw the gate diagram, and verify it against the XOR truth table. Then express your design in Verilog using only the `&`, `|`, and `~` operators (no `^`).

2. **4-bit ripple-carry adder.** Write a Verilog module that instantiates four full adders (define a `full_adder` module first) and chains them together to form a 4-bit ripple-carry adder. The module should have inputs `a[3:0]`, `b[3:0]`, and `c_in`, and outputs `sum[3:0]` and `c_out`. Write a testbench that exercises all corner cases: $0 + 0$, $15 + 1$, $8 + 8$, $15 + 15$.

3. **Overflow detector.** The SAMPO ALU detects signed overflow using the expression `(a[15] == b[15]) && (add_result[15] != a[15])`. Prove that this expression correctly detects overflow for all possible combinations of operand signs and result sign. Construct a truth table with columns for `a[15]`, `b[15]`, and `result[15]`, and verify that the overflow column matches the expected behavior.

4. **ALU extension.** Add a new operation to the SAMPO ALU: a *bit count* function (`ALU_POPCNT`) that counts the number of 1 bits in operand `a`. Write the Verilog for this operation as a combinational expression (hint: summing all 16 individual bits works, though more efficient tree-based approaches exist). Test it in a simulation by modifying the ALU testbench in `verilog/tb/alu_tb.v`.

5. **Boolean simplification.** The following Boolean expression can be simplified using the laws from Table 15.6 and De Morgan's theorems: $F = \overline{(\overline{A} \cdot B)} \cdot (A + \overline{B})$. Simplify F step by step, showing which law you apply at each step. Verify your answer by constructing truth tables for both the original and simplified expressions.

What's Next. With combinational logic under our belt, we can compute any function of the current inputs—but we cannot remember anything. In Chapter 16, we add the dimension of time: flip-flops that capture values on clock edges, registers that group flip-flops into word-width storage elements, and RAM that provides the SAMPO's 64 KB address space. These sequential elements are what transform a collection of gates into a machine that can execute a sequence of instructions.

Chapter 16

Sequential Logic

In Chapter 15, we built circuits whose outputs depend only on their current inputs. Change an input, and the output changes—instantly (after propagation delay) and with no memory of what came before. But a CPU must remember things: the current instruction, the contents of its registers, the values stored in RAM. A CPU must also execute instructions in order, one after another, synchronized to a master clock. This requires *sequential logic*—circuits whose outputs depend not only on the current inputs but also on the circuit's internal *state*.

The difference between combinational and sequential logic is the difference between a calculator and a computer. A calculator performs a single operation on its inputs. A computer remembers intermediate results, loops through instructions, and builds complex behaviors from sequences of operations. The key to all of this is a tiny circuit called a *flip-flop*.

16.1 Flip-Flops: The Memory Element

A *flip-flop* is a circuit that stores a single bit. It has a data input, a clock input, and an output that holds the stored value. On each rising edge of the clock, the flip-flop captures whatever value is present at its data input and holds it stable at its output until the next clock edge.

Before we get to the workhorse D flip-flop, let us trace the evolution from simpler memory elements.

16.1.1 The SR Latch

The simplest memory element is the *SR latch* (Set-Reset latch), built from two cross-coupled NOR gates or two cross-coupled NAND gates. The NOR-based version has two inputs: S (set) and R (reset).

Table 16.1: SR latch truth table (NOR implementation)

S	R	Q	Action
0	0	Q_{prev}	Hold (no change)
0	1	0	Reset
1	0	1	Set
1	1	—	Invalid (forbidden)

When $S = 1$ and $R = 0$, the output Q is forced to 1. When $S = 0$ and $R = 1$, the output is forced to 0. When both inputs are 0, the latch *holds* its previous value—this is the memory behavior. The combination $S = 1, R = 1$ is forbidden because it attempts to both set and reset simultaneously, producing undefined behavior.

The SR latch is level-sensitive: its output can change at any time the inputs change. This makes it difficult to use in synchronous designs where we need changes to happen at precise moments. The solution is to add a clock.

16.1.2 The D Flip-Flop

The *D flip-flop* (data flip-flop) is the fundamental storage element of modern digital design. It has a data input D, a clock input, and an output Q. On the rising edge of the clock, the flip-flop samples D and transfers it to Q. Between clock edges, Q holds its value regardless of what happens at D.

Table 16.2: D flip-flop behavior

Clock Edge	Q_{next}	Action
Rising	D	Capture input
No edge	Q_{prev}	Hold

In Verilog, a D flip-flop is described using an `always` block sensitive to the positive clock edge:

Listing 16.1: A D flip-flop in Verilog

```
reg q;
always @(posedge clk) begin
    q <= d;
end
```

The `posedge clk` means "on the rising edge of the clock." The non-blocking assignment `<=` is used (not blocking `=`) because this is sequential logic—we will discuss the critical distinction between blocking and non-blocking assignment in Chapter 17.

Most practical flip-flops also include an asynchronous reset input. When the reset signal is asserted (typically active-low in the SAMPO design), the flip-flop output is forced to a known value regardless of the clock:

Listing 16.2: D flip-flop with asynchronous active-low reset

```
reg q;
always @(posedge clk or negedge rst_n) begin
    if (!rst_n)
        q <= 1'b0;
    else
        q <= d;
end
```

This pattern appears throughout the SAMPO design. The `rst_n` signal (active-low, indicated by the `_n` suffix) ensures that all flip-flops start in a known state after power-up or a manual reset.

16.1.3 The JK Flip-Flop

The *JK flip-flop* resolves the SR latch's forbidden state. It has two inputs, J and K, and toggles its output when both are asserted simultaneously:

Table 16.3: JK flip-flop truth table

J	K	Q_{next}	Action
0	0	Q_{prev}	Hold
0	1	0	Reset
1	0	1	Set
1	1	$\overline{Q_{\text{prev}}}$	Toggle

The toggle behavior makes the JK flip-flop useful for building counters and frequency dividers. However, in FPGA-based designs like the SAMPO, the D flip-flop is the native storage element—the FPGA fabric provides D flip-flops directly, and any JK behavior is synthesized from D flip-flops with additional logic.

Flip-Flops in FPGAs

The ECP5 FPGA that hosts the SAMPO provides a D flip-flop in every logic slice, paired with a lookup table (LUT). The synthesis tool maps all sequential elements in our Verilog to these physical D flip-flops. A JK flip-flop, if described in Verilog, would be implemented as a D flip-flop with a small LUT computing the next-state logic: $D = J \cdot \overline{Q} + \overline{K} \cdot Q$.

16.2 Registers: Groups of Flip-Flops

A single flip-flop stores one bit. To store a 16-bit value—the natural word size of the SAMPO—we need 16 flip-flops working in parallel, all clocked by the same signal. This group of flip-flops is a *register*.

16.2.1 A 16-Bit Register

A 16-bit register with synchronous load and asynchronous reset looks like this in Verilog:

Listing 16.3: A 16-bit register with write enable

```
reg [15:0] data;
always @(posedge clk or negedge rst_n) begin
    if (!rst_n)
        data <= 16'h0000;
    else if (wr_en)
        data <= wr_data;
end
```

The `wr_en` (write enable) signal controls when new data is captured. When `wr_en` is low, the register holds its current value—the flip-flops simply recirculate their outputs back to their inputs. When `wr_en` goes high, the next rising clock edge captures `wr_data` into the register.

16.2.2 The SAMPO Register File

The SAMPO has 16 general-purpose registers (`R0` through `R15`), each 16 bits wide. These are organized into a *register file*—a small, fast memory with multiple read and write ports. The implementation lives in `verilog/rtl/regfile.v`.

Let us look at the register storage declaration:

Listing 16.4: Register storage in the SAMPO register file

```
// Main register file (R0-R15)
reg [15:0] regs [0:15];

// Alternate registers (R4'-R11', stored as indices 0-7)
reg [15:0] alt_regs [0:7];
```

The `regs` array is a bank of 16 registers, each 16 bits wide. That is $16 \times 16 = 256$ flip-flops for the main register set. The `alt_regs` array provides 8 additional "shadow" registers for the **EXX** instruction, adding another $8 \times 16 = 128$ flip-flops. The total register file requires 384 flip-flops.

The register file provides two simultaneous read ports:

Listing 16.5: Dual read ports in the SAMPO register file

```
// Read ports - R0 always returns 0
assign rd_data1 = (rd_addr1 == 4'h0) ? 16'h0000 : regs[rd_addr1];
assign rd_data2 = (rd_addr2 == 4'h0) ? 16'h0000 : regs[rd_addr2];
```

These are *combinational* read ports—the output updates immediately when the address changes, without waiting for a clock edge. Each read port is a 16-to-1 multiplexer: the 4-bit address selects one of 16 registers, and the selected 16-bit value appears at the output. The special case for `R0` forces the output to zero regardless of what is stored in `regs[0]`.

Writing, however, is *sequential*—it happens on the clock edge:

Listing 16.6: Write logic in the SAMPO register file

```
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        // Reset all registers to 0
        for (i = 0; i < 16; i = i + 1) begin
            regs[i] <= 16'h0000;
        end
        for (i = 0; i < 8; i = i + 1) begin
            alt_regs[i] <= 16'h0000;
        end
    end else begin
        // Handle EXX: swap R4-R11 with alternate set
        if (exx) begin
            for (i = 0; i < 8; i = i + 1) begin
                regs[i + 4] <= alt_regs[i];
                alt_regs[i] <= regs[i + 4];
            end
        end
        else begin
            if (wr_en && (wr_addr != 4'h0)) begin
                regs[wr_addr] <= wr_data;
            end
            if (sp_wr_en) begin
                regs[2] <= sp_wr_data;  // R2 = SP
            end
        end
    end
end
```

Several details in this code are worth studying closely:

- **Reset loop:** On reset (`!rst_n`), every register is cleared to zero using a `for` loop. The synthesis tool unrolls this into 24 parallel reset connections—one for each register in both the main and alternate sets.
- **Write protection for `R0`:** The condition `wr_addr != 4'h0` prevents any write to `R0`. Combined with the read-port multiplexer that forces `R0`'s output to zero, this guarantees that `R0` always reads as zero, regardless of what instructions attempt.
- **EXX swap:** The **`EXX`** instruction swaps `R4` through `R11` with their alternates. Because non-blocking assignment (`<=`) is used, all 16 swaps happen simultaneously on the same clock edge. The right-hand side values are all read *before* any writes take effect.
- **Dual write ports:** The register file has a dedicated second write port for the stack pointer (`R2`). This allows **`PUSH`** and **`POP`** instructions to update both the destination register and the stack pointer in a single clock cycle, rather than requiring two separate write cycles.

Register File

A *register file* is a small, multi-ported memory built from flip-flops. Unlike RAM (which is typically single-ported or dual-ported), a register file provides fast, parallel access to a small number of locations. The SAMPO register file has two read ports and two write ports, allowing the CPU to read two operands and write a result plus an updated stack pointer in a single clock cycle.

16.3 Counters and Shift Registers

Flip-flops can be connected in patterns that produce useful behaviors beyond static storage. Two of the most common patterns are *counters* (which cycle through a sequence of values) and *shift registers* (which move data one position per clock cycle).

16.3.1 Binary Counters

A binary counter increments its value by one on each clock edge. An n-bit counter cycles through 2^n states before wrapping around to zero.

Listing 16.7: A 16-bit binary counter in Verilog

```
reg [15:0] count;
always @(posedge clk or negedge rst_n) begin
    if (!rst_n)
        count <= 16'h0000;
```

```
    else
        count <= count + 1;
end
```

The SAMPO uses counters in several places. The program counter (PC) is essentially a counter with extra logic: it increments by 2 (the size of a 16-bit instruction word) during normal execution and loads an arbitrary value when a branch or jump is taken. The cycle counter in the testbench is a straightforward incrementing counter that tracks how many clock cycles the simulation has run:

Listing 16.8: Cycle counter used in the SAMPO testbench

```
wire [31:0] cycles;
```

The program counter is more sophisticated than a basic counter because it must support multiple update modes: increment for sequential execution, load for branches and jumps, and hold during multi-cycle operations. But at its core, it is a register (a group of flip-flops) with combinational logic that computes the next value.

16.3.2 Shift Registers

A *shift register* is a chain of flip-flops where each flip-flop's output feeds the next flip-flop's input. On each clock edge, data moves one position through the chain.

Listing 16.9: An 8-bit shift register in Verilog

```
reg [7:0] shift_reg;
always @(posedge clk) begin
    if (load)
        shift_reg <= parallel_in;
    else if (shift_en)
        shift_reg <= {shift_reg[6:0], serial_in};
end
```

This shift register supports two modes: parallel load (capturing all 8 bits at once) and serial shift (moving each bit one position to the left and inserting `serial_in` at the bottom). Shift registers are the building blocks of serial communication interfaces like UART, where data is transmitted one bit at a time.

In the SAMPO design, UART transmission involves a shift register that is loaded with a byte in parallel and then shifts it out one bit per baud period. The shift operation itself is a sequential process—one flip-flop transfers its value to the next on each clock edge—even though the SAMPO's combinational barrel shifter in the ALU performs shifts in a single cycle.

Combinational vs. Sequential Shifting

The SAMPO has both kinds of shift circuits. The ALU's shift operations (`ALU_SLL`, `ALU_SRL`, `ALU_SRA`) are *combinational*: they shift by any amount in a single clock cycle using a barrel shifter. The UART's shift register is *sequential*: it shifts by one position per clock cycle, taking multiple cycles to transmit a full byte. Both are built from the same fundamental elements—multiplexers and flip-flops—but arranged differently.

16.4 Clock Domains and Timing

Every sequential circuit operates relative to a clock signal—a periodic square wave that defines when state transitions occur. The clock is the heartbeat of a synchronous digital system.

16.4.1 The System Clock

The SAMPO's clock is a single, global signal that drives every flip-flop in the design. On the ULX3S FPGA board, the system oscillator provides a 25 MHz clock, giving a clock period of 40 ns. The testbench in `verilog/tb/sampo_tb.v` generates a faster 100 MHz clock for simulation:

Listing 16.10: Clock generation in the SAMPO testbench

```
// Parameters
parameter CLK_PERIOD = 10;  // 100 MHz

// Clock generation
initial clk = 0;
always #(CLK_PERIOD/2) clk = ~clk;
```

The `always #(CLK_PERIOD/2) clk = ~clk` statement toggles the clock signal every 5 ns (half of 10 ns), producing a 100 MHz square wave. This is simulation-only syntax—on real hardware, the clock comes from a crystal oscillator or PLL.

16.4.2 Clock Edges and Synchronous Design

In a *synchronous design*, all flip-flops are triggered by the same clock edge (typically the rising edge). This guarantees that state transitions happen at well-defined moments, making the circuit's behavior predictable and analyzable.

The SAMPO is a fully synchronous design. Every register update, every memory write, every state machine transition happens on the rising edge of `clk`. Between clock edges, the combinational logic settles to its new values, and the next clock edge captures those values into the flip-flops.

This creates a simple mental model for understanding the CPU's behavior:

1. Rising clock edge: all flip-flops capture their inputs.
2. Combinational logic computes new values based on the flip-flop outputs.
3. The new values propagate through gates and wires, settling before the next clock edge.
4. The next rising clock edge captures the new values.

This cycle repeats indefinitely at the clock frequency, driving the CPU through its state machine and executing instructions.

16.4.3 Multiple Clock Domains

A clock domain is a group of flip-flops driven by the same clock signal. The SAMPO primarily uses a single clock domain, but interfaces to the outside world—such as the UART—operate at different rates. The UART transmitter shifts data out at the baud rate (115,200 bits per second), which is far slower than the 25 MHz system clock.

When signals must cross between clock domains, special synchronization circuits are needed to avoid *metastability*—a condition where a flip-flop captures an input that is changing at the exact moment of the clock edge, causing the output to settle to an unpredictable value.

Metastability

Metastability occurs when a flip-flop's setup or hold time is violated—that is, when the data input changes too close to the clock edge. The flip-flop enters an unstable state between `0` and `1` and may take an arbitrarily long time to resolve. In digital systems, metastability is managed (not eliminated) by using synchronizer chains: two or more flip-flops in series, giving the signal time to resolve to a valid level before it is used by downstream logic.

Single Clock Domain Discipline

For the SAMPO design, we keep things manageable by using a single clock domain for all CPU logic. The UART interface handles rate differences using counters and enables rather than separate clocks. This avoids clock domain crossing issues entirely within the CPU core, leaving synchronization concerns to the board-level interface where external signals enter the FPGA.

16.5 Setup and Hold Time

For a flip-flop to reliably capture its input, the data must be stable for a minimum period *before* the clock edge (*setup time*, t_{su}) and must remain stable for a minimum period *after* the clock edge (*hold time*, t_{h}). If either constraint is violated, the flip-flop may enter metastability.

16.5.1 The Timing Constraint

Consider a pair of flip-flops connected through combinational logic:

1. Flip-flop A captures a value on clock edge n.
2. The output of flip-flop A propagates through combinational logic.
3. Flip-flop B must capture the result on clock edge $n + 1$.

For this to work, the combinational logic must complete its computation within one clock period, minus the setup time of flip-flop B and the clock-to-output delay of flip-flop A:

$$t_{\text{clk-to-Q}} + t_{\text{comb}} + t_{\text{su}} \leq T_{\text{clk}} \tag{16.1}$$

where T_{clk} is the clock period. If the combinational path is too slow, we must either reduce the clock frequency or restructure the logic.

16.5.2 Critical Paths in the SAMPO

The *critical path* is the longest combinational path between any two flip-flops in the design. It determines the maximum clock frequency at which the circuit can operate. In the SAMPO, the critical path typically runs through the ALU—specifically through the carry chain of the adder and the overflow detection logic.

Consider the addition path:

1. The register file outputs propagate operand values.
2. The 16-bit adder computes the sum (carry ripples through 16 stages).
3. Overflow detection logic checks the sign bits.
4. The result and flags propagate to the writeback logic.

On the ECP5 FPGA at 25 MHz, the clock period is 40 ns—generous enough for the SAMPO's combinational paths. Faster FPGAs or higher clock frequencies would require more careful attention to the critical path, potentially adding pipeline stages to break long combinational paths into shorter segments.

16.5.3 Hold Time and Clock Skew

Hold time violations occur when data changes too quickly *after* the clock edge. This can happen when the combinational path between two flip-flops is extremely short (nearly zero delay) and the clock arrives at the destination flip-flop later than the source flip-flop (clock skew).

$$t_{\text{clk-to-Q}} + t_{\text{comb}} \geq t_{\text{h}} + t_{\text{skew}} \tag{16.2}$$

In FPGA designs, the clock distribution network is carefully balanced to minimize skew, and the place-and-route tool checks for hold time violations automatically. If a violation is detected, the tool inserts delay elements (buffer cells) to slow down the data path.

Timing Analysis is Automatic

When we synthesize the SAMPO for the ECP5 FPGA, the `nextpnr-ecp5` place-and-route tool performs static timing analysis automatically. It reports the maximum achievable clock frequency and flags any paths that violate timing constraints. We will see this in action in **??**.

16.6 RAM: SRAM Cells and Address Decoding

Registers provide fast storage for a handful of values, but a CPU needs access to much more memory. The SAMPO has a 64 KB address space, far too large to build from individual flip-flops (that would require $64 \times 1024 \times 8 = 524{,}288$ flip-flops for byte-level storage). Instead, we use *RAM*—random access memory.

16.6.1 SRAM Cells

Static RAM (SRAM) stores each bit in a circuit of cross-coupled inverters (similar to the SR latch), with access transistors controlled by a word line. Unlike dynamic RAM (DRAM), which stores bits as charges on capacitors that must be periodically refreshed, SRAM holds its value as long as power is applied.

Each SRAM cell typically uses six transistors (a "6T cell"):

- Two cross-coupled inverters (4 transistors) form the storage element.
- Two access transistors connect the storage element to the bit lines when the word line is active.

On an FPGA, RAM is not built from individual transistors in the fabric. Instead, the ECP5 provides dedicated *block RAM* (BRAM) resources—pre-built SRAM arrays embedded in the FPGA die. Each ECP5 block RAM is 18 Kbit (2,048 bytes organized as 16-bit words). The synthesis tool maps our Verilog memory declarations to these block RAM resources automatically.

16.6.2 The SAMPO RAM Module

The SAMPO's 64 KB RAM is implemented in `verilog/rtl/ram.v`. Let us examine the complete module:

Listing 16.11: The SAMPO RAM module (`verilog/rtl/ram.v`)

```
module ram #(
    parameter INIT_FILE = ""
) (
    input  wire        clk,
    input  wire [15:0] addr,
    input  wire [15:0] wdata,
    output reg  [15:0] rdata,
    input  wire        we,
    input  wire [1:0]  be,      // Byte enables: [0] = low, [1] = high
    input  wire        en
);

    // 64KB = 32K x 16-bit words
    reg [15:0] mem [0:32767];

    // Word-aligned address (ignore LSB)
    wire [14:0] word_addr = addr[15:1];

    // Initialize memory from hex file if specified
    initial begin
        if (INIT_FILE != "") begin
            $readmemh(INIT_FILE, mem);
        end
    end

    // Synchronous read and write
    always @(posedge clk) begin
        if (en) begin
            if (we) begin
                // Write with byte enables
                if (be[0]) mem[word_addr][7:0]  <= wdata[7:0];
                if (be[1]) mem[word_addr][15:8] <= wdata[15:8];
            end
            // Read (always returns full word)
            rdata <= mem[word_addr];
        end
    end

endmodule
```

This module illustrates several important concepts about RAM design:

Address Decoding

The 64 KB memory is organized as 32,768 words of 16 bits each. Since each word occupies two bytes, the byte address is converted to a word address by discarding the least-significant bit: `wire [14:0] word_addr = addr[15:1]`. This 15-bit word address selects one of the $2^{15} = 32{,}768$ locations in the `mem` array.

The address decoding is implicit in Verilog's array indexing. When we write `mem[word_addr]`, the synthesis tool generates the necessary decoder logic to activate the correct row in the SRAM array.

Byte Enables

The SAMPO supports both word (16-bit) and byte (8-bit) memory access. The 2-bit `be` (byte enable) signal controls which bytes within a word are written:

- `be[0]` enables writing the low byte (bits 7:0).
- `be[1]` enables writing the high byte (bits 15:8).

For a word write (**`SW`** instruction), both byte enables are asserted. For a byte write (**`SB`** instruction), only one enable is asserted. For reads, the full 16-bit word is always returned; byte extraction is handled by the CPU's data path.

Synchronous Operation

The RAM is *synchronous*: both reads and writes occur on the rising edge of the clock. The `rdata` output is registered—it updates one clock cycle after the address is presented. This is in contrast to *asynchronous* RAM, where the read data appears as soon as the address changes (like the register file's read ports).

Synchronous RAM is the natural choice for FPGA designs because the block RAM resources on the ECP5 are inherently synchronous. Attempting to describe asynchronous RAM in Verilog for an FPGA would force the synthesis tool to use fabric LUTs and flip-flops instead of the dedicated block RAM—a wasteful and slower approach.

Memory Initialization

The parameter `INIT_FILE` allows the RAM to be pre-loaded with a program or test data before simulation begins. The **`$readmemh`** system task reads hexadecimal values from a text file and loads them into the memory array. This is how the SAMPO testbench loads the program to be executed:

Listing 16.12: RAM instantiation with initialization in the testbench

```
soc #(
    .RESET_VECTOR(16'h0100),
    .RAM_INIT_FILE(RAM_INIT_FILE)
) uut (
    .clk(clk),
    .rst_n(rst_n),
    // ...
);
```

The `RAM_INIT_FILE` parameter is passed from the testbench through the SoC module down to the RAM module. For simulation, the default is a hello-world program; for ISA tests, it points to the specific test program being verified.

Parameterized Modules

The `#(parameter INIT_FILE = "")` syntax creates a parameterized module—a module template that can be customized at instantiation time. This is one of Verilog's most powerful features for creating reusable designs. We will explore parameters in depth in Chapter 17.

16.6.3 RAM Capacity and Organization

Let us put the SAMPO's memory in perspective:

Table 16.4: Memory hierarchy in the SAMPO

Memory	Size	Entries	Technology	Access
Register file	32 bytes	16 × 16-bit	Flip-flops	Combinational read
Alt. registers	16 bytes	8 × 16-bit	Flip-flops	Swap via **EXX**
Main RAM	65,536 bytes	32,768 × 16-bit	Block RAM	Synchronous

The register file is fastest (combinational read, zero wait states) but smallest. RAM is larger but slower (one cycle read latency). This trade-off between speed and capacity is a fundamental theme in computer architecture, extending all the way up to modern processors with their multi-level cache hierarchies.

16.7 Exercises

1. **Toggle flip-flop.** Write a Verilog module that implements a T (toggle) flip-flop: on each rising clock edge, if the `t` input is high, the output inverts; otherwise, it holds. Include an

asynchronous active-low reset. Write a testbench that drives the toggle input with a pattern and verifies the output. How does this relate to the JK flip-flop with $J = K$?

2. **4-bit counter with enable.** Extend the basic counter from Section 16.3.1 to include a `count_en` input. The counter should only increment when `count_en` is high. Add a `count_dir` input that selects between counting up and counting down. Write a testbench that exercises both modes and tests the wraparound behavior ($15 \to 0$ for up, $0 \to 15$ for down).

3. **Register file analysis.** The SAMPO register file allows two reads and two writes per clock cycle (the main write port and the SP write port). Draw a timing diagram showing a **POP** instruction that reads the stack pointer, loads a value from memory, writes the loaded value to the destination register, and updates the stack pointer—all coordinated across clock cycles. Identify which signals change on which clock edges.

4. **Synchronous vs. asynchronous RAM.** The SAMPO RAM module uses synchronous reads (`rdata` is registered). Modify the module to use asynchronous reads (make `rdata` a continuous assignment instead of a registered output). Simulate both versions and compare the timing. Why does the synchronous version require an extra clock cycle for reads? What are the advantages of synchronous reads for FPGA implementation?

5. **Memory initialization.** Write a Python script that generates a hex file compatible with `$readmemh`. The file should contain 256 words where word n contains the value $n^2 \bmod 65536$. Load this file into the SAMPO RAM module in a testbench and verify that reading address `0x000A` (word 5) returns $5^2 = 25 =$ `0x0019`.

What's Next. We now understand the two fundamental categories of digital logic: combinational circuits that compute functions of their inputs, and sequential circuits that store state and evolve over time. In Chapter 17, we bring these concepts together by learning Verilog—the hardware description language we use to describe the SAMPO's design. We will see how Verilog's module system, continuous assignments, and always blocks map directly to the combinational and sequential circuits we have studied, and we will run our first simulations using Icarus Verilog and GTKWave.

Chapter 17

Verilog Fundamentals

In Chapters 15 and 16, we studied the logic gates, flip-flops, and memory elements that make up digital circuits. Now we need a language to describe these circuits precisely enough that a tool can convert our description into real hardware. That language is *Verilog*.

Verilog is a *hardware description language* (HDL)—not a programming language. This distinction matters. A C program describes a sequence of operations that execute one at a time on a processor. A Verilog module describes a *structure* of interconnected hardware that operates continuously and in parallel. When you write `a & b` in Verilog, you are not instructing a processor to perform an AND operation—you are instantiating AND gates in silicon.

The SAMPO CPU is written entirely in Verilog, spread across eight source files in the `verilog/rtl/` directory. In this chapter, we learn the language constructs that make those files work: modules, wires, registers, assignments, and always blocks. By the end, you will be able to read any file in the SAMPO project and understand what hardware it describes.

17.1 Module Structure: Ports, Wires, and Regs

The fundamental unit of Verilog design is the *module*. A module defines a block of hardware with a name, a set of input and output ports, and internal logic. Modules can instantiate other modules, building up a hierarchy of components.

17.1.1 Module Declaration

Let us examine the SAMPO ALU module declaration in `verilog/rtl/alu.v`:

Listing 17.1: The ALU module declaration

```
module alu (
    input  wire [15:0] a,
    input  wire [15:0] b,
    input  wire [3:0]  op,
    input  wire        carry_in,
    output reg  [15:0] result,
    output wire        flag_n,
    output wire        flag_z,
    output reg         flag_c,
    output reg         flag_v
);
```

Every module begins with the keyword `module` followed by the module name and a port list enclosed in parentheses. The port list declares each signal's direction (`input` or `output`), type (`wire` or `reg`), and width (e.g., `[15:0]` for 16 bits). The module ends with `endmodule`.

17.1.2 Wires vs. Regs

The two data types you will encounter most often are `wire` and `reg`. Despite the name, `reg` does *not* always mean a hardware register:

`wire` Represents a physical connection (a wire). A wire must be driven by a continuous assignment (`assign`) or by the output of a module instance. It cannot be assigned inside an `always` block.

`reg` Represents a variable that can be assigned inside an `always` block. It may or may not become a physical register depending on context: if assigned inside a `posedge`-triggered always block, it synthesizes to a flip-flop. If assigned inside a combinational always block (`always @(*)`), it synthesizes to combinational logic.

Reg Does Not Mean Register

This is the single most confusing aspect of Verilog for newcomers. The keyword `reg` means "a variable that holds its value until reassigned"—it is a *software-level abstraction*, not a hardware declaration. Whether a `reg` becomes a flip-flop or a combinational wire depends entirely on how it is used. In the SAMPO ALU, `result` is declared as `reg` because it is assigned inside a combinational `always @(*)` block—but it synthesizes to pure combinational logic, not a register.

Look at the ALU's outputs: `flag_n` and `flag_z` are declared as `wire` because they are driven by `assign` statements. The `result`, `flag_c`, and `flag_v` outputs are declared as `reg` because they are assigned inside an `always` block. Both groups synthesize to combinational logic—the ALU has no clock input and no flip-flops.

17.1.3 Internal Signals

Inside a module, you declare internal wires and regs to hold intermediate values. The SAMPO ALU declares several internal signals for extended arithmetic:

Listing 17.2: Internal signals in the ALU

```
// Internal signals for 17-bit arithmetic (to capture carry)
wire [16:0] add_result;
wire [16:0] sub_result;
wire [16:0] adc_result;
wire [16:0] sbc_result;

// Signed versions for signed operations
wire signed [15:0] a_signed = $signed(a);
wire signed [15:0] b_signed = $signed(b);
wire signed [31:0] mul_result_signed;
wire        [31:0] mul_result_unsigned;
```

The `wire [16:0]` declaration creates a 17-bit wire—one bit wider than the data path—to capture the carry output from 16-bit addition. The `wire signed` declarations tell the synthesis tool to treat these signals as signed two's complement values, which affects how operations like multiplication and arithmetic shift behave.

17.1.4 Module Instantiation

Modules are connected to each other through *instantiation*. The ALU testbench in `verilog/tb/alu_tb.v` shows how to instantiate the ALU module:

Listing 17.3: Instantiating the ALU in the testbench

```
alu uut (
    .a(a),
    .b(b),
    .op(op),
    .carry_in(carry_in),
    .result(result),
    .flag_n(flag_n),
    .flag_z(flag_z),
    .flag_c(flag_c),
    .flag_v(flag_v)
);
```

The syntax is: `module_name instance_name (.port(signal), ...)`. The instance name `uut` (Unit Under Test) is a convention for testbenches. Each `.port(signal)` connects the module's port to a signal in the enclosing module. This is analogous to plugging wires into the pins of an integrated circuit—you are wiring up a physical component, not calling a function.

The SAMPO testbench in `verilog/tb/sampo_tb.v` instantiates the entire system-on-chip, demonstrating hierarchical module composition:

Listing 17.4: SoC instantiation with parameters in the main testbench

```
soc #(
    .RESET_VECTOR(16'h0100),
    .RAM_INIT_FILE(RAM_INIT_FILE)
) uut (
    .clk(clk),
    .rst_n(rst_n),
    .tx_data(tx_data),
    .tx_valid(tx_valid),
    .tx_ready(tx_ready),
    .rx_data(rx_data),
    .rx_valid(rx_valid),
    .rx_ready(rx_ready),
    .halted(halted),
    .pc(pc),
    .cycles(cycles),
    .led(led)
);
```

The `#(...)` syntax passes parameters to the module, customizing it at instantiation time. Here, the reset vector and RAM initialization file are set for this particular testbench run.

17.2 Continuous Assignment vs. Procedural Blocks

Verilog provides two fundamentally different mechanisms for describing logic: *continuous assignments* and *procedural blocks*. Understanding when to use each is essential to writing correct Verilog.

17.2.1 Continuous Assignment

A continuous assignment uses the `assign` keyword to drive a `wire`. The assignment is "always active"—whenever any signal on the right-hand side changes, the output updates immediately (after propagation delay).

Listing 17.5: Continuous assignments in the SAMPO ALU

```
// Calculate extended results
assign add_result = {1'b0, a} + {1'b0, b};
assign sub_result = {1'b0, a} - {1'b0, b};
assign adc_result = {1'b0, a} + {1'b0, b} + {16'b0, carry_in};
assign sbc_result = {1'b0, a} - {1'b0, b} - {16'b0, carry_in};

// Multiply results
assign mul_result_signed = a_signed * b_signed;
assign mul_result_unsigned = a * b;
```

Each of these `assign` statements describes a piece of combinational hardware that is "always computing." The adder, subtractor, and multiplier circuits all run continuously in parallel, producing updated results whenever any input changes.

The flag outputs use continuous assignment as well:

Listing 17.6: Flag generation via continuous assignment

```
// N flag: negative (MSB of result)
assign flag_n = result[15];

// Z flag: zero
assign flag_z = (result == 16'h0000);
```

These are pure combinational functions: the N flag is a single wire connected to bit 15 of the result, and the Z flag is a 16-input NOR gate (all bits must be zero).

17.2.2 Procedural Blocks

A procedural block uses the `always` keyword and contains statements that execute in sequence (like a traditional programming language). Despite the sequential appearance, the hardware generated depends on the sensitivity list and the assignment style.

The SAMPO ALU uses a combinational procedural block to implement its operation selection:

Listing 17.7: Combinational always block in the ALU

```
always @(*) begin
    // Default values
    result = 16'h0000;
    flag_c = 1'b0;
    flag_v = 1'b0;

    case (op)
```

```
        `ALU_ADD: begin
            result = add_result[15:0];
            flag_c = add_result[16];
            flag_v = add_overflow;
        end
        `ALU_SUB: begin
            result = sub_result[15:0];
            flag_c = ~sub_result[16];
            flag_v = sub_overflow;
        end
        // ... more operations
        default: begin
            result = 16'h0000;
        end
    endcase
end
```

The `@(*)` sensitivity list means "re-evaluate whenever any input changes." This is a combinational block—it describes a network of multiplexers and logic gates, not a sequence of operations.

Sensitivity List

The *sensitivity list* of an `always` block specifies which signals trigger re-evaluation. For combinational logic, use `@(*)` (Verilog-2001 wildcard) to automatically include all signals read within the block. For sequential logic, use `@(posedge clk)` to trigger only on clock edges. An incomplete sensitivity list in a combinational block will cause simulation-synthesis mismatches.

The `initial` block is a special procedural block that executes once at the start of simulation. It is not synthesizable—it exists only for simulation and testbenches:

Listing 17.8: Initial block for memory loading (simulation only)

```
initial begin
    if (INIT_FILE != "") begin
        $readmemh(INIT_FILE, mem);
    end
end
```

Initial Blocks and Synthesis

The `initial` block in the RAM module is used to pre-load memory contents for simulation. Most synthesis tools ignore `initial` blocks, though some FPGA tools (including Yosys) support them for block RAM initialization. In the SAMPO design, the initial memory contents are loaded both in simulation (via `$readmemh`) and on the FPGA (via the synthesis tool's memory initialization support).

17.3 Blocking vs. Non-Blocking Assignment

Verilog has two assignment operators for use inside `always` blocks. Getting this wrong is the number one cause of subtle bugs in Verilog designs.

17.3.1 Blocking Assignment (=)

A *blocking assignment* executes immediately and in order. The right-hand side is evaluated, the left-hand side is updated, and only then does execution proceed to the next statement. Subsequent statements see the updated value.

Listing 17.9: Blocking assignment in a combinational block

```
always @(*) begin
    result = 16'h0000;    // Step 1: set default
    flag_c = 1'b0;        // Step 2: set default

    case (op)
        `ALU_ADD: begin
            result = add_result[15:0];  // Step 3: override
            flag_c = add_result[16];    // Step 4: override
        end
    endcase
end
```

The blocking assignment is appropriate here because this is *combinational logic*. The statements describe a priority: first set defaults, then override them based on the operation. The synthesis tool converts this into a multiplexer structure, not a sequence of operations.

17.3.2 Non-Blocking Assignment (<=)

A *non-blocking assignment* schedules an update for the end of the current simulation time step. All right-hand sides are evaluated first, using the current values. Then all left-hand sides are updated

simultaneously. This models the behavior of real flip-flops, where all outputs change at the same instant (the clock edge).

Listing 17.10: Non-blocking assignment in the register file

```
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        for (i = 0; i < 16; i = i + 1) begin
            regs[i] <= 16'h0000;
        end
    end else begin
        if (exx) begin
            for (i = 0; i < 8; i = i + 1) begin
                regs[i + 4] <= alt_regs[i];
                alt_regs[i] <= regs[i + 4];
            end
        end
    end
end
```

The **EXX** swap is a perfect illustration of why non-blocking assignment matters. The two assignments inside the loop:

```
regs[i + 4] <= alt_regs[i];
alt_regs[i] <= regs[i + 4];
```

both read their right-hand sides *before* either write takes effect. This means both `regs[i + 4]` and `alt_regs[i]` are read using their old values, and then both are updated simultaneously. The result is a true swap—no temporary variable needed.

If blocking assignment were used instead:

```
regs[i + 4] = alt_regs[i];    // Overwrites regs[i+4]!
alt_regs[i] = regs[i + 4];    // Reads the NEW value -- BUG!
```

the first assignment would overwrite `regs[i + 4]` before the second assignment reads it, destroying the old value and making the swap fail.

The Golden Rules of Assignment

1. Use **blocking assignment** (=) in **combinational** always blocks (`always @(*)`).
2. Use **non-blocking assignment** (<=) in **sequential** always blocks (`always @(posedge clk)`).
3. **Never mix** blocking and non-blocking assignments to the same signal.

Violating these rules creates simulation-synthesis mismatches: the simulator produces one behavior and the synthesized hardware produces another. This class of bug is extremely difficult to track down because the simulation "works" but the hardware does not.

17.4 Combinational vs. Sequential Always Blocks

Now let us solidify the distinction between the two kinds of `always` blocks by examining both in the SAMPO codebase.

17.4.1 Combinational Always Blocks

A combinational always block has the sensitivity list `@(*)` and uses blocking assignment. It describes logic whose output is a pure function of its inputs—no memory, no clock.

The SAMPO ALU is the canonical example:

Listing 17.11: Combinational always block: the ALU's case statement

```
always @(*) begin
    result = 16'h0000;
    flag_c = 1'b0;
    flag_v = 1'b0;

    case (op)
        `ALU_ADD: begin
            result = add_result[15:0];
            flag_c = add_result[16];
            flag_v = add_overflow;
        end
        `ALU_SUB: begin
            result = sub_result[15:0];
            flag_c = ~sub_result[16];
            flag_v = sub_overflow;
        end
        `ALU_AND: begin
            result = a & b;
        end
```

```
        `ALU_OR: begin
            result = a | b;
        end
        `ALU_XOR: begin
            result = a ^ b;
        end
        // ... more operations
        default: begin
            result = 16'h0000;
        end
    endcase
end
```

The default assignments at the top are critical. In a combinational block, every output *must* be assigned a value for every possible input combination. If a signal is not assigned in some branch of an `if` or `case` statement, the synthesis tool infers a *latch*—an unintended memory element that holds the previous value.

Unintentional Latches

If you forget a `default` case or fail to assign all outputs in every branch of a combinational block, the synthesis tool will infer a latch. Latches are almost never what you want in a synchronous FPGA design. They are level-sensitive (not edge-triggered), making timing analysis difficult and behavior unpredictable. The SAMPO ALU avoids this by setting default values for `result`, `flag_c`, and `flag_v` before the `case` statement, ensuring they are assigned in every path.

17.4.2 Sequential Always Blocks

A sequential always block is sensitive to a clock edge (and optionally a reset) and uses non-blocking assignment. It describes flip-flops and registers.

The SAMPO register file and RAM module both use sequential blocks. Here is the RAM's read/write logic:

Listing 17.12: Sequential always block: RAM read and write

```
always @(posedge clk) begin
    if (en) begin
        if (we) begin
            if (be[0]) mem[word_addr][7:0]  <= wdata[7:0];
            if (be[1]) mem[word_addr][15:8] <= wdata[15:8];
        end
        rdata <= mem[word_addr];
    end
```

```
end
```

This block is triggered only on the rising edge of `clk`. The non-blocking assignments ensure that reads and writes behave correctly:

- `rdata <= mem[word_addr]` reads from the memory array and captures the value in the `rdata` output register. If a write is happening to the same address in the same cycle, the read returns the *old* value (before the write), because non-blocking assignments update simultaneously at the end of the time step.
- The `mem[word_addr][7:0] <= wdata[7:0]` write takes effect at the end of the time step, so it does not affect the read in the same cycle.

This "read-before-write" behavior is characteristic of synchronous RAM and matches the behavior of the ECP5 FPGA's block RAM primitives.

Recognizing the Pattern

When reading SAMPO source code, you can instantly classify any `always` block:

- `always @(*)` with `=` ⇒ **combinational logic** (gates, muxes)
- `always @(posedge clk)` with `<=` ⇒ **sequential logic** (flip-flops, registers, RAM)

Every always block in the SAMPO project follows one of these two patterns.

17.5 Parameters and Generate Statements

Good hardware designs are reusable. Verilog provides *parameters* for configuring modules and *generate statements* for creating repetitive structures.

17.5.1 Parameters

A parameter is a compile-time constant that customizes a module when it is instantiated. The SAMPO RAM module uses a parameter for its initialization file:

Listing 17.13: Parameterized RAM module

```
module ram #(
    parameter INIT_FILE = ""
) (
    input  wire        clk,
    input  wire [15:0] addr,
    input  wire [15:0] wdata,
    output reg  [15:0] rdata,
    input  wire        we,
```

```
    input  wire [1:0]  be,
    input  wire        en
);
```

The `#(parameter INIT_FILE = "")` syntax declares a parameter with a default value (empty string). When the module is instantiated, the parameter can be overridden:

Listing 17.14: Overriding a parameter at instantiation

```
ram #(
    .INIT_FILE("programs/hello.hex")
) main_ram (
    .clk(clk),
    .addr(mem_addr),
    .wdata(mem_wdata),
    .rdata(mem_rdata),
    .we(mem_we),
    .be(mem_be),
    .en(mem_en)
);
```

Parameters are resolved at compile time. They do not consume any hardware resources—they configure the synthesis tool's behavior. Common uses include:

- Data widths (`parameter WIDTH = 16`)
- Memory sizes (`parameter DEPTH = 32768`)
- Initialization files (as in the RAM module)
- Address ranges and reset vectors

The SAMPO testbench also uses parameters extensively:

Listing 17.15: Testbench parameters

```
parameter CLK_PERIOD = 10;  // 100 MHz
parameter MAX_CYCLES = 500000;
parameter RAM_INIT_FILE = "../programs/hello.hex";
```

For constants that should not be overridden from outside the module, use `localparam` instead of `parameter`. Local parameters are visible only within the module and cannot be changed at instantiation time. The ALU operation codes in `verilog/rtl/sampo_pkg.vh` use `` `define `` macros rather than `localparam`—both approaches achieve the same effect of defining named constants, but `` `define `` macros are globally visible across all files that include the header.

17.5.2 The Package Header

The SAMPO project defines all its constants in `verilog/rtl/sampo_pkg.vh`, which is included at the top of every source file:

Listing 17.16: Including the package header

```
`include "sampo_pkg.vh"
```

This file defines opcodes, ALU operations, CPU states, and other constants using `` `define `` macros:

Listing 17.17: Constants from `sampo_pkg.vh`

```
// ALU operations (active during execution)
`define ALU_ADD     4'h0
`define ALU_SUB     4'h1
`define ALU_AND     4'h2
`define ALU_OR      4'h3
`define ALU_XOR     4'h4

// CPU states
`define ST_RESET     4'h0
`define ST_FETCH     4'h1
`define ST_FETCH_EXT 4'h2
`define ST_DECODE    4'h3
`define ST_EXECUTE   4'h4
`define ST_MEMORY    4'h5
`define ST_WRITEBACK 4'h6
`define ST_HALTED    4'h7
```

The `` `ifndef/`define/`endif `` guard at the top of the file prevents double inclusion—the same technique used in C header files:

Listing 17.18: Include guard pattern

```
`ifndef SAMPO_PKG_VH
`define SAMPO_PKG_VH

// ... definitions ...

`endif // SAMPO_PKG_VH
```

17.5.3 Generate Statements

The `generate` statement creates multiple instances of hardware based on a compile-time loop or conditional. While the SAMPO does not use explicit generate blocks (preferring `for` loops inside always blocks, which the synthesis tool unrolls), the concept is worth understanding.

A generate loop creates repeated hardware structures:

Listing 17.19: Generate loop example: a ripple-carry adder

```
genvar i;
generate
    for (i = 0; i < 16; i = i + 1) begin : adder_chain
        full_adder fa (
            .a(a[i]),
            .b(b[i]),
            .c_in(carry[i]),
            .sum(sum[i]),
            .c_out(carry[i+1])
        );
    end
endgenerate
```

This generates 16 full adder instances, each named `adder_chain[0]` through `adder_chain[15]`, forming a ripple-carry adder. The `genvar` keyword declares the loop variable, which must be a compile-time constant.

Generate conditionals select between different hardware implementations based on parameters:

Listing 17.20: Generate conditional example

```
generate
    if (WIDTH <= 16) begin : narrow_alu
        // Small multiplier for narrow data paths
        assign product = a * b;
    end else begin : wide_alu
        // Pipelined multiplier for wider data paths
        mult_pipeline mult_inst (.a(a), .b(b), .p(product));
    end
endgenerate
```

17.6 The Icarus Verilog Simulator

Writing Verilog is only half the job. We need to *simulate* our designs to verify they behave correctly before committing them to hardware. The SAMPO project uses *Icarus Verilog* (iverilog), a free, open-source Verilog simulator.

17.6.1 Compilation and Execution

Icarus Verilog uses a two-step flow: first compile the Verilog source to an intermediate format, then execute the simulation.

The SAMPO Makefile in `verilog/Makefile` defines targets for each testbench:

Listing 17.21: Compiling the ALU testbench

```
iverilog -Irtl -o build/alu_tb rtl/alu.v tb/alu_tb.v
```

The `-Irtl` flag adds the `rtl/` directory to the include search path, so the `` `include "sampo_pkg.vh" `` directive can find the package header. The `-o build/alu_tb` flag specifies the output file. The remaining arguments list the source files to compile.

For the full system testbench, all RTL source files must be included:

Listing 17.22: Compiling the full system testbench

```
iverilog -Irtl -o build/sampo_tb \
    rtl/alu.v rtl/shifter.v rtl/regfile.v rtl/decode.v \
    rtl/cpu.v rtl/ram.v rtl/uart.v rtl/soc.v \
    tb/sampo_tb.v
```

Execution is straightforward:

Listing 17.23: Running the ALU testbench

```
cd build && ./alu_tb
```

The simulator executes the compiled design and prints output from `$display` and `$write` calls. For the ALU testbench, the output shows each test group and whether any mismatches were detected.

17.6.2 Using the Makefile

The SAMPO Makefile provides convenient targets that handle compilation and execution:

Listing 17.24: Running tests via Make

```
# Run ALU unit tests
make test-alu

# Run register file unit tests
make test-regfile

# Run all unit tests
make unit-tests

# Run main simulation with hello.hex
make test

# Run all ISA instruction tests
make isa-tests
```

The Makefile also handles the full FPGA synthesis pipeline:

Listing 17.25: FPGA build targets

```
# Synthesize with Yosys
make synth

# Place and route with nextpnr-ecp5
make pnr

# Generate bitstream
make bit

# Full FPGA build (synth + pnr + bit)
make fpga

# Upload to ULX3S board
make prog
```

For ISA tests, the Makefile assembles the test programs, converts them to hex format, compiles the testbench with the appropriate init file, and runs the simulation:

Listing 17.26: Running a single ISA test

```
make isa-test_alu
```

This single command triggers a cascade of build steps:

1. Assemble `examples/tests/test_alu.s` using the SAMPO assembler.
2. Convert the binary to hex format with `bin2hex.py`.
3. Compile the testbench with the test hex file as the RAM init parameter.
4. Execute the simulation and report results.

17.7 Testbench Structure

A *testbench* is a Verilog module that exercises a design by providing inputs and checking outputs. Testbenches are not synthesizable—they use simulation-only constructs like delays, `$display`, and `initial` blocks.

17.7.1 Anatomy of a Testbench

Every testbench follows the same basic structure:

1. **Signal declarations:** Create `reg` signals for inputs (driven by the testbench) and `wire` signals for outputs (driven by the DUT).
2. **DUT instantiation:** Instantiate the design under test.
3. **Clock generation:** For sequential designs, generate a periodic clock signal.
4. **Stimulus and checking:** Apply inputs and verify outputs.

The ALU testbench in `verilog/tb/alu_tb.v` demonstrates this pattern clearly:

Listing 17.27: ALU testbench signal declarations and DUT instantiation

```
`timescale 1ns/1ps

`include "sampo_pkg.vh"

module alu_tb;

    reg  [15:0] a, b;
    reg  [3:0]  op;
    reg         carry_in;
    wire [15:0] result;
    wire        flag_n, flag_z, flag_c, flag_v;

    alu uut (
        .a(a),
        .b(b),
        .op(op),
```

```
        .carry_in(carry_in),
        .result(result),
        .flag_n(flag_n),
        .flag_z(flag_z),
        .flag_c(flag_c),
        .flag_v(flag_v)
    );
```

The inputs (`a`, `b`, `op`, `carry_in`) are declared as `reg` because the testbench drives them. The outputs (`result`, flags) are declared as `wire` because they are driven by the DUT.

17.7.2 The Check Task

A well-structured testbench uses a reusable *task* for verifying results. The SAMPO ALU testbench defines a `check` task:

Listing 17.28: Self-checking task in the ALU testbench

```
integer errors;

task check;
    input [15:0] expected;
    input expected_n, expected_z, expected_c, expected_v;
    begin
        if (result !== expected || flag_n !== expected_n ||
            flag_z !== expected_z || flag_c !== expected_c ||
            flag_v !== expected_v) begin
            $display("ERROR: op=%h a=%h b=%h c_in=%b", op, a, b, carry_in);
            $display("  Got:      result=%h N=%b Z=%b C=%b V=%b",
                     result, flag_n, flag_z, flag_c, flag_v);
            $display("  Expected: result=%h N=%b Z=%b C=%b V=%b",
                     expected, expected_n, expected_z, expected_c, expected_v);
            errors = errors + 1;
        end
    end
endtask
```

The `!==` operator performs a case comparison that distinguishes `x` (unknown) and `z` (high-impedance) values from `0` and `1`. This is stronger than the regular `!=` operator, which treats `x` comparisons as unknown.

17.7.3 Test Stimulus

The `initial` block drives the test vectors:

Listing 17.29: Test vectors for the ALU (excerpt)

```
initial begin
    errors = 0;
    carry_in = 0;

    $display("=== ALU Testbench ===");

    // Test ADD
    $display("Testing ADD...");
    op = `ALU_ADD;
    a = 16'h0001; b = 16'h0002; #10; check(16'h0003, 0, 0, 0, 0);
    a = 16'hFFFF; b = 16'h0001; #10; check(16'h0000, 0, 1, 1, 0);
    a = 16'h7FFF; b = 16'h0001; #10; check(16'h8000, 1, 0, 0, 1);
    a = 16'h8000; b = 16'h8000; #10; check(16'h0000, 0, 1, 1, 1);

    // Test SUB
    $display("Testing SUB...");
    op = `ALU_SUB;
    a = 16'h0005; b = 16'h0003; #10; check(16'h0002, 0, 0, 1, 0);
    a = 16'h0003; b = 16'h0005; #10; check(16'hFFFE, 1, 0, 0, 0);
    // ...
```

Each test vector follows the pattern: set inputs, wait for propagation (`#10` delays 10 time units), then check. The `#10` delay gives the combinational logic time to settle. For the ALU (purely combinational), any non-zero delay suffices; for sequential designs, you would wait for clock edges instead.

17.7.4 The System Testbench

The main testbench in `verilog/tb/sampo_tb.v` is more sophisticated. It simulates the entire SoC, including clock generation, reset sequencing, UART output capture, and pass/fail detection:

Listing 17.30: Reset sequence in the system testbench

```
initial begin
    $display("=== Sampo CPU Testbench ===");
    $display("RAM init file: %s", RAM_INIT_FILE);

    // Initialize
    rst_n = 0;
    #100;
    rst_n = 1;

    $display("CPU started at PC=0x%04X", pc);
```

The reset sequence holds `rst_n` low for 100 time units (10 clock cycles at 100 MHz), allowing all flip-flops to initialize. Then `rst_n` goes high and the CPU begins executing from the reset vector.

The testbench uses a `fork-join` construct to implement a timeout mechanism:

Listing 17.31: Timeout mechanism using fork-join

```
fork
    begin : timeout_block
        #(CLK_PERIOD * MAX_CYCLES);
        $display("TIMEOUT: Max cycles (%d) reached", MAX_CYCLES);
        disable wait_halt;
    end

    begin : wait_halt
        wait(halted);
        #10;
        disable timeout_block;
    end
join
```

Two parallel threads race: one waits for the CPU to halt, and the other counts down a maximum number of cycles. Whichever finishes first disables the other. This prevents the simulation from running forever if the program has a bug that never reaches the **`HALT`** instruction.

The testbench also captures UART output character by character, scanning each line for the strings "PASS" or "FAIL" to provide automated test result reporting. This is how the ISA test suite detects whether each test passed or failed without requiring manual inspection of the output.

Listing 17.32: UART capture and PASS/FAIL detection (simplified)

```
always @(posedge clk) begin
    if (tx_valid && tx_ready) begin
        uart_buffer[uart_idx] = tx_data;
        uart_idx = uart_idx + 1;
        if (tx_data >= 8'h20 && tx_data < 8'h7F) begin
            $write("%c", tx_data);
        end else if (tx_data == 8'h0A) begin
            $write("\n");
            check_line;
            line_idx = 0;
        end
    end
end
```

17.8 Waveform Viewing with GTKWave

Text output from `$display` is useful for pass/fail testing, but debugging hardware requires seeing *how signals change over time*. Waveform viewers display the value of every signal at every point in the simulation, making it possible to trace cause and effect through the circuit.

17.8.1 Generating Waveform Data

The SAMPO testbench generates waveform data using VCD (Value Change Dump) format:

Listing 17.33: VCD dump configuration in the testbench

```
initial begin
    $dumpfile("sampo.vcd");
    $dumpvars(0, sampo_tb);
end
```

The `$dumpfile` call specifies the output file name. The `$dumpvars(0, sampo_tb)` call tells the simulator to record every signal in the `sampo_tb` module and all its sub-modules (the `0` means "all levels of hierarchy"). After simulation completes, the VCD file contains a complete record of every signal transition.

17.8.2 Launching GTKWave

GTKWave is a free waveform viewer that reads VCD files. The Makefile provides a convenient target:

Listing 17.34: Opening waveforms in GTKWave

```
make wave
```

This runs the simulation (if needed) and opens the VCD file in GTKWave. You can also launch it manually:

Listing 17.35: Manual GTKWave launch

```
gtkwave build/sampo.vcd &
```

17.8.3 Reading Waveforms

When GTKWave opens, the left panel shows the module hierarchy. Expanding the hierarchy reveals all signals in each module. Drag signals from the hierarchy panel to the waveform display to view them.

For debugging the SAMPO CPU, the most important signals to watch are:

- `clk` and `rst_n`: the clock and reset signals establish the timing reference.
- `state`: the CPU's FSM state shows which phase of instruction execution is active (FETCH, DECODE, EXECUTE, MEMORY, WRITEBACK).
- `pc`: the program counter shows which instruction is being executed.
- `instr`: the current instruction word reveals what the CPU is doing.
- `alu_result`, `flags`: the ALU output and flag register show the results of computation.
- `mem_addr`, `mem_rdata`, `mem_wdata`: memory interface signals reveal load/store behavior.

GTKWave Display Formats

Right-click on a signal in GTKWave and choose "Data Format" to change how values are displayed. For instruction words and addresses, hexadecimal is most readable. For flags and control signals, binary or decimal works better. For the FSM state, you can create a "Translate Filter" file that maps state numbers to human-readable names (FETCH, DECODE, etc.), making the waveform much easier to read.

17.8.4 Debug Tracing

The SAMPO testbench includes optional cycle-by-cycle debug output for the first 100 cycles:

Listing 17.36: Debug trace output in the testbench

```
always @(posedge clk) begin
    if (rst_n && cycles < 100) begin
        $display("Cycle %5d: PC=%04X St=%d Ins=%04X R4=%04X R5=%04X Flags=%02X",
            cycles, pc, uut.cpu_inst.state, uut.cpu_inst.instr,
            uut.cpu_inst.regfile_inst.regs[4],
            uut.cpu_inst.regfile_inst.regs[5],
            uut.cpu_inst.flags);
    end
end
```

This uses hierarchical signal references (`uut.cpu_inst.state`) to reach inside the DUT's module hierarchy and read internal signals. This is a simulation-only feature—you cannot access internal signals of a module from outside in synthesized hardware.

The trace output shows, for each clock cycle:

- The cycle number and program counter.

- The CPU's FSM state (as a number: 1 = FETCH, 3 = DECODE, etc.).
- The current instruction word.
- The values of `R4` and `R5` (useful for watching test results).
- The flags register.

This textual trace and the GTKWave waveforms provide complementary views of the same simulation. The text trace gives a quick summary; the waveforms show every signal in full detail. Together, they form a powerful debugging toolkit that we will use extensively when building and verifying the SAMPO in Chapters 18 and 24 and **??**.

17.9 Exercises

1. **First module.** Write a Verilog module called `comparator` that takes two 16-bit inputs `a` and `b` and produces three 1-bit outputs: `eq` (equal), `lt` (less than, unsigned), and `gt` (greater than, unsigned). Use continuous assignments. Write a testbench that verifies the outputs for at least five input pairs including edge cases (`0x0000` vs. `0xFFFF`, equal values, adjacent values).

2. **Blocking vs. non-blocking experiment.** Write two versions of a simple 4-stage shift register: one using blocking assignment and one using non-blocking assignment inside a `posedge`-triggered always block. Simulate both and compare the waveforms. Explain why the blocking version produces incorrect behavior. What does the blocking version synthesize to, and why is it different from the intended design?

3. **Parameterized counter.** Write a parameterized counter module with parameters for width (`WIDTH`) and maximum count (`MAX_COUNT`). The counter should increment on each clock edge when enabled, and wrap to zero when it reaches `MAX_COUNT`. Instantiate it twice in a testbench: once as an 8-bit counter that counts to 255, and once as a 4-bit counter that counts to 9 (for a BCD digit). Simulate both and verify the wrap-around behavior.

4. **Run the SAMPO ALU tests.** Navigate to the `verilog/` directory in the SAMPO project and run `make test-alu`. Read the source code of both `rtl/alu.v` and `tb/alu_tb.v` before running the tests. After the tests pass, add a new test case to the testbench that verifies the `ALU_NEG` operation with input `0x0001`, checking that the result is `0xFFFF` with `flag_n=1`, `flag_z=0`, `flag_c=1`, and `flag_v=0`.

5. **Waveform exploration.** Run `make test` in the `verilog/` directory to execute the hello-world program, then open the generated waveform with `make wave`. In GTKWave, find the `state` signal inside the CPU module and watch it cycle through the FSM states. Identify one complete instruction fetch-decode-execute cycle. How many clock cycles does a typical instruction take? What state does the CPU spend the most time in?

What's Next. With the foundations of digital logic and Verilog in hand, we are ready to build the SAMPO CPU itself. In Chapter 18, we dive deep into the SAMPO's arithmetic logic unit—not just the concept we introduced here, but the complete implementation with every operation, every flag corner case, and every design decision explained. We will take the ALU from a Verilog source file to a verified, synthesizable module, using the simulation and waveform techniques we learned in this chapter.

Part VI

Building the CPU in Verilog

Chapter 18

The Arithmetic Logic Unit

Every instruction the SAMPO executes eventually flows through one component: the arithmetic logic unit. Whether we are adding two registers, comparing a value before a branch, or passing an immediate through to the register file, the ALU sits at the heart of the datapath. In this chapter, we open up `verilog/rtl/alu.v`, trace every operation it supports, and examine how it generates the four arithmetic flags that drive conditional branching.

18.1 ALU Design Overview

The SAMPO ALU is a purely combinational module—it has no clock input, no state, and no memory. Present it with two 16-bit operands and a 4-bit operation code, and the result appears at the output within a single propagation delay. This is the fundamental building block that the CPU's state machine will invoke during the EXECUTE phase.

Listing 18.1: ALU module port declaration

```
module alu (
    input  wire [15:0] a,
    input  wire [15:0] b,
    input  wire [3:0]  op,
    input  wire        carry_in,
    output reg  [15:0] result,
    output wire        flag_n,
    output wire        flag_z,
    output reg         flag_c,
    output reg         flag_v
);
```

The interface is deliberately minimal. The inputs `a` and `b` are the two 16-bit operands, `op` selects one of 16 possible operations, and `carry_in` provides the current state of the carry flag for add-with-carry and subtract-with-borrow instructions. The outputs consist of the 16-bit `result` and four condition flags: Negative (`flag_n`), Zero (`flag_z`), Carry (`flag_c`), and Overflow (`flag_v`).

A Pure Combinational Module

Notice that there is no `clk` or `rst_n` port. The ALU is entirely combinational—every output is a function of the current inputs alone. The CPU's state machine drives the ALU inputs and samples the outputs at the right moment in the pipeline. This separation of concerns keeps the ALU design clean and testable in isolation.

18.1.1 The Datapath at a Glance

The ALU handles three broad categories of operations:

1. **Arithmetic**: addition, subtraction, add-with-carry, subtract-with-borrow, negation, multiplication, and division.
2. **Logical**: bitwise AND, OR, XOR, and NOT.
3. **Pass-through**: passing operand `a` or `b` through unchanged, used for moves, loads, and other data-routing tasks.

The ALU also contains shift operations (**SLL**, **SRL**, **SRA**) that support the extended-format variable-shift instructions. Fixed-count shifts are handled by the dedicated shifter module (see Chapter 20), but when the CPU needs to shift by an arbitrary amount encoded in an immediate, it routes through the ALU instead.

All 16 possible values of the 4-bit `op` field are assigned, as shown in Table 18.1. There are no reserved codes—every combination produces a defined result.

18.2 Operation Selection via Function Code

The heart of the ALU is a single combinational `always @(*)` block containing a `case` statement that dispatches on the 4-bit `op` input. Before entering the case, default values are set for the outputs that might not be assigned in every branch:

Listing 18.2: Default output assignment in the ALU

```
always @(*) begin
    // Default values
    result = 16'h0000;
    flag_c = 1'b0;
```

Table 18.1: ALU operation codes defined in `sampo_pkg.vh`

Code	Define	Operation
0x0	ALU_ADD	result = a + b
0x1	ALU_SUB	result = a - b
0x2	ALU_AND	result = a & b
0x3	ALU_OR	result = a | b
0x4	ALU_XOR	result = a ^ b
0x5	ALU_SLL	result = a « b[3:0]
0x6	ALU_SRL	result = a » b[3:0]
0x7	ALU_SRA	result = a »> b[3:0]
0x8	ALU_MUL	result = (a * b)[15:0]
0x9	ALU_MULH	result = (a * b)[31:16]
0xA	ALU_DIV	result = a / b (signed)
0xB	ALU_REM	result = a % b (signed)
0xC	ALU_PASS_A	result = a
0xD	ALU_PASS_B	result = b
0xE	ALU_NOT	result = ~a
0xF	ALU_NEG	result = -a

```
    flag_v = 1'b0;

    case (op)
        `ALU_ADD: begin
            // ...
        end
        // ... remaining operations
    endcase
end
```

Always Assign Defaults

In a combinational `always @(*)` block, every output must be assigned in every path through the logic, or the synthesis tool will infer a *latch*—a storage element that holds the previous value. Setting defaults at the top of the block ensures no latches are created, even if some case branches do not explicitly set every output. The `result` defaults to zero, and the carry and overflow flags default to cleared. This means logical operations like **AND** and **OR** can rely on the defaults for C and V without explicitly clearing them.

The operation code that arrives at the ALU's `op` input is not the same as the instruction opcode in bits [15:12] of the instruction word. The instruction decoder (Chapter 21) translates each instruction into the appropriate ALU operation code. For example, both the **`ADD`** instruction (opcode `0x0`) and the **`ADDI`** instruction (opcode `0x5`) set `alu_op` to `ALU_ADD` (`0x0`). The **`CMP`** instruction, which is a subtraction that discards its result, sets `alu_op` to `ALU_SUB`.

Function Code vs. Opcode

The *instruction opcode* lives in bits [15:12] of the SAMPO instruction word and selects the instruction category. The *ALU function code* is a separate 4-bit value generated by the decoder that selects which ALU operation to perform. Many different instructions map to the same ALU function code: **`ADD`**, **`ADDI`**, **`ADDIX`**, and even **`POP`** (which adds 2 to SP) all use `ALU_ADD`.

18.3 Addition and Subtraction with Carry

Addition and subtraction are the most critical operations in any ALU. They need to produce not only a 16-bit result but also carry and overflow information for conditional branching. The SAMPO uses 17-bit intermediate results to capture the carry output naturally.

18.3.1 Extended-Width Intermediate Signals

Before the main case statement, the ALU pre-computes all four arithmetic results using 17-bit wires:

Listing 18.3: 17-bit intermediate arithmetic results

```
// Internal signals for 17-bit arithmetic (to capture carry)
wire [16:0] add_result;
wire [16:0] sub_result;
wire [16:0] adc_result;
wire [16:0] sbc_result;

// Calculate extended results
assign add_result = {1'b0, a} + {1'b0, b};
assign sub_result = {1'b0, a} - {1'b0, b};
assign adc_result = {1'b0, a} + {1'b0, b} + {16'b0, carry_in};
assign sbc_result = {1'b0, a} - {1'b0, b} - {16'b0, carry_in};
```

By zero-extending both operands to 17 bits before the arithmetic, the carry or borrow naturally appears in bit [16] of the result. This avoids manual carry detection logic and lets the Verilog synthesis tool map the operation directly to the FPGA's carry chain.

Why Pre-Compute All Four?

You might wonder why we compute all four results (`add_result`, `sub_result`, `adc_result`, `sbc_result`) even though only one will be used. Since these are `assign` statements—pure combinational logic—the synthesis tool is free to optimize away any unused hardware. In practice, the tool recognizes which result feeds into the mux selected by `op` and synthesizes only the necessary adders. Writing the code this way keeps the `case` statement clean and makes the carry/overflow detection readable.

18.3.2 The ADD Operation

The addition case extracts the lower 16 bits as the result and takes bit [16] as the carry flag:

Listing 18.4: ALU addition

```
`ALU_ADD: begin
    result = add_result[15:0];
    flag_c = add_result[16];
    flag_v = add_overflow;
end
```

When adding `0xFFFF` + `0x0001`, the 17-bit result is `1_0000_0000_0000_0000`. Bit [16] is 1, so carry is set. The lower 16 bits are `0x0000`, so the result wraps around—and the zero flag (computed separately) will also be set. We can verify this directly in the testbench:

Listing 18.5: Testbench: ADD wrapping to zero

```
op = `ALU_ADD;
a = 16'hFFFF; b = 16'h0001; #10;
check(16'h0000, 0, 1, 1, 0);  // N=0, Z=1, C=1, V=0
```

The result is zero (Z=1), carry is set (C=1) because the addition overflowed the 16-bit range, but the signed overflow flag is clear (V=0) because both operands have different signs when interpreted as signed values ($-1 + 1 = 0$, no signed overflow).

18.3.3 The SUB Operation and ARM-Style Carry

Subtraction introduces an important design decision: what does the carry flag mean after a **SUB**?

Listing 18.6: ALU subtraction with ARM-style carry

```
`ALU_SUB: begin
    result = sub_result[15:0];
    // Carry is set when there's NO borrow (a >= b for unsigned)
```

```
    flag_c = ~sub_result[16];
    flag_v = sub_overflow;
end
```

The SAMPO uses ARM-style carry semantics for subtraction: **the carry flag is set when there is no borrow**. This is the *inverse* of the raw borrow bit produced by the subtractor. When `a >= b` (unsigned), the subtraction does not need to borrow, so C=1. When `a < b`, a borrow occurs, and C=0.

ARM-Style vs. x86-Style Carry on Subtraction

Different architectures define the carry flag differently after subtraction:

- **ARM, 6502, SAMPO**: C=1 means *no borrow* (`a >= b` unsigned). Subtraction is computed as `a + ~b + 1`, and C is the carry out of that addition.
- **x86, Z80**: C=1 means *borrow occurred* (`a < b` unsigned). The carry flag directly reflects the borrow.

The SAMPO follows the ARM convention. This means **`BLTU`** (branch if unsigned less than) checks for C=0 (borrow occurred), not C=1. The branch condition table in the decoder reflects this—see Chapter 21.

The testbench confirms both cases:

Listing 18.7: Testbench: SUB carry behavior

```
op = `ALU_SUB;
a = 16'h0005; b = 16'h0003; #10;
check(16'h0002, 0, 0, 1, 0);  // 5-3=2, C=1 (no borrow)

a = 16'h0003; b = 16'h0005; #10;
check(16'hFFFE, 1, 0, 0, 0);  // 3-5 underflows, C=0 (borrow)
```

In the first case, 5 is greater than 3, so no borrow is needed and C=1. In the second, 3 is less than 5, a borrow occurs, and C=0. The result `0xFFFE` is the two's complement representation of -2, and the N flag is set because bit [15] is 1.

18.3.4 Multiplication and Division

The ALU also handles multiplication and division, which the SAMPO provides as two-operand instructions: `Rd = Rd op Rs`. The implementation uses Verilog's built-in `*`, `/`, and `%` operators, relying on the synthesis tool to instantiate the appropriate hardware multiplier or divider.

Listing 18.8: Signed and unsigned multiply intermediate signals

```
// Signed versions for signed operations
wire signed [15:0] a_signed = $signed(a);
wire signed [15:0] b_signed = $signed(b);
wire signed [31:0] mul_result_signed;
wire        [31:0] mul_result_unsigned;

// Multiply results
assign mul_result_signed = a_signed * b_signed;
assign mul_result_unsigned = a * b;
```

Multiplication produces a 32-bit result from two 16-bit inputs. The **MUL** instruction returns the lower 16 bits, while **MULH** returns the upper 16 bits—allowing software to reconstruct the full 32-bit product when needed:

Listing 18.9: MUL and MULH operations

```
`ALU_MUL: begin
    result = mul_result_signed[15:0];
end

`ALU_MULH: begin
    result = mul_result_signed[31:16];
end
```

Division includes protection against division by zero. Rather than causing undefined behavior or an exception, the SAMPO returns a defined sentinel value:

Listing 18.10: Division with zero check

```
`ALU_DIV: begin
    if (b == 16'h0000) begin
        result = 16'hFFFF;  // Division by zero
    end else begin
        result = $signed(a_signed / b_signed);
    end
end

`ALU_REM: begin
    if (b == 16'h0000) begin
        result = a;  // Remainder unchanged on div by zero
    end else begin
        result = $signed(a_signed % b_signed);
    end
end
```

Division by Zero Returns Defined Values

When the divisor `b` is zero, **`DIV`** returns `0xFFFF` (all ones) and **`REM`** returns the original dividend `a` unchanged. This is a deliberate design choice that avoids the need for a division-by-zero exception mechanism while still producing a detectable result. Software should check for zero divisors before executing division if it needs to handle the error case.

18.4 Flag Generation: N, Z, C, V

The SAMPO FLAGS register contains six flags, but the ALU generates four of them: Negative (N), Zero (Z), Carry (C), and Overflow (V). The Half-carry (H) and Interrupt (I) flags are managed elsewhere. The flag positions within the 8-bit FLAGS register are defined in `sampo_pkg.vh`:

Listing 18.11: Flag bit positions in the FLAGS register

```
`define FLAG_N 7
`define FLAG_Z 6
`define FLAG_C 5
`define FLAG_V 4
`define FLAG_H 3
`define FLAG_I 2
```

18.4.1 The N and Z Flags

The Negative and Zero flags are the same for every ALU operation—they depend only on the result, not on how that result was computed. They are generated outside the `case` statement as continuous assignments:

Listing 18.12: N and Z flag generation

```
// N flag: negative (MSB of result)
assign flag_n = result[15];

// Z flag: zero
assign flag_z = (result == 16'h0000);
```

The N flag is bit [15] of the result—the most significant bit, which in two's complement representation indicates a negative number. The Z flag is asserted when every bit of the result is zero.

These flags are unconditionally driven by the ALU outputs. The CPU's state machine decides *which* operations actually update the FLAGS register. For example, **MOV** passes a value through via ALU_PASS_B and the ALU will produce N and Z values, but the CPU does not update the FLAGS register for a **MOV** instruction.

18.4.2 The C Flag

The Carry flag has different meanings depending on the operation:

- **ADD**: Carry out of the 16-bit addition. Set when the unsigned result exceeds `0xFFFF`.
- **SUB**: *Inverted* borrow—set when no borrow occurs (`a >= b` unsigned). This is the ARM-style convention.
- **SLL/SRL/SRA**: The last bit shifted out of the value.
- **NEG**: Set when the result is non-zero (i.e., the original value was non-zero).
- **Logical ops** (AND, OR, XOR, NOT): Always cleared (defaults to 0).
- **Pass-through** (PASS_A, PASS_B): Always cleared.

Because the carry flag's default is 0 (set at the top of the `always` block), only operations that actively produce carry need to assign `flag_c` in their case branch. The logical and pass-through operations inherit the default.

18.4.3 The V Flag

The Overflow flag indicates signed arithmetic overflow—a condition where the result of an operation cannot be represented in 16-bit two's complement. This flag is critical for signed comparison branches (**BLT**, **BGE**, **BGT**, **BLE**).

Like the carry flag, the overflow flag defaults to 0 and is only set by operations where signed overflow is meaningful: addition, subtraction, and negation.

18.5 Overflow Detection

Signed overflow occurs when the sign of the result is inconsistent with the signs of the operands. The rules differ for addition and subtraction.

18.5.1 Addition Overflow

For addition, overflow occurs when two operands of the *same* sign produce a result of the *opposite* sign. Adding two positive numbers should produce a positive result; if it wraps to negative, that is overflow. Adding two negative numbers should produce a negative result; if it wraps to positive, that

is overflow. Adding operands of opposite signs can never overflow—the result is always between the two operands.

Listing 18.13: Addition overflow detection

```
wire add_overflow = (a[15] == b[15]) &&
                    (add_result[15] != a[15]);
wire adc_overflow = (a[15] == b[15]) &&
                    (adc_result[15] != a[15]);
```

The logic reads: "if the sign bits of `a` and `b` match (both positive or both negative), but the sign bit of the result differs, then overflow has occurred."

Consider a concrete example. Adding `0x7FFF` (32,767, the largest positive 16-bit signed integer) to `0x0001` (1):

Listing 18.14: Testbench: signed addition overflow

```
op = `ALU_ADD;
a = 16'h7FFF; b = 16'h0001; #10;
check(16'h8000, 1, 0, 0, 1);  // N=1, Z=0, C=0, V=1
```

The result is `0x8000` ($-32,768$ in two's complement). Both operands were positive (bit [15] = 0), but the result is negative (bit [15] = 1)—signed overflow. Notice that the carry flag is *not* set, because the addition did not exceed the unsigned 16-bit range. The overflow flag (V=1) and the carry flag (C=0) convey independent information.

18.5.2 Subtraction Overflow

For subtraction, overflow occurs when the operands have *different* signs and the result's sign matches the *subtrahend* (the value being subtracted):

Listing 18.15: Subtraction overflow detection

```
wire sub_overflow = (a[15] != b[15]) &&
                    (sub_result[15] == b[15]);
wire sbc_overflow = (a[15] != b[15]) &&
                    (sbc_result[15] == b[15]);
```

Why different signs? Subtracting a negative from a positive should yield a larger positive number. If the result turns negative instead, the subtraction has overflowed. Similarly, subtracting a positive from a negative should produce a more-negative result; if it wraps to positive, that is overflow.

The testbench exercises this case:

Listing 18.16: Testbench: signed subtraction overflow

```
op = `ALU_SUB;
a = 16'h8000; b = 16'h0001; #10;
check(16'h7FFF, 0, 0, 1, 1);  // N=0, Z=0, C=1, V=1
```

Here we subtract 1 from $-32,768$. The expected mathematical result is $-32,769$, which cannot be represented in 16-bit two's complement. The ALU produces `0x7FFF` (32,767) with V=1, signaling the overflow. The carry flag is 1 because `0x8000` $\geq$ `0x0001` unsigned (no borrow needed).

18.5.3 The Overflow and Branch Connection

The overflow flag works together with the negative flag to implement signed comparisons. After a **`CMP`** (which is a subtraction that discards the result), the branch conditions are:

Table 18.2: Signed branch conditions using N and V flags

Branch	Condition	Meaning
`BLT`	$N \neq V$	Signed less than
`BGE`	$N = V$	Signed greater or equal
`BGT`	$Z = 0$ and $N = V$	Signed greater than
`BLE`	$Z = 1$ or $N \neq V$	Signed less or equal

The key insight is that $N \neq V$ means "the true mathematical result was negative." If no overflow occurred (V=0), then N directly tells us the sign of the result. If overflow *did* occur (V=1), the sign bit is wrong, and $N \neq V$ corrects for it.

18.6 Pass-Through and Bitwise Operations

Not every trip through the ALU involves arithmetic. Several instruction types need the ALU to pass data through unchanged or perform bitwise manipulation.

18.6.1 Pass-Through: PASS_A and PASS_B

Listing 18.17: Pass-through operations

```
`ALU_PASS_A: begin
    result = a;
end

`ALU_PASS_B: begin
    result = b;
```

```
end
```

These operations copy one input directly to the output. Neither carry nor overflow is set (both keep their default value of 0). The decoder uses `ALU_PASS_B` for several purposes:

- **`MOV Rd, Rs1`**: pass the source register value to the destination.
- **`LIX Rd, imm16`**: pass the 16-bit immediate (placed on the `b` input) to the register file.
- Default decoder output: when the decoder does not need the ALU for an instruction, it defaults to `ALU_PASS_B`.

`ALU_PASS_A` is less commonly used but available for instructions that need to route the `a` operand to the result.

18.6.2 Bitwise AND, OR, and XOR

The three bitwise operations are straightforward—each applies the corresponding operation bit by bit:

Listing 18.18: Bitwise operations

```
`ALU_AND: begin
    result = a & b;
    // C and V cleared for logical ops
end

`ALU_OR: begin
    result = a | b;
end

`ALU_XOR: begin
    result = a ^ b;
end
```

Bitwise operations affect the N and Z flags (via the continuous `assign` statements) but always clear C and V. The comment in the **`AND`** case calls this out explicitly. For **`OR`** and **`XOR`**, the defaults handle it—`flag_c` and `flag_v` were initialized to 0 at the top of the block.

The **`TEST`** instruction, which performs **`AND`** without storing the result, maps to `ALU_AND` in the decoder. The CPU simply does not write the result to the register file, but the flags are updated normally. This is exactly how ARM's `TST` instruction works.

Listing 18.19: Testbench: bitwise operations

```
// Test AND
op = `ALU_AND;
a = 16'hF0F0; b = 16'h0FF0; #10;
check(16'h00F0, 0, 0, 0, 0);  // Only common bits

// Test XOR: identical values cancel to zero
op = `ALU_XOR;
a = 16'hFFFF; b = 16'hFFFF; #10;
check(16'h0000, 0, 1, 0, 0);  // Z=1
```

18.6.3 NOT: Bitwise Complement

The **NOT** instruction computes the bitwise complement (one's complement) of its operand. It uses only the a input; the b input is ignored:

Listing 18.20: Bitwise NOT

```
`ALU_NOT: begin
    result = ~a;
end
```

Like the other logical operations, **NOT** clears C and V while setting N and Z based on the result:

Listing 18.21: Testbench: NOT operation

```
op = `ALU_NOT;
a = 16'h0000; b = 16'h0000; #10;
check(16'hFFFF, 1, 0, 0, 0);  // ~0x0000 = 0xFFFF, N=1

a = 16'hFFFF; b = 16'h0000; #10;
check(16'h0000, 0, 1, 0, 0);  // ~0xFFFF = 0x0000, Z=1

a = 16'hAAAA; b = 16'h0000; #10;
check(16'h5555, 0, 0, 0, 0);  // Alternating bits flip
```

18.6.4 NEG: Two's Complement Negation

Negation computes the two's complement of a, equivalent to `0 - a`:

Listing 18.22: Two's complement negation

```
`ALU_NEG: begin
    // NEG is 0 - a
```

```
    result = -a;
    flag_c = (a != 16'h0000);  // Carry set if result non-zero
    flag_v = (a == 16'h8000);  // Overflow only for -32768
end
```

The carry and overflow logic for **NEG** deserves careful attention:

- **Carry**: Set when the input is non-zero. This follows the convention that `NEG` is equivalent to `SUB 0, a`: if a is non-zero, then $0 < a$ unsigned, so no borrow from $0 - a$ would set C=0 in x86 convention, but in the ARM convention (which inverts), the carry is set when the input is non-zero. The practical effect is that the only value that clears carry is zero itself, since $-0 = 0$.
- **Overflow**: Set only when a is `0x8000` ($-32,768$). This is the unique value in 16-bit two's complement that has no positive counterpart—negating it produces `0x8000` again, the same value, which is an overflow condition.

Listing 18.23: Testbench: NEG edge cases

```
op = `ALU_NEG;
a = 16'h0001; b = 16'h0000; #10;
check(16'hFFFF, 1, 0, 1, 0);  // -1 -> FFFF

a = 16'h0000; b = 16'h0000; #10;
check(16'h0000, 0, 1, 0, 0);  // -0 -> 0, C=0

a = 16'h8000; b = 16'h0000; #10;
check(16'h8000, 1, 0, 1, 1);  // -(-32768) overflows, V=1
```

The last test case is particularly interesting: negating `0x8000` yields `0x8000`. The magnitude 32,768 cannot be represented as a positive 16-bit signed integer (the maximum is 32,767), so the result wraps back to itself. The overflow flag correctly signals this anomaly.

18.6.5 Shift Operations in the ALU

The ALU includes shift-left-logical, shift-right-logical, and shift-right-arithmetic operations. These are used by the extended-format variable-shift instructions (**SLLX**, **SRLX**, **SRAX**), which encode the shift amount in a 16-bit immediate. Fixed-count shifts (by 1, 4, or 8) are handled by the dedicated shifter module (Chapter 20).

Listing 18.24: Variable-width shift operations in the ALU

```
`ALU_SLL: begin
    result = a << b[3:0];
    flag_c = (b[3:0] != 0) ? a[16 - b[3:0]] : 1'b0;
```

```
end

`ALU_SRL: begin
    result = a >> b[3:0];
    flag_c = (b[3:0] != 0) ? a[b[3:0] - 1] : 1'b0;
end

`ALU_SRA: begin
    result = $signed(a) >>> b[3:0];
    flag_c = (b[3:0] != 0) ? a[b[3:0] - 1] : 1'b0;
end
```

The shift amount is taken from the low 4 bits of `b`, allowing shifts of 0 to 15 positions. The carry flag captures the last bit shifted out of the value. For a left shift, this is the bit at position `16 - shift_amount` (the highest bit that gets pushed out the top). For right shifts, it is the bit at position `shift_amount - 1` (the lowest bit that gets pushed out the bottom). When the shift amount is zero, no bits are shifted out, so carry is cleared.

The arithmetic right shift (`ALU_SRA`) uses Verilog's `>>>` operator with `$signed(a)` to ensure sign extension—the vacated high bits are filled with copies of the sign bit, preserving the value's sign.

LUI Uses ALU_SLL

The **`LUI`** (Load Upper Immediate) instruction is decoded as an `ALU_SLL` with a shift amount of 8. The decoder sets `alu_b` to `0x0008` and `alu_op` to `ALU_SLL`, so the ALU performs `Rs1 << 8`, placing the source register's low byte into the upper byte of the result. This reuse of existing ALU operations means **`LUI`** requires no additional hardware.

18.7 Testing the ALU

The ALU testbench in `verilog/tb/alu_tb.v` exercises every operation with carefully chosen input values designed to hit corner cases. The testbench instantiates the ALU as a unit under test (UUT) and uses a `check` task that compares all five outputs (result and four flags) against expected values:

Listing 18.25: The testbench check task

```
task check;
    input [15:0] expected;
    input expected_n, expected_z, expected_c, expected_v;
    begin
        if (result !== expected || flag_n !== expected_n ||
            flag_z !== expected_z || flag_c !== expected_c ||
            flag_v !== expected_v) begin
```

```
            $display("ERROR: op=%h a=%h b=%h c_in=%b",
                     op, a, b, carry_in);
            $display("  Got:      result=%h N=%b Z=%b C=%b V=%b",
                     result, flag_n, flag_z, flag_c, flag_v);
            $display("  Expected: result=%h N=%b Z=%b C=%b V=%b",
                     expected, expected_n, expected_z,
                     expected_c, expected_v);
            errors = errors + 1;
        end
    end
endtask
```

The task uses `!==` (identity inequality) rather than `!=` (logical inequality) to catch x and z values, which can hide bugs in simulation. If any output contains an unknown value, the identity comparison will catch it even when logical comparison might not.

The test strategy covers several categories of inputs for each operation:

- **Normal cases**: Small values that produce unremarkable results, verifying basic functionality.
- **Boundary values**: `0x0000`, `0x7FFF`, `0x8000`, `0xFFFF`—the extremes of the unsigned and signed ranges.
- **Overflow triggers**: Value pairs that cross the signed boundary (e.g., `0x7FFF` + `0x0001`).
- **Carry triggers**: Value pairs that cross the unsigned boundary (e.g., `0xFFFF` + `0x0001`).
- **Zero results**: Pairs that cancel to zero, verifying the Z flag.

The addition tests alone exercise four such scenarios:

Listing 18.26: ALU addition test cases

```
// Normal addition
a = 16'h0001; b = 16'h0002; #10;
check(16'h0003, 0, 0, 0, 0);

// Unsigned overflow (wraps to zero)
a = 16'hFFFF; b = 16'h0001; #10;
check(16'h0000, 0, 1, 1, 0);

// Signed overflow (positive -> negative)
a = 16'h7FFF; b = 16'h0001; #10;
check(16'h8000, 1, 0, 0, 1);

// Both carry and signed overflow
a = 16'h8000; b = 16'h8000; #10;
```

```
check(16'h0000, 0, 1, 1, 1);
```

That last case—`0x8000` + `0x8000`—is worth studying. Both operands are $-32,768$ in signed interpretation. Their sum is $-65,536$, which overflows the signed range (V=1). The unsigned sum is 65,536, which overflows the 16-bit unsigned range (C=1). The result wraps to zero (Z=1). A single addition simultaneously triggers three of the four flags.

Exercises

1. **Trace a CMP instruction through the ALU.** The **`CMP Rd, Rs1`** instruction is decoded as an `ALU_SUB`. For `R4` = `0x0010` and `R5` = `0x0020`, work out the values of all four flags (N, Z, C, V) after **`CMP R4, R5`**. Then determine which of **`BLT`**, **`BGE`**, **`BEQ`**, **`BNE`** would be taken.

2. **Add an ADC operation.** The ALU source already computes `adc_result` and `adc_overflow` but does not include an `ALU_ADC` case in the `case` statement (ADC is handled by the CPU routing the carry separately). Write the case branch for an `ALU_ADC` operation that uses these pre-computed signals. What would the carry and overflow outputs be?

3. **Test the MULH overflow case.** Using the testbench pattern shown in this chapter, write test vectors for **`MULH`** that verify the upper 16 bits of the product for these input pairs: (a) 256×256, (b) -1×-1, (c) $-32,768 \times 2$. Predict the results by hand first, then verify in simulation.

4. **Investigate the NEG anomaly.** Modify the testbench to test **`NEG`** with every power of two from `0x0001` to `0x8000`. For which input is `flag_v` set? Explain why this is the only case.

5. **Design a flags-only ALU output.** Some architectures have a separate "flags ALU" that generates flags without producing a result, saving power when only a comparison is needed. Sketch a Verilog module that takes two 16-bit inputs and produces only N, Z, C, V for subtraction, without generating the full 16-bit result. What hardware resources does this save compared to the full ALU?

What's Next

The ALU can compute any arithmetic or logical result the SAMPO needs, but it operates on raw data presented to its inputs. Where do those operands come from? The answer is the register file—a 16-entry array of 16-bit registers with dual read ports, a hardwired zero register, and a bank-switching trick for fast context saves. In the next chapter, we will build it.

Chapter 19

The Register File

Registers are the fastest storage a CPU has. Every arithmetic operation reads its operands from and writes its result to the register file, making it the most heavily accessed component in the entire design. In this chapter, we examine the SAMPO's register file: a 16-entry array of 16-bit registers with dual read ports, a dedicated stack pointer write port, a hardwired zero register, and a bank-switching mechanism for fast context saves. The implementation lives in `verilog/rtl/regfile.v`—a compact module that nonetheless embodies several important design decisions.

19.1 Register Array Design

The SAMPO provides 16 general-purpose registers, `R0` through `R15`, each 16 bits wide. In addition, eight *alternate* registers shadow `R4` through `R11`, used by the **`EXX`** instruction for bank switching. The module declaration reflects this dual-array structure:

Listing 19.1: Register file module declaration

```
module regfile (
    input  wire        clk,
    input  wire        rst_n,
    input  wire [3:0]  rd_addr1,
    input  wire [3:0]  rd_addr2,
    input  wire [3:0]  wr_addr,
    output wire [15:0] rd_data1,
    output wire [15:0] rd_data2,
    input  wire [15:0] wr_data,
    input  wire        wr_en,
    input  wire        exx,
    // Second write port for SP updates (PUSH/POP)
```

```
    input  wire [15:0] sp_wr_data,
    input  wire        sp_wr_en
);
```

This interface provides:

- **Two read ports** (`rd_addr1`/`rd_data1` and `rd_addr2`/`rd_data2`): the CPU reads two operands simultaneously during the DECODE phase.
- **One general write port** (`wr_addr`/`wr_data`/`wr_en`): used during WRITEBACK to store results.
- **One dedicated SP write port** (`sp_wr_data`/`sp_wr_en`): allows updating the stack pointer (`R2`) simultaneously with a general register write, essential for **`POP`**.
- **An EXX signal**: swaps registers `R4`–`R11` with their alternates in a single clock cycle.

Internally, the registers are stored as two Verilog arrays:

Listing 19.2: Register storage arrays

```
// Main register file (R0-R15)
reg [15:0] regs [0:15];

// Alternate registers (R4'-R11', stored as indices 0-7)
reg [15:0] alt_regs [0:7];
```

The main array `regs` holds all 16 architectural registers. The alternate array `alt_regs` holds eight shadow copies, indexed 0 through 7, corresponding to `R4`′ through `R11`′. When **`EXX`** fires, these two sets swap.

Register Conventions

The SAMPO assigns special roles to several registers by convention:

- `R0` (`ZERO`): hardwired to zero, writes are discarded.
- `R1` (`RA`): return address, set by **`JALR`**/**`JALX`**.
- `R2` (`SP`): stack pointer, manipulated by **`PUSH`**/**`POP`**.
- `R3` (`FP`/`GP`): frame pointer or global pointer.
- `R4`–`R7` (`A0`–`A3`): function arguments.
- `R8`–`R11` (`T0`–`T3`): temporaries.
- `R12`–`R15` (`S0`–`S3`): callee-saved registers.

19.2 Read and Write Ports

19.2.1 Dual Read Ports

The register file provides two independent, *combinational* read ports. Each port takes a 4-bit address and produces a 16-bit data output:

Listing 19.3: Combinational read ports

```
// Read ports - R0 always returns 0
assign rd_data1 = (rd_addr1 == 4'h0) ? 16'h0000
                                     : regs[rd_addr1];
assign rd_data2 = (rd_addr2 == 4'h0) ? 16'h0000
                                     : regs[rd_addr2];
```

These are `assign` statements—pure combinational logic with no clock dependency. When the CPU places an address on `rd_addr1`, the corresponding register value appears on `rd_data1` after a brief propagation delay. Both ports can be read simultaneously, which the CPU uses during the DECODE phase to fetch both source operands (`rs1` and `rs2`) in the same clock cycle.

The conditional expression in each `assign` implements the `R0` hardwired-zero behavior, which we discuss in detail in Section 19.3.

Why Combinational Reads?

Making the read ports combinational (as opposed to synchronous, where data appears one cycle after the address) allows the CPU to read operands and latch them in the same cycle. This is critical for the multi-cycle FSM architecture: the DECODE state reads register values, and they must be stable by the end of that cycle so they can be latched into internal registers (`rs1_val` and `rs2_val` in `cpu.v`) on the next clock edge.

19.2.2 The General Write Port

The primary write port is synchronous—writes take effect on the rising edge of the clock:

Listing 19.4: General write port (inside the synchronous always block)

```
if (wr_en && (wr_addr != 4'h0)) begin
    // Write to register (except R0)
    regs[wr_addr] <= wr_data;
end
```

Two conditions must be met for a write to occur: `wr_en` must be asserted, and the destination must not be `R0`. The CPU asserts `wr_en` during the WRITEBACK state for instructions that produce a result. The guard on `wr_addr != 4'h0` provides a second layer of protection for the zero register, complementing the read-side override.

19.2.3 The Dedicated SP Write Port

The second write port exists solely to update `R2` (the stack pointer):

Listing 19.5: Dedicated SP write port

```
// SP write port (R2) - for PUSH/POP stack pointer updates
// This allows writing both Rd and SP in the same cycle
if (sp_wr_en) begin
    regs[2] <= sp_wr_data;  // R2 = SP
end
```

This port is necessary because of the **POP** instruction. When the CPU executes **POP Rd**, it must perform two register writes in the same cycle:

1. Write the loaded value from memory into `Rd`.
2. Increment the stack pointer: `R2` = `R2` + 2.

With a single write port, these would require two separate clock cycles, adding latency to every **POP** instruction. The dedicated SP port allows both writes to happen simultaneously during WRITEBACK.

Why Not a Full Dual-Write Port?

A general-purpose dual-write register file would require more complex arbitration logic to handle the case where both ports write to the same address. By restricting the second port to always write `R2`, the design avoids this complexity. The only conflict scenario would be **POP R2** (popping into the stack pointer itself), which is an unusual but valid operation—the SP port write and the general port write would both target `R2`, with the SP port's value taking effect since it is assigned last in the `always` block.

19.3 R0 Hardwired to Zero

The SAMPO follows the RISC tradition of hardwiring register 0 to zero. This is implemented with a two-pronged approach:

Read side: The `assign` statements for both read ports return `0x0000` when the address is `0x0`:

Listing 19.6: R0 read-side zero enforcement

```
assign rd_data1 = (rd_addr1 == 4'h0) ? 16'h0000
                                     : regs[rd_addr1];
```

Write side: The write enable check explicitly rejects writes to address 0:

Listing 19.7: R0 write-side protection

```
if (wr_en && (wr_addr != 4'h0)) begin
    regs[wr_addr] <= wr_data;
end
```

The testbench verifies both aspects—first confirming that `R0` reads as zero, then attempting to write to it and confirming the write is silently discarded:

Listing 19.8: Testbench: R0 is hardwired to zero

```
// Test R0 hardwired to zero
check_read(4'h0, 16'h0000);

// Try to write to R0 - should be ignored
wr_addr = 4'h0;
wr_data = 16'hDEAD;
wr_en = 1;
@(posedge clk);
wr_en = 0;
#1;
check_read(4'h0, 16'h0000);
```

Even writing `0xDEAD` to `R0` has no effect—it still reads as zero.

Why a Zero Register?

A hardwired zero register may seem wasteful—it sacrifices one register from the file. But it provides enormous utility:

- **ADD Rd, Rs, R0** becomes a move without needing a separate MOV instruction in the ALU.
- **SUB Rd, R0, Rs** becomes negation.
- **CMP Rs, R0** tests whether a register is zero.
- Branch targets can use R0 as a "discard" destination for instructions where only the flags matter.

The SAMPO does include a dedicated **MOV** instruction, but the zero register still simplifies many code patterns and decoder paths. This convention is shared with MIPS, RISC-V, and many other RISC architectures.

19.4 EXX Bank Switching

The **EXX** instruction, inspired by the Zilog Z80's exchange instruction, swaps registers R4 through R11 with an alternate set. This provides a fast way to save and restore eight registers without touching the stack, which is useful for interrupt service routines or context switches.

19.4.1 The Swap Mechanism

When the exx signal is asserted, the register file swaps the contents of regs[4] through regs[11] with alt_regs[0] through alt_regs[7]:

Listing 19.9: EXX bank swap implementation

```
// Handle EXX: swap R4-R11 with alternate set
if (exx) begin
    // Swap R4-R11 with alt_regs[0-7]
    for (i = 0; i < 8; i = i + 1) begin
        regs[i + 4] <= alt_regs[i];
        alt_regs[i] <= regs[i + 4];
    end
end
```

This is a true swap—both assignments use non-blocking (<=) assignment, so they happen simultaneously at the clock edge. The value of regs[i+4] that feeds into alt_regs[i] is the value *before* the clock edge, not the new value being assigned. This is a fundamental property of non-blocking assignment in Verilog.

Non-Blocking Assignment is Essential Here

If blocking assignment (=) were used instead, the swap would fail. The first iteration would overwrite `regs[4]` with `alt_regs[0]`, and then the assignment `alt_regs[0] = regs[4]` would capture the *already-overwritten* value. Non-blocking assignment (<=) ensures both sides of the swap see the pre-edge values, implementing a true parallel swap without temporary variables.

19.4.2 What EXX Does Not Touch

Only `R4` through `R11` participate in the exchange. The "system" registers—`R0` (zero), `R1` (return address), `R2` (stack pointer), `R3` (frame pointer)—are left unchanged. The callee-saved registers `R12` through `R15` are also not swapped.

Table 19.1: EXX swap scope

Registers	Alias	Swapped by EXX?
`R0`–`R3`	ZERO, RA, SP, FP	No
`R4`–`R7`	A_0–A_3	Yes
`R8`–`R11`	T_0–T_3	Yes
`R12`–`R15`	S_0–S_3	No

This design means an interrupt handler can execute **`EXX`**, use `R4`–`R11` freely, execute **`EXX`** again to restore the original values, and return—without a single memory access for register saving.

19.4.3 EXX vs. Regular Writes

The register file's `always` block is structured so that **`EXX`** and regular writes are mutually exclusive:

Listing 19.10: EXX and write port are mutually exclusive

```
if (exx) begin
    // Swap R4-R11 with alt_regs[0-7]
    for (i = 0; i < 8; i = i + 1) begin
        regs[i + 4] <= alt_regs[i];
        alt_regs[i] <= regs[i + 4];
    end
end
// Handle register write (can happen same cycle as EXX)
// Note: if both exx and wr_en, write happens to swapped value
else begin
    if (wr_en && (wr_addr != 4'h0)) begin
        regs[wr_addr] <= wr_data;
    end
```

```
    if (sp_wr_en) begin
        regs[2] <= sp_wr_data;
    end
end
```

The `else` ensures that when `exx` is active, the normal write path is disabled. This is correct because the CPU state machine activates `exx` during the EXECUTE phase and performs register writes during WRITEBACK—they never occur in the same clock cycle.

19.4.4 Verifying the Double Swap

The testbench verifies the key property that two consecutive **EXX** operations restore the original register state:

Listing 19.11: Testbench: EXX double-swap verification

```
// Write known values to R4-R11
for (i = 4; i < 12; i = i + 1) begin
    wr_addr = i[3:0];
    wr_data = 16'hAA00 + i;
    wr_en = 1;
    @(posedge clk);
    wr_en = 0;
end

// Execute first EXX
exx = 1;
@(posedge clk);
exx = 0;

// R4-R11 now have alternate values (0 from reset)
for (i = 4; i < 12; i = i + 1) begin
    check_read(i[3:0], 16'h0000);
end

// R0-R3 and R12-R15 should be unchanged
check_read(4'h0, 16'h0000);
check_read(4'h1, 16'h1001);
check_read(4'hC, 16'h100C);
check_read(4'hF, 16'h100F);

// Execute second EXX - original values restored
exx = 1;
@(posedge clk);
```

```
exx = 0;

for (i = 4; i < 12; i = i + 1) begin
    check_read(i[3:0], 16'hAA00 + i);
end
```

The test writes distinctive values (`0xAA04` through `0xAA0B`) to `R4`–`R11`, then executes **`EXX`**. Since the alternate registers were cleared to zero at reset, `R4`–`R11` should now read as zero. Registers outside the swap range should retain their values. A second **`EXX`** restores the original values.

19.5 Synchronous Write, Combinational Read

The register file's timing model is a deliberate asymmetry:

- **Reads are combinational**: output data changes immediately when the address input changes (within propagation delay). No clock edge is needed.
- **Writes are synchronous**: data is captured on the rising edge of `clk` when `wr_en` is asserted. The new value is visible on the next read after the write edge.

19.5.1 The Read-After-Write Question

What happens if you read and write the same register address in the same clock cycle? With the SAMPO's design:

- The *read* returns the *old* value (the value before the write takes effect at the clock edge).
- The *write* takes effect at the clock edge.
- On the next cycle, a read would return the newly written value.

This is consistent with the "write-first" or "read-old-data" convention. The SAMPO's multi-cycle FSM avoids read-after-write hazards by structuring the pipeline so that reads (DECODE) and writes (WRITEBACK) always occur in different clock cycles.

19.5.2 Reset Behavior

On reset, all registers (both main and alternate) are cleared to zero:

Listing 19.12: Register file reset

```
if (!rst_n) begin
    // Reset all registers to 0
    for (i = 0; i < 16; i = i + 1) begin
        regs[i] <= 16'h0000;
```

```
    end
    for (i = 0; i < 8; i = i + 1) begin
        alt_regs[i] <= 16'h0000;
    end
end
```

This is an asynchronous reset (triggered by `negedge rst_n` in the sensitivity list). All 24 registers—16 main plus 8 alternate—are zeroed. In practice, software must initialize the stack pointer (R2) before using **PUSH** or **POP**, since it starts at zero.

Stack Pointer Initialization

Since all registers are zero after reset, the first instructions of any SAMPO program should initialize the stack pointer. A common pattern is:

```
    LIX SP, 0xFFFE    ; Stack grows downward from top of RAM
```

This places the stack at the top of the 64KB address space, growing downward as values are pushed.

19.5.3 FPGA Synthesis Considerations

On an FPGA, the register file will be synthesized from flip-flops rather than block RAM, because the combinational read ports and small size (24 entries × 16 bits = 384 bits total) make flip-flop implementation efficient. The ECP5 FPGA used by the SAMPO has plentiful flip-flop resources, so this is not a concern for our design. Larger register files (32 or 64 entries) in more complex CPUs might warrant block RAM with a registered read path, but that would add a cycle of latency.

Exercises

1. **Trace a PUSH/POP sequence.** Walk through the register file port usage for the sequence **PUSH R4**, **POP R5**. For each clock cycle, identify which ports are active (`rd_addr1`, `rd_addr2`, `wr_addr`, `sp_wr_data`), what data appears on each, and when the actual writes occur.

2. **EXX and interrupt handling.** Write a short SAMPO assembly sequence that demonstrates using **EXX** in an interrupt handler. The handler should save context with **EXX**, do some work using R4–R7, restore context with a second **EXX**, and return with **RETI**. How many clock cycles does this save compared to pushing and popping eight registers?

3. **POP R2 conflict.** Analyze what happens when the CPU executes `POP R2` (popping a value into the stack pointer itself). Both the general write port and the SP write port target `R2`. Based on the ordering in the `always` block, which value wins? Is this the correct behavior?

4. **Add read port forwarding.** Modify the register file to implement "write-first" forwarding: if a read and write to the same address occur in the same cycle, the read should return the *new* value being written. Write the modified read port logic and explain why the SAMPO's FSM does not actually need this feature.

What's Next

With the register file providing fast operand access and the ALU computing results, we need one more computational unit: the shifter. Shift and rotate operations are fundamental to bit manipulation, data packing, and multiplication by powers of two. The SAMPO's barrel shifter handles 16 different shift and rotate functions in a single combinational stage, and we will examine all of them in the next chapter.

Chapter 20

The Shifter

Shift and rotate operations appear everywhere in low-level programming: multiplying or dividing by powers of two, extracting bit fields from packed data, computing hash functions, and implementing serial protocols. Many CPU architectures fold shifting into the ALU, but the SAMPO uses a separate dedicated module—the barrel shifter. This keeps the ALU focused on arithmetic and logical operations while giving the shifter its own clean namespace of 16 functions. In this chapter, we explore the implementation in `verilog/rtl/shifter.v`, trace every operation, and examine how each one sets the carry output.

20.1 Barrel Shifter Design

Barrel Shifter

A *barrel shifter* is a combinational circuit that can shift or rotate its input by any number of positions in a single clock cycle. Unlike a serial shifter, which shifts one bit per clock, a barrel shifter uses a network of multiplexers to select the correct output in constant time. The name comes from the image of rotating a barrel to different positions—the output "pops out" at the desired alignment without stepping through intermediate positions.

The SAMPO's shifter module has a straightforward interface:

Listing 20.1: Shifter module port declaration

```
module shifter (
    input  wire [15:0] value,
    input  wire [3:0]  func,
    input  wire        carry_in,
```

```
    output reg  [15:0] result,
    output reg         carry_out
);
```

The inputs are:

- `value`: the 16-bit value to shift or rotate.
- `func`: a 4-bit function code selecting one of 16 operations.
- `carry_in`: the current carry flag, needed for rotate-through-carry operations.

The outputs are:

- `result`: the 16-bit shifted or rotated value.
- `carry_out`: the carry flag produced by the operation (typically the last bit shifted out).

Like the ALU, the shifter is purely combinational—no clock, no state. The CPU drives the inputs during the EXECUTE phase and latches the outputs into `exec_result` (in `cpu.v`), with the carry output updating the C flag in the FLAGS register.

20.1.1 The 16 Shift Functions

The function codes are defined in `sampo_pkg.vh` and organized into three groups based on shift amount:

The first eight operations (codes `0x0`–`0x7`) shift or rotate by 1 position, plus the byte swap. The next four (`0x8`–`0xB`) shift by 4 positions—one nibble. The final four (`0xC`–`0xF`) shift by 8 positions—one byte. This encoding gives the programmer common shift amounts without requiring multi-cycle or variable-amount shift hardware in the fast path.

Variable Shifts Use the ALU

For arbitrary shift amounts (e.g., "shift left by the value in R5"), the SAMPO uses the extended instructions **SLLX**, **SRLX**, and **SRAX**, which route through the ALU's shift operations (`ALU_SLL`, `ALU_SRL`, `ALU_SRA`) rather than the dedicated shifter. See Chapter 18 for details.

The entire module body is a single combinational `always @(*)` block with default assignments followed by a `case` statement:

Listing 20.2: Shifter default assignments

```
always @(*) begin
    // Default values
    result = value;
```

Table 20.1: Shifter function codes

Code	Define	Operation	Amount
0x0	SHIFT_SLL1	Shift left logical	1
0x1	SHIFT_SRL1	Shift right logical	1
0x2	SHIFT_SRA1	Shift right arithmetic	1
0x3	SHIFT_ROL1	Rotate left	1
0x4	SHIFT_ROR1	Rotate right	1
0x5	SHIFT_RCL1	Rotate left through carry	1
0x6	SHIFT_RCR1	Rotate right through carry	1
0x7	SHIFT_SWAP	Byte swap	—
0x8	SHIFT_SLL4	Shift left logical	4
0x9	SHIFT_SRL4	Shift right logical	4
0xA	SHIFT_SRA4	Shift right arithmetic	4
0xB	SHIFT_ROL4	Rotate left	4
0xC	SHIFT_SLL8	Shift left logical	8
0xD	SHIFT_SRL8	Shift right logical	8
0xE	SHIFT_SRA8	Shift right arithmetic	8
0xF	SHIFT_ROL8	Rotate left	8

```
    carry_out = carry_in;

    case (func)
        // ... operations
    endcase
end
```

The defaults are significant: if no case matches (which cannot happen since all 16 codes are assigned), the result passes through unchanged and the carry is preserved. This is also the behavior for a shift of zero—though no function code explicitly means "shift by 0," the default would produce that behavior.

20.2 Shift Functions: SLL, SRL, SRA

The three fundamental shift operations—logical left, logical right, and arithmetic right—form the backbone of the shifter. Each is available in three variants: shift by 1, by 4, and by 8.

20.2.1 Shift Left Logical (SLL)

A logical left shift moves all bits toward the most significant end, filling vacated low bits with zeros. The bit that "falls off" the top becomes the carry output.

Listing 20.3: Shift left logical by 1

```
`SHIFT_SLL1: begin
    // Shift left logical by 1
    result = {value[14:0], 1'b0};
    carry_out = value[15];
end
```

The Verilog concatenation `{value[14:0], 1'b0}` takes bits 14 down to 0 of the input and appends a zero at the bottom. Bit 15 of the original value is expelled into `carry_out`. This is equivalent to multiplying by 2 (for unsigned values), which is why left shifts are commonly used as a fast multiply.

The by-4 and by-8 variants follow the same pattern:

Listing 20.4: Shift left logical by 4 and by 8

```
`SHIFT_SLL4: begin
    // Shift left logical by 4
    result = {value[11:0], 4'b0000};
    carry_out = value[15];  // Last bit shifted out
end

`SHIFT_SLL8: begin
    // Shift left logical by 8
    result = {value[7:0], 8'b0};
    carry_out = value[15];
end
```

SLL by 4 for Nibble Assembly

Shifting left by 4 is useful for building values nibble by nibble. For example, to construct a specific value in a register:

```
    ADDI R4, 0x05    ; R4 = 0x0005
    SLL4 R4, R4      ; R4 = 0x0050
    ADDI R4, 0x03    ; R4 = 0x0053
```

Each **SLL4** shifts the accumulated value up by one nibble, making room for the next digit.

20.2.2 Shift Right Logical (SRL)

A logical right shift moves bits toward the least significant end, filling vacated high bits with zeros. The bit that falls off the bottom becomes carry:

Listing 20.5: Shift right logical by 1

```
`SHIFT_SRL1: begin
    // Shift right logical by 1
    result = {1'b0, value[15:1]};
    carry_out = value[0];
end
```

The by-4 and by-8 variants:

Listing 20.6: Shift right logical by 4 and 8

```
`SHIFT_SRL4: begin
    // Shift right logical by 4
    result = {4'b0000, value[15:4]};
    carry_out = value[3];  // Last bit shifted out
end

`SHIFT_SRL8: begin
    // Shift right logical by 8
    result = {8'b0, value[15:8]};
    carry_out = value[7];
end
```

Logical right shift divides by powers of 2 for *unsigned* values. For signed values, we need arithmetic right shift.

20.2.3 Shift Right Arithmetic (SRA)

An arithmetic right shift preserves the sign bit by replicating the most significant bit into the vacated positions:

Listing 20.7: Shift right arithmetic by 1

```
`SHIFT_SRA1: begin
    // Shift right arithmetic by 1 (preserve sign)
    result = {value[15], value[15:1]};
    carry_out = value[0];
end
```

The key difference from **SRL** is the fill value: `value[15]` (the sign bit) replaces the hardcoded `1'b0`. If the value is positive (bit 15 = 0), SRA and SRL produce identical results. If negative (bit 15 = 1), SRA fills with ones, maintaining the negative sign.

Listing 20.8: Shift right arithmetic by 4 and 8

```
`SHIFT_SRA4: begin
    // Shift right arithmetic by 4
    result = {{4{value[15]}}, value[15:4]};
    carry_out = value[3];
end

`SHIFT_SRA8: begin
    // Shift right arithmetic by 8
    result = {{8{value[15]}}, value[15:8]};
    carry_out = value[7];
end
```

The Verilog replication syntax `{4{value[15]}}` creates 4 copies of the sign bit, which are concatenated with the shifted value. This sign extension ensures that **SRA** correctly divides signed values by powers of 2 (rounding toward negative infinity).

SRA Rounding Behavior

For negative odd numbers, **SRA** by 1 rounds toward negative infinity, not toward zero. For example, -3 (`0xFFFD`) shifted right by 1 produces -2 (`0xFFFE`), not -1. This matches the mathematical definition of floor division by 2, which is the conventional behavior for arithmetic right shifts across most architectures.

20.3 Rotate Functions: ROL, ROR

Rotate operations are circular shifts: bits that fall off one end wrap around to the other, so no information is lost. The SAMPO provides several rotate variants.

20.3.1 Rotate Left (ROL)

Listing 20.9: Rotate left by 1

```
`SHIFT_ROL1: begin
    // Rotate left by 1
    result = {value[14:0], value[15]};
    carry_out = value[15];
end
```

Bit 15 wraps around to bit 0, while all other bits shift left by one position. The carry output captures the bit that rotated around. The by-4 and by-8 variants:

Listing 20.10: Rotate left by 4 and 8

```
`SHIFT_ROL4: begin
    // Rotate left by 4
    result = {value[11:0], value[15:12]};
    carry_out = value[15];
end

`SHIFT_ROL8: begin
    // Rotate left by 8 (same as byte swap with carry)
    result = {value[7:0], value[15:8]};
    carry_out = value[15];
end
```

ROL8 and SWAP

Rotate left by 8 and byte swap (**SWAP**) produce the same result: the high and low bytes are exchanged. The difference is in the carry output—**ROL8** sets carry to `value[15]`, while **SWAP** leaves carry unchanged. If you need to swap bytes without affecting flags, use **SWAP**.

20.3.2 Rotate Right (ROR)

Listing 20.11: Rotate right by 1

```
`SHIFT_ROR1: begin
    // Rotate right by 1
    result = {value[0], value[15:1]};
    carry_out = value[0];
end
```

Bit 0 wraps around to bit 15. The carry captures the bit that rotated. Note that only a by-1 variant of rotate right is provided. For larger right rotations, the programmer can use the equivalent left rotation: rotating right by n is the same as rotating left by $16 - n$.

20.3.3 Rotate Through Carry: RCL and RCR

The rotate-through-carry operations treat the carry flag as part of the value, creating a 17-bit circular shift:

Listing 20.12: Rotate left and right through carry

```
`SHIFT_RCL1: begin
    // Rotate left through carry by 1
    result = {value[14:0], carry_in};
    carry_out = value[15];
end

`SHIFT_RCR1: begin
    // Rotate right through carry by 1
    result = {carry_in, value[15:1]};
    carry_out = value[0];
end
```

In **RCL**, the carry flag shifts into bit 0, and bit 15 shifts out into the carry. In **RCR**, carry shifts into bit 15, and bit 0 shifts out into carry. The result is a 17-bit rotation where the carry flag acts as an extra bit between the MSB and LSB.

Rotate Through Carry

A *rotate through carry* treats the processor's carry flag as an extension of the register, forming a $(n + 1)$-bit ring: for the SAMPO, a 17-bit ring consisting of the 16-bit register value and the carry flag. Each rotation moves every bit one position around this ring. This is invaluable for multi-word shift operations, where a shift in one word must feed its outgoing bit into the next word.

These operations are essential for multi-precision arithmetic. To shift a 32-bit value stored across two registers (R4:R5, with R4 holding the high word), you can shift the low word first, which captures the outgoing bit in the carry, then rotate the high word through carry to incorporate it:

Listing 20.13: 32-bit left shift using RCL

```
    ; Shift R4:R5 left by 1 (R4 = high, R5 = low)
    SLL R5, R5, 1    ; Shift low word, MSB goes to carry
    RCL R4, R4, 1    ; Rotate high word through carry
```

20.4 SWAP: Byte Swap

The byte swap operation exchanges the high and low bytes of the input value:

Listing 20.14: SWAP operation

```
`SHIFT_SWAP: begin
```

```
    // Swap high and low bytes
    result = {value[7:0], value[15:8]};
    carry_out = carry_in;  // Carry unchanged
end
```

Unlike the shift and rotate operations, **SWAP** does not modify the carry flag—it passes `carry_in` through to `carry_out` unchanged. This is a deliberate design choice: byte swapping is a data formatting operation rather than an arithmetic one, so it should not disturb the condition flags.

Byte swapping is commonly needed when converting between big-endian and little-endian byte orders. Since the SAMPO is a little-endian machine, data received from big-endian network protocols or file formats can be converted with a single **SWAP** instruction:

Listing 20.15: Endianness conversion with SWAP

```
    LW R4, (R5)      ; Load 16-bit value (little-endian)
    SWAP R4, R4       ; Convert to big-endian byte order
```

20.5 Carry Output for Each Operation

The carry output from the shifter updates the C flag in the FLAGS register. The CPU handles this during the EXECUTE phase in `cpu.v`:

Listing 20.16: Shift flag update in cpu.v

```
end else if (dec_opcode == `OP_SHIFT) begin
    flags[`FLAG_N] <= shift_result[15];
    flags[`FLAG_Z] <= (shift_result == 16'h0000);
    flags[`FLAG_C] <= shift_carry;
    flags[`FLAG_V] <= 1'b0;
end
```

After a shift operation, all four relevant flags are updated: N reflects the sign of the result, Z indicates whether the result is zero, C captures the carry output from the shifter, and V is always cleared (shift operations do not generate signed overflow).

The carry output follows a consistent pattern across operations:

Table 20.2: Carry output behavior by operation type

Operation	Carry Output
SLL (any amount)	Last bit shifted out the top (`value[15]`)
SRL (any amount)	Last bit shifted out the bottom
SRA (any amount)	Last bit shifted out the bottom
ROL (any amount)	Bit that wrapped around (`value[15]`)
ROR	Bit that wrapped around (`value[0]`)
RCL	Bit expelled from MSB (`value[15]`)
RCR	Bit expelled from LSB (`value[0]`)
SWAP	Unchanged (`carry_in`)

Carry from Multi-Bit Shifts

For shifts by 4 or 8, the "last bit shifted out" is defined as the most extreme bit involved in the shift. For **SRL4**, this is `value[3]`—the lowest of the four bits being pushed out the bottom. For **SRL8**, it is `value[7]`. This convention matches most architectures and provides a useful "remainder" bit for division-by-power-of-two operations.

20.5.1 Carry as a Multi-Word Shift Connector

The carry output's role extends beyond flag setting. In multi-precision arithmetic, the carry from one shift feeds into the carry input of the next, connecting multiple registers into a single wide shift operation. The rotate-through-carry instructions (**RCL** and **RCR**) are designed specifically for this purpose.

Consider a 64-bit right shift across four registers (`R4:R5:R6:R7`, most significant word first):

Listing 20.17: 64-bit right shift using RCR

```
; Shift R4:R5:R6:R7 right by 1
SRL  R4, R4, 1    ; Shift MSW, bit 0 -> carry
RCR  R5, R5, 1    ; Carry -> bit 15, bit 0 -> carry
RCR  R6, R6, 1    ; Carry -> bit 15, bit 0 -> carry
RCR  R7, R7, 1    ; Carry -> bit 15, bit 0 -> carry (LSB lost)
```

The carry flag acts as a 1-bit conveyor belt, passing each word's outgoing bit into the next word's incoming position. Without rotate-through-carry, this operation would require explicit bit extraction and insertion, costing several additional instructions per word.

20.6 Integration with the CPU

The CPU instantiates the shifter alongside the ALU and decoder as a top-level functional unit:

Listing 20.18: Shifter instantiation in cpu.v

```
shifter shifter_inst (
    .value(shift_value),
    .func(shift_func),
    .carry_in(flags[`FLAG_C]),
    .result(shift_result),
    .carry_out(shift_carry)
);
```

The `carry_in` is connected directly to the carry flag bit in the FLAGS register, providing the current carry state for **RCL** and **RCR** operations. The CPU's input selection logic drives the shifter inputs:

Listing 20.19: Shifter input selection in cpu.v

```
always @(*) begin
    shift_value = rs1_val;
    shift_func = dec_shift_func;
end
```

The source register value (`rs1_val`, latched during the DECODE phase) feeds the shifter's data input, and the function code from the decoder selects the operation. During the EXECUTE phase, the CPU checks whether the current instruction is a shift operation and, if so, routes the shifter's output to the execution result instead of the ALU's output:

Listing 20.20: Selecting shifter result over ALU result

```
if (dec_opcode == `OP_SHIFT) begin
    exec_result <= shift_result;
end else begin
    exec_result <= alu_result;
end
```

This multiplexing means the ALU and shifter are both computing results in parallel during every EXECUTE phase, but only one result is captured. The unused output is simply discarded. This is a common trade-off in hardware design: computing speculatively and selecting later is cheaper than conditional computation.

Exercises

1. **Trace a multi-word shift.** Starting with the 32-bit value `0x12345678` stored in `R4:R5` (high:low), trace the values of both registers and the carry flag through a 4-position left shift implemented using **`SLL4`** and **`ROL4`** (or **`RCL`**). What is the final 32-bit value?

2. **Implement SWAP using other operations.** The **`SWAP`** instruction exchanges the high and low bytes. Show how to achieve the same result using only **`SLL8`**, **`SRL8`**, and **`OR`**, without using **`SWAP`**. How many instructions and clock cycles does your sequence require compared to a single **`SWAP`**?

3. **Test all carry outputs.** Write a testbench (following the pattern in `alu_tb.v`) that exercises each of the 16 shifter functions with the input value `0xA5C3`. For each operation, predict the result and carry output by hand, then verify in simulation.

4. **Sign-extension via SRA.** To sign-extend an 8-bit value in the low byte of a register to a full 16-bit signed value, one approach is to shift left by 8 then arithmetic-shift right by 8. Write a SAMPO assembly sequence that does this and explain why it works. What are the limitations compared to the **`LB`** (load byte, sign-extended) instruction?

5. **Design a configurable shifter.** The SAMPO shifter uses fixed shift amounts (1, 4, 8) encoded in the function code. Sketch a design for a shifter that takes a separate 4-bit shift amount input and implements SLL, SRL, SRA, and ROL for any shift from 0 to 15. How would the Verilog differ? What are the hardware cost implications on an FPGA?

What's Next

We now have three computational modules—the ALU, the register file, and the shifter—but nothing to direct them. How does the CPU know which ALU operation to perform, which registers to read, or whether an instruction needs memory access? That is the job of the instruction decoder. In the next chapter, we crack open the SAMPO's 16-bit instruction word, extract every field, and generate the web of control signals that orchestrate execution.

Chapter 21

The Instruction Decoder

If the ALU is the muscle of the CPU, the decoder is the nervous system. Every instruction that enters the SAMPO passes through the decoder, which cracks open the 16-bit instruction word, extracts register addresses and immediate values, and generates the constellation of control signals that direct the rest of the datapath. A mistake in the decoder means a wrong register read, a misrouted ALU operation, or a branch that goes to the wrong address. In this chapter, we walk through `verilog/rtl/decode.v` line by line, understanding how each instruction format maps to control signals.

21.1 Opcode Field Extraction

The SAMPO's instruction word is divided into fields whose positions are fixed by convention. The decoder extracts them with simple wire assignments:

Listing 21.1: Instruction field extraction

```
// Extract instruction fields
assign opcode = instr[15:12];
assign rd     = instr[11:8];
assign rs1    = instr[7:4];
assign rs2    = instr[3:0];
assign func   = instr[3:0];
```

These five assignments decompose any 16-bit instruction into its constituent parts:

- `opcode` (bits [15:12]): the 4-bit primary opcode, selecting one of 16 instruction categories.
- `rd` (bits [11:8]): the destination register address.

- `rs1` (bits [7:4]): the first source register address.
- `rs2` (bits [3:0]): the second source register address.
- `func` (bits [3:0]): the function code, which shares the same bits as `rs2`.

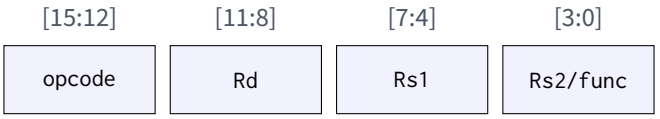


Figure 21.1: Standard R-type instruction field layout

Notice that `rs2` and `func` are assigned from the same bits. This is the fundamental encoding trick of the SAMPO: in R-type instructions (like **`ADD Rd, Rs1, Rs2`**), bits [3:0] hold the second source register. In instructions that need a function code instead (like shift operations where bits [3:0] select SLL vs. SRL vs. SRA), the same bits serve as the function selector. The decoder determines which interpretation to use based on the opcode.

Additional fields are derived for specific instruction categories:

Listing 21.2: Derived instruction fields

```
// Extended sub-opcode
assign ext_sub = instr[3:0];

// Shift function (same as func for SHIFT opcode)
assign shift_func = func;

// MulDiv function
assign muldiv_func = func;

// Branch condition (stored in rd field for branch instructions)
assign branch_cond = rd;
```

The `branch_cond` assignment is worth noting: for branch instructions (**`BEQ`**, **`BNE`**, etc.), the condition code occupies the same bits as `rd`—bits [11:8]. Branch instructions do not have a destination register, so those bits are repurposed. This field reuse is how the SAMPO fits 16 branch conditions into the same instruction format.

21.2 Control Signal Generation

The decoder's primary output is a set of control signals that tell the rest of the CPU what to do. These are generated in a single combinational `always @(*)` block containing a case statement on the opcode:

Listing 21.3: Decoder control signal defaults

```
always @(*) begin
    // Default all outputs
    alu_op      = `ALU_PASS_B;
    mem_load    = 1'b0;
    mem_store   = 1'b0;
    mem_byte    = 1'b0;
    mem_signed  = 1'b0;
    mem_offset  = 16'h0000;
    reg_write   = 1'b0;
    is_jump     = 1'b0;
    is_jump_reg = 1'b0;
    is_branch   = 1'b0;
    is_call     = 1'b0;
    is_ret      = 1'b0;
    is_halt     = 1'b0;
    is_nop      = 1'b0;
    is_exx      = 1'b0;
    is_push     = 1'b0;
    is_pop      = 1'b0;
    is_cmp      = 1'b0;
    is_test     = 1'b0;
    // ... and many more
```

Every control signal defaults to its inactive state at the top of the block. Only the relevant signals are set to 1 within each opcode's case branch. This default-then-override pattern ensures no latch inference and makes it clear which signals each instruction activates.

Control Signals

A *control signal* is a single-bit wire that enables or configures a specific datapath component. The decoder generates control signals for:

- **ALU operation** (`alu_op`): which function the ALU performs.
- **Register write** (`reg_write`): whether the result is written to a register.
- **Memory access** (`mem_load`, `mem_store`): whether a memory read or write occurs.
- **Branch/Jump** (`is_branch`, `is_jump`): whether the PC changes.
- **Special operations**: flags for **`EXX`**, **`PUSH`**, **`POP`**, **`HALT`**, etc.

Let us trace through the simplest cases first. The basic ALU instructions set only two things—the ALU operation and the register write enable:

Listing 21.4: Decode for basic ALU instructions

```
`OP_ADD: begin
    alu_op = `ALU_ADD;
    reg_write = 1'b1;
end

`OP_SUB: begin
    alu_op = `ALU_SUB;
    reg_write = 1'b1;
end

`OP_AND: begin
    alu_op = `ALU_AND;
    reg_write = 1'b1;
end

`OP_OR: begin
    alu_op = `ALU_OR;
    reg_write = 1'b1;
end

`OP_XOR: begin
    alu_op = `ALU_XOR;
    reg_write = 1'b1;
end
```

Each of these instructions reads two source registers, performs an ALU operation, and writes the result to the destination register. The memory signals, branch signals, and special operation flags all remain at their defaults (inactive).

21.3 Register Selection: Rd, Rs1, Rs2

The decoder extracts register addresses from fixed bit positions, but different instruction types interpret those positions differently. The CPU's register file control logic (in `cpu.v`) uses the decoder outputs along with additional context to configure the read and write addresses.

For most R-type instructions, the mapping is straightforward:

- `rd` = bits [11:8] = destination register
- `rs1` = bits [7:4] = first source operand
- `rs2` = bits [3:0] = second source operand

But several instructions remap these fields. The CPU handles this during the DECODE phase:

Listing 21.5: Register address remapping in cpu.v

```
`ST_DECODE: begin
    if (dec_opcode == `OP_ADDI) begin
        rf_rd_addr1 = dec_rd;  // ADDI uses Rd as source
    end else if (dec_is_cmp || dec_is_test) begin
        rf_rd_addr1 = dec_rd;   // CMP/TEST: Rd is first operand
        rf_rd_addr2 = dec_rs1;  // Rs1 is second operand
    end else if (dec_is_push) begin
        rf_rd_addr1 = `REG_SP;
        rf_rd_addr2 = dec_rs1;  // Source register
    end else if (dec_is_pop) begin
        rf_rd_addr1 = `REG_SP;
    end else if (dec_is_muldiv) begin
        rf_rd_addr1 = dec_rd;   // MUL/DIV: Rd is first operand
        rf_rd_addr2 = dec_rs1;
    end else if (dec_opcode == `OP_STORE) begin
        rf_rd_addr1 = dec_rs1;  // Base address
        rf_rd_addr2 = dec_rd;   // Data to store (in rd field)
    end
end
```

Several noteworthy remappings:

- **ADDI**: This is a two-operand instruction (`Rd = Rd + imm8`). The destination register is also the source, so `rf_rd_addr1` reads from `dec_rd` instead of the default `dec_rs1`.
- **CMP/TEST**: These comparison instructions use the `rd` field as the first operand (the value being compared), not as a destination. The result is discarded—only flags are updated.

- **PUSH**: Reads the stack pointer (`R2`) as the base address and the source register from `dec_rs1` as the value to push.
- **MUL/DIV**: Two-operand instructions where `Rd = Rd op Rs1`. The destination register is also the first source operand.
- **STORE**: The "destination" field actually holds the *source* data register (the value to store), while `rs1` holds the base address register.

Store Encoding Reversal

In store instructions, the register field at bits [11:8] (normally `rd`, the *destination*) holds the *source data* register. This is because stores write to memory, not to a register, so there is no destination register. The field is repurposed for the data source. This means the CPU must swap the read addresses: `rf_rd_addr1` gets the base address from `dec_rs1`, while `rf_rd_addr2` gets the data from `dec_rd`. Forgetting this swap is a common source of bugs in CPU implementations.

21.4 Immediate Extraction and Sign Extension

Three types of immediate values are extracted from the instruction word:

Listing 21.6: Immediate value extraction

```
// Sign-extend 8-bit immediate (bits 7:0)
assign imm8_sext = {{8{instr[7]}}, instr[7:0]};

// Branch offset: sign-extend 8-bit offset and multiply by 2
assign offset8 = {{7{instr[7]}}, instr[7:0], 1'b0};

// Jump offset: sign-extend 12-bit offset and multiply by 2
assign offset12 = {{3{instr[11]}}, instr[11:0], 1'b0};
```

21.4.1 The 8-Bit Signed Immediate

The `imm8_sext` output sign-extends the 8-bit immediate in bits [7:0] to 16 bits. The Verilog replication `{8{instr[7]}}` creates 8 copies of the sign bit (bit 7), prepending them to the original 8 bits. This is used by **`ADDI`**, which adds a signed 8-bit immediate (-128 to $+127$) to the destination register.

21.4.2 The Branch Offset

Branch instructions encode a signed 8-bit offset that is multiplied by 2 (since all instructions are word-aligned at even addresses). The `offset8` output performs this in one step: it sign-extends to

15 bits and appends a zero bit at the bottom, effectively computing `sign_extend(offset) * 2`. The resulting range is -256 to $+254$ bytes from the instruction following the branch.

21.4.3 The Jump Offset

Jump instructions use a 12-bit offset occupying the entire lower 12 bits of the instruction word. The `offset12` output sign-extends to 15 bits and appends a zero, giving a range of -4096 to $+4094$ bytes. This wider range reflects the fact that jumps typically target more distant locations than conditional branches.

Why Multiply by 2?

Since all SAMPO instructions are 16 bits (2 bytes) wide, instruction addresses are always even. Encoding the offset as a number of *words* rather than bytes doubles the range. A branch with an 8-bit offset field can reach ± 128 words = ± 256 bytes, instead of just ± 128 bytes. This word-offset encoding is shared with ARM (Thumb), RISC-V compressed, and many other RISC architectures.

21.5 Extended Instruction Detection

Extended instructions use opcode `0xF` and occupy two 16-bit words. The decoder detects them with a single comparison:

Listing 21.7: Extended instruction detection

```
assign is_extended = (opcode == `OP_EXTENDED);
```

When `is_extended` is asserted, the CPU's state machine inserts an additional FETCH_EXT state to read the second instruction word (`imm16`). The extended sub-opcode in bits [3:0] of the first word determines the specific operation:

Listing 21.8: Extended instruction decode

```
`OP_EXTENDED: begin
    case (ext_sub)
        `EXT_ADDIX: begin
            alu_op = `ALU_ADD;
            reg_write = 1'b1;
        end
        `EXT_SUBIX: begin
            alu_op = `ALU_SUB;
            reg_write = 1'b1;
        end
```

```
        // ... more cases
        `EXT_LIX: begin
            alu_op = `ALU_PASS_B;
            reg_write = 1'b1;
        end
        `EXT_JX: begin
            is_jump = 1'b1;
        end
        `EXT_JALX: begin
            is_jump = 1'b1;
            is_call = 1'b1;
            reg_write = 1'b1;
        end
    endcase
end
```

The extended instructions provide full 16-bit immediate operands. For example, **`LIX Rd, imm16`** loads an arbitrary 16-bit constant into a register, and **`ADDIX Rd, Rs1, imm16`** adds a full 16-bit immediate. The trade-off is size: extended instructions occupy 4 bytes instead of 2, and require an extra clock cycle for the second fetch.

Table 21.1: Extended instruction sub-opcodes

Sub	Mnemonic	Operation
0x0	**ADDIX**	Rd = Rs1 + imm16
0x1	**SUBIX**	Rd = Rs1 - imm16
0x2	**ANDIX**	Rd = Rs1 & imm16
0x3	**ORIX**	Rd = Rs1 | imm16
0x4	**XORIX**	Rd = Rs1 ^ imm16
0x5	**LWX**	Rd = mem16[Rs1 + imm16]
0x6	**SWX**	mem16[Rs1 + imm16] = Rd
0x7	**LIX**	Rd = imm16
0x8	**JX**	PC = imm16
0x9	**JALX**	RA = PC + 4; PC = imm16
0xA	**CMPIX**	flags = Rd - imm16
0xB	**INX**	Rd = port_read(imm16[7:0])
0xC	**OUTX**	port_write(imm16[7:0], Rs1)
0xD	**SLLX**	Rd = Rs1 « imm16[3:0]
0xE	**SRLX**	Rd = Rs1 » imm16[3:0]
0xF	**SRAX**	Rd = Rs1 »> imm16[3:0]

21.6 Load/Store and Branch/Jump Control

21.6.1 Load Instructions

Load instructions (opcode `0x6`) use the `func` field to select the load type and offset:

Listing 21.9: Load instruction decode

```
`OP_LOAD: begin
    mem_load = (func != `LOAD_LUI);
    reg_write = 1'b1;

    case (func)
        `LOAD_LW:  begin mem_offset = 16'h0000; end
        `LOAD_LB:  begin
            mem_offset = 16'h0000;
            mem_byte = 1'b1;
            mem_signed = 1'b1;
        end
        `LOAD_LBU: begin
            mem_offset = 16'h0000;
            mem_byte = 1'b1;
            mem_signed = 1'b0;
        end
        4'h3: begin mem_offset = 16'h0002; end  // LW +2
        4'h4: begin mem_offset = 16'h0004; end  // LW +4
        4'h5: begin mem_offset = 16'h0006; end  // LW +6
        4'h6: begin mem_offset = 16'hFFFE; end  // LW -2
        4'h7: begin mem_offset = 16'hFFFC; end  // LW -4
        `LOAD_LUI: begin alu_op = `ALU_SLL; end
        default:   begin mem_offset = 16'h0000; end
    endcase
end
```

Several design choices are visible here:

- **Embedded offsets**: Rather than requiring a separate immediate for load offsets, the SAMPO encodes common offsets ($+2$, $+4$, $+6$, -2, -4) directly in the function code. This means **`LW Rd, 4(Rs1)`** is a single 16-bit instruction, not a 32-bit extended instruction.

- **Byte loads**: The `mem_byte` signal tells the memory subsystem to read a single byte, and `mem_signed` controls whether it is sign-extended (**`LB`**) or zero-extended (**`LBU`**).

- **LUI is not a memory operation**: Load Upper Immediate (**LUI**) shares the load opcode but does not access memory at all. Instead, it routes through the ALU as a left shift by 8 (`ALU_SLL`). The `mem_load` signal is gated: `mem_load = (func != LOAD_LUI)`.

21.6.2 Store Instructions

Listing 21.10: Store instruction decode

```
`OP_STORE: begin
    mem_store = 1'b1;

    case (func)
        `STORE_SW: begin mem_offset = 16'h0000; end
        `STORE_SB: begin
            mem_offset = 16'h0000;
            mem_byte = 1'b1;
        end
        4'h2: begin mem_offset = 16'h0002; end  // SW +2
        4'h3: begin mem_offset = 16'h0004; end  // SW +4
        4'h4: begin mem_offset = 16'h0006; end  // SW +6
        4'h5: begin mem_offset = 16'hFFFE; end  // SW -2
        4'h6: begin mem_offset = 16'hFFFC; end  // SW -4
        default: begin mem_offset = 16'h0000; end
    endcase
end
```

Store instructions set `mem_store` but not `reg_write`—stores modify memory, not registers. The offset encoding follows the same pattern as loads, using two's complement negative offsets for backward addressing.

21.6.3 Branch Instructions

Branch decode is the most minimal:

Listing 21.11: Branch instruction decode

```
`OP_BRANCH: begin
    is_branch = 1'b1;
end
```

The single `is_branch` signal is all the decoder needs to produce. The actual branch condition evaluation happens in the CPU core using the `branch_cond` field (extracted from `rd`) and the current FLAGS register. The branch offset comes from `offset8`.

21.6.4 Jump Instructions

Jump instructions are more complex because three different instruction types share opcode `0x9`: the relative jump (**J**), the register jump (**JR**), and the register jump-and-link (**JALR**).

Listing 21.12: Jump instruction decode

```
`OP_JUMP: begin
    if (rd == 4'hF && (func == 4'h0)) begin
        // JR Rs1
        is_jump_reg = 1'b1;
        is_ret = (rs1 == `REG_RA);  // JR R1 is return
    end else if (func == 4'h1 && rd != 4'hF) begin
        // JALR Rd, Rs1
        is_jump_reg = 1'b1;
        is_call = 1'b1;
        reg_write = 1'b1;
    end else begin
        // J offset12
        is_jump = 1'b1;
    end
end
```

The decoder distinguishes these three forms using specific bit patterns—the details of which we explore in Section 21.9.

21.7 I/O Control Signals

The SAMPO provides four I/O instructions in opcode `0xD`, differentiated by the function code:

Listing 21.13: I/O instruction decode

```
`OP_IO: begin
    case (func)
        `IO_INI: begin
            is_io_in = 1'b1;
            io_port_imm = 1'b1;
            reg_write = 1'b1;
        end
        `IO_OUTI: begin
            is_io_out = 1'b1;
            io_port_imm = 1'b1;
        end
        `IO_IN: begin
            is_io_in = 1'b1;
```

```
            reg_write = 1'b1;
        end
        `IO_OUT: begin
            is_io_out = 1'b1;
        end
    endcase
end
```

The `io_port_imm` signal distinguishes between the two port addressing modes:

- **Immediate port** (**`INI`**, **`OUTI`**): the port address is a 4-bit value embedded in the `rs1` field of the instruction.
- **Register port** (**`IN`**, **`OUT`**): the port address comes from a register.

The I/O port value is selected by a multiplexer driven by `io_port_imm`:

Listing 21.14: I/O port address selection

```
assign io_port = io_port_imm ? {4'b0, rs1} : imm16[7:0];
```

For immediate-port instructions, the 4-bit port number from `rs1` is zero-extended to 8 bits. For extended I/O instructions (**`INX`**, **`OUTX`**), the full 8-bit port address comes from the `imm16` second word.

The UART Lives at Ports `0x80`–`0x81`

The most common I/O target is the UART, at port `0x80` (status) and `0x81` (data). Since these addresses exceed the 4-bit range of **`INI`**/**`OUTI`**, UART access must use the extended instructions **`INX`** and **`OUTX`**. See Chapter 23 for the UART implementation.

21.8 The Misc Opcode: A Kitchen Drawer

Opcode `0xC` is the catch-all category for instructions that do not fit neatly elsewhere. It includes stack operations, comparisons, moves, block operations, and register exchange:

Listing 21.15: Misc instruction decode (selected cases)

```
`OP_MISC: begin
    case (func)
        `MISC_PUSH: begin
            is_push = 1'b1;
            mem_store = 1'b1;
            alu_op = `ALU_SUB;  // SP - 2
        end
```

```
            `MISC_POP: begin
                is_pop = 1'b1;
                mem_load = 1'b1;
                reg_write = 1'b1;
                alu_op = `ALU_ADD;  // SP + 2
            end
            `MISC_CMP: begin
                is_cmp = 1'b1;
                alu_op = `ALU_SUB;
            end
            `MISC_TEST: begin
                is_test = 1'b1;
                alu_op = `ALU_AND;
            end
            `MISC_MOV: begin
                alu_op = `ALU_PASS_B;
                reg_write = 1'b1;
            end
            `MISC_EXX: begin
                is_exx = 1'b1;
            end
            `MISC_GETF: begin
                is_getf = 1'b1;
                reg_write = 1'b1;
            end
            `MISC_SETF: begin
                is_setf = 1'b1;
            end
        endcase
    end
```

Notice how **PUSH** and **POP** set both memory signals and ALU operations. **PUSH** needs a memory store (to write the register value to the stack) and a subtraction (to decrement SP by 2). **POP** needs a memory load (to read from the stack) and an addition (to increment SP by 2). These compound operations require careful coordination in the CPU state machine, which we explore in Chapter 22.

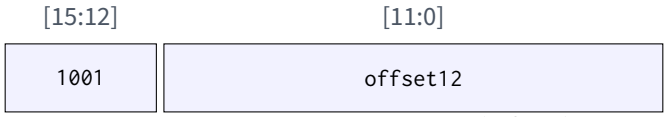
CMP is decoded as `ALU_SUB` with `is_cmp` asserted. The `is_cmp` flag tells the CPU to update flags but *not* write the subtraction result to a register (`reg_write` stays at its default of 0). Similarly, **TEST** uses `ALU_AND` with `is_test`, updating flags without storing the AND result.

21.9 The J/JR/JALR Decode Conflict

Perhaps the trickiest part of the SAMPO's instruction encoding is the jump opcode (`0x9`), which must distinguish between three fundamentally different instructions that share the same opcode. Let us examine the bit patterns carefully.

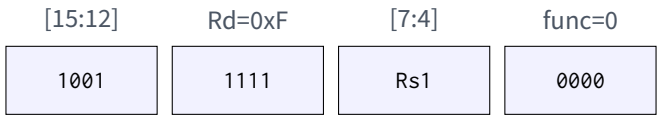
21.9.1 The Three Jump Forms

The relative jump **`J`** uses the full 12 bits [11:0] as a signed offset:



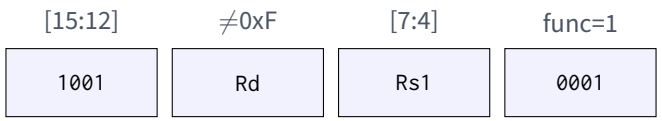
J: PC = PC + 2 + sign_ext(offset12) × 2

The register jump **`JR Rs1`** places a sentinel value in the `rd` field and zero in the `func` field:



JR: PC = Rs1

The jump-and-link **`JALR Rd, Rs1`** uses a non-oxF value in `rd` and `0x1` in `func`:



JALR: Rd = PC + 2; PC = Rs1

21.9.2 The Conflict

The problem is that the 12-bit offset field of **`J`** overlaps with the `rd`, `rs1`, and `func` fields used by **`JR`** and **`JALR`**. When the decoder sees opcode `0x9`, it must determine which of the three forms is present *before* interpreting the remaining bits.

The decoder resolves this with a priority chain:

Listing 21.16: Jump decode priority chain

```
if (rd == 4'hF && (func == 4'h0)) begin
    // JR Rs1
    is_jump_reg = 1'b1;
```

```
    is_ret = (rs1 == `REG_RA);
end else if (func == 4'h1 && rd != 4'hF) begin
    // JALR Rd, Rs1
    is_jump_reg = 1'b1;
    is_call = 1'b1;
    reg_write = 1'b1;
end else begin
    // J offset12
    is_jump = 1'b1;
end
```

The logic works as follows:

1. **Check for JR first**: If bits [11:8] are `0xF` and bits [3:0] are `0x0`, this is a register jump. The `rd = 0xF` pattern is a sentinel—it would encode a relative jump whose offset has bits [11:8] = `0xF`, which is still a valid offset but is sacrificed to enable register jumps.
2. **Check for JALR**: If bits [3:0] are `0x1` and `rd` is *not* `0xF`, this is a jump-and-link-register. The `func == 1` distinguishes it from both **`JR`** (which has `func == 0`) and regular **`J`** (which interprets those bits as offset).
3. **Fall through to J**: If neither special pattern matches, the instruction is a relative jump using the full 12-bit offset.

Lost Jump Offsets

This encoding scheme means that certain 12-bit offset values are "shadowed" by the **`JR`** and **`JALR`** patterns. Specifically, any offset whose bits [11:8] = `0xF` and bits [3:0] = `0x0` or `0x1` will be decoded as a register jump instead of a relative jump. The assembler must be aware of these forbidden offsets and use the extended **`JX`** instruction (which uses an absolute 16-bit address from the second word) when a relative jump would produce a conflicting bit pattern. In practice, this only affects jumps to very specific distances that rarely occur naturally.

21.9.3 JR R1 as Return

The decoder includes a special detection for the common case of returning from a subroutine:

```
is_ret = (rs1 == `REG_RA);  // JR R1 is return
```

When **`JR`** is used with `R1` (the return address register), the decoder marks this as a return. While the `is_ret` signal does not change the execution behavior (the PC is still loaded from `R1`), it can be useful for debugging and tracing—a testbench or debugger can distinguish between a generic register jump and a function return.

21.10 System Instructions

System instructions (opcode `0xE`) use the `rd` field—bits [11:8]—as the function selector, since these instructions do not have a destination register:

Listing 21.17: System instruction decode

```
`OP_SYSTEM: begin
    case (rd)  // System func is in rd field
        `SYS_NOP:  is_nop = 1'b1;
        `SYS_HALT: is_halt = 1'b1;
        `SYS_DI:   is_di = 1'b1;
        `SYS_EI:   is_ei = 1'b1;
        `SYS_RETI: is_reti = 1'b1;
        `SYS_SCF:  is_scf = 1'b1;
        `SYS_CCF:  is_ccf = 1'b1;
        default:   begin end
    endcase
end
```

Each system instruction sets a single control flag. The CPU state machine handles the actual effects:

- **`NOP`**: No operation—the state machine advances through its states without doing anything.
- **`HALT`**: Transitions the state machine to HALTED, stopping execution.
- **`DI`/`EI`**: Clear or set the interrupt enable flag (bit [2] of FLAGS).
- **`SCF`**: Set the carry flag.
- **`CCF`**: Complement (toggle) the carry flag.

Exercises

1. **Decode by hand.** Given the instruction word `0x5407`, identify the opcode, extract all fields, and determine the complete set of control signals the decoder produces. What instruction is this? What ALU operation is selected? Which register is the destination?
2. **Trace a PUSH.** The instruction **`PUSH R4`** is encoded as `0xC040` (opcode `0xC`, rd=`0x0`, rs1=`0x4`, func=`0x0`). Walk through the decoder and list every control signal that is set to 1. Then explain how the CPU uses these signals to decrement SP, compute the store address, and write the register value to memory.

3. **Find the conflict.** What happens if the assembler tries to encode **`J`** with an offset that produces the bit pattern `0x9F40`? Decode this instruction both as a **`J`** (with offset) and as a **`JR`** (with register). Which interpretation does the decoder choose? How should the assembler handle this?

4. **Add a new system instruction.** Suppose you want to add a **`SWAPF`** instruction that swaps the upper and lower nibbles of the FLAGS register. Assign it an unused system function code, write the decoder case, and describe what changes would be needed in `cpu.v` to implement the actual swap.

5. **Decode timing analysis.** The decoder is purely combinational. Using the decoder as an example, identify the longest chain of logic gates from an input change (`instr` changing) to the last output stabilizing. Which output has the most complex logic? How does this affect the maximum clock frequency?

What's Next

The decoder generates the control signals, but it does not execute anything—it is a purely combinational translator from instruction bits to control wires. The component that *uses* those signals, orchestrating the entire fetch–decode–execute cycle, is the CPU state machine. In the next chapter, we examine the 8-state FSM that brings the SAMPO to life, including a debugging story about a subtle pipeline hazard that took several sessions to track down.

Chapter 22

The CPU State Machine

We have now built every functional unit the SAMPO needs: the ALU computes results, the register file stores operands, the shifter handles bit manipulation, and the decoder translates instructions into control signals. What remains is the conductor—the finite state machine that orchestrates all of these components through the fetch–decode–execute cycle. In this chapter, we open `verilog/rtl/cpu.v` and trace the 8-state FSM that brings the SAMPO to life. We will also encounter a subtle bug involving the memory interface that required careful detective work to diagnose.

22.1 Multi-Cycle CPU Design Philosophy

The SAMPO uses a multi-cycle design: each instruction takes several clock cycles to complete, with a different phase of execution happening in each cycle. This contrasts with single-cycle designs (where every instruction completes in one clock period) and pipelined designs (where multiple instructions overlap in different stages).

The multi-cycle approach has clear trade-offs:

- **Advantage**: Each clock cycle performs only one major operation (memory access, ALU computation, register write), so the critical path is shorter and the clock can run faster.
- **Advantage**: The same hardware (ALU, memory interface) can be reused across different phases of execution, reducing resource usage.
- **Disadvantage**: Instructions take multiple cycles, so throughput is lower than a pipelined design at the same clock frequency.
- **Advantage**: The design is significantly simpler than a pipeline. No forwarding logic, no hazard detection, no pipeline stalls to manage.

For a teaching CPU like the SAMPO, the multi-cycle approach strikes the right balance: the design is understandable enough to study yet complete enough to execute real programs.

22.2 The 8-State FSM

The CPU's state machine has eight states, defined in `sampo_pkg.vh`:

Listing 22.1: CPU state definitions

```
`define ST_RESET     4'h0
`define ST_FETCH     4'h1
`define ST_FETCH_EXT 4'h2
`define ST_DECODE    4'h3
`define ST_EXECUTE   4'h4
`define ST_MEMORY    4'h5
`define ST_WRITEBACK 4'h6
`define ST_HALTED    4'h7
```

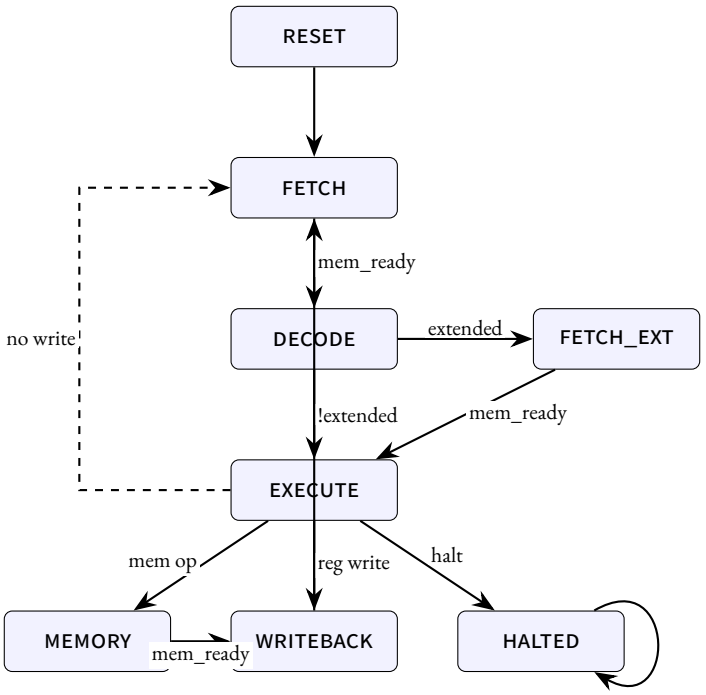


Figure 22.1: The SAMPO CPU state machine

Each state has a specific responsibility:

1. RESET: Initialize the program counter to the reset vector and clear flags.
2. FETCH: Read the instruction word from memory at the current PC.
3. FETCH_EXT: Read the second word of a 32-bit extended instruction.
4. DECODE: Latch register values for the source operands.
5. EXECUTE: Perform the ALU/shift operation, evaluate branches, update flags and PC.
6. MEMORY: Perform a memory load or store.
7. WRITEBACK: Write results to the register file.
8. HALTED: Stop execution permanently.

22.3 State Transitions and Timing

The state register and next-state logic are separated into two `always` blocks, following standard synchronous FSM design practice:

Listing 22.2: State register

```
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        state <= `ST_RESET;
    end else begin
        state <= next_state;
    end
end
```

The next-state logic is a combinational block:

Listing 22.3: Next state logic

```
always @(*) begin
    next_state = state;

    case (state)
        `ST_RESET:     next_state = `ST_FETCH;

        `ST_FETCH: begin
            if (mem_ready)
                next_state = `ST_DECODE;
        end
```

```
        `ST_DECODE: begin
            if (dec_is_extended)
                next_state = `ST_FETCH_EXT;
            else
                next_state = `ST_EXECUTE;
        end

        `ST_FETCH_EXT: begin
            if (mem_ready)
                next_state = `ST_EXECUTE;
        end

        `ST_EXECUTE: begin
            if (dec_is_halt)
                next_state = `ST_HALTED;
            else if (dec_mem_load || dec_mem_store)
                next_state = `ST_MEMORY;
            else if (dec_reg_write)
                next_state = `ST_WRITEBACK;
            else
                next_state = `ST_FETCH;
        end

        `ST_MEMORY: begin
            if (mem_ready)
                next_state = `ST_WRITEBACK;
        end

        `ST_WRITEBACK: next_state = `ST_FETCH;

        `ST_HALTED: next_state = `ST_HALTED;

        default: next_state = `ST_RESET;
    endcase
end
```

Multi-Cycle Instruction Timing

Different instructions take different numbers of clock cycles:

- **ALU register ops** (e.g., `ADD`): FETCH → DECODE → EXECUTE → WRITEBACK = 4 cycles minimum
- **Branches (taken or not)**: FETCH → DECODE → EXECUTE → FETCH = 3–4 cycles
- **Load/Store**: FETCH → DECODE → EXECUTE → MEMORY → WRITEBACK = 5 cycles
- **Extended instructions**: Add 1 cycle for FETCH_EXT
- **NOP**: FETCH → DECODE → EXECUTE → FETCH = 3–4 cycles

The FETCH and MEMORY states also depend on `mem_ready`, which adds at least one cycle for synchronous RAM.

Several transitions deserve closer examination.

22.3.1 The Execute Exit Decision

The EXECUTE state has *four* possible exit paths, determined by the instruction type:

1. **HALT**: Transition to HALTED and stop.
2. **Memory operation**: If the instruction requires a load or store, go to MEMORY.
3. **Register write**: If the instruction writes a result (but does not need memory), go to WRITEBACK.
4. **None of the above**: For branches, NOP, and other instructions that neither access memory nor write registers, go directly to FETCH.

This priority order matters. A load instruction sets both `dec_mem_load` and `dec_reg_write`, but the memory check takes priority, routing execution through MEMORY first. The register write will happen later in WRITEBACK.

22.4 FETCH and FETCH_EXT

22.4.1 The Fetch Cycle

During FETCH, the CPU presents the program counter on the memory address bus and waits for the memory to respond:

Listing 22.4: Memory interface during FETCH

```
`ST_FETCH: begin
```

```
    mem_addr = pc;
    mem_valid = 1'b1;
end
```

The memory interface uses a valid/ready handshake: the CPU asserts `mem_valid` to request an access, and the memory responds with `mem_ready` when the data is available. For the synchronous RAM in the SAMPO SoC, `mem_ready` is simply `mem_valid` delayed by one clock cycle—so every memory access takes exactly one cycle.

When `mem_ready` arrives, the instruction word is latched:

Listing 22.5: Instruction latch on mem_ready

```
`ST_FETCH: begin
    if (mem_ready) begin
        instr <= mem_rdata;
    end
end
```

Once latched, the decoder immediately begins producing control signals from the `instr` register (the decoder is combinational and always active), though these signals are not *used* until the appropriate later state.

22.4.2 Fetching Extended Instructions

If the decoder detects an extended instruction (`dec_is_extended`), the state machine inserts the FETCH_EXT state before EXECUTE. This state reads the second 16-bit word:

Listing 22.6: Memory interface during FETCH_EXT

```
`ST_FETCH_EXT: begin
    mem_addr = pc + 16'h2;
    mem_valid = 1'b1;
end
```

The address is PC + 2—the word immediately following the first instruction word. When the data arrives, it is stored in the `imm16` register:

Listing 22.7: Latching the extended immediate

```
`ST_FETCH_EXT: begin
    if (mem_ready) begin
        imm16 <= mem_rdata;
    end
```

```
end
```

The decoder's `imm16` input is connected to this register, so extended instruction fields like the 16-bit immediate for **LIX** or the absolute address for **JX** become available to the control logic once FETCH_EXT completes.

22.5 EXECUTE: ALU, Branches, and Register Writes

The EXECUTE state is the busiest—it handles ALU computation, branch evaluation, flag updates, PC advancement, and several special operations. Let us examine each responsibility.

22.5.1 ALU Input Selection

The ALU inputs are configured by a combinational block that routes different operand sources depending on the instruction type:

Listing 22.8: ALU input muxing (selected cases)

```
always @(*) begin
    alu_a = rs1_val;
    alu_b = rs2_val;
    alu_op = dec_alu_op;

    case (dec_opcode)
        `OP_ADDI: begin
            alu_a = rs1_val;       // Rd value (read via rs1)
            alu_b = dec_imm8_sext; // Sign-extended immediate
        end

        `OP_MULDIV: begin
            alu_a = rs1_val;  // Rd value (first operand)
            alu_b = rs2_val;  // Rs1 value (second operand)
        end

        `OP_EXTENDED: begin
            case (dec_ext_sub)
                `EXT_ADDIX, `EXT_SUBIX, `EXT_ANDIX,
                `EXT_ORIX, `EXT_XORIX: begin
                    alu_a = rs1_val;
                    alu_b = imm16;
                end
                `EXT_LIX: begin
                    alu_b = imm16;
                end
```

```
                `EXT_SLLX, `EXT_SRLX, `EXT_SRAX: begin
                    alu_a = rs1_val;
                    alu_b = {12'b0, imm16[3:0]};
                end
            endcase
        end
    endcase
end
```

The default case routes `rs1_val` to `alu_a` and `rs2_val` to `alu_b`, which is correct for R-type instructions like **`ADD Rd, Rs1, Rs2`**. Special cases override one or both inputs:

- **`ADDI`** replaces `alu_b` with the sign-extended 8-bit immediate.
- Extended ALU instructions replace `alu_b` with `imm16` (the 16-bit immediate from the second word).
- Extended shift instructions truncate the immediate to 4 bits for the shift amount.
- **`LIX`** passes the 16-bit immediate through `ALU_PASS_B`.

22.5.2 PC Update Logic

The program counter update during EXECUTE handles six different cases:

Listing 22.9: Program counter update

```
`ST_EXECUTE: begin
    if (dec_is_halt) begin
        // PC unchanged
    end else if (dec_is_branch && branch_taken) begin
        pc <= pc + 16'h2 + dec_offset8;
    end else if (dec_is_jump && !dec_is_extended) begin
        pc <= pc + 16'h2 + dec_offset12;
    end else if (dec_is_jump && dec_is_extended) begin
        pc <= imm16;  // JX or JALX
    end else if (dec_is_jump_reg) begin
        pc <= rs1_val;
    end else if (dec_is_extended) begin
        pc <= pc + 16'h4;  // Extended = 4 bytes
    end else begin
        pc <= pc + 16'h2;  // Normal = 2 bytes
    end
```

Each case deserves attention:

- **HALT**: The PC does not change. The state machine transitions to HALTED and stays there.
- **Taken branch**: PC advances by 2 (past the branch instruction itself) plus the sign-extended, doubled offset. The offset is relative to the instruction *following* the branch, not the branch itself.
- **Relative jump**: Same calculation with the wider 12-bit offset.
- **Extended jump (JX/JALX)**: The PC is loaded with the absolute address from `imm16`.
- **Register jump (JR/JALR)**: The PC is loaded directly from the source register value.
- **Extended instruction (non-jump)**: The PC advances by 4 bytes to skip both the opcode word and the immediate word.
- **Normal instruction**: The PC advances by 2 bytes.

22.5.3 Branch Condition Evaluation

Branch evaluation happens in a separate combinational block that the state machine samples during EXECUTE:

Listing 22.10: Branch condition evaluation

```
always @(*) begin
    case (dec_branch_cond)
        `BR_EQ:  branch_taken = flags[`FLAG_Z];
        `BR_NE:  branch_taken = ~flags[`FLAG_Z];
        `BR_LT:  branch_taken = flags[`FLAG_N] != flags[`FLAG_V];
        `BR_GE:  branch_taken = flags[`FLAG_N] == flags[`FLAG_V];
        `BR_LTU: branch_taken = ~flags[`FLAG_C];
        `BR_GEU: branch_taken = flags[`FLAG_C];
        `BR_MI:  branch_taken = flags[`FLAG_N];
        `BR_PL:  branch_taken = ~flags[`FLAG_N];
        `BR_VS:  branch_taken = flags[`FLAG_V];
        `BR_VC:  branch_taken = ~flags[`FLAG_V];
        `BR_CS:  branch_taken = flags[`FLAG_C];
        `BR_CC:  branch_taken = ~flags[`FLAG_C];
        `BR_GT:  branch_taken = ~flags[`FLAG_Z] &&
                     (flags[`FLAG_N] == flags[`FLAG_V]);
        `BR_LE:  branch_taken = flags[`FLAG_Z] ||
                     (flags[`FLAG_N] != flags[`FLAG_V]);
        `BR_HI:  branch_taken = flags[`FLAG_C] &&
                     ~flags[`FLAG_Z];
        `BR_LS:  branch_taken = ~flags[`FLAG_C] ||
                     flags[`FLAG_Z];
        default: branch_taken = 1'b0;
```

```
    endcase
end
```

All 16 branch conditions are decoded, providing a rich set of signed and unsigned comparisons. The signed conditions (**BLT**, **BGE**, **BGT**, **BLE**) use the $N \neq V$ relationship discussed in Chapter 18. The unsigned conditions (**BLTU**/**BCC**, **BGEU**/**BCS**, **BHI**, **BLS**) use the carry flag with the ARM-style convention where C=1 means "no borrow."

BLTU Checks for C=0

Because the SAMPO uses ARM-style carry (C=1 means no borrow after subtraction), the unsigned-less-than branch **BLTU** checks for C=0, *not* C=1. This is the opposite of x86's **JB** instruction. The carry flag after **CMP A, B** is 1 when $A \geq B$ unsigned, so "less than" means carry is clear.

22.5.4 Flag Updates

Flag updates during EXECUTE follow a priority chain that determines which operations affect which flags:

Listing 22.11: Flag update priority chain (simplified)

```
if (dec_is_setf) begin
    flags <= rs1_val[7:0];
end else if (dec_is_ei) begin
    flags[`FLAG_I] <= 1'b1;
end else if (dec_is_di) begin
    flags[`FLAG_I] <= 1'b0;
end else if (dec_is_scf) begin
    flags[`FLAG_C] <= 1'b1;
end else if (dec_is_ccf) begin
    flags[`FLAG_C] <= ~flags[`FLAG_C];
end else if (dec_opcode == `OP_SHIFT) begin
    flags[`FLAG_N] <= shift_result[15];
    flags[`FLAG_Z] <= (shift_result == 16'h0000);
    flags[`FLAG_C] <= shift_carry;
    flags[`FLAG_V] <= 1'b0;
end else if (/* arithmetic ops */) begin
    flags[`FLAG_N] <= alu_flag_n;
    flags[`FLAG_Z] <= alu_flag_z;
    flags[`FLAG_C] <= alu_flag_c;
    flags[`FLAG_V] <= alu_flag_v;
end else if (/* logical ops */) begin
    flags[`FLAG_N] <= alu_flag_n;
```

```
    flags[`FLAG_Z] <= alu_flag_z;
    flags[`FLAG_C] <= 1'b0;
    flags[`FLAG_V] <= 1'b0;
end else if (dec_is_muldiv) begin
    flags[`FLAG_N] <= alu_flag_n;
    flags[`FLAG_Z] <= alu_flag_z;
end
```

The rules are:

- **SETF** overwrites the entire FLAGS register from a source register.
- **EI**/**DI** modify only the I flag.
- **SCF**/**CCF** modify only the C flag.
- Shift operations set N, Z, C from the shifter and clear V.
- Arithmetic operations (ADD, SUB, ADDI, CMP, extended ALU ops) set all four flags from the ALU.
- Logical operations (AND, OR, XOR, TEST) set N and Z from the ALU, but always clear C and V.
- Multiply/divide operations set only N and Z.
- Operations not listed (MOV, PUSH, POP, branches, jumps, NOP) do not modify flags.

22.5.5 Execution Result Selection

During EXECUTE, the CPU latches the execution result into `exec_result`, selecting from multiple sources:

Listing 22.12: Execution result multiplexing

```
if (dec_opcode == `OP_SHIFT) begin
    exec_result <= shift_result;
end else if (dec_is_getf) begin
    exec_result <= {8'h00, flags};
end else if (dec_is_call) begin
    exec_result <= dec_is_extended ? (pc + 16'h4)
                                   : (pc + 16'h2);
end else if (dec_is_io_in) begin
    exec_result <= {8'h00, io_rdata};
end else begin
    exec_result <= alu_result;
end
```

Four special cases override the default ALU result:

- **Shift instructions**: Use the shifter output instead of the ALU.
- **GETF**: Return the FLAGS register, zero-extended to 16 bits.
- **Call instructions**: Return the return address (PC + 2 for normal, PC + 4 for extended).
- **I/O input**: Return the data read from the I/O port, zero-extended.

22.6 MEMORY and WRITEBACK

22.6.1 The MEMORY State

The MEMORY state handles all memory loads and stores. The memory address and data are configured by a combinational block:

Listing 22.13: Memory interface during MEMORY state

```
`ST_MEMORY: begin
    mem_valid = 1'b1;

    if (dec_is_push) begin
        mem_addr = rs1_val - 16'h2;  // SP - 2
        mem_wdata = rs2_val;          // Source register
        mem_we = 1'b1;
    end else if (dec_is_pop) begin
        mem_addr = rs1_val;           // Current SP
    end else if (dec_is_extended &&
                 dec_ext_sub == `EXT_LWX) begin
        mem_addr = rs1_val + imm16;
    end else if (dec_is_extended &&
                 dec_ext_sub == `EXT_SWX) begin
        mem_addr = rs1_val + imm16;
        mem_wdata = rs2_val;
        mem_we = 1'b1;
    end else if (dec_mem_store) begin
        mem_addr = rs1_val + dec_mem_offset;
        mem_wdata = dec_mem_byte ?
            {rs2_val[7:0], rs2_val[7:0]} : rs2_val;
        mem_we = 1'b1;
        if (dec_mem_byte) begin
            mem_be = mem_addr[0] ? 2'b10 : 2'b01;
        end
    end else if (dec_mem_load) begin
        mem_addr = rs1_val + dec_mem_offset;
```

```
    end
end
```

Several details are noteworthy:

- **PUSH address**: The store address is SP -2, because the SAMPO's stack grows downward (toward lower addresses). The stack pointer is decremented *before* the store, so the data lands at the new stack top.
- **Byte stores**: For **SB**, the data byte is duplicated into both halves of the 16-bit write data (`{rs2_val[7:0], rs2_val[7:0]}`). The byte enable signal (`mem_be`) selects which byte of the word to actually write, based on whether the address is odd or even. An even address writes the low byte (`2'b01`); an odd address writes the high byte (`2'b10`).
- **Extended loads/stores**: **LWX** and **SWX** add the full 16-bit immediate to the base register for the effective address.

When a load completes (`mem_ready` asserts), the data is latched into `mem_result`:

Listing 22.14: Load data latching with byte extension

```
`ST_MEMORY: begin
    if (mem_ready && dec_mem_load) begin
        if (dec_mem_byte) begin
            if (mem_addr[0]) begin
                // Odd address: high byte
                mem_result <= dec_mem_signed ?
                    {{8{mem_rdata[15]}}, mem_rdata[15:8]} :
                    {8'h00, mem_rdata[15:8]};
            end else begin
                // Even address: low byte
                mem_result <= dec_mem_signed ?
                    {{8{mem_rdata[7]}}, mem_rdata[7:0]} :
                    {8'h00, mem_rdata[7:0]};
            end
        end else begin
            mem_result <= mem_rdata;
        end
    end
end
```

Byte loads require post-processing: the RAM always returns a full 16-bit word, so the CPU must extract the correct byte based on the address alignment and apply sign or zero extension. For even

addresses, the low byte (bits [7:0]) is relevant; for odd addresses, the high byte (bits [15:8]). The `dec_mem_signed` signal selects sign extension (**LB**) versus zero extension (**LBU**).

22.6.2 The WRITEBACK State

The WRITEBACK state performs the actual register file write:

Listing 22.15: Register file control during WRITEBACK

```
`ST_WRITEBACK: begin
    rf_wr_en = dec_reg_write && (dec_rd != `REG_ZERO);

    if (dec_mem_load) begin
        rf_wr_data = mem_result;
    end else if (dec_is_pop) begin
        rf_wr_data = mem_result;
        rf_wr_addr = dec_rd;
    end else begin
        rf_wr_data = exec_result;
    end

    // PUSH: update SP via main port
    if (dec_is_push) begin
        rf_wr_en = 1'b1;
        rf_wr_addr = `REG_SP;
        rf_wr_data = rs1_val - 16'h2;
    end

    // POP: write SP via dedicated SP port
    if (sp_update_pending) begin
        rf_sp_wr_en = 1'b1;
        rf_sp_wr_data = sp_new_val;
    end
end
```

The write data source depends on the instruction:

- **Loads and POP**: The write data comes from `mem_result` (the value read from memory).
- **All other instructions**: The write data comes from `exec_result` (computed during EXECUTE).
- **PUSH**: Overrides the write to update SP (`R2`) with the decremented value.

22.7 The Pipeline Hazard: The MEMORY to FETCH Bug

This is a debugging story. During early testing, a subtle bug caused incorrect instruction fetches after store instructions. Understanding the bug—and the fix—teaches an important lesson about multi-cycle FSM design.

22.7.1 The Original Design

In the original implementation, the state machine's EXECUTE exit logic was:

Listing 22.16: Original (buggy) EXECUTE exit logic

```
// ORIGINAL (buggy)
`ST_EXECUTE: begin
    if (dec_is_halt)
        next_state = `ST_HALTED;
    else if (dec_mem_load || dec_mem_store)
        next_state = `ST_MEMORY;
    else if (dec_reg_write)
        next_state = `ST_WRITEBACK;
    else
        next_state = `ST_FETCH;
end
```

And the MEMORY exit was:

Listing 22.17: Original (buggy) MEMORY exit logic

```
// ORIGINAL (buggy)
`ST_MEMORY: begin
    if (mem_ready) begin
        if (dec_reg_write)
            next_state = `ST_WRITEBACK;
        else
            next_state = `ST_FETCH;  // BUG: stores go here
    end
end
```

For store instructions, `dec_reg_write` is 0 (stores do not write to registers), so after MEMORY the machine would transition directly to FETCH.

22.7.2 The Problem

The bug: `mem_ready` is a registered signal—it is the delayed version of `mem_valid`. When MEMORY completes a store and transitions directly to FETCH, the following timing occurs:

1. MEMORY: CPU asserts `mem_valid` with the store address. On the next clock edge, `mem_ready` goes high (the RAM acknowledges the store).
2. MEMORY → FETCH: `mem_ready` is high, so the state machine transitions to FETCH.
3. FETCH: CPU changes `mem_addr` to the PC for instruction fetch and asserts `mem_valid`. But `mem_ready` *is still high from the previous cycle* because it has not yet had time to deassert and reassert.
4. The FETCH state sees `mem_ready` = 1 and immediately latches `mem_rdata` as the instruction—but `mem_rdata` still holds the *old* data from the store operation, not the newly fetched instruction.

The result: the CPU fetches stale data from the memory bus instead of the next instruction. The instruction appears corrupted, and execution goes off the rails.

Stale Signals Across State Transitions

When a handshake signal like `mem_ready` persists across a state transition, the new state may see it before the memory has had time to respond to the new request. This is a classic multi-cycle FSM hazard. The fix must ensure that `mem_ready` from one transaction cannot be confused with the response to a different transaction.

22.7.3 The Fix

The fix is deceptively straightforward: **always route through WRITEBACK after MEMORY, even for stores.**

Listing 22.18: Fixed MEMORY exit logic

```
`ST_MEMORY: begin
    if (mem_ready) begin
        // Always go through WRITEBACK after MEMORY.
        // For loads: writes loaded data to register file.
        // For stores/PUSH: allows mem_ready to deassert
        // before next FETCH (prevents stale rdata latch),
        // and lets PUSH update the stack pointer.
        next_state = `ST_WRITEBACK;
    end
end
```

The WRITEBACK state does not assert `mem_valid`. This means `mem_ready` deasserts during the WRITEBACK cycle (since it is a delayed version of `mem_valid`, and `mem_valid` went low). When the machine then transitions to FETCH, `mem_ready` starts low, and the fetch must wait one cycle for the new memory access to complete. No stale data is latched.

For store instructions that do not write to registers, the WRITEBACK state is essentially a one-cycle "cooldown" that breaks the hazard. For PUSH instructions, the WRITEBACK state serves double duty: it provides the cooldown *and* lets the stack pointer update happen (see Section 22.8).

The Comment Documents the Bug

The comment in the source code—*"allows mem_ready to deassert before next FETCH (prevents stale rdata latch)"*—captures the debugging insight. Always document bug fixes. A future reader (or a future version of yourself) will want to know *why* stores go through writeback when they have nothing to write.

22.8 PUSH/POP: Dual Write

Stack operations are the most complex instructions in the SAMPO because they require two register updates in a single instruction: one to the data register and one to the stack pointer.

22.8.1 PUSH Execution

`PUSH Rs` must:

1. Decrement SP by 2.
2. Store the source register's value at the new SP.

The timeline:

- DECODE: Read SP (`R2`) via `rf_rd_addr1` and the source register via `rf_rd_addr2`.
- EXECUTE: Nothing special for PUSH (PC advances normally).
- MEMORY: Write the source register value to address SP -2. The address computation happens in the combinational memory interface block.
- WRITEBACK: Write the decremented SP value (SP -2) to `R2` via the main write port.

Listing 22.19: PUSH register write during WRITEBACK

```
// Handle PUSH: update SP via main port
if (dec_is_push) begin
    rf_wr_en = 1'b1;
```

```
    rf_wr_addr = `REG_SP;
    rf_wr_data = rs1_val - 16'h2;
end
```

The SP update uses the *main* write port, overriding any other write. This works because **PUSH** has no destination register (`dec_reg_write` = 0), so the main port would otherwise be idle.

22.8.2 POP Execution

POP Rd must:

1. Read the value at the current SP from memory.
2. Write that value to the destination register Rd.
3. Increment SP by 2.

This requires *two* simultaneous register writes: one to Rd (the loaded value) and one to R2/SP (the incremented pointer). This is why the register file has a dedicated SP write port.

The timeline:

- DECODE: Read SP via `rf_rd_addr1`.
- EXECUTE: PC advances normally.
- MEMORY: Read from address SP. When `mem_ready` arrives, the loaded data is latched into `mem_result`. The new SP value is also computed and stored:

Listing 22.20: POP: preparing the SP update

```
`ST_MEMORY: begin
    if (mem_ready && dec_is_pop) begin
        sp_update_pending <= 1'b1;
        sp_new_val <= rs1_val + 16'h2;
    end
end
```

- WRITEBACK: Write the loaded value to Rd via the main port, and simultaneously write the incremented SP via the dedicated SP port:

Listing 22.21: POP: dual register write

```
`ST_WRITEBACK: begin
    // Main port: write loaded value to Rd
    rf_wr_en = dec_reg_write && (dec_rd != `REG_ZERO);
```

```
    rf_wr_data = mem_result;
    rf_wr_addr = dec_rd;

    // SP port: write incremented SP
    if (sp_update_pending) begin
        rf_sp_wr_en = 1'b1;
        rf_sp_wr_data = sp_new_val;
    end
end
```

The two writes happen on the same clock edge: the main write port carries Rd's new value, and the SP write port carries SP's new value. After the clock edge, `sp_update_pending` is cleared.

22.9 Instruction Execution Walkthroughs

Let us trace several instructions through the complete state machine, cycle by cycle. Assume the clock frequency is such that each state takes exactly one clock cycle (with synchronous RAM providing one-cycle access).

22.9.1 Walkthrough: ADD R4, R5, R6

Table 22.1: Cycle-by-cycle execution of **`ADD R4, R5, R6`**

Cycle	State	Activity
1	FETCH	`mem_addr = PC`, `mem_valid = 1`. RAM begins read.
2	FETCH	`mem_ready` rises. `instr <= mem_rdata` (`0x0456`). Transition to DECODE.
3	DECODE	`rf_rd_addr1 = 5` (Rs1), `rf_rd_addr2 = 6` (Rs2). Combinational reads produce values. `rs1_val <= rf_rd_data1`, `rs2_val <= rf_rd_data2`. Decoder outputs: `alu_op = ALU_ADD`, `reg_write = 1`.
4	EXECUTE	ALU computes `rs1_val + rs2_val`. Result latched: `exec_result <= alu_result`. Flags updated: N, Z, C, V from ALU. PC advances: `pc <= pc + 2`. Since `reg_write = 1`, next state is WRITEBACK.
5	WRITEBACK	`rf_wr_en = 1`, `rf_wr_addr = 4` (Rd), `rf_wr_data = exec_result`. Register R4 is written on clock edge. Transition to FETCH.

Total: 5 clock cycles.

22.9.2 Walkthrough: LIX R4, 0x1234

This is a 32-bit extended instruction.

Table 22.2: Cycle-by-cycle execution of **`LIX R4, 0x1234`**

Cycle	State	Activity
1	FETCH	Read first word at PC.
2	FETCH	`instr <= 0xF407` (opcode=oxF, Rd=4, Rs1=0, sub=7).
3	DECODE	Decoder sees `is_extended = 1`. Transition to FETCH_EXT.
4	FETCH_EXT	`mem_addr = PC + 2`, read second word.
5	FETCH_EXT	`imm16 <= 0x1234`. Transition to EXECUTE.
6	EXECUTE	ALU: `PASS_B` with `alu_b = imm16 = 0x1234`. `exec_result <= 0x1234`. PC advances by 4.
7	WRITEBACK	Write `0x1234` to R4.

Total: 7 clock cycles. The extra FETCH_EXT cycle is the cost of 32-bit instructions.

22.9.3 Walkthrough: SW (R5), R4

Table 22.3: Cycle-by-cycle execution of **`SW (R5), R4`**

Cycle	State	Activity
1	FETCH	Read instruction.
2	FETCH	`instr` latched.
3	DECODE	Read R5 (base addr) and R4 (store data). Note: R4 is encoded in the Rd field for stores, so `rf_rd_addr2 = dec_rd`.
4	EXECUTE	PC advances by 2. `mem_store = 1`, so next state is MEMORY.
5	MEMORY	`mem_addr = rs1_val`, `mem_wdata = rs2_val`, `mem_we = 1`. RAM writes the data.
6	MEMORY	`mem_ready` rises. Transition to WRITEBACK (not directly to FETCH—this is the bug fix).
7	WRITEBACK	No register write (`dec_reg_write = 0`). This cycle serves as the `mem_ready` cooldown.

Total: 7 clock cycles. The WRITEBACK "cooldown" cycle is the fix for the MEMORY→FETCH hazard.

22.9.4 Walkthrough: BEQ (taken)

Table 22.4: Cycle-by-cycle execution of **`BEQ +6`** (taken)

Cycle	State	Activity
1	FETCH	Read instruction.
2	FETCH	`instr` latched (e.g., `0x8003`).
3	DECODE	Decoder: `is_branch = 1`, `branch_cond = 0x0` (BEQ). No registers read (branch does not use Rd/Rs1).
4	EXECUTE	`branch_taken` evaluated: Z flag is 1, so taken. PC updated: `pc <= pc + 2 + offset8`. No `reg_write`, no `mem_load/store`, so next state is FETCH.

Total: 4 clock cycles. Branches are the fastest multi-cycle path because they skip both MEMORY and WRITEBACK.

Exercises

1. **Walkthrough: PUSH R4.** Create a cycle-by-cycle table (following the format in this chapter) for the **`PUSH R4`** instruction. Track the values on `mem_addr`, `mem_wdata`, `rf_wr_addr`, and `rf_wr_data` in each state. Assume SP initially contains `0xFFFE`.

2. **Counting cycles.** A program consists of the following loop body: **`LW R4, (R5)`**, **`ADDI R4, 1`**, **`SW (R5), R4`**, **`BNE loop`**. Calculate the total number of clock cycles per iteration (assuming all memory accesses complete in one cycle and the branch is taken). If the CPU runs at 25 MHz, how many iterations per second can it execute?

3. **Reproduce the MEMORY→FETCH bug.** Modify the next-state logic in `cpu.v` to restore the original buggy behavior (stores go directly from MEMORY to FETCH). Run the ISA test suite and observe which tests fail. Use GTKWave to examine the waveforms around the failing instruction and confirm that `mem_rdata` contains stale data at the moment of the bad fetch.

4. **Add a pipeline register.** An alternative fix for the MEMORY→FETCH hazard would be to add a pipeline register that holds the fetched instruction, breaking the dependency on `mem_ready` timing. Sketch this modification. What are the trade-offs compared to the "always go through WRITEBACK" approach?

5. **Performance analysis.** Calculate the average cycles per instruction (CPI) for a program that is 40% ALU register operations, 25% loads, 15% stores, 10% branches (half taken, half not), 5%

extended instructions, and 5% PUSH/POP. Use the cycle counts from the walkthroughs in this chapter.

What's Next

The CPU state machine orchestrates the datapath, but a CPU alone is not a usable system. It needs memory to store programs and data, and peripherals to communicate with the outside world. In the next chapter, we wrap the CPU, RAM, and a UART controller into a complete System on Chip, ready for synthesis onto an FPGA.

Chapter 23

System on Chip

A CPU that cannot communicate with the outside world is an expensive paperweight. In this chapter, we wire the SAMPO CPU core to memory and a UART controller, forming a complete System on Chip that can run programs, print messages to a serial terminal, and receive input—all from a single Verilog module. The SoC top level lives in `verilog/rtl/soc.v`, with the RAM in `verilog/rtl/ram.v` and the UART in `verilog/rtl/uart.v`.

23.1 SoC Architecture Overview

The SAMPO SoC consists of three components:

1. **The CPU core** (`cpu.v`): the state machine, ALU, register file, shifter, and decoder we built in the preceding chapters.

2. **A 64KB synchronous RAM** (`ram.v`): stores both program code and data. Initialized from a hex file at simulation start.

3. **A UART controller** (`uart.v`): provides serial communication via I/O ports, compatible with the MC6850 status/data register model.

The CPU communicates with RAM through a memory bus (address, read data, write data, write enable, byte enables) and with the UART through a separate I/O bus (port address, read data, write data, read/write strobes). This separation of memory and I/O spaces is a deliberate design choice—the SAMPO uses *port-mapped* I/O rather than memory-mapped I/O.

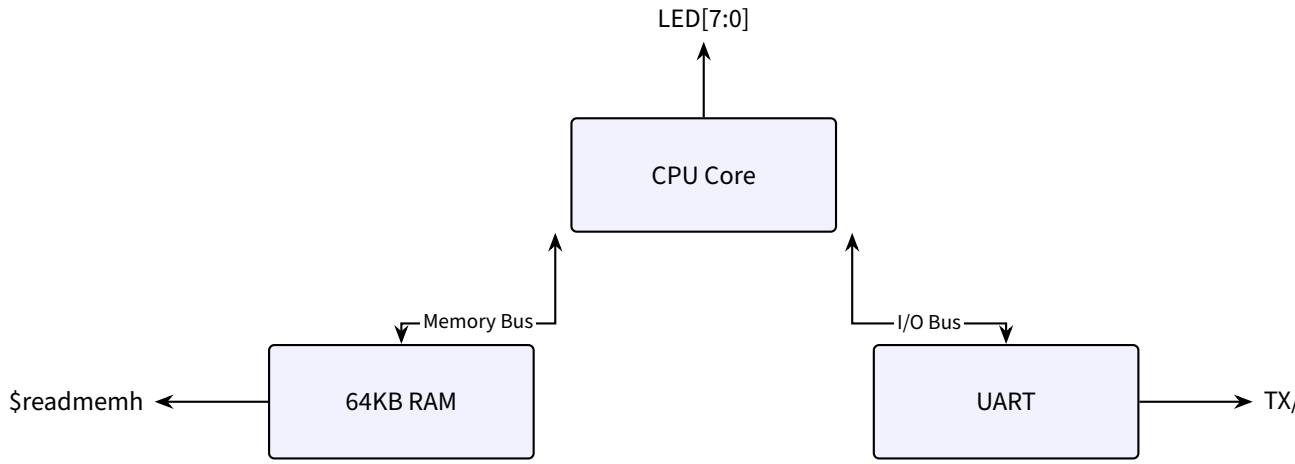


Figure 23.1: The SAMPO SoC block diagram

Port-Mapped vs. Memory-Mapped I/O

In *memory-mapped I/O*, peripherals are accessed via load and store instructions to specific memory addresses. In *port-mapped I/O*, peripherals are accessed via dedicated **`IN`** and **`OUT`** instructions on a separate address space. The SAMPO uses port-mapped I/O, keeping the full 64KB memory space available for code and data. The I/O port space is 8 bits wide (256 ports), separate from the 16-bit memory address space.

The SoC module's port declaration captures these interfaces:

Listing 23.1: SoC module declaration

```
module soc #(
    parameter RESET_VECTOR = 16'h0100,
    parameter RAM_INIT_FILE = ""
) (
    input  wire        clk,
    input  wire        rst_n,

    // External UART interface
    output wire [7:0]  tx_data,
    output wire        tx_valid,
    input  wire        tx_ready,
    input  wire [7:0]  rx_data,
    input  wire        rx_valid,
    output wire        rx_ready,
```

```
    // Status outputs
    output wire        halted,
    output wire [15:0] pc,
    output wire [31:0] cycles,

    // LED output
    output wire [7:0]  led
);
```

Two parameters control the SoC's configuration:

- `RESET_VECTOR`: the address where the CPU begins execution after reset. Defaults to `0x0100`, leaving the first 256 bytes available for an interrupt vector table.
- `RAM_INIT_FILE`: the path to a hex file that initializes the RAM contents at simulation start. This is how programs are loaded.

23.2 Memory Bus: addr, rdata, wdata, we, be

The memory bus connects the CPU to the RAM with five signal groups:

Listing 23.2: CPU-to-memory bus signals

```
wire [15:0] cpu_mem_addr;
wire [15:0] cpu_mem_rdata;
wire [15:0] cpu_mem_wdata;
wire        cpu_mem_we;
wire [1:0]  cpu_mem_be;
wire        cpu_mem_valid;
wire        cpu_mem_ready;
```

The valid/ready handshake follows a straightforward protocol:

1. The CPU asserts `mem_valid` with the address (and write data/enables for stores).
2. The RAM performs the operation on the next clock edge.
3. `mem_ready` asserts one cycle after `mem_valid`, signaling that `mem_rdata` is valid (for reads) or the write has completed.

The SoC generates `mem_ready` by simply delaying `mem_valid` by one clock cycle:

Listing 23.3: Memory ready generation

```
// Memory ready is delayed by 1 cycle for synchronous RAM
```

Table 23.1: Memory bus signals

Signal	Width	Direction	Purpose
`mem_addr`	16	CPU → RAM	Byte address. Bit [0] is used for byte-enable selection; bits [15:1] form the word address.
`mem_rdata`	16	RAM → CPU	Read data. Always returns a full 16-bit word.
`mem_wdata`	16	CPU → RAM	Write data. For byte writes, the byte is duplicated in both halves.
`mem_we`	1	CPU → RAM	Write enable. High for stores, low for loads.
`mem_be`	2	CPU → RAM	Byte enables. `[0]` = low byte, `[1]` = high byte. Both set for word operations.
`mem_valid`	1	CPU → RAM	Request valid. Asserted when the CPU wants a memory access.
`mem_ready`	1	SoC → CPU	Response ready. Asserted when data is available (reads) or write is complete (writes).

```
reg mem_valid_d;
always @(posedge clk or negedge rst_n) begin
    if (!rst_n)
        mem_valid_d <= 1'b0;
    else
        mem_valid_d <= cpu_mem_valid;
end
assign cpu_mem_ready = mem_valid_d;
```

Why Not Use the RAM's Own Ready Signal?

For synchronous RAM, the read data is always available exactly one cycle after the address is presented. There is no variable latency. Using a delayed version of `mem_valid` as the ready signal is the most straightforward approach and avoids adding output logic to the RAM module. If the design were extended to support external SDRAM or flash with variable latency, a true ready signal from the memory controller would be needed.

23.3 Synchronous RAM

The RAM module is compact—42 lines of Verilog that implement a 64KB synchronous memory with byte-enable support:

Listing 23.4: Complete RAM module

```
module ram #(
    parameter INIT_FILE = ""
) (
    input  wire        clk,
    input  wire [15:0] addr,
    input  wire [15:0] wdata,
    output reg  [15:0] rdata,
    input  wire        we,
    input  wire [1:0]  be,
    input  wire        en
);

    // 64KB = 32K x 16-bit words
    reg [15:0] mem [0:32767];

    // Word-aligned address (ignore LSB)
    wire [14:0] word_addr = addr[15:1];

    // Initialize from hex file if specified
    initial begin
        if (INIT_FILE != "") begin
            $readmemh(INIT_FILE, mem);
        end
    end

    // Synchronous read and write
    always @(posedge clk) begin
        if (en) begin
            if (we) begin
                // Write with byte enables
                if (be[0]) mem[word_addr][7:0]  <= wdata[7:0];
                if (be[1]) mem[word_addr][15:8] <= wdata[15:8];
            end
            // Read (always returns full word)
            rdata <= mem[word_addr];
        end
    end

endmodule
```

23.3.1 Memory Organization

The 64KB address space is organized as 32,768 words of 16 bits each. The 16-bit byte address from the CPU is converted to a 15-bit word address by dropping the LSB: `wire [14:0] word_addr = addr [15:1]`.

Since the SAMPO is little-endian, the low byte of each word is at the even address and the high byte is at the odd address. When the CPU performs a word access, it presents the even address, and the RAM returns the full 16-bit word. For byte accesses, the CPU presents the byte address (even or odd), and the SoC uses byte enables and address bit [0] to select the correct byte.

Little-Endian Byte Order

In a little-endian machine, the least significant byte is stored at the lowest address. For a 16-bit value `0x1234` stored at address `0x0100`:

- Address `0x0100` (even) contains `0x34` (low byte).
- Address `0x0101` (odd) contains `0x12` (high byte).

This matches the x86 and ARM conventions. The RAM module stores words with the low byte in bits [7:0] and the high byte in bits [15:8].

23.4 Address Decoding and Byte Enables

In the SAMPO SoC, address decoding is straightforward because there is only one memory device. All addresses from `0x0000` to `0xFFFF` map to the 64KB RAM. There is no need for an address decoder to select between multiple memory devices.

Byte enables, however, require careful handling. The 2-bit `mem_be` signal controls which bytes of a word are written during store operations:

- `2'b11`: Write both bytes (word store, **SW**).
- `2'b01`: Write only the low byte (byte store to even address).
- `2'b10`: Write only the high byte (byte store to odd address).

The CPU generates the correct byte enable based on the address alignment:

Listing 23.5: Byte enable generation for stores (from cpu.v)

```
if (dec_mem_byte) begin
    mem_be = mem_addr[0] ? 2'b10 : 2'b01;
end
```

For byte stores, the data byte is duplicated in both halves of the 16-bit write data. The byte enable then selects which half actually gets written to the RAM:

Listing 23.6: Byte store data duplication (from cpu.v)

```
mem_wdata = dec_mem_byte ?
    {rs2_val[7:0], rs2_val[7:0]} : rs2_val;
```

Why Duplicate the Byte?

Duplicating the byte in both halves of the write data simplifies the RAM design. The RAM does not need to know whether it is performing a byte or word write—it always writes whatever bytes are enabled by `be`. The duplication ensures the correct byte value appears at whichever position the byte enable selects. This is a common technique in memory controller design.

23.4.1 Program Loading with $readmemh

Programs are loaded into the RAM at simulation time using Verilog's `$readmemh` system task:

Listing 23.7: RAM initialization

```
initial begin
    if (INIT_FILE != "") begin
        $readmemh(INIT_FILE, mem);
    end
end
```

The hex file contains one 16-bit word per line in hexadecimal format, stored in word-address order. For example, a program starting at byte address `0x0100` (word address 128) would have its first instruction at line 129 of the hex file (line 1 is word address 0):

Listing 23.8: Converting a binary to hex for $readmemh

```
cd sasm && cargo run --release -- ../examples/hello.s -o hello.bin
cd ../verilog && python3 bin2hex.py ../sasm/hello.bin programs/hello.hex
make sim RAM_INIT_FILE=programs/hello.hex
```

The `bin2hex.py` script converts the assembler's binary output into the hex format that `$readmemh` expects. For FPGA synthesis, the same hex file can be used with the synthesis tool's memory initialization mechanism.

23.5 The UART Controller

The UART (Universal Asynchronous Receiver/Transmitter) provides serial communication between the SAMPO and the outside world. The implementation in `verilog/rtl/uart.v` uses a register model inspired by the Motorola MC6850 ACIA, with two I/O ports:

Table 23.2: UART port map

Port	Read	Write
`0x80`	Status register	(not used)
`0x81`	Receive data	Transmit data

Listing 23.9: UART module port declaration

```
module uart (
    input  wire        clk,
    input  wire        rst_n,

    // CPU I/O interface
    input  wire [7:0]  port,
    input  wire [7:0]  wdata,
    output reg  [7:0]  rdata,
    input  wire        rd,
    input  wire        wr,

    // External UART interface
    output reg  [7:0]  tx_data,
    output reg         tx_valid,
    input  wire        tx_ready,
    input  wire [7:0]  rx_data,
    input  wire        rx_valid,
    output reg         rx_ready
);
```

The module has two interfaces: the CPU-facing I/O bus (port address, data, read/write strobes) and the external UART interface (TX data/valid/ready, RX data/valid/ready). The external interface uses a standard valid/ready handshake, allowing it to connect to a testbench UART model, a physical serial transceiver, or any other compatible interface.

23.5.1 The Status Register

The status register at port `0x80` provides two bits of information:

Listing 23.10: UART status register

```
// Status bits (MC6850-compatible)
// Bit 0: RDRF - Receive Data Register Full
// Bit 1: TDRE - Transmit Data Register Empty (TX ready)
wire [7:0] status;
assign status = {6'b0, tx_ready, rx_pending};
```

- **Bit 0** (RDRF—Receive Data Register Full): Set when received data is available in the RX register. Cleared when the CPU reads the data register.
- **Bit 1** (TDRE—Transmit Data Register Empty): Set when the transmitter is ready to accept a new byte. Cleared while a transmission is in progress.

Software polls the status register before reading or writing data:

Listing 23.11: Polling the UART status register

```
wait_tx:
    INX  R4, 0x80       ; Read UART status
    ANDI R4, R4, 0x02   ; Test TDRE bit
    BEQ  wait_tx         ; Loop until TX ready
    OUTX 0x81, R5        ; Send byte
```

23.5.2 Transmit Logic

The transmit path uses a holding register to decouple the CPU write from the external handshake:

Listing 23.12: UART transmit logic

```
// TX state
reg [7:0] tx_reg;
reg       tx_pending;

always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        tx_valid <= 1'b0;
        tx_pending <= 1'b0;
        tx_data <= 8'h00;
        tx_reg <= 8'h00;
    end else begin
        // CPU write to data port
        if (wr && port == PORT_DATA) begin
            tx_reg <= wdata;
            tx_pending <= 1'b1;
```

```
        end

        // Clear valid after it's accepted
        if (tx_valid && tx_ready) begin
            tx_valid <= 1'b0;
        end

        // Set valid when we have pending data and TX is ready
        if (tx_pending && !tx_valid && tx_ready) begin
            tx_valid <= 1'b1;
            tx_data <= tx_reg;
            tx_pending <= 1'b0;
        end
    end
end
```

The transmit sequence:

1. The CPU writes a byte to port `0x81`. The byte is captured in `tx_reg` and `tx_pending` is set.
2. On the next cycle where the external transmitter is ready (`tx_ready` high) and no previous byte is still being asserted (`tx_valid` low), the byte is transferred to `tx_data` and `tx_valid` is asserted.
3. The external interface accepts the byte (acknowledges with `tx_ready`), and `tx_valid` is de-asserted.

The holding register allows the CPU to write a byte and continue execution without waiting for the external transmitter to finish with the previous byte. If the CPU writes too fast (before the previous byte is consumed), the new byte overwrites `tx_reg` and the old byte may be lost. Software must poll the TDRE status bit to avoid this.

23.5.3 Receive Logic

The receive path mirrors the transmit path with a pending flag:

Listing 23.13: UART receive logic

```
// RX state
reg [7:0] rx_reg;
reg       rx_pending;

always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        rx_ready <= 1'b0;
        rx_pending <= 1'b0;
```

```
        rx_reg <= 8'h00;
    end else begin
        // CPU read from data port clears pending
        if (rd && port == PORT_DATA) begin
            rx_pending <= 1'b0;
        end

        // Accept new data when available and not pending
        if (rx_valid && !rx_pending) begin
            rx_reg <= rx_data;
            rx_pending <= 1'b1;
            rx_ready <= 1'b1;
        end

        // Clear ready after one cycle
        if (rx_ready) begin
            rx_ready <= 1'b0;
        end
    end
end
```

The receive sequence:

1. External data arrives (`rx_valid` asserted). If no previous byte is pending (`rx_pending` = 0), the byte is captured in `rx_reg` and `rx_pending` is set. The `rx_ready` signal pulses for one cycle to acknowledge receipt.
2. The status register reports RDRF = 1 (bit 0).
3. The CPU reads port `0x81`, receiving the byte from `rx_reg`. The read clears `rx_pending`, and the status register returns to RDRF = 0.

If a new byte arrives while the previous one has not been read, it is dropped (`rx_valid && !rx_pending` is false). Software must read received bytes promptly or risk data loss.

23.5.4 CPU Read Path

The read multiplexer is combinational, selecting between the status and data registers based on the port address:

Listing 23.14: UART CPU read path

```
always @(*) begin
    rdata = 8'h00;
    if (rd) begin
```

```
        case (port)
            PORT_STATUS: rdata = status;
            PORT_DATA:   rdata = rx_reg;
            default:     rdata = 8'h00;
        endcase
    end
end
```

Reading from any port other than `0x80` or `0x81` returns zero. This is a safe default—unimplemented ports do not cause errors, they return nothing.

23.6 Baud Rate Generation

The UART module in the SAMPO SoC handles the I/O register interface but does *not* include baud rate generation or the actual serial bit-banging logic. These are handled by the external interface: in simulation, the testbench models the serial protocol directly; on an FPGA, a separate baud rate generator and shift register drive the physical TX/RX pins.

This separation keeps the SoC module portable across different clock frequencies and baud rates. The UART's valid/ready handshake is clock-domain agnostic—it works at any speed, with the external interface controlling the actual serial timing.

Baud Rate for the ULX3S

When targeting the ULX3S FPGA board at 25 MHz, a common baud rate is 115200. The baud rate divisor is $\lfloor 25{,}000{,}000/115{,}200 \rfloor = 217$. The baud rate generator counts 217 clock cycles per bit period, providing the timing reference for the serial shift register. See **??** for the complete FPGA integration.

23.7 Status Register and LED Outputs

The SoC provides an 8-bit LED output that gives visual feedback on the CPU's state:

Listing 23.15: LED status output

```
// LED[7]:   CPU halted
// LED[6]:   UART TX active (data being sent)
// LED[5]:   UART RX data available
// LED[4:0]: Lower 5 bits of PC (shows execution progress)
assign led = {halted, tx_valid, rx_valid, pc[5:1]};
```

Each LED bit conveys specific information:

Table 23.3: LED output assignments

Bit	Source	Meaning
7	`halted`	Lit when the CPU has executed **`HALT`**
6	`tx_valid`	Flickers during UART transmission
5	`rx_valid`	Briefly lit when data is received
4:0	`pc[5:1]`	Shows PC bits 5 through 1, creating a cycling pattern during execution

The lower 5 bits show `pc[5:1]` (not `pc[4:0]`) because bit 0 of the PC is always zero (instructions are word-aligned). Using bits 5:1 gives a 32-position counter that visibly cycles as the CPU executes through a program, providing at-a-glance confirmation that the CPU is running. When the CPU halts, LED 7 lights up and the lower bits freeze at the final PC value.

LEDs as Debugging Tools

On an FPGA board, these LEDs are invaluable for basic debugging. If LED 7 never lights up, the CPU never reached a **`HALT`** instruction—possibly stuck in an infinite loop. If LED 6 never flickers, the program never wrote to the UART. If the lower LEDs show a fixed pattern, the CPU may be stuck at a specific address. This crude but immediate feedback often catches problems before you even connect a logic analyzer.

23.8 Putting It All Together

The SoC top level wires all three components together. The CPU is instantiated with its reset vector parameter:

Listing 23.16: CPU instantiation in the SoC

```
cpu #(
    .RESET_VECTOR(RESET_VECTOR)
) cpu_inst (
    .clk(clk),
    .rst_n(rst_n),
    .mem_addr(cpu_mem_addr),
    .mem_rdata(cpu_mem_rdata),
    .mem_wdata(cpu_mem_wdata),
    .mem_we(cpu_mem_we),
    .mem_be(cpu_mem_be),
    .mem_valid(cpu_mem_valid),
    .mem_ready(cpu_mem_ready),
```

```
    .io_addr(cpu_io_addr),
    .io_rdata(cpu_io_rdata),
    .io_wdata(cpu_io_wdata),
    .io_rd(cpu_io_rd),
    .io_wr(cpu_io_wr),
    .halted(halted),
    .pc_out(pc),
    .cycles_out(cycles)
);
```

The RAM receives the memory bus signals directly:

Listing 23.17: RAM instantiation

```
ram #(
    .INIT_FILE(RAM_INIT_FILE)
) ram_inst (
    .clk(clk),
    .addr(cpu_mem_addr),
    .wdata(cpu_mem_wdata),
    .rdata(cpu_mem_rdata),
    .we(cpu_mem_we),
    .be(cpu_mem_be),
    .en(cpu_mem_valid)
);
```

And the UART connects to the I/O bus:

Listing 23.18: UART instantiation

```
uart uart_inst (
    .clk(clk),
    .rst_n(rst_n),
    .port(cpu_io_addr),
    .wdata(cpu_io_wdata),
    .rdata(cpu_io_rdata),
    .rd(cpu_io_rd),
    .wr(cpu_io_wr),
    .tx_data(tx_data),
    .tx_valid(tx_valid),
    .tx_ready(tx_ready),
    .rx_data(rx_data),
    .rx_valid(rx_valid),
    .rx_ready(rx_ready)
);
```

The `en` input on the RAM is driven by `cpu_mem_valid`. When the CPU is not accessing memory (during EXECUTE, DECODE, or WRITEBACK), `mem_valid` is low, and the RAM holds its previous output. This saves a small amount of power on an FPGA, as the RAM block does not toggle on every clock cycle.

Exercises

1. **Write a "Hello, World!" program.** Write a complete SAMPO assembly program that prints "Hello, World!" to the UART. You will need to poll the status register at port `0x80` for the TDRE bit before each character, then write the character to port `0x81`. Test it in simulation using the SoC testbench.

2. **Add a timer peripheral.** Design a simple countdown timer as a new I/O peripheral. It should have three ports: a control register (start/stop), a count register (16-bit, counts down each clock cycle), and a status register (bit 0 = timer expired). Write the Verilog module and integrate it into the SoC at ports `0x90`–`0x92`. What changes are needed in the SoC to support multiple I/O devices?

3. **Memory map analysis.** The SAMPO SoC has a flat 64KB memory space with no address decoding. Suppose you wanted to add a second RAM block at addresses `0x8000`–`0xFFFF` and restrict the first RAM to `0x0000`–`0x7FFF`. Write the address decoding logic and explain how the `mem_rdata` and `mem_ready` signals would need to be multiplexed.

4. **UART echo program.** Write a SAMPO assembly program that reads characters from the UART and echoes them back. The program should poll the RDRF bit of the status register, read the data register when a byte is available, and then write that byte back to the transmit data register (after checking TDRE). Test it by sending characters through the simulation testbench.

What's Next

We now have a complete System on Chip: a CPU that can execute programs, a RAM to hold them, and a UART to communicate. But how do we know it works? Building hardware that *seems* right is not the same as hardware that *is* right. In the next part of the book, we turn to verification—writing testbenches that exercise every instruction, building an ISA test suite, and using waveform analysis to track down the bugs that inevitably appear when theory meets silicon.

Part VII

Simulation and Verification

Chapter 24

Testbench Design

A CPU design that compiles without errors is not a CPU that works. The gap between "synthesizes cleanly" and "executes instructions correctly" is vast, and the only bridge across it is verification. In this chapter, we build that bridge: the testbenches that let us exercise every corner of the SAMPO before it ever touches an FPGA.

We will dissect the anatomy of a Verilog testbench from clock generation through stimulus to result checking. We will build self-checking testbenches that report pass or fail without human interpretation. We will generate VCD waveform files and learn to read them. And we will encounter a subtle but critical lesson about UART modeling that cost several hours of debugging: the `tx_ready=1` problem.

24.1 Testbench Anatomy

A Verilog testbench is a module with no ports. It exists solely in simulation—it will never be synthesized into hardware. Its job is to instantiate the design under test (DUT), drive its inputs, and observe its outputs. Every testbench in the SAMPO project follows the same structural pattern.

24.1.1 The Module Shell

Open `verilog/tb/alu_tb.v`. The file begins with a timescale directive and the module declaration:

Listing 24.1: ALU testbench: module declaration

```
`timescale 1ns/1ps

`include "sampo_pkg.vh"
```

```
module alu_tb;

    reg  [15:0] a, b;
    reg  [3:0]  op;
    reg         carry_in;
    wire [15:0] result;
    wire        flag_n, flag_z, flag_c, flag_v;
```

Three things to note here. First, the `` `timescale 1ns/1ps `` directive sets the time unit to one nanosecond and the precision to one picosecond. This means that `#10` advances simulation by 10 ns, and the simulator can resolve events down to 1 ps. Second, we include `sampo_pkg.vh` to get access to the ALU operation constants (`` `ALU_ADD ``, `` `ALU_SUB ``, and so on). Third, the module has no port list—testbenches are top-level modules that drive everything internally.

The signal declarations mirror the DUT's ports, but with opposite directions: inputs to the DUT are declared as `reg` (because the testbench drives them), and outputs from the DUT are declared as `wire` (because the testbench observes them).

Design Under Test (DUT)

The *design under test* (often abbreviated DUT or UUT for "unit under test") is the module being verified. In a testbench, the DUT is instantiated as a submodule and connected to the testbench's internal signals. The testbench supplies stimulus on the DUT's inputs and checks correctness on its outputs.

24.1.2 Instantiating the DUT

The DUT is instantiated with explicit port connections:

Listing 24.2: ALU testbench: DUT instantiation

```
    alu uut (
        .a(a),
        .b(b),
        .op(op),
        .carry_in(carry_in),
        .result(result),
        .flag_n(flag_n),
        .flag_z(flag_z),
        .flag_c(flag_c),
        .flag_v(flag_v)
    );
```

The instance is named `uut`—a convention used throughout the SAMPO testbenches. Named port connections (`.port(signal)`) are always preferable to positional connections because they survive port reordering in the DUT without silently breaking the testbench.

Name Your Instances Consistently

Using `uut` (unit under test) as the instance name in every testbench makes it straightforward to write reusable waveform viewer configurations. When debugging in GTKWave, you always know the hierarchy path starts with `uut`.

24.1.3 Clock Generation

The ALU is a purely combinational module, so it needs no clock. But most SAMPO modules are sequential, and sequential logic needs a clock. The system testbench in `verilog/tb/sampo_tb.v` generates its clock with two lines:

Listing 24.3: Clock generation in the system testbench

```
parameter CLK_PERIOD = 10;  // 100 MHz

initial clk = 0;
always #(CLK_PERIOD/2) clk = ~clk;
```

The `initial` block sets the clock to zero at time zero. The `always` block then toggles the clock every half-period, producing a clean 50% duty cycle square wave. With `CLK_PERIOD = 10` (10 ns), each half-period is 5 ns, giving a 100 MHz clock.

The register file testbench uses the same pattern with a shorter period:

Listing 24.4: Clock generation in the register file testbench

```
initial clk = 0;
always #5 clk = ~clk;
```

Here the period is hardcoded at 10 ns (5 ns per half-cycle). This is fine for a unit test that does not need to match the system clock precisely. The key requirement is that the clock toggles at regular intervals and that the testbench author knows the relationship between simulation time and clock cycles.

Why `initial` Before `always`?

The `initial clk = 0` ensures the clock starts in a known state at time zero. Without it, the `always` block would toggle an uninitialized value (`x`), and the first posedge would occur at an unpredictable time. Always initialize your clock.

24.1.4 The Reset Sequence

Every sequential design needs a reset, and the testbench must model the reset correctly. The SAMPO uses active-low asynchronous reset (`rst_n`). The system testbench applies it like this:

Listing 24.5: Reset sequence in the system testbench

```
initial begin
    $display("=== Sampo CPU Testbench ===");
    $display("RAM init file: %s", RAM_INIT_FILE);

    // Initialize
    rst_n = 0;
    #100;
    rst_n = 1;

    $display("CPU started at PC=0x%04X", pc);
end
```

Reset is asserted (driven low) at time zero and held for 100 ns—ten full clock cycles at 100 MHz. This gives every flip-flop in the design time to settle into its reset state. Then reset is deasserted (driven high), and the CPU begins executing from its reset vector.

The register file testbench follows the same pattern:

Listing 24.6: Reset sequence in the register file testbench

```
initial begin
    errors = 0;
    rst_n = 0;
    wr_en = 0;
    exx = 0;
    rd_addr1 = 0;
    rd_addr2 = 0;
    wr_addr = 0;
    wr_data = 0;

    // Reset
    #20;
```

```
        rst_n = 1;
        #10;
    end
```

Note the additional initialization of all input signals to zero during reset. This is critical: in simulation, undriven `reg` signals start as `x` (unknown). If any of these unknowns propagate into the DUT before reset, they can corrupt state in ways that survive the reset deassertion. Always initialize every input before releasing reset.

Initialize All Inputs During Reset

An uninitialized `reg` in Verilog holds the value `x`. If the DUT samples an `x` on a control signal during reset, the result is undefined. Initialize every testbench input to a known value at time zero—not just the ones you think the DUT reads during reset.

24.1.5 Applying Stimulus

With the DUT instantiated, clock running, and reset released, we need to drive the inputs. The approach depends on whether we are testing combinational or sequential logic.

For the purely combinational ALU, stimulus is applied by assigning values and waiting for propagation:

Listing 24.7: Combinational stimulus in the ALU testbench

```
        // Test ADD
        op = `ALU_ADD;
        a = 16'h0001; b = 16'h0002; #10;
        // result is now 0x0003

        a = 16'hFFFF; b = 16'h0001; #10;
        // result is now 0x0000 with carry
```

The `#10` delay (10 ns) gives the combinational logic time to propagate. In practice, a few nanoseconds would suffice, but 10 ns keeps the waveforms readable.

For sequential modules, we synchronize stimulus to clock edges. The register file testbench writes a value and then waits for the next clock edge:

Listing 24.8: Sequential stimulus in the register file testbench

```
        // Write to R0 - should be ignored
        wr_addr = 4'h0;
        wr_data = 16'hDEAD;
```

```
        wr_en = 1;
        @(posedge clk);
        wr_en = 0;
        #1;
        check_read(4'h0, 16'h0000);
```

The `@(posedge clk)` construct suspends execution until the next rising clock edge, ensuring the write is captured by the synchronous register file. The subsequent `#1` (1 ns delay) gives the combinational read path time to settle before we check the output.

This pattern—set up inputs, wait for clock edge, wait for propagation, check outputs—is the fundamental rhythm of sequential testbench writing.

24.1.6 Termination and Summary

Every testbench must terminate. In the ALU testbench, the final statement is:

Listing 24.9: Testbench termination with summary

```
        $display("=== ALU Tests Complete ===");
        if (errors == 0) begin
            $display("PASSED: All tests passed!");
        end else begin
            $display("FAILED: %d errors", errors);
        end

        $finish;
```

The `$finish` system task tells the simulator to stop. Without it, the simulation would run forever (the clock's `always` block never terminates on its own).

The system testbench has a more sophisticated termination strategy. It uses a `fork/join` to race two conditions: the CPU halting or a timeout expiring:

Listing 24.10: Fork/join termination in the system testbench

```
        fork
            begin : timeout_block
                #(CLK_PERIOD * MAX_CYCLES);
                $display("TIMEOUT: Max cycles (%d) reached",
                         MAX_CYCLES);
                disable wait_halt;
            end

            begin : wait_halt
```

```
            wait(halted);
            #10;
            disable timeout_block;
        end
    join
```

The `fork` launches both blocks in parallel. If the CPU asserts `halted` first, the `wait_halt` block wins and disables the timeout. If 500,000 cycles pass without a halt, the timeout fires and disables the halt-wait. Either way, exactly one of the two blocks completes, and the testbench proceeds to print its summary.

Always Include a Timeout

A CPU with a bug may loop forever. Without a timeout, the simulation will run until you manually kill it (or until your disk fills with VCD data). The `MAX_CYCLES` parameter in `sampo_tb.v` provides a hard upper bound. Set it high enough for your longest test program but low enough to catch infinite loops within a reasonable time.

24.2 Self-Checking Testbenches

A testbench that requires a human to read the output and decide whether it passed is a testbench that will eventually lie to you. Self-checking testbenches compare actual results against expected values automatically and report errors without human interpretation.

24.2.1 The Check Task Pattern

The SAMPO testbenches use Verilog `task` blocks to encapsulate the check logic. The ALU testbench defines a `check` task that verifies both the result and all four flags:

Listing 24.11: Self-checking task in the ALU testbench

```
    integer errors;

    task check;
        input [15:0] expected;
        input expected_n, expected_z, expected_c, expected_v;
        begin
            if (result !== expected ||
                flag_n !== expected_n ||
                flag_z !== expected_z ||
                flag_c !== expected_c ||
                flag_v !== expected_v) begin
                $display("ERROR: op=%h a=%h b=%h c_in=%b",
```

```
                    op, a, b, carry_in);
            $display("  Got:      result=%h N=%b Z=%b C=%b V=%b",
                result, flag_n, flag_z, flag_c, flag_v);
            $display("  Expected: result=%h N=%b Z=%b C=%b V=%b",
                expected, expected_n, expected_z,
                expected_c, expected_v);
            errors = errors + 1;
        end
    end
endtask
```

Several design decisions here deserve attention:

- **Use `!==` not `!=`.** The `!==` operator is the "case inequality" that treats `x` and `z` as actual values. If the result contains an unknown bit, `!==` catches it. The regular `!=` would return `x` when comparing against `x`, which evaluates as false in an `if` condition—silently hiding the error.
- **Check everything, not just the result.** ALU bugs often manifest in the flags rather than the result. The SAMPO's ARM-style subtract carry (C=1 means no borrow) is a frequent source of confusion, and the only way to catch carry flag errors is to verify them explicitly.
- **Print context on failure.** When the check fails, the error message includes the inputs (`op`, `a`, `b`, `carry_in`), the actual outputs, and the expected outputs. This makes debugging vastly faster than "test 47 failed."
- **Count errors, do not abort.** The task increments an `errors` counter rather than calling `$finish`. This lets the testbench run to completion, reporting all failures at once rather than stopping at the first one.

Case Equality Operators

Verilog has four equality operators. The logical operators `==` and `!=` return `x` if either operand contains `x` or `z`. The case operators `===` and `!==` compare bit-by-bit, including `x` and `z`, and always return 0 or 1. In testbenches, always use `===` and `!==` to catch propagating unknowns.

24.2.2 Using the Check Task

Each test case is a single line: set inputs, wait for propagation, call `check` with expected values:

Listing 24.12: ALU test cases for ADD

```
    // Test ADD
    $display("Testing ADD...");
    op = `ALU_ADD;
```

```
        a = 16'h0001; b = 16'h0002; #10;
        check(16'h0003, 0, 0, 0, 0);

        a = 16'hFFFF; b = 16'h0001; #10;
        check(16'h0000, 0, 1, 1, 0);  // Overflow to 0

        a = 16'h7FFF; b = 16'h0001; #10;
        check(16'h8000, 1, 0, 0, 1);  // Signed overflow

        a = 16'h8000; b = 16'h8000; #10;
        check(16'h0000, 0, 1, 1, 1);  // Both overflow
```

Each test case documents itself through its arguments. The comments explain the *intent*—what corner case this test exercises—while the `check` arguments specify the exact expected behavior. If the ALU's carry logic ever changes, these check calls will catch it.

24.2.3 The Register File Check Pattern

The register file testbench uses a slightly different check task, oriented around reading a register and comparing:

Listing 24.13: Self-checking task in the register file testbench

```
    task check_read;
        input [3:0] addr;
        input [15:0] expected;
        begin
            rd_addr1 = addr;
            #1;
            if (rd_data1 !== expected) begin
                $display("ERROR: R%d read %h, expected %h",
                         addr, rd_data1, expected);
                errors = errors + 1;
            end
        end
    endtask
```

This task bundles the stimulus (setting the read address) with the check (comparing the output). This is appropriate when the stimulus and check are tightly coupled, as they are for a register file read.

The power of this pattern emerges when testing complex sequences like the **EXX** instruction, which swaps registers `R4`–`R11` with a shadow bank:

Listing 24.14: Testing the EXX swap in the register file testbench

```
        // Write known values to R4-R11
        for (i = 4; i < 12; i = i + 1) begin
            wr_addr = i[3:0];
            wr_data = 16'hAA00 + i;
            wr_en = 1;
            @(posedge clk);
            #1;
            wr_en = 0;
        end

        @(posedge clk); #1;

        // Execute EXX
        exx = 1;
        @(posedge clk); #1;
        exx = 0;

        // R4-R11 should now be 0 (alternate bank was reset)
        for (i = 4; i < 12; i = i + 1) begin
            check_read(i[3:0], 16'h0000);
        end

        // R0-R3 and R12-R15 should be unchanged
        check_read(4'h0, 16'h0000);
        check_read(4'h1, 16'h1001);
        check_read(4'h2, 16'h1002);
        check_read(4'h3, 16'h1003);
```

This test establishes known state (write `0xAA04` through `0xAA0B` to `R4–R11`), performs the operation (**`EXX`**), and then verifies every register. It checks not only that `R4–R11` swapped (the alternate bank was zero from reset), but also that registers outside the swap range (`R0–R3` and `R12–R15`) were not disturbed. Then a second **`EXX`** should restore the original values:

Listing 24.15: Verifying double EXX restores original values

```
        // EXX again to restore
        exx = 1;
        @(posedge clk); #1;
        exx = 0;

        for (i = 4; i < 12; i = i + 1) begin
            check_read(i[3:0], 16'hAA00 + i);
        end
```

A non-self-checking testbench would have required manually inspecting 32 register values across two swap operations. The self-checking version runs in milliseconds and reports its verdict with a single line of output.

24.2.4 System-Level Self-Checking

The system testbench in `sampo_tb.v` takes self-checking to the next level. Instead of checking individual signals, it monitors the UART output for the strings "PASS" and "FAIL" emitted by test programs running on the CPU:

Listing 24.16: PASS/FAIL detection in the system testbench

```
// Line buffer for detecting PASS/FAIL
reg [7:0]  line_buf [0:255];
integer    line_idx;

task check_line;
    integer i;
    reg found_pass, found_fail;
    begin
        found_pass = 0;
        found_fail = 0;
        for (i = 0; i < line_idx - 3; i = i + 1) begin
            if (line_buf[i] == "P" &&
                line_buf[i+1] == "A" &&
                line_buf[i+2] == "S" &&
                line_buf[i+3] == "S")
                found_pass = 1;
            if (line_buf[i] == "F" &&
                line_buf[i+1] == "A" &&
                line_buf[i+2] == "I" &&
                line_buf[i+3] == "L")
                found_fail = 1;
        end
        if (found_pass) pass_count = pass_count + 1;
        if (found_fail) fail_count = fail_count + 1;
    end
endtask
```

The `check_line` task scans each line of UART output for the four-character substrings "PASS" and "FAIL". This is a character-at-a-time search—Verilog has no built-in string operations—but it works reliably for the SAMPO test output format (see Chapter 25).

The testbench calls `check_line` whenever it encounters a newline character (`0x0A`) in the UART data stream. At the end of simulation, it prints a summary:

Listing 24.17: Test results summary

```
        if (pass_count > 0 || fail_count > 0) begin
            $display("Test Results:");
            $display("  PASSED: %0d", pass_count);
            $display("  FAILED: %0d", fail_count);
            if (fail_count > 0)
                $display("  *** FAILURES DETECTED ***");
            else
                $display("  All tests passed!");
        end

        if (fail_count > 0)
            $finish(1);
        else
            $finish(0);
```

The `$finish` call takes an argument: 0 for success, 1 for failure. Some simulators use this as a process exit code, making it possible to integrate the testbench into CI/CD pipelines where a non-zero exit code signals failure.

Self-Checking Testbench

A *self-checking testbench* automatically compares actual DUT outputs against expected values and reports a pass/fail verdict without human interpretation. The three requirements are: (1) known expected values, (2) automatic comparison, and (3) a machine-readable verdict. Testbenches that rely on visual waveform inspection are *not* self-checking.

24.3 VCD Waveform Generation and Analysis

Self-checking testbenches tell you *whether* something is wrong. Waveforms tell you *what* is wrong. When a test fails, the first thing you reach for is the VCD (Value Change Dump) file—a complete recording of every signal in the design over the course of simulation.

24.3.1 Generating VCD Files

The system testbench generates a VCD file with two system calls:

Listing 24.18: VCD generation in the system testbench

```
    initial begin
        $dumpfile("sampo.vcd");
        $dumpvars(0, sampo_tb);
    end
```

The `$dumpfile` call specifies the output filename. The `$dumpvars` call specifies what to record. The first argument is the depth: 0 means "all levels of hierarchy below the given module." The second argument is the scope: `sampo_tb` is the top-level testbench module. Together, these two calls record every signal in the entire design—from the top-level `clk` down through the SoC, CPU, ALU, register file, and RAM.

VCD File Size

Recording every signal for a 500,000-cycle simulation generates VCD files that can reach hundreds of megabytes. For targeted debugging, consider recording fewer signals by specifying a deeper scope (e.g., `$dumpvars(0, uut.cpu_inst)`) or using `$dumpoff`/`$dumpon` to record only during the cycles of interest.

24.3.2 Viewing Waveforms with GTKWave

The Makefile provides a convenient target for viewing waveforms:

Listing 24.19: Opening waveforms in GTKWave

```
make wave
```

This runs `gtkwave build/sampo.vcd &`, opening the VCD file in GTKWave. The key signals to add to the waveform view when debugging SAMPO programs are:

Save Your GTKWave Configuration

Once you have arranged the signals you care about, save the configuration as a `.gtkw` file (File → Write Save File). Next time you open the VCD, you can reload the same signal list and display settings without manually adding signals one by one.

24.3.3 Debug Trace Output

In addition to VCD files, the system testbench prints a textual trace of the first 100 cycles to the console. This is often enough to diagnose boot-up issues without opening a waveform viewer:

Listing 24.20: Debug trace output for the first 100 cycles

```
    always @(posedge clk) begin
```

Table 24.1: Essential signals for SAMPO waveform debugging

Signal Path	Type	Purpose
`clk`	Clock	Time reference
`rst_n`	Reset	Reset boundary
`uut.cpu_inst.state`	4-bit	FSM state
`uut.cpu_inst.pc`	16-bit	Program counter
`uut.cpu_inst.instr`	16-bit	Current instruction
`uut.cpu_inst.flags`	8-bit	Status flags
`uut.cpu_inst.mem_addr`	16-bit	Memory address bus
`uut.cpu_inst.mem_rdata`	16-bit	Memory read data
`uut.cpu_inst.mem_we`	1-bit	Memory write enable
`uut.cpu_inst.mem_ready`	1-bit	Memory ready
`uut.cpu_inst.alu_result`	16-bit	ALU output
`halted`	1-bit	CPU halted

```
        if (rst_n && cycles < 100) begin
            $display(
                "Cycle %5d: PC=%04X St=%d Ins=%04X"
                " R4=%04X R5=%04X Flags=%02X",
                cycles, pc, uut.cpu_inst.state,
                uut.cpu_inst.instr,
                uut.cpu_inst.regfile_inst.regs[4],
                uut.cpu_inst.regfile_inst.regs[5],
                uut.cpu_inst.flags);
        end
    end
```

This block uses hierarchical references to reach into the CPU and register file—a simulation-only feature that the testbench uses liberally. The output looks like:

Listing 24.21: Sample debug trace output

```
Cycle     1: PC=0100 St=1 Ins=0000 R4=0000 R5=0000 Flags=00
Cycle     2: PC=0100 St=3 Ins=5404 R4=0000 R5=0000 Flags=00
Cycle     3: PC=0100 St=4 Ins=5404 R4=0000 R5=0000 Flags=00
Cycle     4: PC=0102 St=1 Ins=5404 R4=0004 R5=0000 Flags=00
```

Reading this trace, we can follow the FSM: FETCH (St=1) reads the instruction at `0x0100`, DECODE (St=3) latches register values, EXECUTE (St=4) performs the operation, and then back to FETCH at `0x0102`. The instruction `0x5404` is an **`ADDI`** that added 4 to `R4`.

24.3.4 Reading VCD Signals

When debugging from waveforms, the approach is always the same:

1. Find the **symptom**: the cycle where the output first diverges from expected behavior.
2. Trace **backward** through the state machine to find the state that produced the wrong output.
3. Examine the **inputs to that state**: were the operands correct? Were the control signals correct?
4. If the inputs were wrong, recurse: go to the state that produced those inputs and repeat.

This backward-tracing technique is the core of waveform debugging, and we will apply it in detail in Chapter 26 when we track down the MEMORY→FETCH pipeline hazard.

24.4 Parameterized Testbenches

The SAMPO system testbench is designed to verify not one program but hundreds, using parameterization to swap programs without modifying or recompiling the testbench source.

24.4.1 Parameters for Configuration

The system testbench declares three parameters that control its behavior:

Listing 24.22: Testbench parameters in sampo_tb.v

```
parameter CLK_PERIOD = 10;   // 100 MHz
parameter MAX_CYCLES = 500000;
parameter RAM_INIT_FILE = "../programs/hello.hex";
```

- `CLK_PERIOD` controls the simulation clock speed.
- `MAX_CYCLES` sets the timeout—how long to wait before declaring that the program has failed to halt.
- `RAM_INIT_FILE` specifies the hex file that initializes the RAM, which effectively selects which program the CPU will execute.

24.4.2 Parameter Overrides from the Command Line

The Makefile overrides the `RAM_INIT_FILE` parameter when compiling the testbench for a specific ISA test:

Listing 24.23: Parameter override in the Makefile

```
iverilog -Irtl -Psampo_tb.RAM_INIT_FILE=\"../programs/test_alu.hex\" \
    -o build/sampo_tb_test_alu \
    rtl/alu.v rtl/shifter.v rtl/regfile.v rtl/decode.v \
    rtl/cpu.v rtl/ram.v rtl/uart.v rtl/soc.v \
    tb/sampo_tb.v
```

The `-P` flag to Icarus Verilog overrides a parameter using the syntax `-Pmodule.PARAM=value`. The escaped quotes (`\"...\"`) ensure the file path is passed as a string literal. This compiles a separate executable for each test program, with the program baked into the binary.

Separate Executables per Test

The Makefile creates a separate compiled testbench for each ISA test: `build/sampo_tb_test_alu`, `build/sampo_tb_test_loadstore`, and so on. While this uses more disk space, it allows tests to run in parallel and ensures that a crash in one test does not prevent others from running.

24.4.3 The Makefile Pattern

The Makefile uses Make's pattern rules to handle the entire assemble-convert-compile-run pipeline for any test:

Listing 24.24: ISA test automation in the Makefile

```
# List of all ISA tests
ISA_TESTS = test_alu test_addi test_shift test_muldiv \
            test_loadstore test_branch test_jump \
            test_stack test_misc test_extended

# Assemble test source to binary
../examples/tests/%.bin: ../examples/tests/%.s
  cd ../sasm && cargo run --release -- $< -o $@

# Convert test binary to hex
programs/%.hex: ../examples/tests/%.bin
  python3 bin2hex.py $< $@
```

Running a single test is a one-command operation:

Listing 24.25: Running a single ISA test

```
make isa-test_alu
```

Running all ten tests is equally straightforward:

Listing 24.26: Running all ISA tests

```
make isa-tests
```

This triggers the assembly of each `.s` file, conversion to hex, compilation of a parameterized testbench, and execution—reporting pass or fail for each test program.

24.4.4 Benefits of Parameterization

Parameterization provides several concrete benefits:

1. **Single testbench source**. All ten ISA tests and any ad-hoc test programs use the same `sampo_tb.v`. Bug fixes to the testbench immediately apply to all tests.
2. **Reproducible builds**. Each test's hex file is baked into the compiled testbench at compile time via `$readmemh` in the RAM module. There is no runtime file loading that could fail due to path issues.
3. **CI/CD integration**. The `make isa-tests` target returns a non-zero exit code if any test fails, making it suitable for continuous integration pipelines.
4. **Extensibility**. Adding a new test is a matter of writing the assembly file and adding its name to the `ISA_TESTS` list. The Makefile handles the rest.

24.5 The tx_ready=1 Lesson

This section describes a debugging session that consumed more time than it should have, and the lesson it teaches about testbench fidelity.

24.5.1 The Symptom

During early development of the SAMPO, the CPU was executing programs correctly in the emulator but producing garbled or missing UART output in Verilog simulation. Some characters appeared, some were dropped, and the pattern was not consistent between runs of different programs.

The test programs used a standard pattern to send a character via UART: poll the status port (`0x80`) until the transmitter is ready, then write the character to the data port (`0x81`). In the emulator, this worked perfectly. In Verilog simulation, characters were vanishing.

24.5.2 The Root Cause

The first version of the testbench modeled the UART with a single line:

Listing 24.27: First (broken) UART model

```
initial begin
    tx_ready = 1;  // Always ready
end
```

The problem: this makes the testbench's UART accept bytes instantly. The CPU sends a character, sees `tx_ready` still high on the very next poll, and immediately sends the next character. But the SoC's UART module is not stateless—it has internal state that tracks whether a transmission is in progress. When the testbench kept `tx_ready` asserted while the CPU was still in the middle of an I/O write cycle, the UART module's handshake logic became confused.

More subtly, the `tx_ready` signal fed back through the UART module to the I/O status register at `0x80`. With `tx_ready` always asserted, the status register always reported "ready," causing the CPU's busy-wait loop to exit immediately every time. On real hardware, the UART would deassert ready during transmission, forcing the CPU to wait. The testbench was bypassing a timing constraint that the real system depended on.

24.5.3 The Fix

The fix was to model realistic UART transmission timing. The corrected testbench introduces a delay counter that deasserts `tx_ready` for a configurable number of cycles after each byte is accepted:

Listing 24.28: Corrected UART model with transmission delay

```
// At 12.5 MHz FPGA clock, 115200 baud:
// 10 bits per byte * ~108 cycles per bit = ~1080 cycles
parameter TX_BYTE_CYCLES = 108 * 10;

reg [15:0] tx_delay_cnt;

initial begin
    tx_ready = 1;
    tx_delay_cnt = 0;
end
```

```
    always @(posedge clk) begin
        if (tx_valid && tx_ready) begin
            // Accept byte and start "transmission"
            uart_buffer[uart_idx] = tx_data;
            uart_idx = uart_idx + 1;

            // Display character immediately
            if (tx_data >= 8'h20 && tx_data < 8'h7F)
                $write("%c", tx_data);
            else if (tx_data == 8'h0A)
                $write("\n");

            $fflush();

            // Simulate transmission delay
            tx_ready <= 0;
            tx_delay_cnt <= TX_BYTE_CYCLES;
        end else if (tx_delay_cnt > 0) begin
            tx_delay_cnt <= tx_delay_cnt - 1;
            if (tx_delay_cnt == 1)
                tx_ready <= 1;
        end
    end
```

The key change is the state machine in the `always` block. When the UART accepts a byte (`tx_valid && tx_ready`), it immediately deasserts `tx_ready` and starts counting down from `TX_BYTE_CYCLES`. Only when the counter reaches 1 does `tx_ready` go high again, mimicking the time a real UART spends shifting bits out on the serial line.

The delay is calibrated to match the FPGA clock rate. At 12.5 MHz with 115200 baud, each bit takes approximately 108 clock cycles, and each byte (start bit + 8 data bits + stop bit = 10 bits) takes about 1080 cycles. The simulation clock runs at 100 MHz, but the delay constant matches the FPGA rate to exercise the same busy-wait iteration count that the CPU would encounter on real hardware.

Match Your Testbench to Your Target

The purpose of simulation is to predict hardware behavior. If your testbench models an interface as infinitely fast when the real hardware has finite latency, your simulation will pass tests that fail on hardware. Model the timing of external interfaces—even if approximately—to catch timing-dependent bugs before they reach the FPGA.

24.5.4 Capturing UART Output

The corrected UART model also captures output for display and analysis. The testbench maintains a 4096-byte buffer and an index:

Listing 24.29: UART output capture buffer

```
reg [7:0]  uart_buffer [0:4095];
integer    uart_idx;
```

Each accepted byte is stored in the buffer and simultaneously displayed using `$write`. The `$fflush()` call after each character ensures that output appears in real time during simulation, rather than being buffered until the simulation ends. This is important for long-running tests where you want to see progress as it happens.

The character display logic handles three cases: printable ASCII characters (`0x20`–`0x7E`) are printed as-is, newlines (`0x0A`) trigger a newline and invoke the `check_line` task for PASS/FAIL detection, carriage returns (`0x0D`) are silently ignored, and all other bytes are printed in hex notation (`[XX]`) to make non-printable characters visible.

24.5.5 The Broader Lesson

The `tx_ready` bug illustrates a general principle of testbench design: *the testbench is part of the system under test*. When the testbench models an external interface incorrectly, it does not just change the test—it changes the behavior of the DUT in ways that make the test results unreliable.

Common examples of testbench infidelity include:

- **Always-ready peripherals**: As we saw, a UART that accepts bytes instantly hides busy-wait bugs.
- **Zero-latency memory**: A memory that responds in the same cycle as the request hides pipeline stalls and hazards.
- **Ideal clocks**: A clock with no jitter or skew hides timing margin issues.
- **Missing bus contention**: A testbench that never drives conflicting values hides tri-state or arbitration bugs.

The SAMPO SoC's memory already models one-cycle latency correctly—the `mem_ready` signal is the `mem_valid` signal delayed by one clock cycle through a flip-flop in `rtl/soc.v`:

Listing 24.30: Memory ready generation in the SoC

```
reg mem_valid_d;
always @(posedge clk or negedge rst_n) begin
```

```
        if (!rst_n)
            mem_valid_d <= 1'b0;
        else
            mem_valid_d <= cpu_mem_valid;
    end
    assign cpu_mem_ready = mem_valid_d;
```

This means every memory access takes at least two clock cycles (request on cycle N, data available on cycle $N + 1$), matching the behavior of the synchronous RAM on the FPGA. The CPU's FSM was designed around this latency from the start, which is why the FETCH state waits for `mem_ready` before latching the instruction.

Testbench Bugs Are System Bugs

A testbench that models external interfaces incorrectly will either hide real bugs or create phantom bugs that do not exist in hardware. Both outcomes undermine confidence in the verification process. Treat testbench code with the same rigor as RTL code—review it, test it, and fix it when it lies.

Exercises

1. **Add a Shifter Testbench.** The shifter module (`rtl/shifter.v`) has no dedicated testbench. Write `tb/shifter_tb.v` that tests each shift function (`SLL1`, `SRL1`, `SRA1`, `ROL1`, `ROR1`, **`SWAP`**, and the 4-bit and 8-bit variants). Use the `check` task pattern from the ALU testbench. Verify that arithmetic right shift preserves the sign bit and that rotate functions preserve all bits.

2. **Selective VCD Recording.** Modify the `$dumpvars` call in `sampo_tb.v` to record only the CPU core's signals (not the RAM contents). Compare the resulting VCD file size against the full dump for a 10,000-cycle simulation. What percentage reduction do you observe?

3. **Variable UART Delay.** Modify the UART model in `sampo_tb.v` to accept a `TX_BYTE_CYCLES` parameter. Run the same test program with values of 1, 100, 1080, and 5000. At what delay value does the test first start passing correctly? What happens at a delay of 1 cycle?

4. **Extend the Debug Trace.** The debug trace in `sampo_tb.v` prints only the first 100 cycles. Modify it to print a trace line only when the state machine enters EXECUTE (state 4), and include the decoded opcode. Run a small test program and verify that every executed instruction appears in the trace.

5. **Error Injection.** Introduce a deliberate bug into the ALU (for example, make **`SUB`** compute $A + B$ instead of $A - B$) and run the ALU testbench. Verify that the testbench catches the

error and reports the correct number of failures. Then fix the bug and confirm all tests pass again.

What's Next With the testbench infrastructure in place, we need something to test. In the next chapter, we build the ISA test suite: ten assembly programs that systematically exercise every instruction in the SAMPO instruction set, using UART output and the self-checking testbench to automatically verify 132 individual test cases.

Chapter 25

The ISA Test Suite

Building a CPU is an act of optimism: you design the state machine, wire up the ALU, connect the register file, and hope that when you load a program, the right things happen. The ISA test suite is where optimism meets reality. It is a collection of ten assembly programs that systematically exercise every instruction in the SAMPO instruction set, producing 132 individual PASS/FAIL verdicts via the UART. When all 132 tests pass, you can say with confidence that the CPU executes its instruction set correctly. When one fails, you know exactly which instruction, which operand combination, and which expected-versus-actual values to investigate.

This chapter walks through the test strategy, the framework code shared by all tests, the UART reporting mechanism, and two test programs in detail: `test_alu.s`, which validates the arithmetic and logic core, and `test_loadstore.s`, which ultimately led to the discovery of the MEMORY→FETCH pipeline hazard.

25.1 Test Strategy

The strategy is straightforward: one test program per instruction category, where each program exercises every variant and boundary condition for its category. The ten test programs map directly to the SAMPO instruction set's natural groupings:

Each test follows the same pattern:

1. **Set up operands.** Load known values into registers using **`LIX`**.
2. **Execute the instruction under test.** Perform the operation.
3. **Compare the result.** Move the result into `R4` and the expected value into `R5`, then call the `check_eq` framework routine.

Table 25.1: ISA test suite programs

File	Instructions Tested	Tests
`test_alu.s`	**ADD**, **SUB**, **AND**, **OR**, **XOR**, **NEG**, flags	20
`test_addi.s`	**ADDI** with sign extension, boundaries	10
`test_shift.s`	**SLL**, **SRL**, **SRA**, **ROL**, **ROR**, **SWAP**, multi-bit shifts	17
`test_muldiv.s`	**MUL**, **MULH**, **DIV**, **DIVU**, **REM**, **REMU**	12
`test_loadstore.s`	**LW**, **LB**, **LBU**, **SW**, **SB**, **LIX**, offsets	11
`test_branch.s`	All 16 branch conditions, taken and not-taken	24
`test_jump.s`	**J**, **JR**, **JALR**, **JX**, **JALX**	6
`test_stack.s`	**PUSH**, **POP**, **CMP**, **TEST**, **MOV**	12
`test_misc.s`	**EXX**, **GETF**, **SETF**, **SCF**, **CCF**, **NOP**	9
`test_extended.s`	**ADDIX**, **SUBIX**, **ANDIX**, **ORIX**, **XORIX**, **SLLX**, **SRLX**, **SRAX**	11
	Total	**132**

4. **Report.** The framework prints "PASS" or "FAIL" via the UART. The testbench detects these strings and tallies the results.

This strategy has several deliberate properties:

- **Each program is self-contained.** It includes its own copy of the test framework, its own string data, and its own test cases. No external dependencies, no shared libraries.
- **Programs are independent.** A failure in `test_shift.s` does not affect `test_alu.s`. You can run any subset of the tests.
- **Every test uses the UART.** This means the tests exercise I/O instructions alongside the instructions under test. If the UART is broken, all tests fail—which is exactly the behavior you want, because a CPU that cannot communicate its results is not useful.
- **Tests run on both simulation and real hardware.** The same assembly programs work in the Icarus Verilog testbench and on the ULX3S FPGA. The only difference is how the output is captured.

Why Not a Single Monolithic Test?

A single test program that exercises all 132 cases would be more compact, but it has a fatal flaw: if an early instruction is broken (say, **ADD**), every subsequent test that uses **ADD** in its setup code will also fail, producing a cascade of unrelated failures. Separate programs isolate failures and make debugging tractable.

25.2 Test Framework Helpers

Every test program includes the same framework code: two routines that handle string output and result checking. These are copied into each test file rather than linked from a shared library, because the SAMPO assembler produces standalone binaries with no linker.

25.2.1 The `print_str` Routine

The `print_str` routine sends a null-terminated string to the UART, character by character:

Listing 25.1: The print_str framework routine

```
print_str:
        PUSH R12
        MOV R12, R4
ps_loop:
        LBU R5, (R12)
        CMP R5, R0
        BEQ ps_done
ps_wait:
        INI R6, ACIA_STATUS
        AND R7, R6, R6
        ADDI R7, -2
        BNE ps_wait
        OUTI ACIA_DATA, R5
        ADDI R12, 1
        J ps_loop
ps_done:
        POP R12
        JR RA
```

The routine takes the string address in `R4`. It saves `R12` (which is outside the **`EXX`** swap range and thus safe to use as a loop variable), copies the address, and enters a byte-at-a-time loop.

For each character, it:

1. Loads the byte at the current address with **`LBU`** (load byte unsigned).
2. Compares against `R0` (zero) to check for the null terminator.
3. Busy-waits on the UART status port (`0x80`) until the transmitter is ready.
4. Writes the character to the UART data port (`0x81`) using **`OUTI`**.
5. Increments the pointer and loops.

The UART Busy-Wait

The busy-wait sequence (`INI R6, ACIA_STATUS` / `AND R7, R6, R6` / `ADDI R7, -2` / `BNE ps_wait`) is checking whether the UART status register equals `0x02` (transmitter ready). The `AND R7, R6, R6` copies the status value into R7, then `ADDI R7, -2` subtracts 2. If the result is non-zero, the transmitter is busy and we loop. This is the pattern that motivated the `tx_ready` testbench fix discussed in Section 24.5.

25.2.2 The check_eq Routine

The `check_eq` routine is the core of the self-checking mechanism. It compares R4 (actual result) with R5 (expected result) and prints "PASS" or "FAIL":

Listing 25.2: The check_eq framework routine

```
check_eq:
        PUSH RA
        CMP R4, R5
        BNE ce_fail
        LIX R4, str_pass
        JALX print_str
        POP RA
        JR RA
ce_fail:
        LIX R4, str_fail
        JALX print_str
        POP RA
        JR RA

str_pass:
        .asciz "PASS\n"
str_fail:
        .asciz "FAIL\n"
```

The routine saves the return address (because it calls `print_str` internally via **JALX**), performs the comparison with **CMP**, and branches on inequality. If equal, it prints "PASS\n"; if not, "FAIL\n". The newline is important—it triggers the testbench's line scanner (the `check_line` task from Section 24.2.4) to tally the result.

The calling convention is:

- R4: actual value
- R5: expected value

- Returns to caller after printing the verdict

Every test case follows the same three-step pattern before calling `check_eq`:

1. Print the test label (e.g., "ADD basic: ").
2. Execute the instruction under test, leaving the result in some register.
3. Move the result to `R4` and the expected value to `R5`.

Reusable Test Pattern

When adding a new test case to any test program, follow this template:

```
        LIX R4, s_my_test      ; Print label
        JALX print_str
        ; ... set up operands and execute instruction ...
        MOV R4, Rresult         ; Actual result
        LIX R5, expected_value  ; Expected result
        JALX check_eq
```

Every test in the suite follows this exact structure, making the code easy to read and extend.

25.3 UART-Based Test Reporting

The UART is the SAMPO's only communication channel. It is the sole mechanism by which a running program can report results to the outside world, whether that world is a Verilog testbench or a serial terminal connected to the FPGA.

25.3.1 The Reporting Protocol

Each test produces one line of output in the format:

```
test_label PASS
```

or:

```
test_label FAIL
```

The test label is a human-readable string that identifies the test case (e.g., "ADD basic:"). The verdict ("PASS" or "FAIL") is always the last word on the line, followed by a newline character.

This format is designed for dual consumption:

- **Human-readable**: An engineer watching the serial terminal sees each test result scroll by with a descriptive label.
- **Machine-parseable**: The testbench's `check_line` task scans for the four-character substrings "PASS" and "FAIL" and tallies them. No complex parsing is needed.

25.3.2 The Full Output Flow

Each test program produces a header, a series of test results, and a footer:

Listing 25.3: Typical test program output (test_alu.s)

```
=== ALU Tests ===
ADD basic: PASS
ADD zero: PASS
ADD carry out: PASS
ADD overflow: PASS
ADD to R0: PASS
SUB basic: PASS
SUB to zero: PASS
SUB underflow: PASS
AND mask: PASS
AND zero: PASS
OR basic: PASS
OR all bits: PASS
XOR toggle: PASS
XOR self zero: PASS
NEG basic: PASS
NEG zero: PASS
ADD C flag: PASS
ADD V flag: PASS
SUB C flag: PASS
AND clr C/V: PASS
Done.
```

The header ("=== ALU Tests ===") identifies which test program is running. The "Done." footer confirms that the program reached its **HALT** instruction normally, rather than hanging in an infinite loop or crashing into uninitialized memory.

25.3.3 Testbench Integration

On the testbench side (in `sampo_tb.v`), the UART model captures each byte, builds lines in a buffer, and calls `check_line` at each newline. At the end of simulation, the testbench prints a summary and returns a non-zero exit code if any FAILs were detected:

Listing 25.4: Testbench summary output

```
=== Sampo CPU Testbench ===
RAM init file: ../programs/test_alu.hex

CPU started at PC=0x0100
UART output:
----------------------------------------
=== ALU Tests ===
ADD basic: PASS
...
AND clr C/V: PASS
Done.
----------------------------------------

Simulation complete:
  Final PC:    0x0340
  Cycles:      328471
  UART chars:  347
  Status:      HALTED

Test Results:
  PASSED: 20
  FAILED: 0
  All tests passed!
```

This integration means that running `make isa-test_alu` produces a single pass/fail result suitable for automation. No human needs to read the output unless a failure occurs.

25.4 Test Coverage Table

Table 25.2 breaks down what each test program verifies. The test counts reflect the number of `check_eq` or equivalent pass/fail calls in each program.

Table 25.2: Detailed test coverage by program

Program	Test Cases	Count
`test_alu.s`	ADD: basic, zero, carry out, signed overflow, write to R0 (5). SUB: basic, to zero, underflow (3). AND: mask, with zero (2). OR: basic, all bits (2). XOR: toggle, self-zero (2). NEG: basic, zero (2). Flags: ADD carry, ADD overflow, SUB carry (ARM-style), AND clears C/V (4).	20
`test_addi.s`	Positive immediate (+5), negative immediate (−3), result zero, max positive (+127), max negative (−128), carry from `0xFFFF`, signed overflow from `0x7FFF`, add to large value, negative-to-negative, preserves upper byte.	10
`test_shift.s`	SLL by 1, SRL by 1, SRA positive, SRA negative (sign preservation), ROL by 1, ROR by 1, SWAP bytes, SLL carry out, SRL carry out. Multi-bit: SLL4, SRL4, SRA4, SLL8, SRL8, SRA8, ROL4, ROL8.	17
`test_muldiv.s`	MUL: basic (3×4), by zero, large (100×200), overflow (low word), by 1 (identity). MULH: high word. DIV: basic (12÷4), by zero. DIVU: unsigned. REM: basic (13%4), by zero (returns dividend). REMU: unsigned.	12
`test_loadstore.s`	SW/LW round-trip (two values), SB/LBU round-trip, LB sign extension (`0x80`→`0xFF80`), LBU zero extension (`0xFF`→`0x00FF`), LW with positive offset (+2, +4), LW with negative offset (−2), LIX (non-zero, zero), multi-word store/load.	11
`test_branch.s`	All 12 unique branch conditions tested for both taken and not-taken: BEQ, BNE, BLT, BGE, BLTU, BGEU, BMI, BPL, BGT, BLE, BHI, BLS.	24
`test_jump.s`	J forward, JR via register, JALR (call and return), JX absolute, JALX (call and return), JALX return address verification.	6

continued on next page

Program	Test Cases	Count
`test_stack.s`	PUSH/POP single register, PUSH/POP LIFO order (two pushes, two pops), CMP equal (Z flag), CMP less than, CMP greater than, TEST zero result, TEST non-zero, TEST clears C/V, MOV basic, MOV from R0, SP restoration after PUSH/POP pair.	12
`test_misc.s`	SETF/GETF round-trip (two values), SCF (set carry), CCF toggle ($1 \rightarrow 0$, $0 \rightarrow 1$), NOP (execution continues), EXX swap and restore, EXX alternate bank preserved, EXX preserves R13.	9
`test_extended.s`	ADDIX, SUBIX, ANDIX, ORIX, XORIX (5 ALU-immediate ops). SLLX, SRLX, SRAX (3 extended shifts). SLLX by 0 (identity), ADDIX by 0 (identity), ADDIX carry.	11

25.5 Writing test_alu.s

The ALU test program is the natural starting point for the test suite. If the ALU does not work, nothing else will. It exercises every ALU operation defined in `sampo_pkg.vh` and verifies both results and flag behavior.

25.5.1 Program Structure

The program starts at the reset vector (`0x0100`) with a jump to the test body, followed by the framework routines:

Listing 25.5: test_alu.s program entry

```
        .org 0x0100

.equ ACIA_STATUS 0x80
.equ ACIA_DATA   0x81

        J test_main

; ---- Test Framework ----
; ... print_str and check_eq routines ...

; ---- Tests ----
test_main:
```

```
        LIX R2, 0xFFF0
```

The first instruction in `test_main` initializes the stack pointer (`R2`) to `0xFFF0`, near the top of the 64 KB address space. This is critical: the framework routines use **`PUSH`** and **`POP`**, and without a valid stack pointer, the first **`PUSH`** would write to address 0 (the default value of an uninitialized register), potentially corrupting the program's own code in memory.

25.5.2 Arithmetic Tests

The ADD tests cover four categories: basic operation, zero operands, carry generation, and signed overflow:

Listing 25.6: ADD tests in test_alu.s

```
        ; ADD basic: 0x1000 + 0x2000 = 0x3000
        LIX R4, s_add_basic
        JALX print_str
        LIX R8, 0x1000
        LIX R9, 0x2000
        ADD R10, R8, R9
        MOV R4, R10
        LIX R5, 0x3000
        JALX check_eq

        ; ADD carry out: 0xFFFF + 1 = 0x0000
        LIX R4, s_add_carry
        JALX print_str
        LIX R8, 0xFFFF
        LIX R9, 0x0001
        ADD R10, R8, R9
        MOV R4, R10
        MOV R5, R0
        JALX check_eq
```

The carry-out test verifies that `0xFFFF + 0x0001` wraps around to `0x0000`. This is a 16-bit arithmetic test—the result must be zero, not 0x10000.

An important edge case is writing to `R0`:

Listing 25.7: Testing ADD to the zero register

```
        ; ADD to R0: result discarded
        LIX R4, s_add_r0
        JALX print_str
```

```
        LIX R8, 0x1234
        LIX R9, 0x5678
        ADD R0, R8, R9
        MOV R4, R0
        MOV R5, R0
        JALX check_eq
```

This verifies the hardwired zero register: even though we compute `0x1234 + 0x5678 = 0x68AC` and write it to `R0`, the register should still read as zero. If the register file incorrectly allows writes to `R0`, this test catches it.

25.5.3 Flag Tests

The last four tests in `test_alu.s` verify flag behavior, which is where most ALU bugs hide. These tests use **GETF** to read the flags register and then mask the relevant bits:

Listing 25.8: Testing the carry flag after ADD

```
        ; ADD carry flag: 0xFFFF + 1 sets C
        LIX R4, s_flag_add_c
        JALX print_str
        LIX R8, 0xFFFF
        LIX R9, 0x0001
        ADD R10, R8, R9
        GETF R4
        ; Expect: N=0, Z=1, C=1, V=0 -> 0110 = 0x60
        LIX R8, 0x00F0
        AND R4, R4, R8
        LIX R5, 0x0060
        JALX check_eq
```

The **GETF** instruction reads the 8-bit flags register into a general-purpose register. The flags layout is: N(bit 7), Z(bit 6), C(bit 5), V(bit 4), H(bit 3), I(bit 2). To check only the NZCV bits, the test masks with `0x00F0` (extracting bits 7–4) and compares against the expected value `0x0060` (Z=1, C=1, all others 0).

The SUB carry test is particularly important because the SAMPO uses ARM-style carry semantics:

Listing 25.9: Testing ARM-style SUB carry behavior

```
        ; SUB no-borrow flag: 5 - 3 sets C=1 (no borrow)
        LIX R4, s_flag_sub_c
        JALX print_str
        LIX R8, 0x0005
```

```
        LIX R9, 0x0003
        SUB R10, R8, R9
        GETF R4
        ; Expect: C=1 (no borrow) -> C bit = 0x20
        LIX R8, 0x0020
        AND R4, R4, R8
        LIX R5, 0x0020
        JALX check_eq
```

ARM-Style Subtract Carry

In the SAMPO, **SUB** sets C=1 when there is *no borrow* (i.e., $A \geq B$ unsigned). This is the ARM/Cortex convention, opposite to x86/Z80 where the carry flag indicates a borrow. Getting this wrong breaks unsigned comparisons (**BLTU**, **BGEU**) and multi-precision arithmetic. The test suite explicitly verifies this behavior.

25.5.4 String Data

Each test case requires a label string. These are collected at the end of the program:

Listing 25.10: String data for test labels

```
str_header:
        .asciz "=== ALU Tests ===\n"
str_done:
        .asciz "Done.\n"
s_add_basic:
        .asciz "ADD basic: "
s_add_zero:
        .asciz "ADD zero: "
s_add_carry:
        .asciz "ADD carry out: "
; ... and so on for each test ...
```

The `.asciz` directive emits a null-terminated ASCII string. The trailing space after each label ("ADD basic: ") provides visual separation between the label and the PASS/FAIL verdict in the output.

25.6 Writing test_loadstore.s: Finding the Pipeline Hazard

The load/store test program is significant not just for what it tests, but for what it found. During development, `test_loadstore.s` was the test that exposed the MEMORY→FETCH pipeline

hazard—a bug where the CPU would latch stale memory data when transitioning directly from the MEMORY state to the FETCH state.

25.6.1 Basic Store/Load Round-Trip

The fundamental test is a store-then-load round-trip: write a known value to memory, read it back, and verify the result:

Listing 25.11: Store/load round-trip test

```
test_main:
        LIX R2, 0xFFF0

        ; Use 0x8000 as test data area
        LIX R13, 0x8000

        ; SW/LW round-trip: store 0x1234 then load it back
        LIX R4, s_sw_lw
        JALX print_str
        LIX R9, 0x1234
        SW (R13), R9
        LW R4, (R13)
        LIX R5, 0x1234
        JALX check_eq
```

The test uses address `0x8000` as its data area—halfway through the 64 KB address space, well clear of the program code starting at `0x0100`. `R13` (outside the **EXX** swap range) holds the base address throughout the test.

The sequence **SW** followed by **LW** to the same address is the most basic memory test, but it exercises the full memory pipeline: the EXECUTE state computes the effective address, the MEMORY state performs the write (or read), and the WRITEBACK state stores the loaded value into the destination register.

25.6.2 Byte Access and Sign Extension

Byte loads exercise the sign extension and zero extension logic in the CPU's MEMORY state handler:

Listing 25.12: Byte access tests in test_loadstore.s

```
        ; LB sign extension: store 0x80, LB gives 0xFF80
        LIX R4, s_lb_sign
        JALX print_str
        SW (R13), R0
        LIX R9, 0x0080
        SB (R13), R9
```

```
        LB R4, (R13)
        LIX R5, 0xFF80
        JALX check_eq

        ; LBU zero extension: store 0xFF, LBU gives 0x00FF
        LIX R4, s_lbu_zero
        JALX print_str
        SW (R13), R0
        LIX R9, 0x00FF
        SB (R13), R9
        LBU R4, (R13)
        LIX R5, 0x00FF
        JALX check_eq
```

The critical test here is the **LB** (load byte, signed) of the value `0x80`. Bit 7 is set, so sign extension should fill the upper byte with ones, producing `0xFF80`. Meanwhile, **LBU** (load byte, unsigned) of the value `0xFF` should produce `0x00FF`—the upper byte is zero regardless of the loaded byte's sign bit.

In the CPU implementation (`rtl/cpu.v`), this logic lives in the MEMORY state:

Listing 25.13: Byte load sign/zero extension in the CPU

```
`ST_MEMORY: begin
    if (mem_ready && dec_mem_load) begin
        if (dec_mem_byte) begin
            if (mem_addr[0]) begin
                // Odd address: high byte
                mem_result <= dec_mem_signed ?
                    {{8{mem_rdata[15]}}, mem_rdata[15:8]} :
                    {8'h00, mem_rdata[15:8]};
            end else begin
                // Even address: low byte
                mem_result <= dec_mem_signed ?
                    {{8{mem_rdata[7]}}, mem_rdata[7:0]} :
                    {8'h00, mem_rdata[7:0]};
            end
        end else begin
            mem_result <= mem_rdata;
        end
    end
end
```

The address's LSB (`mem_addr[0]`) determines which byte to extract from the word-sized memory read. The `dec_mem_signed` signal selects between sign extension (replicate bit 7) and zero extension (fill with zeros). These tests catch errors in both the byte selection and the extension logic.

25.6.3 Offset Addressing

The offset addressing tests verify that the CPU correctly computes effective addresses with positive and negative displacements:

Listing 25.14: Load with positive and negative offsets

```
        ; LW offset +2: store at addr+2, load with 2(addr)
        LIX R4, s_lw_p2
        JALX print_str
        LIX R8, 0x8010
        LIX R9, 0x5678
        LIX R10, 0x8012
        SW (R10), R9
        LW R4, 2(R8)
        LIX R5, 0x5678
        JALX check_eq

        ; LW offset -2: load from addr-2
        LIX R4, s_lw_m2
        JALX print_str
        LIX R8, 0x8022
        LIX R9, 0xDEF0
        LIX R10, 0x8020
        SW (R10), R9
        LW R4, -2(R8)
        LIX R5, 0xDEF0
        JALX check_eq
```

The positive offset test stores `0x5678` at address `0x8012`, then loads it using base address `0x8010` with offset +2. The negative offset test stores at `0x8020` and loads via `0x8022` with offset −2. Both must compute the same effective address. The CPU computes this in the MEMORY state as `rs1_val + dec_mem_offset`.

25.6.4 The Hazard That Hid Here

During development, the load/store tests initially failed intermittently. A **`SW`** followed by a **`LW`** would sometimes return the wrong value—not a random value, but the *previous* instruction's data. The pattern was maddening: the test would pass on some programs but fail on others, depending on instruction ordering.

The root cause was a state machine transition bug. The original CPU design allowed the MEMORY state to transition directly to FETCH for store instructions (since stores do not need a writeback). But `mem_ready` is generated by delaying `mem_valid` through a flip-flop. When the CPU went from MEMORY directly to FETCH, the new FETCH state asserted `mem_valid` for the instruction fetch. One cycle later, `mem_ready` went high—but it was responding to the fetch request, not the previous store. The CPU then latched the instruction word as if it were memory data from the store.

This hazard was subtle because it only manifested when a store was followed by an instruction that the CPU needed to fetch. We will examine the waveform evidence and the fix in detail in Chapter 26. The fix was to always route through the WRITEBACK state after MEMORY, even for stores, giving `mem_ready` time to deassert before the next fetch.

Tests Find Bugs

The load/store test program did not just verify that loads and stores work—it verified the *interaction* between memory operations and instruction fetches. This kind of emergent bug cannot be found by testing instructions in isolation. It only appears when one instruction's memory access affects the next instruction's fetch. This is why we test with real programs, not just unit tests.

25.7 Makefile Automation

Running ten test programs manually would be tedious and error-prone. The `verilog/Makefile` automates the entire process, from assembly through simulation to result reporting.

25.7.1 The Test List

All ten tests are listed in a single Make variable:

Listing 25.15: ISA test list in the Makefile

```
ISA_TESTS = test_alu test_addi test_shift test_muldiv \
            test_loadstore test_branch test_jump \
            test_stack test_misc test_extended
```

25.7.2 The Assembly Pipeline

Each test goes through a three-stage pipeline:

1. **Assemble**: The `.s` source file is assembled to a `.bin` binary by the SAMPO assembler (written in Rust).

2. **Convert**: The binary is converted to Verilog hex format using `bin2hex.py`, producing a `.hex` file that can be loaded by `$readmemh`.

3. **Compile**: A parameterized testbench is compiled with Icarus Verilog, with the hex file path baked into the `RAM_INIT_FILE` parameter.

The Makefile expresses this as a chain of pattern rules:

Listing 25.16: Assembly pipeline in the Makefile

```
# Assemble test source to binary
../examples/tests/%.bin: ../examples/tests/%.s
  cd ../sasm && cargo run --release -- $< -o $@

# Convert test binary to hex
programs/%.hex: ../examples/tests/%.bin
  python3 bin2hex.py $< $@

# Compile testbench for a specific test program
build/sampo_tb_%: $(RTL_SRCS) tb/sampo_tb.v programs/%.hex
  iverilog -Irtl \
      -Psampo_tb.RAM_INIT_FILE=\"../programs/$*.hex\" \
      -o $@ \
      rtl/alu.v rtl/shifter.v rtl/regfile.v \
      rtl/decode.v rtl/cpu.v rtl/ram.v \
      rtl/uart.v rtl/soc.v tb/sampo_tb.v
```

Make's pattern matching (`%`) means these rules automatically apply to any test name. When you run `make isa-test_muldiv`, Make traces the dependency chain backward: the compiled testbench needs the hex file, which needs the binary, which needs the source. Each step is executed only if its output is older than its inputs.

25.7.3 Running Individual Tests

The Makefile generates a phony target for each test:

Listing 25.17: Individual test targets

```
$(addprefix isa-,$(ISA_TESTS)): isa-%: build/sampo_tb_%
  @echo "=== Running $* ==="
  @cd build && ./sampo_tb_$*
```

The `addprefix` function generates targets `isa-test_alu`, `isa-test_addi`, etc. Each target depends on the corresponding compiled testbench, which in turn triggers the full assembly pipeline. Running a test is a single command:

Listing 25.18: Running a single ISA test

```
make isa-test_alu
```

25.7.4 Running All Tests

The `isa-tests` target runs all ten tests in sequence:

Listing 25.19: Running the complete ISA test suite

```
make isa-tests
```

This iterates through the `ISA_TESTS` list, running each test as a dependency. If any test returns a non-zero exit code (indicating failures), Make stops and reports the error. This behavior is desirable during development—when a test fails, you want to fix it before moving on.

25.7.5 The Precious Directive

The Makefile includes a `.PRECIOUS` directive that prevents Make from deleting intermediate files:

Listing 25.20: Preserving intermediate files

```
.PRECIOUS: programs/%.hex ../examples/tests/%.bin
```

Without this, Make would delete the `.hex` and `.bin` files after the testbench is compiled, since they are intermediate targets. But we want to keep them around for debugging: when a test fails, examining the hex file (and disassembling the binary) is often the fastest path to understanding the failure.

25.7.6 Adding a New Test

To add a new test program to the suite:

1. Write the assembly file in `examples/tests/`, following the framework pattern.
2. Add the test name (without the `.s` extension) to the `ISA_TESTS` list in the Makefile.
3. Run `make isa-<name>` to verify it passes.
4. Run `make isa-tests` to confirm all existing tests still pass.

No Makefile rules need to be added—the pattern rules handle everything automatically.

Watch Mode for Test Development

When developing a new test, use this pattern for rapid iteration:

```
# Edit the test source, then:
make isa-test_mytest
# ... read the output ...
# Fix the test or the CPU, repeat.
```

Make will only reassemble the changed source file and recompile the testbench. The turnaround time is typically under five seconds.

Exercises

1. **Count Your Tests.** Run all ten ISA tests with `make isa-tests` and verify that the total PASS count across all programs equals 132. If it does not, identify which program has an unexpected count and investigate.
2. **Write a NOT Test.** The **NOT** instruction is exercised in the ALU testbench (`alu_tb.v`) but not in the ISA test suite. Add three test cases to `test_alu.s`: NOT of zero (`0x0000` → `0xFFFF`), NOT of all ones (`0xFFFF` → `0x0000`), and NOT of a mixed pattern (`0xAAAA` → `0x5555`). Run the test and verify all three pass.
3. **Write a Block Transfer Test.** Write a new test program `test_block.s` that stores ten consecutive words at address `0x8000`, then uses **LW** with incrementing offsets to read each word back and verify it. This tests the interaction between offset addressing and sequential memory access.
4. **Break and Detect.** Introduce a deliberate bug in `rtl/alu.v`: make **XOR** compute **OR** instead. Run `make isa-test_alu` and verify that exactly the right tests fail. Then fix the bug and confirm all tests pass again.
5. **Measure Simulation Time.** Record the wall-clock time for running each of the ten ISA tests. Which test takes the longest? Why? (Hint: look at the UART character count—each character requires a busy-wait that burns many simulation cycles.)

What's Next The test suite tells us *that* something is broken. The next chapter teaches us how to find out *why*. We will trace through VCD waveforms, follow signals through the state machine, and dissect the MEMORY→WRITEBACK fix that the load/store tests demanded—turning a failed test into a working CPU.

Chapter 26

Debugging Hardware

Every hardware designer has a moment where the waveform viewer becomes their entire world: a grid of colored traces, a cursor blinking at the cycle where everything went wrong, and the dawning realization that the bug is not in the code you are looking at but in the code that ran three states earlier. Debugging hardware is a discipline unto itself, and the SAMPO project provided ample opportunity to practice it.

This chapter catalogs the bugs we encountered, the techniques we used to find them, and the fixes we applied. We start with common Verilog pitfalls that affect any design, then trace through the waveforms that revealed the MEMORY→FETCH pipeline hazard, and finally discuss the gap between simulation and synthesis—where a design that works in Icarus Verilog fails on the FPGA.

26.1 Common Verilog Bugs

Before we dive into the SAMPO-specific bugs, let us catalog the Verilog mistakes that plague every hardware project. These are patterns that the language permits but that produce incorrect or unexpected behavior.

26.1.1 Blocking vs. Non-Blocking Assignment

This is the single most common source of simulation-synthesis mismatches. The rule is straightforward but easy to forget:

- Use **non-blocking** (<=) for sequential logic (`always @(posedge clk)`).
- Use **blocking** (=) for combinational logic (`always @(*)`).

The SAMPO follows this convention throughout. The state machine's sequential block in `rtl/cpu.v` uses non-blocking:

Listing 26.1: Correct use of non-blocking assignment in sequential logic

```
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        state <= `ST_RESET;
    end else begin
        state <= next_state;
    end
end
```

The next-state combinational block uses blocking:

Listing 26.2: Correct use of blocking assignment in combinational logic

```
always @(*) begin
    next_state = state;

    case (state)
        `ST_RESET: begin
            next_state = `ST_FETCH;
        end
        // ...
    endcase
end
```

Mixing Assignment Types

Using blocking assignment (=) in a sequential `always` block causes the simulator to update the variable immediately within the block, before other assignments execute. Synthesis tools may interpret the same code differently, producing a circuit that behaves differently from simulation. Never mix blocking and non-blocking assignments to the same signal.

Consider what happens if the state machine were written with blocking assignment:

Listing 26.3: Broken: blocking assignment in sequential logic (DO NOT DO THIS)

```
// BROKEN - DO NOT USE
always @(posedge clk) begin
    state = next_state;  // Blocking: updates immediately
    pc = pc + 2;         // This sees the NEW state value!
end
```

With blocking assignment, the `pc = pc + 2` would see the updated value of `state`, because blocking assignment completes before the next statement executes. With non-blocking (`<=`), all assignments sample their right-hand-side values at the beginning of the clock edge and update simultaneously at the end—matching the behavior of real flip-flops.

26.1.2 Incomplete Sensitivity Lists

In Verilog-2001 and later, using `always @(*)` for combinational blocks automatically infers the sensitivity list from the signals read in the block. All of the SAMPO's combinational blocks use this form. The older alternative—manually listing every input signal—is a notorious source of bugs:

Listing 26.4: Broken: incomplete sensitivity list

```
// BROKEN - missing carry_in
always @(a or b or op) begin  // carry_in is missing!
    case (op)
        ALU_ADC: result = a + b + carry_in;
    endcase
end
```

If `carry_in` changes while `a`, `b`, and `op` remain constant, the block will not re-evaluate, and `result` will hold a stale value. The simulation will appear correct if `carry_in` always changes at the same time as other inputs, which makes this bug intermittent and maddening.

Always Use `@(*)`

There is no good reason to write a manual sensitivity list for combinational logic. The `@(*)` construct was introduced precisely to eliminate this class of bugs. All the SAMPO's `always` blocks for combinational logic use it.

26.1.3 Latches from Incomplete Case Statements

When a combinational `always` block does not assign a value to an output in every branch of a `case` or `if` statement, the synthesis tool infers a latch to hold the previous value. This is almost never intentional.

The SAMPO decoder in `rtl/decode.v` avoids this by setting all outputs to default values at the top of the `always` block:

Listing 26.5: Default values prevent latch inference in the decoder

```
always @(*) begin
    // Default all outputs
    alu_op      = `ALU_PASS_B;
```

```
    mem_load    = 1'b0;
    mem_store   = 1'b0;
    mem_byte    = 1'b0;
    mem_signed  = 1'b0;
    mem_offset  = 16'h0000;
    reg_write   = 1'b0;
    is_jump     = 1'b0;
    is_jump_reg = 1'b0;
    is_branch   = 1'b0;
    is_call     = 1'b0;
    is_ret      = 1'b0;
    is_halt     = 1'b0;
    is_nop      = 1'b0;
    is_exx      = 1'b0;
    is_push     = 1'b0;
    is_pop      = 1'b0;
    // ... many more ...

    case (opcode)
        `OP_ADD: begin
            alu_op = `ALU_ADD;
            reg_write = 1'b1;
        end
        // ...
    endcase
end
```

The 28 default assignments at the top ensure that every output has a defined value for every possible input combination. Without these defaults, any opcode not explicitly listed in the case statement would leave outputs undefined, and the synthesis tool would infer latches.

Inferred Latch

An *inferred latch* is a latch that the synthesis tool creates because a combinational output is not assigned in every branch of the logic. Unlike a flip-flop, a latch is level-sensitive (it holds its value while the enable is low), which makes it susceptible to glitches and timing violations. Inferred latches are almost always bugs. Synthesis tools will warn about them—treat these warnings as errors.

26.1.4 Uninitialized Signals

In simulation, uninitialized `reg` signals hold the value `x` (unknown). Unlike some software languages where uninitialized variables contain garbage, Verilog's `x` propagates through logic: `x + 1 = x`, `x & 0 = 0`, `x & 1 = x`. This propagation can corrupt the entire design if not caught early.

The SAMPO's reset logic initializes every register to a known value:

Listing 26.6: Complete reset initialization in the CPU

```
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        pc <= RESET_VECTOR;
        flags <= 8'h00;
        instr <= 16'h0000;
        imm16 <= 16'h0000;
        cycle_count <= 32'h0;
        rs1_val <= 16'h0000;
        rs2_val <= 16'h0000;
        exec_result <= 16'h0000;
        mem_result <= 16'h0000;
        sp_update_pending <= 1'b0;
        sp_new_val <= 16'h0000;
    end else begin
        // ...
    end
end
```

Every `reg` in the CPU is explicitly initialized during reset. This is not just good practice—it is mandatory for FPGA implementation, because FPGAs do not have a hardware reset that clears all flip-flops. The reset logic must do it explicitly.

26.1.5 Width Mismatches

Verilog silently truncates or zero-extends values when assigning between signals of different widths. This is a fertile source of bugs:

Listing 26.7: Width mismatch example

```
wire [15:0] result;
wire [7:0]  partial;
assign result = partial;  // Upper 8 bits silently zeroed
```

In the SAMPO, the ALU's carry computation explicitly uses 17-bit arithmetic to capture the carry:

Listing 26.8: Explicit widening for carry detection

```
{carry_out, alu_result} = op_a + op_b;
```

The concatenation `{carry_out, alu_result}` creates a 17-bit target, ensuring the carry bit is captured rather than silently discarded.

26.2 Waveform Debugging

When a self-checking test reports "FAIL" and the console trace is not sufficient to identify the cause, waveform analysis is the next tool. The approach is systematic: find the symptom, trace backward through the state machine, and identify where the signals first diverge from expected behavior.

26.2.1 Setting Up the Waveform View

After running a failing test (e.g., `make isa-test_loadstore`), open the VCD file:

Listing 26.9: Opening the VCD file for analysis

```
gtkwave build/sampo.vcd
```

Add the essential signals from Table 24.1 in Chapter 24. For memory-related debugging, also add:

Table 26.1: Additional signals for memory debugging

Signal	Purpose
`uut.cpu_inst.mem_valid`	Memory request active
`uut.cpu_inst.mem_ready`	Memory response ready
`uut.cpu_inst.mem_addr`	Address being accessed
`uut.cpu_inst.mem_rdata`	Data read from memory
`uut.cpu_inst.mem_wdata`	Data written to memory
`uut.cpu_inst.mem_we`	Write enable
`uut.cpu_inst.exec_result`	Execution stage result
`uut.cpu_inst.mem_result`	Memory stage result
`uut.cpu_inst.rf_wr_data`	Register file write data
`uut.cpu_inst.rf_wr_en`	Register file write enable
`uut.cpu_inst.rf_wr_addr`	Register file write address

26.2.2 The Backward Trace Method

Waveform debugging is inherently a backward process. You start at the cycle where the wrong value appears and work backward through the pipeline:

1. **Find the wrong output.** Look at the register write during WRITEBACK: is the wrong value being written to the wrong register, or is the right register getting the wrong value?

2. **Trace the data source.** The value written to the register file comes from either `exec_result` (for ALU operations) or `mem_result` (for loads). Which path was used? Was the selection correct?

3. **Check the execution.** If the value came from the ALU, check the ALU inputs (`alu_a`, `alu_b`, `alu_op`). Were the operands correct? Was the ALU operation correct?

4. **Check the decode.** If the operands were wrong, check the register file read addresses during the DECODE state. Were the correct source registers addressed?

5. **Check the fetch.** If the decode produced wrong signals, check the instruction word latched during FETCH. Was the correct instruction fetched from the correct address?

Each step moves one pipeline stage backward, from writeback toward fetch. In a design as deep as the SAMPO's eight-state FSM, the bug might be four or five states before the symptom.

Use GTKWave Markers

Press `M` in GTKWave to place a named marker at the current cursor position. Place markers at key events: the cycle where the wrong value appears, the start of the instruction's fetch, and any state transitions of interest. This lets you quickly navigate between related events even when they are thousands of cycles apart.

26.2.3 Reading the State Machine

The `state` signal (4 bits) encodes the current FSM state using the constants from `sampo_pkg.vh`:

A normal instruction follows the sequence 1→3→4→6→1 (fetch→decode→execute→writeback→fetch). Instructions that access memory add state 5 between execute and writeback: 1→3→4→5→6→1. Extended instructions insert state 2 after decode: 1→3→2→4→6→1.

When debugging, the first thing to check is whether the state sequence is correct. An unexpected state transition often points directly at the bug.

Table 26.2: State machine encoding

Value	State	Meaning
0	RESET	Initializing registers to reset values
1	FETCH	Reading instruction from memory at PC
2	FETCH_EXT	Reading second word for 32-bit instructions
3	DECODE	Latching register operands
4	EXECUTE	Performing ALU/shift operation, updating PC and flags
5	MEMORY	Performing load or store
6	WRITEBACK	Writing result to register file
7	HALTED	CPU stopped, waiting for reset

26.3 Signal Tracing Through the State Machine

Let us trace a specific instruction through the pipeline to establish what the waveforms should look like when everything works correctly. Consider the instruction `LW R4, (R13)` at address `0x0140`, where R13 contains `0x8000`.

26.3.1 Fetch (State 1)

In the FETCH state:

- `mem_addr` = `0x0140` (the current PC)
- `mem_valid` = 1
- One cycle later, `mem_ready` = 1
- `mem_rdata` contains the instruction word (let's say `0x64D0` for `LW R4, (R13)`)
- The CPU latches: `instr` ← `0x64D0`

26.3.2 Decode (State 3)

In the DECODE state:

- The decoder extracts: opcode = `0x6` (`OP_LOAD`), rd = 4, rs1 = `0xD` (13), func = 0 (`LOAD_LW`)
- `dec_mem_load` = 1, `dec_reg_write` = 1
- The register file reads R13 via `rf_rd_addr1` = 13
- At the next clock edge: `rs1_val` ← `0x8000`

26.3.3 Execute (State 4)

In the EXECUTE state:

- Since `dec_mem_load` is asserted, the next state is MEMORY
- The PC advances: PC ← `0x0142` (current PC + 2)
- No flag updates occur (loads do not affect flags)

26.3.4 Memory (State 5)

In the MEMORY state:

- `mem_addr` = `rs1_val` + `dec_mem_offset` = `0x8000` + 0 = `0x8000`
- `mem_valid` = 1, `mem_we` = 0 (read)
- One cycle later, `mem_ready` = 1
- `mem_rdata` contains the word at `0x8000`
- The CPU latches: `mem_result` ← `mem_rdata`
- Next state: WRITEBACK

26.3.5 Writeback (State 6)

In the WRITEBACK state:

- `rf_wr_en` = 1 (because `dec_reg_write` is set and rd ≠ 0)
- `rf_wr_addr` = 4 (the destination register)
- `rf_wr_data` = `mem_result` (the loaded value)
- Next state: FETCH, beginning the next instruction

When debugging a load instruction, compare each of these values in the waveform against the expected values. The first point where reality diverges from expectation is where the bug lives.

26.4 The MEMORY to WRITEBACK Fix in Detail

This is the story of the most significant bug found during SAMPO development. It required waveform analysis, an understanding of the memory timing protocol, and a careful redesign of the state machine transitions.

26.4.1 The Symptom

The load/store test program (`test_loadstore.s`) was failing intermittently. Store operations appeared to work, but loads sometimes returned the wrong value. The wrong value was not random—it was the *instruction word of the next instruction*, not the data from the memory address being loaded. This was the key clue.

26.4.2 The Original State Machine

In the original CPU design, the next-state logic for the EXECUTE state looked approximately like this:

Listing 26.10: Original (buggy) next-state logic for EXECUTE

```
// ORIGINAL (BUGGY) VERSION - DO NOT USE
`ST_EXECUTE: begin
    if (dec_is_halt) begin
        next_state = `ST_HALTED;
    end else if (dec_mem_load || dec_mem_store) begin
        next_state = `ST_MEMORY;
    end else if (dec_reg_write) begin
        next_state = `ST_WRITEBACK;
    end else begin
        next_state = `ST_FETCH;
    end
end

`ST_MEMORY: begin
    if (mem_ready) begin
        if (dec_mem_load) begin
            next_state = `ST_WRITEBACK;
        end else begin
            // Stores go directly to FETCH
            next_state = `ST_FETCH;
        end
    end
end
```

The critical line is the `else` branch in MEMORY: for store instructions, the CPU transitioned directly from MEMORY to FETCH. This seemed logical—stores do not write a register, so why go through WRITEBACK?

26.4.3 The Timing Diagram

To understand the bug, we must understand how `mem_ready` is generated. In `rtl/soc.v`:

Listing 26.11: Memory ready generation in the SoC

```
reg mem_valid_d;
always @(posedge clk or negedge rst_n) begin
    if (!rst_n)
        mem_valid_d <= 1'b0;
    else
        mem_valid_d <= cpu_mem_valid;
end
assign cpu_mem_ready = mem_valid_d;
```

The `mem_ready` signal is simply `mem_valid` delayed by one clock cycle through a flip-flop. This means:

- Cycle N: CPU asserts `mem_valid`.
- Cycle $N + 1$: `mem_ready` goes high. Data on `mem_rdata` is valid.

Now consider what happens when a **`SW`** instruction goes from MEMORY directly to FETCH:

Table 26.3: Timing of the MEMORY→FETCH hazard

Cycle	State	`mem_valid`	`mem_ready`	`mem_addr`	`mem_rdata`
N	MEMORY	1	0 (prev.)	`0x8000` (store addr)	—
$N + 1$	MEMORY	1	1	`0x8000`	data at `0x8000`
$N + 2$	FETCH	1	**1 (stale!)**	`0x0142` (next PC)	**stale data**
$N + 3$	DECODE	0	1	—	instr at `0x0142`

The problem is at cycle $N + 2$. The CPU has just entered FETCH and is asserting `mem_valid` for the instruction fetch at the new PC (`0x0142`). But `mem_ready` is *still high* from the previous cycle's store—it is the delayed version of the store's `mem_valid`, which was high in cycle $N + 1$.

The FETCH state checks `if (mem_ready)` to decide whether to latch the instruction. At cycle $N + 2$, `mem_ready` is high (from the store), so the CPU latches `mem_rdata`. But `mem_rdata` has not yet been updated with the instruction data—the RAM's synchronous read of `0x0142` will not produce valid data until cycle $N + 3$. The CPU latches whatever was on `mem_rdata` from the previous memory operation.

This is the hazard: the CPU interprets stale memory data as an instruction word.

26.4.4 The Waveform Evidence

In the VCD waveform, the hazard is visible as a suspicious pattern in the `state` trace. Instead of the normal fetch dwell time (two cycles: one to request, one to receive), the fetch completes in a single cycle because `mem_ready` is already high when the state machine enters FETCH.

The smoking gun is the `instr` signal: it latches a value that does not match the instruction word at the current PC. If you look up the latched value in the hex dump, it corresponds to either the previous memory read data or a partially updated bus value.

26.4.5 The Fix

The fix is to never go directly from MEMORY to FETCH. Instead, all instructions that pass through MEMORY—both loads *and* stores—must go through WRITEBACK first. The WRITEBACK state does not assert `mem_valid`, so it provides a "dead" cycle where `mem_ready` deasserts naturally:

Listing 26.12: Fixed next-state logic: always go through WRITEBACK

```
`ST_MEMORY: begin
    if (mem_ready) begin
        // Always go through WRITEBACK after MEMORY.
        // For loads: writes loaded data to register file.
        // For stores/PUSH: allows mem_ready to deassert
        // before next FETCH (prevents stale rdata latch),
        // and lets PUSH update the stack pointer.
        next_state = `ST_WRITEBACK;
    end
end
```

The comment in the actual `rtl/cpu.v` source documents both the functional reason (register writes for loads) and the timing reason (letting `mem_ready` deassert). This is the kind of comment that saves future developers hours of confusion.

26.4.6 Why WRITEBACK Works

With the fix, the timing looks like this:

Table 26.4: Corrected timing with WRITEBACK between MEMORY and FETCH

Cycle	State	mem_valid	mem_ready	mem_addr	Notes
N	MEMORY	1	0	0x8000	Store request
$N+1$	MEMORY	1	1	0x8000	Store completes
$N+2$	WRITEBACK	0	1	—	No mem access; ready still high
$N+3$	FETCH	1	**0**	0x0142	Fetch request; ready is now 0
$N+4$	FETCH	1	1	0x0142	Fetch completes with correct data

The key is cycle $N+3$: when FETCH begins, `mem_ready` is 0 because the previous cycle (WRITEBACK) did not assert `mem_valid`. The fetch state sees `mem_ready` = 0 and waits. One cycle later (cycle $N+4$),

the RAM responds with the correct instruction word, `mem_ready` goes high, and the CPU latches the correct instruction.

The extra WRITEBACK cycle for stores costs one clock per store instruction. In practice, this is negligible: the UART busy-wait (which is the dominant loop in test programs) burns thousands of cycles per character, so one extra cycle per store is unmeasurable.

26.4.7 The PUSH Side Effect

The WRITEBACK state also turned out to be the correct place to handle the **PUSH** instruction's stack pointer update. During **PUSH**, the MEMORY state writes the data to `SP-2`. The stack pointer itself must be decremented by 2 after the write completes. In WRITEBACK, this happens via the register file write port:

Listing 26.13: PUSH stack pointer update in WRITEBACK

```
`ST_WRITEBACK: begin
    // Handle PUSH: update SP via main port
    if (dec_is_push) begin
        rf_wr_en = 1'b1;
        rf_wr_addr = `REG_SP;
        rf_wr_data = rs1_val - 16'h2;
    end

    // Handle POP: write Rd via main port,
    // SP via dedicated SP port
    if (sp_update_pending) begin
        rf_sp_wr_en = 1'b1;
        rf_sp_wr_data = sp_new_val;
    end
end
```

For **POP**, both the popped value (written to the destination register via the main write port) and the stack pointer increment (written via the dedicated SP write port) happen simultaneously in WRITEBACK. The dedicated `sp_wr` port exists precisely to handle this case where two registers need updating in a single cycle.

Design Validation Through Bug Fixes

The MEMORY→WRITEBACK fix was not just a bug fix—it was a validation of the state machine's design. The WRITEBACK state's existence was originally motivated by register writes for load instructions. The fix revealed that it also served as a timing barrier for memory operations and as a hook for stack pointer updates. A state that originally had one purpose turned out to have three, all of which the design supported correctly once the transition logic was fixed.

26.5 Simulation vs. Synthesis Mismatches

A design that works in simulation but fails on the FPGA—or vice versa—is a special kind of frustrating. The root causes are always the same: the simulator and the synthesis tool interpret the Verilog differently, or the FPGA has timing constraints that the simulation does not model.

26.5.1 Initial Value Differences

In Icarus Verilog simulation, `reg` signals start as `x` (unknown) unless explicitly initialized. On an FPGA (specifically the Lattice ECP5 used on the ULX3S), flip-flops power up in a state defined by the bitstream, typically zero for ECP5.

This means a design that "works" on the FPGA despite having uninitialized registers may fail in simulation because `x` propagation catches the error. This is actually a benefit of simulation: it catches initialization bugs that the FPGA masks by accident.

The SAMPO handles this by initializing every register in the reset logic, making behavior identical between simulation and hardware. The register file's reset clears all 16 registers (plus the 8 alternate registers for **EXX**) to zero.

26.5.2 Memory Initialization

The RAM module uses `$readmemh` to initialize its contents:

Listing 26.14: Memory initialization in the RAM module

```
initial begin
    if (INIT_FILE != "") begin
        $readmemh(INIT_FILE, mem);
    end
end
```

The `initial` block is a simulation-only construct. On the FPGA, RAM initialization happens through a different mechanism: the synthesis tool embeds the hex file into the bitstream, which the FPGA's block RAM initialization circuitry loads at power-up.

This creates a subtle difference: in simulation, any memory location not specified in the hex file contains `x`. On the ECP5 block RAM, uninitialized memory contains zero (for ECP5 block RAM). If a program accidentally reads from an uninitialized address, simulation will propagate `x` while hardware will return zero. Both are wrong, but they are wrong in different ways.

Test with Complete Memory Images

Ensure your hex files cover the entire address range your program will access. Reads from uninitialized addresses are a common source of simulation-synthesis mismatches. The SAMPO test programs use a dedicated data area starting at `0x8000`, which is always explicitly written before being read.

26.5.3 Timing-Dependent Behavior

The most insidious simulation-synthesis mismatches come from timing dependencies that the RTL simulation does not model. In RTL simulation (like Icarus Verilog), all combinational logic resolves in zero time, and all sequential logic updates at the clock edge. On the FPGA, combinational logic has propagation delay, and if a signal path is too long, it may not settle before the next clock edge.

The SAMPO's 12.5 MHz FPGA clock provides generous timing margins (80 ns clock period). At this frequency, even the longest combinational paths in the design (the ALU's multiply, the decoder's full fanout) have time to settle. Faster clock frequencies would require timing analysis and potentially pipeline insertion—topics covered in Chapter 27.

26.6 Timing Violations and Metastability

Timing violations occur when a signal changes too close to the clock edge for the receiving flip-flop to reliably sample it. Metastability occurs when a flip-flop receives an input that violates its setup or hold time constraints, causing the output to settle to an unpredictable value (or oscillate between 0 and 1 for a brief period).

26.6.1 Setup and Hold Time

Every flip-flop has two critical timing parameters:

Setup and Hold Time

The *setup time* (t_{su}) is the minimum time before the clock edge that the data input must be stable. The *hold time* (t_h) is the minimum time after the clock edge that the data input must remain stable. Together, they define a window around the clock edge during which the data must not change. Violating either constraint results in unpredictable output behavior.

In the SAMPO, the primary timing concern is the combinational paths between flip-flops. The longest path determines the maximum clock frequency. The `nextpnr-ecp5` place-and-route tool performs static timing analysis and reports the critical path. At 12.5 MHz, the SAMPO has substantial positive slack on all paths.

26.6.2 Clock Domain Crossings

The SAMPO's FPGA implementation uses a single clock domain for the CPU core, RAM, and UART interface logic. However, the external UART signals (serial TX/RX lines) are asynchronous relative to the system clock. The FPGA top module in `fpga/ulx3s_top.v` includes synchronizer flip-flops for the UART receive line:

The synchronizer is a chain of two or three flip-flops that reduces the probability of metastability to negligible levels. Without it, the asynchronous UART RX signal could violate the setup/hold time of the receiving flip-flop, causing metastable behavior that propagates through the design.

26.6.3 Reset Synchronization

The FPGA board's reset button generates an asynchronous signal. The `fpga/reset_sync.v` module synchronizes it to the system clock domain:

Listing 26.15: Reset synchronizer structure

```
// Reset synchronizer: 2-stage synchronizer for
// asynchronous reset input
reg [1:0] sync_reg;

always @(posedge clk or negedge raw_rst_n) begin
    if (!raw_rst_n) begin
        sync_reg <= 2'b00;
    end else begin
        sync_reg <= {sync_reg[0], 1'b1};
    end
end

assign rst_n = sync_reg[1];
```

The asynchronous reset drives all flip-flops (including the synchronizer) to their reset state immediately. When reset is released, the `1'b1` value propagates through the two-stage synchronizer, producing a synchronized, glitch-free reset deassertion on the system clock domain.

Never Use an Unsynchronized Reset

An asynchronous reset that deasserts near a clock edge can cause different flip-flops in the design to come out of reset on different clock cycles. This produces an inconsistent initial state that manifests as random, unreproducible failures. Always synchronize the reset deassertion, even if the assertion is asynchronous.

26.6.4 Debugging Timing Issues

Timing violations do not appear in RTL simulation because the simulator does not model gate delays. They manifest on the FPGA as:

- **Intermittent failures**: The design works sometimes and fails other times, depending on temperature, voltage, and the specific place-and-route result.
- **Frequency-dependent behavior**: The design works at a low clock frequency but fails at a higher one.
- **Position-dependent behavior**: Changing unrelated logic (which changes the place-and-route) makes the bug appear or disappear.

The SAMPO avoids these issues through conservative clock frequency selection and clean synchronous design practices. The `--freq 25` flag passed to `nextpnr-ecp5` tells the tool to target 25 MHz timing, providing 2× margin over the actual 12.5 MHz clock. Any timing violations at 25 MHz would indicate paths that are dangerously close to failing at 12.5 MHz.

Read the Timing Report

After place and route, check the timing report generated by `nextpnr-ecp5`. It lists the critical path and the slack (how much margin you have). Negative slack means a timing violation. Even with positive slack, paths with very small slack are fragile and should be investigated.

Exercises

1. **Reproduce the Hazard.** Modify the next-state logic in `rtl/cpu.v` to restore the original (buggy) behavior: make MEMORY transition directly to FETCH for store instructions. Run `make isa-test_loadstore` and confirm that it fails. Examine the VCD waveforms around the failure point. Can you identify the cycle where `mem_ready` is stale? Restore the fix when done.

2. **Trace an Instruction.** Pick any instruction from `test_alu.s` (e.g., the SUB that tests underflow). Run the test, open the VCD, and trace that specific instruction through all pipeline stages: fetch, decode, execute, writeback. Verify that every signal matches the expected values described in Section 26.3. Record the cycle numbers for each state transition.
3. **Inject and Detect a Latch.** Remove the default assignment for `is_halt` at the top of the decoder's `always` block (in `rtl/decode.v`). Recompile with Icarus Verilog and check whether it reports a latch inference warning. What behavior change, if any, does this cause in simulation?
4. **Timing Analysis.** Run the full FPGA synthesis flow (`make synth pnr`) and examine the timing report from `nextpnr-ecp5`. What is the critical path in the design? How much slack does it have at 25 MHz? At what frequency would the design start failing?

What's Next With the CPU verified in simulation and its most significant bug understood and fixed, we are ready to leave the simulator behind and enter the physical world. The next part of the book covers FPGA implementation: how to take the Verilog design we have simulated and tested, synthesize it into a netlist, map it onto real FPGA fabric, and run the SAMPO on an actual ULX3S board.

Part VIII

FPGA Implementation

Chapter 27

FPGA Fundamentals

We have spent the previous chapters designing, simulating, and verifying the SAMPO CPU entirely in software. Icarus Verilog and GTKWave have been our laboratory — every signal visible, every state transition inspectable at leisure. But a CPU that lives only in simulation is like a ship that never touches water. In this chapter, we step into the world of Field-Programmable Gate Arrays, the devices that will turn our Verilog into real, running hardware.

If you have never worked with FPGAs before, this chapter will give you the conceptual foundation you need. If you have, it will ground those concepts in the specific device family — the Lattice ECP5 — that we target for the SAMPO build. Either way, by the end of this chapter you will understand exactly what happens when Yosys reads our Verilog: what physical structures it maps onto, why some designs fit and others do not, and how a Look-Up Table turns arbitrary Boolean logic into silicon.

27.1 FPGA Architecture Overview

An FPGA is, at its heart, a sea of configurable logic blocks connected by a programmable routing network. Unlike an ASIC, where the logic gates are etched permanently into silicon, an FPGA's function is determined by a *bitstream* — a configuration file that is loaded into the device at power-up. Change the bitstream, and the same chip becomes an entirely different circuit.

Field-Programmable Gate Array (FPGA)

A *Field-Programmable Gate Array* is an integrated circuit containing an array of configurable logic blocks (CLBs), programmable interconnect, block RAM, and I/O cells. The device's function is determined by a configuration bitstream loaded into SRAM cells that control multiplexers, look-up tables, and routing switches throughout the chip.

Every FPGA, regardless of vendor, contains the same fundamental building blocks:

1. **Configurable Logic Blocks (CLBs)** — Each CLB contains one or more Look-Up Tables (LUTs) that implement combinational logic, plus flip-flops for sequential logic. This is where your `always @(posedge clk)` blocks and `assign` statements become hardware.

2. **Programmable Interconnect** — A network of wires and programmable switches that connects CLBs to each other and to I/O pins. The routing network typically consumes more die area than the logic blocks themselves.

3. **Block RAM (BRAM)** — Dedicated memory blocks that implement synchronous RAM far more efficiently than building memory from flip-flops. Our SAMPO uses block RAM for its 64KB address space.

4. **I/O Blocks (IOBs)** — Configurable input/output cells at the chip's periphery. Each IOB can be configured for different voltage standards, pull-up/pull-down resistors, and drive strengths.

5. **Clock Management** — Phase-Locked Loops (PLLs) and clock distribution networks that generate and distribute clock signals throughout the device.

Table 27.1 summarizes these building blocks and their role in the SAMPO design.

Table 27.1: FPGA building blocks and their role in the SAMPO design.

Block	Function	SAMPO Use
LUT4	Any 4-input Boolean function	ALU, decoder, FSM logic
Flip-Flop	1-bit storage (D-type)	Registers, state, PC
Block RAM	Dedicated synchronous memory	64KB main memory
IOB	External pin interface	Clock, UART, LEDs, buttons
PLL	Clock synthesis and management	25 MHz → 12.5 MHz division
Carry Chain	Fast ripple-carry for arithmetic	ALU addition, subtraction

27.1.1 How FPGAs Differ from Simulation

When we simulate the SAMPO with Icarus Verilog, the tool interprets our Verilog as a software model. Every signal update happens in a deterministic event queue. On an FPGA, the situation is fundamentally different:

- **Real propagation delay.** In simulation, combinational logic evaluates in zero time (within a time step). On an FPGA, signals propagate through LUTs and routing with real, measurable delay — typically nanoseconds.

- **Clock frequency matters.** The SAMPO ALU's longest combinational path determines the maximum clock speed. Our nextpnr timing report tells us this path needs approximately 81 ns, capping the frequency at around 12.34 MHz.
- **Metastability.** External signals (like the UART receive pin or button inputs) are asynchronous to the FPGA's clock domain. They must be synchronized through a chain of flip-flops — something simulation does not enforce.
- **Resource limits.** A simulation can instantiate unlimited logic. An FPGA has a fixed number of LUTs, flip-flops, and block RAMs.

Simulation Passes Do Not Guarantee FPGA Success

A design that passes all simulation tests may still fail on real hardware due to timing violations, metastability, or resource exhaustion. The FPGA toolchain (Yosys + nextpnr) performs its own analysis — pay attention to its warnings.

27.2 Look-Up Tables: Any N-Input Function

The Look-Up Table is the atomic unit of combinational logic in an FPGA. A k-input LUT can implement *any* Boolean function of k inputs — AND, OR, XOR, multiplexer, comparator, or any arbitrary truth table. It does this not with gates, but with a small SRAM: a $2^k \times 1$-bit memory whose address lines are the function inputs and whose stored data is the truth table.

Look-Up Table (LUT)

A k-input *Look-Up Table* is a $2^k \times 1$-bit SRAM that implements an arbitrary Boolean function. The k input signals serve as the address; the stored bit at that address is the output. The "program" for a LUT is the 2^k-bit truth table loaded from the FPGA's configuration bitstream.

Most modern FPGAs use 4-input LUTs (LUT4s), though some families offer LUT5s or LUT6s. The ECP5 family that we target uses LUT4s — each implementing any function of up to 4 inputs.

27.2.1 A Concrete Example: 2-to-1 Multiplexer

Consider the 2-to-1 multiplexer used throughout the SAMPO — for instance, in the ALU input selection logic:

Listing 27.1: ALU input multiplexing (from `rtl/cpu.v`).

```
always @(*) begin
    alu_a = rs1_val;
    alu_b = rs2_val;
```

```
    alu_op = dec_alu_op;

    case (dec_opcode)
        `OP_ADDI: begin
            alu_a = rs1_val;  // Rd value (read via rs1)
            alu_b = dec_imm8_sext;
        end
    endcase
end
```

For a single bit of the `alu_b` mux, the truth table is:

sel	dec_imm8_sext[n]	rs2_val[n]	(unused)	alu_b[n]
0	0	0	X	0
0	0	1	X	1
0	1	0	X	0
0	1	1	X	1
1	0	0	X	0
1	0	1	X	0
1	1	0	X	1
1	1	1	X	1

The synthesis tool (Yosys) computes this truth table and programs it into the LUT's SRAM. At runtime, the input signals drive the SRAM address lines, and the correct output appears after the LUT's propagation delay — typically around 0.3–0.5 ns on an ECP5.

27.2.2 LUT Utilization in the SAMPO

The SAMPO CPU consumes roughly 1,500–2,500 LUTs depending on which peripherals are included. Table 27.2 breaks down the estimate by module.

Table 27.2: Estimated LUT utilization for SAMPO modules.

Module	LUTs (estimated)
Register file (16 × 16-bit)	~256
ALU (16-bit with flags)	~200–400
Instruction decoder	~300–500
CPU state machine & control logic	~500–1,000
UART TX + RX	~200–300
Total	**~1,500–2,500**

The ECP5-85F on the ULX3S board provides 83,640 LUTs. Our design uses roughly 2–3% of the available logic — plenty of room for additional peripherals, debug infrastructure, or even a second CPU core.

Viewing LUT Usage

After synthesis, Yosys prints a resource summary showing the exact number of LUTs, flip-flops, and other primitives consumed by your design. We will see how to read this output in Chapter 29.

27.3 Carry Chains for Arithmetic

A LUT4 can implement any 4-input function, but addition is a special case that the FPGA architects optimized with dedicated hardware. Consider what happens when Yosys synthesizes the SAMPO ALU's addition:

Listing 27.2: ALU addition from `rtl/alu.v` — synthesized using carry chains.

```
`ALU_ADD: begin
    {carry_out, alu_result} = op_a + op_b;
    overflow = (op_a[15] == op_b[15]) &&
               (alu_result[15] != op_a[15]);
end
```

A 16-bit addition requires computing the carry from bit 0 through bit 15. If each carry had to propagate through the general-purpose routing network, the delay would be enormous — potentially tens of nanoseconds just for the carry chain.

FPGAs solve this with *dedicated carry logic*: hardwired connections between adjacent LUTs that bypass the routing network entirely. On the ECP5, each logic cell (called a "slice") contains a LUT4, a flip-flop, and carry-chain hardware. The carry output of one slice connects directly to the carry input of the next slice in the same column, with a propagation delay of only about 50 ps per stage.

Carry Chain

A *carry chain* is a dedicated, hardwired path between adjacent logic cells that propagates arithmetic carry signals. By bypassing the general-purpose routing network, carry chains reduce the delay of multi-bit addition from $O(n \times t_{\text{route}})$ to $O(n \times t_{\text{carry}})$, where $t_{\text{carry}} \ll t_{\text{route}}$.

For the SAMPO's 16-bit ALU, the carry chain spans 16 slices in a vertical column. The total carry propagation delay is approximately $16 \times 50\,\text{ps} = 0.8\,\text{ns}$ — a small fraction of the overall critical path.

27.3.1 Impact on Timing

The SAMPO's critical path is not the carry chain itself, but the combination of:

1. Register file read (flip-flop to LUT input)
2. ALU computation (LUTs + carry chain)
3. Flag generation (additional LUTs after the ALU result)
4. Multiplexer selection (result routing back to write data)

The nextpnr timing report for our ECP5 build reports a maximum frequency of approximately 12.34 MHz, corresponding to a critical path of about 81 ns. This is why we divide the 25 MHz board clock down to 12.5 MHz in `fpga/ulx3s_top.v`:

Listing 27.3: Clock divider in `fpga/ulx3s_top.v`.

```
// Clock divider: 25 MHz -> 12.5 MHz
// The CPU's combinational ALU path needs ~81ns (nextpnr reports max
// freq of 12.34 MHz). Running at 12.5 MHz (80ns period) gives some
// timing margin while keeping reasonable performance.
reg clk_div;
wire clk_cpu = clk_div;
initial clk_div = 1'b0;
always @(posedge clk_25mhz) begin
    clk_div <= ~clk_div;
end
```

Without the carry chain optimization, the ALU addition alone could consume 20–30 ns, pushing the maximum frequency even lower.

Why Not a Faster Clock?

Running the CPU at 12.5 MHz when the critical path allows 12.34 MHz leaves very little timing margin. In practice, temperature variations and voltage fluctuations can shift the delay. A more conservative design would target 10 MHz or use pipelining to break the critical path. We discuss pipelining opportunities in Chapter 35.

27.4 Block RAM vs. Distributed RAM

The SAMPO needs 64KB of memory for its address space. There are two fundamentally different ways to build memory on an FPGA: block RAM and distributed RAM.

27.4.1 Block RAM

Block RAMs are dedicated, hardened memory macros embedded in the FPGA fabric. On the ECP5, each block RAM primitive (called an EBR, for Embedded Block RAM) provides 18 Kbits of synchronous dual-port memory. The ECP5-85F contains 208 of these blocks, for a total of approximately 3.7 Mbits (about 462 KB) of block RAM.

Key characteristics of block RAM:

- **Synchronous.** Reads and writes happen on the clock edge. Read data appears one cycle after the address is presented — this is why our SoC needs the `mem_ready` signal.
- **Configurable width.** Each EBR can be configured as $16\text{K} \times 1$, $8\text{K} \times 2$, $4\text{K} \times 4$, $2\text{K} \times 9$, $1\text{K} \times 18$, or 512×36.
- **True dual-port.** Each EBR has two independent ports with separate address, data, and enable signals.
- **Zero LUT cost.** Block RAM does not consume any LUTs — it is a separate resource.

Our SAMPO memory module in `rtl/ram.v` infers block RAM because it follows the synchronous read/write pattern that Yosys recognizes:

Listing 27.4: Synchronous RAM in `rtl/ram.v` — infers block RAM.

```
// 64KB = 32K x 16-bit words
reg [15:0] mem [0:32767];

// Word-aligned address (ignore LSB)
wire [14:0] word_addr = addr[15:1];

// Initialize memory from hex file if specified
initial begin
    if (INIT_FILE != "") begin
        $readmemh(INIT_FILE, mem);
    end
end

// Synchronous read and write
always @(posedge clk) begin
    if (en) begin
        if (we) begin
            // Write with byte enables
            if (be[0]) mem[word_addr][7:0]  <= wdata[7:0];
            if (be[1]) mem[word_addr][15:8] <= wdata[15:8];
        end
        // Read (always returns full word)
```

```
        rdata <= mem[word_addr];
    end
end
```

The critical detail is the registered read: `rdata <= mem[word_addr]` inside an `always @(posedge clk)` block. This tells Yosys that the read is synchronous, which matches the block RAM primitive. If we had written `assign rdata = mem[word_addr]` (a combinational read), Yosys would be forced to use distributed RAM instead — consuming thousands of LUTs.

Synchronous Reads Are Required for Block RAM

Block RAM primitives on the ECP5 (and most other FPGAs) only support synchronous reads. If your Verilog describes a combinational (asynchronous) read, the synthesis tool must implement the memory using LUTs and flip-flops as distributed RAM. For a 64KB memory, this would require over 32,000 flip-flops — more than most FPGAs contain. Always use registered reads for large memories.

27.4.2 Distributed RAM

Distributed RAM is built from the LUT4s themselves. Since a LUT4 is a 16×1-bit SRAM, it can serve as a tiny memory cell. By combining multiple LUTs, you can build small memories without consuming block RAM.

Distributed RAM makes sense for:

- Very small memories (a few words)
- Memories that need combinational (same-cycle) reads
- Register files that need multiple read ports

The SAMPO's 16-register file in `rtl/regfile.v` is a candidate for distributed RAM, since it requires two simultaneous read ports and is small enough (16 entries $\times$ 16 bits = 256 bits) that LUT-based implementation is efficient.

27.4.3 Block RAM Utilization

Our 64KB memory requires $64 \times 1024 \times 8 = 524,288$ bits of storage. Each ECP5 EBR provides 18,432 bits. Configured as $1\text{K} \times 18$ (which accommodates our 16-bit data width), we need:

$$\lceil 32{,}768 / 1{,}024 \rceil = 32 \text{ block RAMs}$$

The ECP5-85F has 208 EBRs, so our memory uses about 15% of the available block RAM. This leaves ample room for additional memories such as a frame buffer, instruction cache, or debug trace buffer.

Check Your Block RAM Usage

If nextpnr reports that your design exceeds available block RAM, consider whether some memories can be reduced in size. The SAMPO does not need the full 64KB for most test programs — you could reduce the memory to 16KB or even 8KB during development.

27.5 Clock Management: PLLs and Dividers

Every synchronous digital circuit needs a clock, and FPGAs provide dedicated hardware for generating, distributing, and managing clock signals.

27.5.1 The Clock Distribution Network

FPGA vendors invest significant die area in clock distribution networks — dedicated low-skew routing resources that deliver the clock signal to every flip-flop with minimal timing variation. On the ECP5, these are called *clock spines* and *clock branches*.

Clock signals must travel from the clock input pin to every flip-flop in the design. The path length varies — flip-flops near the clock input receive the clock earlier than those far away. This variation is called *clock skew*. The dedicated clock network minimizes skew to a few hundred picoseconds, ensuring that all flip-flops see the clock edge at nearly the same time.

Clock Skew

Clock skew is the difference in arrival time of a clock edge at two different flip-flops. Excessive skew can cause setup or hold time violations, leading to unreliable operation. FPGA clock distribution networks are designed to keep skew below 500 ps.

27.5.2 Phase-Locked Loops

A PLL is an analog circuit that generates a new clock signal from a reference clock. PLLs can:

- **Multiply** the input frequency (e.g., 25 MHz → 100 MHz)
- **Divide** the input frequency (e.g., 25 MHz → 12.5 MHz)
- **Phase-shift** the output relative to the input
- **Duty-cycle correct** a clock with unequal high/low periods

The ECP5 contains up to 4 PLLs, each capable of generating multiple output clocks from a single reference. A PLL is the "proper" way to derive a CPU clock from the board oscillator.

27.5.3 Our Approach: Toggle Divider

For the SAMPO build, we use a simpler approach — a single flip-flop that toggles on every rising edge of the 25 MHz clock, producing a 12.5 MHz output:

Listing 27.5: Toggle divider in `fpga/ulx3s_top.v`.

```
reg clk_div;
wire clk_cpu = clk_div;
initial clk_div = 1'b0;
always @(posedge clk_25mhz) begin
    clk_div <= ~clk_div;
end

localparam CLK_FREQ  = 12_500_000;
localparam BAUD_RATE = 115200;
```

This toggle divider is the simplest possible clock divider: one flip-flop, one inversion. It produces a clean 50% duty-cycle clock at exactly half the input frequency.

Why Not Use a PLL?

A PLL would give us more flexibility — we could generate any frequency, not just integer divisions of the input clock. However, PLLs require vendor-specific instantiation and are more complex to configure. For a first FPGA implementation, the toggle divider keeps things straightforward and portable. We discuss the trade-offs further in Section 27.6.5.

The downside of using a fabric-based divider (rather than a PLL) is that the derived clock does not automatically use the dedicated clock network. The synthesis tool may need a constraint or attribute to route `clk_cpu` onto a global clock spine. On the ECP5 with nextpnr, the tool is generally smart enough to promote frequently-used nets to global routing, but it is worth checking the place-and-route log.

27.6 The ECP5 Family

The Lattice Semiconductor ECP5 is a mid-range FPGA family that has become the workhorse of the open-source FPGA ecosystem. It offers a compelling combination of capacity, features, and — crucially — full open-source toolchain support through Project Trellis, Yosys, and nextpnr.

27.6.1 ECP5 Architecture

The ECP5 fabric is organized into a grid of *tiles*, each containing a *programmable logic cell* (PLC). Each PLC contains four *slices*, and each slice contains:

- Two LUT4s (each implementing any 4-input function)
- Two flip-flops (D-type, with set/reset and enable)
- Carry chain logic for arithmetic
- Multiplexers for cascading LUTs into wider functions

This means each PLC contains 8 LUT4s and 8 flip-flops.

27.6.2 ECP5 Family Members

The ECP5 comes in several sizes:

Table 27.3: ECP5 family members.

Device	LUT4s	EBRs	PLLs	Notes
ECP5-12F	12,288	32	2	Entry-level
ECP5-25F	24,576	56	2	Popular for small projects
ECP5-45F	44,820	108	4	Mid-range
ECP5-85F	83,640	208	4	Used on ULX3S

The ULX3S board that we target ships with the ECP5-85F — the largest member of the family. This is generous overkill for the SAMPO, but it means we never need to worry about running out of resources during development.

27.6.3 ECP5 I/O

The ECP5 I/O system supports multiple voltage standards:

- **LVCMOS33** — 3.3V CMOS, used for most ULX3S connections (LEDs, buttons, UART)
- **LVCMOS25** — 2.5V CMOS
- **LVCMOS12** — 1.2V CMOS
- **LVDS** — Low-Voltage Differential Signaling, used for high-speed interfaces
- **SSTL** — Stub Series Terminated Logic, used for DDR memory interfaces

Each I/O pin has configurable:

- Pull-up or pull-down resistors (internal)
- Drive strength (2, 4, 6, 8, or 12 mA)
- Slew rate (slow or fast)
- Input/output delay

These are configured through the pin constraints file (`.lpf`), which we examine in Chapter 28. For example, the ULX3S button pins use LVCMOS33 with internal pull-ups because the buttons are active-low:

Listing 27.6: Button pin constraint from `fpga/ulx3s_v20.lpf`.

```
LOCATE COMP "btn[0]" SITE "D6";
IOBUF PORT "btn[0]" PULLMODE=UP IO_TYPE=LVCMOS33;
```

27.6.4 ECP5 Block RAM Details

Each ECP5 EBR (Embedded Block RAM) is an 18 Kbit true dual-port memory that can be configured in several aspect ratios:

Table 27.4: ECP5 EBR configuration modes.

Configuration	Depth × Width
16K × 1	16,384 entries, 1 bit each
8K × 2	8,192 entries, 2 bits each
4K × 4	4,096 entries, 4 bits each
2K × 9	2,048 entries, 9 bits each
1K × 18	1,024 entries, 18 bits each
512 × 36	512 entries, 36 bits each

For the SAMPO's 16-bit-wide memory, the 1K × 18 mode is the natural fit — it provides 1,024 entries of 18 bits each (we use 16 of the 18 bits). Each EBR covers 2KB of address space (1K words × 2 bytes/word), so our 64KB memory requires 32 EBRs.

The `$readmemh` initialization in `rtl/ram.v` maps directly to the EBR's initialization feature. The synthesis tool converts the hex file contents into the EBR's `INITVAL` parameters, embedding the program directly into the bitstream. This means our SAMPO program is ready to execute the instant the FPGA finishes configuration — no bootloader required.

27.6.5 ECP5 PLL Resources

The ECP5-85F contains 4 PLLs, each capable of:

- Input frequency range: 8–400 MHz
- Output frequency range: 3.125–400 MHz
- Fractional division for non-integer frequency ratios
- Multiple output clocks per PLL (primary + up to 3 secondary)
- Dynamic phase adjustment

For our toggle-divider approach, we do not instantiate any PLLs, leaving all 4 available for future expansion. If we wanted to run the SAMPO at a non-power-of-two frequency — say 10 MHz — we would need a PLL:

Listing 27.7: Example ECP5 PLL instantiation (not used in current SAMPO design).

```
// ECP5 PLL to generate 10 MHz from 25 MHz input
// CLKI_DIV=5, CLKFB_DIV=2, CLKOP_DIV=50
EHXPLLL #(
    .CLKI_DIV(5),
    .CLKFB_DIV(2),
    .FEEDBK_PATH("CLKOP"),
    .CLKOP_DIV(50),
    .CLKOP_ENABLE("ENABLED"),
    .CLKOP_FPHASE(0),
    .CLKOP_CPHASE(49)
) pll_inst (
    .CLKI(clk_25mhz),
    .CLKOP(clk_10mhz),
    .LOCK(pll_locked),
    .RST(1'b0),
    .ENCLKOP(1'b1)
);
```

PLL Lock Signal

A PLL takes several microseconds to lock after power-up or reset. The `LOCK` output indicates when the PLL output is stable. In a production design, you would hold the CPU in reset until `pll_locked` is asserted. Our toggle divider avoids this complexity since it produces a valid clock on the very first input edge.

27.6.6 Open-Source Toolchain Support

The ECP5 was one of the first commercial FPGA families to receive full open-source toolchain support, thanks to the *Project Trellis* reverse-engineering effort. Project Trellis documented the ECP5's bitstream format, enabling the open-source toolchain:

1. **Yosys** reads Verilog and synthesizes it into ECP5 primitives (LUT4s, flip-flops, EBRs)
2. **nextpnr-ecp5** places those primitives onto the physical device and routes the connections
3. **ecppack** (from Project Trellis) converts the placed-and-routed design into a bitstream that can be loaded onto the FPGA

This is the complete pipeline we use for the SAMPO FPGA build. No vendor tools, no license servers, no proprietary software. The entire process runs on a laptop with tools installable through package managers:

Listing 27.8: Installing the open-source ECP5 toolchain on macOS.

```
# Open-source FPGA tools (macOS)
brew install yosys nextpnr-ecp5
# Or from the oss-cad-suite bundle:
# https://github.com/YosysHQ/oss-cad-suite-build
```

We examine this toolchain in detail in Chapter 29.

Exercises

1. **LUT truth table.** Write out the complete 16-entry truth table for a LUT4 that implements a 4-input AND gate. Then write the truth table for a 2-bit equality comparator (inputs: A_1, A_0, B_1, B_0; output: $A = B$). How many LUT4s would you need for a 4-bit equality comparator?

2. **Carry chain analysis.** The SAMPO ALU performs 16-bit addition. If each carry stage adds 50 ps of delay, what is the carry chain propagation delay for a 32-bit addition? At what clock frequency would the carry chain alone become the critical path, assuming the rest of the logic path adds 5 ns of delay?

3. **Block RAM sizing.** Suppose you wanted to add a second memory bank to the SAMPO for a dedicated video frame buffer storing a 320×240 monochrome display (1 bit per pixel). How many ECP5 EBRs would this require? What percentage of the ECP5-85F's EBRs would the CPU memory plus frame buffer consume?

4. **Clock domain exploration.** The SAMPO currently uses a toggle divider to go from 25 MHz to 12.5 MHz. What clock frequencies could you generate with a chain of 1, 2, or 3 toggle dividers? What frequencies would require a PLL?

5. **Resource estimation.** Using the LUT estimates from Table 27.2, determine the smallest ECP5 family member (from Table 27.3) that could fit the SAMPO CPU with 64KB of memory. What if you reduced the memory to 16KB?

What's Next

We now understand the physical substrate that our Verilog will map onto: LUTs for logic, carry chains for arithmetic, block RAM for memory, and PLLs for clocking. In Chapter 28, we turn our attention to the specific board that hosts our FPGA — the ULX3S. We will examine its hardware layout, pin assignments, USB-serial interface, and the constraints file that tells the toolchain how to connect our design to the physical world.

Chapter 28

The ULX3S Board

An FPGA by itself is a bare chip — rows of pins, no blinking lights, no easy way to get data in or out. A development board wraps that chip in everything you need for a working system: a clock oscillator, power regulation, a USB-serial bridge, buttons, LEDs, and a JTAG programming interface. The ULX3S is the board we have chosen for the SAMPO build, and in this chapter we will walk through every piece of it that matters for getting our CPU running.

The ULX3S is an open-source hardware design created by the FPGA community, built around the Lattice ECP5 FPGA. It pairs the largest ECP5 variant — the 85F — with a generous set of peripherals, all at a price that makes FPGA development accessible. More importantly for our purposes, it works flawlessly with the fully open-source toolchain: Yosys, nextpnr-ecp5, and openFPGALoader.

28.1 ULX3S Hardware Overview

The ULX3S board provides everything we need to run the SAMPO CPU as a physical system. Let us start with the top-level module that ties it all together, then work outward to the individual peripherals:

Listing 28.1: Top-level port list from `fpga/ulx3s_top.v`.

```
module ulx3s_top (
    input         clk_25mhz,    // 25 MHz onboard oscillator

    // Buttons (active-low)
    input  [6:0]  btn,          // BTN0 = reset, BTN1-6 = user

    // LEDs
    output [7:0]  led,
```

```
    // FTDI USB-Serial
    output        ftdi_rxd,     // FPGA TX -> FTDI RX
    input         ftdi_txd      // FTDI TX -> FPGA RX
);
```

This is the interface between our design and the physical world. Every signal in this port list corresponds to a real wire on the PCB, connected to a real component.

Table 28.1 summarizes the board components we use.

Table 28.1: ULX3S components used by the SAMPO design.

Component	Pins	Purpose
25 MHz oscillator	`G2`	System clock source
7 buttons	`D6`, `R1`, `T1`, …	Reset (`btn[0]`) and user input
8 LEDs	`B2`–`H3`	Status display (halted, UART activity, PC bits)
FTDI FT231X	`L4`, `M1`	USB-to-serial bridge for UART I/O
ECP5-85F	—	FPGA fabric (83,640 LUTs, 208 EBRs)

The ULX3S also has 32MB of SDRAM, an ESP32 module, HDMI output, a microSD slot, and numerous GPIO headers — but the SAMPO design uses none of these. Our CPU is deliberately minimal: clock, reset, serial I/O, and a few status LEDs.

Board Versions

The ULX3S has gone through several PCB revisions. Our constraints file targets the v2.0+ layout. The pin assignments for buttons, LEDs, and FTDI are the same across recent board revisions, but always verify your constraints file matches your specific board version.

28.2 Pin Constraints File

The pin constraints file — called an `.lpf` file (Lattice Preference File) for ECP5 targets — tells the place-and-route tool which FPGA pin to assign to each signal in our top-level module. Without this file, nextpnr has no way to know that `clk_25mhz` should be connected to pin `G2`, or that `led[0]` drives the LED connected to pin `B2`.

28.2.1 Anatomy of a Constraints File

Our constraints file at `fpga/ulx3s_v20.lpf` contains two types of statements for each signal:

Listing 28.2: Clock pin constraint from `fpga/ulx3s_v20.lpf`.

```
# Clock - 25 MHz onboard oscillator
LOCATE COMP "clk_25mhz" SITE "G2";
IOBUF PORT "clk_25mhz" PULLMODE=NONE IO_TYPE=LVCMOS33;
FREQUENCY PORT "clk_25mhz" 25 MHZ;
```

Each signal needs at minimum two declarations:

1. **`LOCATE COMP`** assigns the signal to a physical pin. The `SITE` value (e.g., `"G2"`) is the ball/pin name from the ECP5 package pinout. The signal name in quotes must exactly match the port name in your Verilog module.

2. **`IOBUF PORT`** configures the I/O buffer for that pin. It sets the voltage standard (`IO_TYPE`), pull-up/pull-down resistors (`PULLMODE`), and drive strength (`DRIVE`).

The optional `FREQUENCY PORT` statement tells the timing analysis engine that this pin carries a clock signal at the specified frequency. This is critical for accurate timing analysis — without it, nextpnr cannot check whether your design meets timing at the intended clock speed.

28.2.2 Button Constraints

The ULX3S has 7 push buttons, active-low (pressed = 0, released = 1). We use `btn[0]` as the CPU reset:

Listing 28.3: Button constraints from `fpga/ulx3s_v20.lpf`.

```
# Buttons (active-low)
LOCATE COMP "btn[0]" SITE "D6";
IOBUF PORT "btn[0]" PULLMODE=UP IO_TYPE=LVCMOS33;
LOCATE COMP "btn[1]" SITE "R1";
IOBUF PORT "btn[1]" PULLMODE=UP IO_TYPE=LVCMOS33;
LOCATE COMP "btn[2]" SITE "T1";
IOBUF PORT "btn[2]" PULLMODE=UP IO_TYPE=LVCMOS33;
LOCATE COMP "btn[3]" SITE "R18";
IOBUF PORT "btn[3]" PULLMODE=UP IO_TYPE=LVCMOS33;
LOCATE COMP "btn[4]" SITE "V1";
IOBUF PORT "btn[4]" PULLMODE=UP IO_TYPE=LVCMOS33;
LOCATE COMP "btn[5]" SITE "U1";
IOBUF PORT "btn[5]" PULLMODE=UP IO_TYPE=LVCMOS33;
LOCATE COMP "btn[6]" SITE "H16";
IOBUF PORT "btn[6]" PULLMODE=UP IO_TYPE=LVCMOS33;
```

PULLMODE=UP enables the internal pull-up resistor. When the button is not pressed, the pull-up holds the signal at 3.3V (logic 1). When pressed, the button shorts the pin to ground (logic 0). This is why the buttons are "active-low" — pressing them drives the signal low.

28.2.3 LED Constraints

Listing 28.4: LED constraints from `fpga/ulx3s_v20.lpf`.

```
# LEDs
LOCATE COMP "led[0]" SITE "B2";
IOBUF PORT "led[0]" PULLMODE=NONE IO_TYPE=LVCMOS33 DRIVE=4;
LOCATE COMP "led[1]" SITE "C2";
IOBUF PORT "led[1]" PULLMODE=NONE IO_TYPE=LVCMOS33 DRIVE=4;
```

The LEDs use DRIVE=4 (4 mA drive strength). This is sufficient to illuminate a typical SMD LED through the board's current-limiting resistors. No pull-up is needed since the FPGA actively drives these outputs.

28.2.4 FTDI UART Constraints

Listing 28.5: FTDI serial constraints from `fpga/ulx3s_v20.lpf`.

```
# FTDI USB-Serial
LOCATE COMP "ftdi_rxd" SITE "L4";
IOBUF PORT "ftdi_rxd" PULLMODE=NONE IO_TYPE=LVCMOS33 DRIVE=4;
LOCATE COMP "ftdi_txd" SITE "M1";
IOBUF PORT "ftdi_txd" PULLMODE=NONE IO_TYPE=LVCMOS33;
```

Note the naming convention: `ftdi_rxd` is the FTDI chip's RX pin, which means it receives data *from* the FPGA. So `ftdi_rxd` is an *output* from our design's perspective ("FPGA TX → FTDI RX"), and `ftdi_txd` is an *input* ("FTDI TX → FPGA RX"). This cross-over naming can be confusing; look at the port direction in the Verilog to resolve any ambiguity.

Signal Name Direction

The names `ftdi_rxd` and `ftdi_txd` describe the direction from the *FTDI chip*'s perspective, not the FPGA's. From our Verilog code's perspective:

- `ftdi_rxd` is an `output` (we transmit to the FTDI)
- `ftdi_txd` is an `input` (we receive from the FTDI)

Mixing these up will result in a design that compiles and programs correctly but produces no serial output.

28.3 Clock Configuration

The ULX3S carries a 25 MHz crystal oscillator connected to pin G2. This is the only external clock source we use.

28.3.1 From 25 MHz to 12.5 MHz

As we discussed in Section 27.3.1, the SAMPO's critical path limits the maximum clock frequency to about 12.34 MHz. We use a toggle flip-flop to divide the 25 MHz oscillator by 2:

Listing 28.6: Clock division in `fpga/ulx3s_top.v`.

```
reg clk_div;
wire clk_cpu = clk_div;
initial clk_div = 1'b0;
always @(posedge clk_25mhz) begin
    clk_div <= ~clk_div;
end

localparam CLK_FREQ  = 12_500_000;
localparam BAUD_RATE = 115200;
```

The `CLK_FREQ` parameter is passed to the UART modules so they can compute the correct baud rate divisor. Getting this wrong would produce garbled serial output.

28.3.2 Clock Domain Considerations

The toggle divider creates a second clock domain: `clk_25mhz` (the raw oscillator) and `clk_cpu` (the divided clock). In our design, *everything* runs on `clk_cpu` — the CPU, the SoC, the UARTs, and the reset synchronizer all use the divided clock. This avoids clock-domain crossing issues entirely.

The only module that sees `clk_25mhz` directly is the toggle flip-flop itself. Since a single flip-flop cannot have a timing violation with itself, this is safe by construction.

Keep Clock Domains Minimal

Every clock domain boundary requires synchronization logic (typically two flip-flops in series) to avoid metastability. By running the entire SAMPO system on a single derived clock, we eliminate this class of bugs entirely. If you add peripherals that need a different clock frequency, carefully consider whether they need their own clock domain or can share the CPU clock.

28.4 FTDI USB-Serial Interface

The ULX3S includes an FTDI FT231X USB-to-serial converter. When you plug the board into your computer via USB, this chip appears as a virtual serial port (e.g., `/dev/cu.usbserial-D01263` on macOS or `/dev/ttyUSB0` on Linux). The FT231X handles all the USB protocol complexity — from the FPGA's perspective, it is a pair of TTL-level serial lines.

28.4.1 The Communication Path

The full communication path from a SAMPO program to your terminal looks like this:

1. SAMPO CPU executes **`OUTI`** `R5, 0x81` — writes a byte to the UART data port
2. The SoC's `uart` module (in `rtl/uart.v`) latches the byte and asserts `tx_valid`
3. The hardware `uart_tx` module (in `fpga/uart_tx.v`) serializes the byte into 8N1 format on `ftdi_rxd`
4. The FT231X chip receives the serial data and buffers it
5. The FT231X presents the data to the host computer over USB
6. Your terminal program (`screen`, `picocom`, `minicom`) displays the character

This path involves two separate UARTs. The "software" UART in `rtl/uart.v` is an MC6850-compatible register interface that the CPU sees through its I/O port system. The "hardware" UARTs in `fpga/uart_tx.v` and `fpga/uart_rx.v` handle the actual serial-line timing.

28.4.2 The Hardware UART Transmitter

The transmitter in `fpga/uart_tx.v` is a state machine that converts parallel bytes into serial 8N1 frames:

Listing 28.7: UART TX state machine from `fpga/uart_tx.v`.

```
// Baud rate divisor: number of clock ticks per bit
localparam CLKS_PER_BIT = CLK_FREQ / BAUD_RATE;
// Counter width needed
localparam CNT_W = $clog2(CLKS_PER_BIT);

// States
localparam S_IDLE  = 2'd0;
localparam S_START = 2'd1;
localparam S_DATA  = 2'd2;
localparam S_STOP  = 2'd3;
```

At 12.5 MHz with 115,200 baud, each bit period is:

$$\text{CLKS_PER_BIT} = \frac{12{,}500{,}000}{115{,}200} = 108 \text{ clocks}$$

The transmitter holds the output pin at the correct level for exactly 108 clock cycles per bit. A complete 8N1 frame takes 10 bit periods (1 start + 8 data + 1 stop):

$$10 \times 108 = 1{,}080 \text{ clocks} = 86.4 \ \mu\text{s per character}$$

The state machine drives the `ftdi_rxd` pin directly:

Listing 28.8: UART TX data shifting from `fpga/uart_tx.v`.

```
S_DATA: begin
    tx_pin <= shift_reg[0];  // LSB first
    if (baud_tick) begin
        shift_reg <= {1'b0, shift_reg[7:1]};
        if (bit_idx == 3'd7) begin
            state <= S_STOP;
        end else begin
            bit_idx <= bit_idx + 1;
        end
        baud_cnt <= CLKS_PER_BIT[CNT_W-1:0] - 1;
    end else begin
        baud_cnt <= baud_cnt - 1;
    end
end
```

Data is shifted out LSB first, which is the standard for RS-232 / UART communication. The `baud_tick` signal goes high for one clock when the baud counter reaches zero, triggering the transition to the next bit.

28.4.3 The Hardware UART Receiver

The receiver in `fpga/uart_rx.v` is more complex because it must detect the start of a frame from an asynchronous signal and sample each bit at the center of its period. It uses 16× oversampling:

Listing 28.9: UART RX oversampling from `fpga/uart_rx.v`.

```
// 16x oversampling clock divisor
localparam CLKS_PER_SAMPLE = CLK_FREQ / (BAUD_RATE * 16);
```

At 12.5 MHz and 115,200 baud:

$$\text{CLKS_PER_SAMPLE} = \frac{12{,}500{,}000}{115{,}200 \times 16} \approx 6 \text{ clocks}$$

The receiver runs at 16× the baud rate, giving it 16 sample points within each bit period. It detects the falling edge of the start bit, waits 8 sample periods (half a bit period) to reach the center, then samples every 16 periods thereafter to capture each data bit at its midpoint.

Input Synchronization

Because `ftdi_txd` is asynchronous to our clock domain, the receiver includes a 2-stage synchronizer to prevent metastability:

Listing 28.10: Metastability protection in `fpga/uart_rx.v`.

```
// Input synchronizer (2-FF to prevent metastability)
reg rx_sync1, rx_sync2;
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        rx_sync1 <= 1'b1;
        rx_sync2 <= 1'b1;
    end else begin
        rx_sync1 <= rx_pin;
        rx_sync2 <= rx_sync1;
    end
end
wire rx_in = rx_sync2;
```

The two flip-flops (`rx_sync1` and `rx_sync2`) give any metastable state time to resolve before the signal is used by the rest of the receiver logic. The reset value of `1'b1` matches the UART idle state (line high).

Metastability

Metastability occurs when a flip-flop samples an input that changes too close to the clock edge, leaving the flip-flop in an indeterminate state between 0 and 1. A 2-stage synchronizer reduces the probability of metastable output to negligible levels (less than once per century at typical clock rates).

False Start Detection

The receiver includes false-start detection — if the line returns high before the mid-point of the start bit, the receiver returns to idle:

Listing 28.11: False start detection from `fpga/uart_rx.v`.

```
S_START: begin
    if (samp_tick) begin
        if (tick_cnt == 4'd7) begin
            // Mid-bit sample: verify start bit is still low
            if (!rx_in) begin
                state    <= S_DATA;
                tick_cnt <= 0;
                bit_idx  <= 0;
            end else begin
                // False start - go back to idle
                state <= S_IDLE;
            end
        end else begin
            tick_cnt <= tick_cnt + 1;
        end
        samp_cnt <= CLKS_PER_SAMPLE[SAMP_W-1:0] - 1;
    end else begin
        samp_cnt <= samp_cnt - 1;
    end
end
```

This is important for noise immunity. A brief glitch on the serial line might trigger a falling edge, but the false-start check prevents it from being interpreted as a valid frame.

28.4.4 The MC6850-Compatible Interface

The SoC-level UART module in `rtl/uart.v` provides the register interface that the CPU sees. It follows the MC6850 ACIA (Asynchronous Communications Interface Adapter) convention:

Table 28.2: UART port map.

Port	Name	Read	Write
`0x80`	Status	Bit 1: TX ready, Bit 0: RX data available	—
`0x81`	Data	Received byte	Transmit byte

Listing 28.12: Status register from `rtl/uart.v`.

```
// Status bits (MC6850-compatible)
// Bit 0: RDRF - Receive Data Register Full
// Bit 1: TDRE - Transmit Data Register Empty (TX ready)
wire [7:0] status;
assign status = {6'b0, tx_ready, rx_pending};
```

This is the same interface that the SAMPO emulator implements, so programs that work in simulation also work on real hardware without modification.

28.5 LEDs, Buttons, and GPIO

28.5.1 LED Status Display

The 8 LEDs on the ULX3S provide at-a-glance status information about the CPU. The SoC module in `rtl/soc.v` assigns specific meanings to each LED:

Listing 28.13: LED status assignment from `rtl/soc.v`.

```
// LED[7]:   CPU halted
// LED[6]:   UART TX active (data being sent)
// LED[5]:   UART RX data available
// LED[4:0]: Lower 5 bits of PC (shows execution progress)
assign led = {halted, tx_valid, rx_valid, pc[5:1]};
```

This is a minimal but surprisingly useful debug interface:

- **LED 7 (halted):** Lights up when the CPU executes the **`HALT`** instruction. If this LED is on immediately after programming, your program reached **`HALT`**. If it never comes on, the program may be stuck in an infinite loop or crashed.
- **LED 6 (TX active):** Blinks when the CPU is sending serial data. Rapid blinking during program execution is a sign of life. No blinking when you expect output means the UART path is broken.
- **LED 5 (RX available):** Lights when incoming serial data is waiting to be read by the CPU.
- **LEDs 4–0 (PC bits):** Show the lower bits of the program counter, shifted right by 1 (since instructions are word-aligned). These create a binary "progress bar" that visually indicates the CPU is executing code. The pattern changes with every instruction fetch.

The top-level module simply passes these through to the output pins:

Listing 28.14: LED passthrough in `fpga/ulx3s_top.v`.

```
assign led = led_status;
```

Using LEDs for Debugging

When bringing up a new FPGA design, LEDs are often the only debug output that works before the serial port is verified. If you are not getting UART output, try assigning a slow counter to the LEDs to verify that the clock and logic fabric are working. A counter running at the CPU clock frequency with `led[7] = counter[22]` will produce a visible blink at roughly 3 Hz.

28.5.2 Buttons and Reset

The SAMPO uses `btn[0]` as the external reset button. Since the button is active-low (pressed = 0) and asynchronous to the CPU clock, we cannot connect it directly to the `rst_n` signal. The reset synchronizer module in `fpga/reset_sync.v` handles both synchronization and debouncing:

Listing 28.15: Reset synchronizer from `fpga/reset_sync.v`.

```
module reset_sync #(
    parameter CLK_FREQ   = 25_000_000,
    parameter DEBOUNCE_MS = 20       // Debounce time in milliseconds
) (
    input  wire clk,
    input  wire ext_rst_n,    // External reset (active-low, from button)
    output wire rst_n         // Synchronized, debounced reset (active-low)
);

    localparam DEBOUNCE_CYCLES = CLK_FREQ / 1000 * DEBOUNCE_MS;
    localparam CNT_W = $clog2(DEBOUNCE_CYCLES + 1);

    // 2-FF synchronizer for metastability protection
    reg sync1, sync2;
    always @(posedge clk) begin
        sync1 <= ext_rst_n;
        sync2 <= sync1;
    end
```

The module performs three functions:

1. **Synchronization:** A 2-FF chain (`sync1`, `sync2`) prevents metastability from the asynchronous button signal.

2. **Immediate assertion:** When the button is pressed (signal goes low), reset is asserted immediately — no debounce delay on the press.
3. **Debounced release:** When the button is released, a counter waits for 20 ms of stable high signal before de-asserting reset. This prevents the mechanical contact bounce from causing multiple reset pulses.

Listing 28.16: Debounce logic from `fpga/reset_sync.v`.

```
// Debounce counter
reg [CNT_W-1:0] cnt;
reg             rst_out;

always @(posedge clk) begin
    if (!sync2) begin
        // Button pressed (active-low): immediately assert reset
        cnt     <= 0;
        rst_out <= 1'b0;
    end else if (cnt < DEBOUNCE_CYCLES[CNT_W-1:0]) begin
        // Button released: count up before deasserting reset
        cnt <= cnt + 1;
    end else begin
        rst_out <= 1'b1;
    end
end

assign rst_n = rst_out;
```

At 12.5 MHz, 20 ms of debounce requires counting to 250,000 — a counter width of 18 bits.

Contact Bounce

Mechanical push buttons do not transition cleanly between open and closed states. The contacts physically bounce, producing a rapid series of transitions that can last 5–20 ms. Without debouncing, a single button press could trigger dozens of reset pulses. The debounce counter filters out this noise by requiring the signal to remain stable for a specified duration.

The reset synchronizer is instantiated in the top-level module:

Listing 28.17: Reset synchronizer instantiation from `fpga/ulx3s_top.v`.

```
reset_sync #(
    .CLK_FREQ(CLK_FREQ)
) u_reset (
    .clk(clk_cpu),
```

```
    .ext_rst_n(btn[0]),
    .rst_n(rst_n)
);
```

Note that the reset synchronizer runs on `clk_cpu` (12.5 MHz), not `clk_25mhz`. This ensures the debounced reset signal is synchronous to the CPU clock domain.

28.6 JTAG Programming via openFPGALoader

The ULX3S includes JTAG access through the FTDI chip, which doubles as both a USB-serial bridge and a JTAG programmer. The FTDI FT231X on the ULX3S supports MPSSE (Multi-Protocol Synchronous Serial Engine) mode, which openFPGALoader uses to drive the JTAG pins.

28.6.1 openFPGALoader

openFPGALoader is an open-source utility that programs FPGAs from various vendors. For the ULX3S, the command is:

Listing 28.18: Programming the ULX3S with openFPGALoader.

```
openFPGALoader -b ulx3s build/sampo.bit
```

The `-b ulx3s` flag tells openFPGALoader to use the ULX3S board profile, which knows the FTDI chip configuration and JTAG pin assignments.

28.6.2 SRAM vs. Flash Programming

The ECP5 supports two programming modes:

- **SRAM programming** (default): The bitstream is loaded directly into the FPGA's configuration SRAM. This is fast (about 2 seconds for the 85F) but volatile — the configuration is lost when power is removed.
- **Flash programming**: The bitstream is written to an external SPI flash chip on the board. At power-up, the FPGA automatically loads its configuration from flash. This persists across power cycles but takes longer to program.

For development, SRAM programming is ideal. You can iterate quickly: modify Verilog, rebuild, reprogram, and test — all in under 30 seconds.

Listing 28.19: Flash programming for persistent configuration.

```
# Program to volatile SRAM (fast, lost on power-off)
```

```
openFPGALoader -b ulx3s build/sampo.bit

# Program to SPI flash (slower, persists across power cycles)
openFPGALoader -b ulx3s -f build/sampo.bit
```

28.6.3 Verifying the Connection

If openFPGALoader cannot find the board, check:

1. The USB cable is connected (the ULX3S has a micro-USB connector)
2. The FTDI driver is installed and loaded
3. No other program has the serial port open (this can block JTAG access on some operating systems)
4. The user has permission to access the USB device (on Linux, you may need a udev rule)

You can verify JTAG connectivity without programming:

Listing 28.20: Detecting the FPGA over JTAG.

```
openFPGALoader --detect
```

This scans the JTAG chain and reports the device ID. For the ECP5-85F, you should see a device with IDCODE `0x41113043`.

FTDI Dual-Mode

The FTDI chip on the ULX3S serves dual duty: JTAG programming and serial communication. You cannot use both simultaneously — openFPGALoader takes exclusive control of the FTDI during programming. After programming completes, you can open the serial port normally. This is why the Makefile's `fpga-*` targets program first, sleep for 1 second, then open the serial port.

Exercises

1. **Constraint file modification.** The current design uses only 3 of the 7 buttons (`btn[0]` for reset, `btn[1]`–`btn[6]` declared but unused). Modify the constraints file to use `btn[1]` as a "single-step" input. What changes would you need in `fpga/ulx3s_top.v` to bring this signal into the design?

2. **Baud rate calculation.** The UART TX uses `CLKS_PER_BIT = CLK_FREQ / BAUD_RATE`. For a 12.5 MHz clock and 115,200 baud, this gives 108. What is the actual baud rate produced by 108 clocks per bit? What is the percentage error from the target 115,200 baud?

3. **UART timing.** A SAMPO program outputs the string "Hello, Sampo!" (13 characters plus a null terminator). At 115,200 baud with 8N1 framing, how long does it take to transmit all 13 characters? If the CPU clock is 12.5 MHz and each **`OUTI`** instruction takes approximately 6 cycles, can the CPU outrun the UART? What mechanism prevents data loss?

4. **Debounce exploration.** The reset synchronizer uses a 20 ms debounce period. At 12.5 MHz, how wide is the debounce counter (in bits)? What would happen if you reduced the debounce period to 1 ms? What if you eliminated it entirely?

5. **LED encoding.** With LEDs 4–0 showing `pc[5:1]`, what LED pattern would you expect to see when the PC is at `0x0100` (the reset vector)? What about `0x010A`? Draw the expected LED states for each address.

What's Next

We now have a concrete picture of the hardware platform: the ECP5 FPGA, the ULX3S board, and every wire that connects our design to the outside world. In Chapter 29, we will trace the path from Verilog source code to a programming bitstream, following each stage of the open-source toolchain — Yosys synthesis, nextpnr placement and routing, and ecppack bitstream generation.

Chapter 29

Synthesis and Place & Route

In the previous chapters, we learned what an FPGA is and which board we are targeting. Now we face the central question: how does a collection of Verilog files become a bitstream that configures real silicon? The answer is a pipeline of three tools — Yosys, nextpnr-ecp5, and ecppack — each performing a distinct transformation. Every stage of this pipeline is open-source, inspectable, and under our control.

This chapter follows the SAMPO design through each stage, reading the actual Makefile rules that orchestrate the build, interpreting the output each tool produces, and understanding what "timing closure" means when your CPU's critical path is 81 ns.

29.1 The Open-Source FPGA Toolchain

The traditional FPGA development flow uses vendor-provided tools: Xilinx Vivado, Intel Quartus, or Lattice Diamond. These are large, proprietary packages that require licenses and often run only on specific operating systems. The open-source FPGA toolchain provides an alternative that works on any platform:

Table 29.1: The open-source FPGA toolchain stages.

Stage	Tool	Input	Output
Synthesis	Yosys	Verilog (`.v`)	JSON netlist
Place & Route	nextpnr-ecp5	JSON + constraints (`.lpf`)	Text config (`.config`)
Bitstream	ecppack	Text config	Bitstream (`.bit`)
Programming	openFPGALoader	Bitstream	FPGA configured

The flow is a linear pipeline. Each tool consumes the output of the previous one:

$$\text{Verilog} \xrightarrow{\text{Yosys}} \text{JSON} \xrightarrow{\text{nextpnr}} \text{.config} \xrightarrow{\text{ecppack}} \text{.bit} \xrightarrow{\text{openFPGALoader}} \text{FPGA}$$

Synthesis

Synthesis is the process of translating a hardware description (Verilog, VHDL, or another HDL) into a *netlist* — a graph of logic elements (LUTs, flip-flops, block RAMs) and the connections between them. The netlist is technology-specific: Yosys maps the design onto primitives that exist in the target FPGA architecture.

Place and Route

Place and route assigns each logic element in the netlist to a specific physical location on the FPGA (*placement*) and determines the routing paths that connect them (*routing*). This step determines whether the design meets timing requirements.

29.2 Yosys: RTL Synthesis

Yosys is the open-source synthesis tool created by Claire (formerly Clifford) Wolf. It reads Verilog, performs a series of transformations, and outputs a netlist of FPGA primitives.

29.2.1 The Synthesis Command

The Makefile in `verilog/Makefile` defines the synthesis step:

Listing 29.1: Yosys synthesis rule from `verilog/Makefile`.

```
$(BUILD_DIR)/sampo.json: $(RTL_SRCS) $(FPGA_SRCS) \
    $(PROG_DIR)/hello.hex | $(BUILD_DIR)
  yosys -p "read_verilog -I$(RTL_DIR) \
    $(RTL_DIR)/alu.v $(RTL_DIR)/shifter.v \
    $(RTL_DIR)/regfile.v $(RTL_DIR)/decode.v \
    $(RTL_DIR)/cpu.v $(RTL_DIR)/ram.v \
    $(RTL_DIR)/uart.v $(RTL_DIR)/soc.v \
    $(FPGA_DIR)/uart_tx.v $(FPGA_DIR)/uart_rx.v \
    $(FPGA_DIR)/reset_sync.v $(FPGA_DIR)/ulx3s_top.v; \
    synth_ecp5 -top ulx3s_top -json $@"
```

Let us break this command apart:

- `read_verilog -I$(RTL_DIR)` tells Yosys to read the Verilog source files, using `rtl/` as an include path (needed for the `` `include "sampo_pkg.vh" `` directives).
- The source files are listed in dependency order: primitives (`alu.v`, `shifter.v`, `regfile.v`) first, then higher-level modules (`cpu.v`, `soc.v`), and finally the FPGA-specific wrappers (`uart_tx.v`, `uart_rx.v`, `reset_sync.v`, `ulx3s_top.v`).
- `synth_ecp5` invokes the ECP5-specific synthesis flow. This is a Yosys *synthesis script* that performs all the standard transformations — constant propagation, dead-code elimination, technology mapping — targeting ECP5 primitives.
- `-top ulx3s_top` specifies which module is the top level of the design hierarchy. Yosys will optimize away any logic not reachable from this module.
- `-json $@` writes the output netlist in JSON format, which nextpnr can read.

29.2.2 What Yosys Does

The `synth_ecp5` command is actually a sequence of many internal passes. At a high level, Yosys performs these transformations:

1. **Parsing.** Read the Verilog and build an Abstract Syntax Tree (AST). Resolve `` `include `` and `` `define `` directives.
2. **Elaboration.** Expand module instantiations, resolve parameters, and flatten the hierarchy into a single design. Our `soc` module becomes a flat netlist containing the CPU, RAM, UART, and all their sub-modules.
3. **High-level optimization.** Constant propagation, dead-code elimination, and algebraic simplification. If a signal is always zero, Yosys removes all logic that depends on it.
4. **Memory inference.** Yosys recognizes the RAM pattern in `rtl/ram.v` — a synchronous read/write to a `reg` array — and maps it to ECP5 EBR (Embedded Block RAM) primitives rather than building it from flip-flops.
5. **Technology mapping.** Boolean functions are decomposed into 4-input LUTs. Arithmetic operations are mapped to carry chains. Flip-flops are matched to the ECP5's DFF primitives.
6. **Optimization.** LUT merging, constant folding, and other optimizations to reduce resource usage.

29.2.3 Reading the Synthesis Log

Yosys produces verbose output during synthesis. The most important section is the resource summary printed at the end. For the SAMPO design, it looks something like this:

Listing 29.2: Example Yosys resource summary (representative numbers).

```
=== ulx3s_top ===

   Number of wires:                4812
   Number of wire bits:           18234
   Number of public wires:          523
   Number of public wire bits:     5647
   Number of memories:                0
   Number of memory bits:             0
   Number of processes:               0
   Number of cells:                5138
     CCU2C                          238
     DP16KD                          32
     TRELLIS_FF                    1247
     LUT4                          3604
     TRELLIS_IO                      24
```

Let us interpret each line:

- **LUT4: 3604** — The number of 4-input Look-Up Tables. This is the primary measure of logic utilization on the ECP5. The 85F has 83,640 LUTs, so we are using about 4%.
- **TRELLIS_FF: 1247** — The number of flip-flops (D-type registers). This includes the CPU registers, state machine state, program counter, cycle counter, UART shift registers, and all other sequential elements.
- **DP16KD: 32** — The number of 16K dual-port block RAMs. Our 64KB memory requires 32 EBRs, exactly matching our calculation from Section 27.4.3.
- **CCU2C: 238** — The number of carry-chain cells. Each CCU2C contains two carry stages, so 238 cells provide 476 carry stages for arithmetic operations.
- **TRELLIS_IO: 24** — The number of I/O buffers. We use 1 clock input + 7 buttons + 8 LEDs + 2 UART pins = 18, plus some internally-generated I/Os.

Checking for Memory Inference

If the `DP16KD` count is zero when you expect block RAM, Yosys failed to infer your memory as block RAM. Check that your read is registered (inside an `always @(posedge clk)`) and that you are not using the memory's output combinationally in the same cycle as the read.

29.3 nextpnr-ecp5: Placement and Routing

Synthesis gives us a netlist — a list of LUTs, flip-flops, and block RAMs with their connections. But this netlist is abstract: it does not say *where* on the FPGA each element should go, or *which wires* should connect them. That is the job of nextpnr.

29.3.1 The Place and Route Command

From the Makefile:

Listing 29.3: nextpnr place and route rule from `verilog/Makefile`.

```
$(BUILD_DIR)/sampo.config: $(BUILD_DIR)/sampo.json $(LPF)
  $(NEXTPNR_ECP5) --85k --package CABGA381 \
    --json $< --lpf $(LPF) \
    --textcfg $@ --freq 25 --timing-allow-fail
```

The arguments:

- `--85k` selects the ECP5-85F device. This tells nextpnr the exact dimensions of the FPGA grid, the number and location of block RAMs, PLLs, and I/O pins.
- `--package CABGA381` specifies the package type — a 381-ball BGA. Different packages expose different subsets of the FPGA's I/O pins.
- `--json $<` reads the synthesized netlist from Yosys.
- `--lpf $(LPF)` reads the pin constraints from `fpga/ulx3s_v20.lpf`, which we examined in Section 28.2.
- `--textcfg $@` writes the placed-and-routed design as a text configuration file, which ecppack will convert to a binary bitstream.
- `--freq 25` tells the timing analyzer that the design should run at 25 MHz. Since our actual CPU clock is 12.5 MHz (derived internally), this is conservative — but nextpnr needs a frequency to perform timing analysis.
- `--timing-allow-fail` allows the build to complete even if timing constraints are not met. During development, this is useful because timing violations do not always cause functional failures, and you want to test the design on hardware regardless.

–timing-allow-fail in Production

The `--timing-allow-fail` flag should be removed for a production design. Timing violations mean that signals may not arrive at flip-flop inputs before the clock edge, leading to unpredictable behavior. During development, however, it prevents the tool from blocking your iteration cycle over minor timing issues.

29.3.2 What nextpnr Does

nextpnr performs two interleaved operations:

Placement

Placement assigns each logic element to a specific location on the FPGA. The placer considers:

- **I/O constraints.** Pins specified in the `.lpf` file are fixed; the placer cannot move them.
- **Block RAM locations.** EBR primitives exist at specific grid positions; the placer must assign our 32 `DP16KD` instances to physical EBR sites.
- **Carry chain legality.** Carry chains must be placed in vertically adjacent slices within the same column.
- **Timing.** Elements connected by critical paths should be placed close together to minimize routing delay.

nextpnr uses a simulated annealing algorithm: it starts with a random placement, then iteratively swaps elements, accepting swaps that improve the design quality and occasionally accepting worse swaps to escape local minima.

Routing

After placement, the router connects the elements using the FPGA's programmable interconnect. The router must find a path for every net (signal) in the design, using the available routing resources without conflicts (two different signals cannot use the same wire segment).

The routing step is typically the most time-consuming part of the flow. For the SAMPO design on an ECP5-85F, nextpnr takes roughly 30–60 seconds for the complete place-and-route, though this varies with machine speed.

29.3.3 The nextpnr Timing Model

The Makefile specifies the `NEXTPNR_ECP5` variable with a fallback to the oss-cad-suite installation:

Listing 29.4: NEXTPNR_ECP5 variable from `verilog/Makefile`.

```
NEXTPNR_ECP5 ?= $(shell which nextpnr-ecp5 2>/dev/null \
    || echo $(HOME)/oss-cad-suite/bin/nextpnr-ecp5)
```

This accommodates both system-installed and locally-built toolchain versions.

nextpnr performs *static timing analysis* (STA) after placement and routing. It traces every combinational path between flip-flops and calculates the total propagation delay, accounting for:

- LUT delay (the time for inputs to propagate through the look-up table)
- Routing delay (wire propagation through switch matrices)
- Setup time (how early data must arrive before the clock edge)
- Clock skew (variation in clock arrival time across the chip)

The result is a *maximum frequency* (F_{max}) — the fastest clock at which the design can reliably operate.

29.4 ecppack: Bitstream Generation

The final transformation converts the text configuration file into a binary bitstream that the FPGA can load. ecppack is part of the Project Trellis toolkit and understands the ECP5's configuration format.

Listing 29.5: ecppack bitstream generation from `verilog/Makefile`.

```
$(BUILD_DIR)/sampo.bit: $(BUILD_DIR)/sampo.config
  ecppack --compress $< $@
```

The `--compress` flag enables bitstream compression, reducing the file size and programming time. The ECP5 supports transparent decompression during configuration, so there is no performance penalty.

A typical compressed bitstream for the SAMPO design is around 1–2 MB. The uncompressed bitstream for the ECP5-85F would be approximately 10 MB.

ecppack's job is straightforward: it takes the text-format configuration (which specifies the state of every configuration bit in the FPGA) and packs it into the binary format that the FPGA's configuration logic expects. The output `.bit` file is what openFPGALoader sends to the device over JTAG.

29.5 Reading Timing Reports

The timing report from nextpnr is one of the most important outputs of the FPGA build process. It tells you whether your design will work at the intended clock frequency — and if not, where the bottleneck is.

29.5.1 The Critical Path

The *critical path* is the longest combinational delay between any two flip-flops in the design. It determines the maximum clock frequency. For the SAMPO, nextpnr reports a critical path of approximately 81 ns, corresponding to $F_{max} \approx 12.34$ MHz.

A typical critical path report looks like this:

Listing 29.6: Example nextpnr critical path report (representative).

```
Info: Critical path report for clock 'clk_cpu':
Info: curr total
Info:  0.6  0.6  Source cpu_inst.state_SB_LUT4_I3_LC.Q
Info:  2.1  2.7  Net cpu_inst.state[1] (6,22) -> (7,24)
Info:  0.5  3.2  Sink cpu_inst.alu_a_SB_LUT4_I2_LC.I1
Info:  0.5  3.7  Source cpu_inst.alu_a_SB_LUT4_I2_LC.O
Info:  3.2  6.9  Net cpu_inst.alu_a[7] (7,24) -> (11,28)
...
Info: 81.0 81.0  Total path delay
Info: Max frequency for clock 'clk_cpu': 12.34 MHz
```

Each line shows:

- **curr**: The delay contribution of this element (in nanoseconds)
- **total**: Running total delay from the path start
- **Source/Sink**: The logic element (LUT, flip-flop) at each end
- **Net**: The routing path, with grid coordinates showing physical distance

29.5.2 Interpreting the Critical Path

The SAMPO's critical path runs through the execute stage of the CPU. Conceptually, it looks like this:

1. **State register output** → State decode logic
2. **Register file read** → ALU input multiplexer
3. **ALU computation** (LUTs + carry chain)

4. **Flag generation** from ALU result
5. **Result multiplexer** → Write-back data
6. **Setup time** at the destination flip-flop

This is a deep combinational path that spans the entire execution data flow. In a pipelined design, this path would be broken into multiple stages, each taking a single clock cycle. Our multi-cycle FSM design keeps it as one long path, which limits the clock frequency but simplifies the control logic.

29.5.3 Slack

Slack is the difference between the available clock period and the critical path delay. Positive slack means the design meets timing; negative slack means it does not.

$$\begin{aligned} \text{Slack} &= T_{\text{clk}} - T_{\text{critical}} \\ &= 80\,\text{ns} - 81\,\text{ns} \\ &= -1\,\text{ns} \end{aligned}$$

At 12.5 MHz (80 ns period), we have about -1 ns of negative slack. This means we are technically violating timing. In practice, the timing model is conservative, and the design works reliably at 12.5 MHz — but there is no margin for temperature or voltage variation.

Timing Violations in Practice

A small negative slack (a few nanoseconds) does not always cause immediate failures. The FPGA's timing model includes safety margins, and the actual silicon is often faster than the model predicts. However, relying on this is engineering by luck, not by design. For a production system, you would either reduce the clock frequency or pipeline the critical path.

29.6 Resource Utilization

Understanding how your design consumes FPGA resources is essential for planning expansion and identifying optimization opportunities.

29.6.1 Utilization Summary

After place and route, nextpnr prints a utilization summary:

Several observations:

Table 29.2: Typical SAMPO resource utilization on ECP5-85F.

Resource	Used	Available	Utilization
LUT4	~3,600	83,640	~4.3%
Flip-Flops	~1,250	83,640	~1.5%
Block RAM (EBR)	32	208	15.4%
PLL	0	4	0%
I/O Pins	18	365	4.9%

- **We are logic-light.** Using only 4% of the LUTs means the ECP5-85F is vastly oversized for our design. The smallest ECP5 (the 12F with 12,288 LUTs) would comfortably fit the SAMPO.
- **Block RAM is the densest resource.** At 15% utilization, the 64KB memory is our largest consumer of silicon resources. Reducing the memory to 32KB would halve this to 8%.
- **No PLLs used.** Our toggle divider does not require a PLL, leaving all 4 available for future use.
- **Flip-flop usage is modest.** 1,250 flip-flops cover the 16-register file (256 FFs for the main bank, 128 for the EXX shadow bank), the CPU state machine, the program counter, the cycle counter, the UART shift registers, and all other sequential state.

29.6.2 Where Do the LUTs Go?

Without a detailed per-module breakdown from the tools (which Yosys can provide with additional flags), we can estimate the distribution based on the design structure:

- **Instruction decoder** (~500–800 LUTs): The decoder in `rtl/decode.v` generates dozens of control signals from the instruction word. Each control signal is a Boolean function of the opcode, function code, and various bit fields.
- **CPU control logic** (~800–1,200 LUTs): The state machine, next-state logic, program counter update, flag update, and memory interface multiplexing in `rtl/cpu.v`.
- **ALU** (~300–500 LUTs): The 16-bit arithmetic and logic operations, including carry chains (counted separately as CCU2C cells).
- **UART TX + RX** (~200–300 LUTs): Baud rate generators, state machines, shift registers, and synchronizers.
- **Register file** (~200–400 LUTs): Distributed RAM for the register file, plus multiplexers for the two read ports and write port.
- **Shifter, reset sync, glue logic**: The remainder.

Per-Module Utilization

You can get a per-module resource breakdown by running Yosys with the `stat` command after synthesis but before flattening. This shows the resource count for each module separately, helping you identify which modules are the largest consumers.

29.7 The Makefile Build Pipeline

The `verilog/Makefile` orchestrates the entire FPGA build, from assembly to programming. Let us trace the complete pipeline.

29.7.1 Source File Definitions

The Makefile begins by defining the source files:

Listing 29.7: Source file definitions from `verilog/Makefile`.

```
# Directories
RTL_DIR = rtl
TB_DIR = tb
PROG_DIR = programs
FPGA_DIR = fpga
BUILD_DIR = build

# Source files
RTL_SRCS = $(RTL_DIR)/sampo_pkg.vh \
           $(RTL_DIR)/alu.v \
           $(RTL_DIR)/shifter.v \
           $(RTL_DIR)/regfile.v \
           $(RTL_DIR)/decode.v \
           $(RTL_DIR)/cpu.v \
           $(RTL_DIR)/ram.v \
           $(RTL_DIR)/uart.v \
           $(RTL_DIR)/soc.v

FPGA_SRCS = $(FPGA_DIR)/uart_tx.v \
            $(FPGA_DIR)/uart_rx.v \
            $(FPGA_DIR)/reset_sync.v \
            $(FPGA_DIR)/ulx3s_top.v
```

Notice the clean separation: `RTL_SRCS` contains the portable design files (used for both simulation and FPGA), while `FPGA_SRCS` contains the board-specific wrappers that only apply to the ULX3S target. The simulation testbenches in `tb/` never reference any FPGA-specific module.

29.7.2 FPGA Configuration

Listing 29.8: FPGA configuration from `verilog/Makefile`.

```
# FPGA synthesis settings
DEVICE = 85k
PACKAGE = CABGA381
LPF = $(FPGA_DIR)/ulx3s_v20.lpf
NEXTPNR_ECP5 ?= $(shell which nextpnr-ecp5 2>/dev/null \
    || echo $(HOME)/oss-cad-suite/bin/nextpnr-ecp5)
```

The `NEXTPNR_ECP5` variable uses a two-tier lookup: first try the system PATH, then fall back to the oss-cad-suite installation directory. This makes the Makefile work across different installation methods.

29.7.3 The Full FPGA Pipeline

The three FPGA build stages are chained through Make dependencies:

Listing 29.9: FPGA build targets from `verilog/Makefile`.

```
# Yosys synthesis
.PHONY: synth
synth: $(BUILD_DIR)/sampo.json

# nextpnr place & route
.PHONY: pnr
pnr: $(BUILD_DIR)/sampo.config

# ecppack bitstream generation
.PHONY: bit
bit: $(BUILD_DIR)/sampo.bit

# Upload to ULX3S via openFPGALoader
.PHONY: prog
prog: $(BUILD_DIR)/sampo.bit
  openFPGALoader -b ulx3s $<

# Full FPGA build pipeline
.PHONY: fpga
fpga: bit
```

Running `make fpga` triggers the entire chain:

Listing 29.10: Building and programming the SAMPO FPGA design.

```
# Build the bitstream (synthesis + place & route + pack)
make fpga

# Program the FPGA
make prog
```

The Make dependency system ensures that each stage only runs when its inputs have changed. If you modify `rtl/alu.v`, Make will re-run synthesis, place and route, and bitstream generation — but it will not re-assemble the program or re-run any simulation tests.

29.7.4 The Per-Test FPGA Pipeline

The Makefile also supports building FPGA images for individual ISA tests. This pipeline adds an extra step: patching the test program with `hex_loop_patch.py` so it runs in a continuous loop on the FPGA:

Listing 29.11: Per-test FPGA build rules from `verilog/Makefile`.

```
# Create looped hex file
$(PROG_DIR)/%_loop.hex: $(PROG_DIR)/%.hex
  python3 hex_loop_patch.py $< $@

# Synthesize FPGA with a looped test program
$(BUILD_DIR)/sampo_%.json: $(RTL_SRCS) $(FPGA_SRCS) \
    $(PROG_DIR)/%_loop.hex | $(BUILD_DIR)
  sed 's|programs/hello\.hex|programs/$*_loop.hex|g' \
    $(FPGA_DIR)/ulx3s_top.v > \
    $(BUILD_DIR)/ulx3s_top_$*.v
  yosys -p "read_verilog -I$(RTL_DIR) ..."
```

Notice the `sed` command: since the `RAM_INIT_FILE` parameter is hardcoded as `"programs/hello.hex"` in `fpga/ulx3s_top.v`, the Makefile creates a modified copy of the top-level module with the test program's hex file substituted. This is a pragmatic workaround — the alternative would be a more complex parameterization scheme.

29.7.5 Build, Program, and Capture

The combined `fpga-*` targets do everything in one step:

Listing 29.12: Combined FPGA test targets from `verilog/Makefile`.

```
.PHONY: $(addprefix fpga-,$(ISA_TESTS))
$(addprefix fpga-,$(ISA_TESTS)): fpga-%: $(BUILD_DIR)/sampo_%.bit
```

```
  @echo "=== Programming $* to FPGA ==="
  @openFPGALoader -b ulx3s $<
  @sleep 1
  @echo "=== Capturing UART output ==="
  @python3 fpga_capture.py $(SERIAL_PORT) $(SERIAL_BAUD) \
    5 > $(BUILD_DIR)/fpga_$*.log
  @cat $(BUILD_DIR)/fpga_$*.log
```

The `sleep 1` is critical: after programming, the FTDI chip needs a moment to release its JTAG hold and switch back to serial mode. Without this delay, the serial port open would fail.

The `fpga_capture.py` script captures one complete iteration of test output from the looping program, which we explore fully in Chapter 30.

29.7.6 Running All Tests

Listing 29.13: Running all FPGA tests from `verilog/Makefile`.

```
# Run all ISA tests on FPGA sequentially
.PHONY: fpga-tests
fpga-tests:
  @PASS=0; FAIL=0; SKIP=0; \
  for t in $(ISA_TESTS); do \
    echo "========================================"; \
    echo "=== FPGA: $$t ==="; \
    echo "========================================"; \
    if $(MAKE) --no-print-directory fpga-$$t; then \
      PASS=$$((PASS + 1)); \
    else \
      FAIL=$$((FAIL + 1)); \
    fi; \
  done; \
  echo "========================================"; \
  echo "FPGA Test Summary: $$PASS passed, $$FAIL failed"; \
  echo "========================================"; \
  [ $$FAIL -eq 0 ]
```

This target iterates through all 10 ISA test suites, building a separate FPGA bitstream for each test, programming it, and capturing the output. The full run takes approximately 30–45 minutes, dominated by the synthesis and place-and-route time for each test image.

The `ISA_TESTS` variable lists all the test suites:

Listing 29.14: ISA test list from `verilog/Makefile`.

```
ISA_TESTS = test_alu test_addi test_shift test_muldiv \
            test_loadstore test_branch test_jump \
            test_stack test_misc test_extended
```

We cover the FPGA test infrastructure and results in Chapter 31 and **??**.

Exercises

1. **Run synthesis.** Build the SAMPO FPGA image by running `make synth` from the `verilog/` directory. Examine the Yosys output and find the resource summary. How many LUT4s, flip-flops, and block RAMs does your build report? How does it compare to the estimates in this chapter?

2. **Timing analysis.** Run `make pnr` and find the maximum frequency (F_{max}) reported by nextpnr. What is the critical path delay? Try to identify which module the critical path passes through by examining the signal names in the timing report.

3. **Reduce the memory.** Modify `rtl/ram.v` to implement only 16KB of memory instead of 64KB (change the array size from `32767` to `8191` and the address width accordingly). Re-run synthesis and compare the block RAM count. Does the LUT count change? Why or why not?

4. **Smallest ECP5.** Modify the Makefile to target the ECP5-12F instead of the 85F (change `--85k` to `--12k`). Does the design still fit? If not, which resource is exhausted first?

5. **Explore the JSON netlist.** Open the `build/sampo.json` file in a text editor. Find the entry for the CPU's program counter flip-flops. What does the JSON representation of a LUT4 look like? How is the truth table encoded?

What's Next

We now have a bitstream on disk, ready to be loaded onto the FPGA. In Chapter 30, we program the ULX3S for the first time, open the serial port, and watch the SAMPO CPU execute real code on real hardware. Along the way, we will encounter — and solve — the looping problem, the address-shift bug, and the subtleties of J/JR/JALR encoding conflicts that only appear when you run test programs on physical silicon.

Chapter 30

Running on Real Hardware

There is a moment in every CPU design project when the simulation is done, the bitstream is built, and the FPGA is sitting on the desk, waiting. You press the button. The LEDs flicker. The serial port opens. And either your CPU works — or it does not.

This chapter covers that moment and everything that follows. We will program the FPGA, open the serial port, and watch "Hello, Sampo!" scroll across the terminal. Then we will confront the real problems that only appear on hardware: the looping problem (how do you test a CPU that halts after one run?), the J/JR/JALR encoding conflicts (certain jump offsets silently decode as wrong instructions), and the pragmatic solution that makes FPGA testing reliable.

30.1 Programming the FPGA

With the bitstream built (via `make fpga` as described in Chapter 29), programming the ULX3S is a single command:

Listing 30.1: Programming the ULX3S.

```
make prog
```

This runs:

Listing 30.2: The underlying openFPGALoader command.

```
openFPGALoader -b ulx3s build/sampo.bit
```

The programming process takes about 2–3 seconds. You will see output like:

Listing 30.3: Typical openFPGALoader output.

```
write to ram
Open file: done
Parse file: done
Enable Memory access: done
Writing: [========================================] 100.00%
Done
```

The moment `Done` appears, the FPGA is configured and your SAMPO CPU begins executing from the reset vector at `0x0100`. If the programmed image contains `programs/hello.hex`, the CPU will execute the Hello World program immediately — printing its output to the UART and then halting.

30.1.1 Observing the LEDs

The first sign of life is the LEDs. Recall from Section 28.5.1 that the SoC maps status information to the 8 LEDs:

Listing 30.4: LED status from `rtl/soc.v`.

```
// LED[7]:   CPU halted
// LED[6]:   UART TX active (data being sent)
// LED[5]:   UART RX data available
// LED[4:0]: Lower 5 bits of PC (shows execution progress)
assign led = {halted, tx_valid, rx_valid, pc[5:1]};
```

What you should see after programming:

1. A brief flicker across LEDs 4–0 as the CPU fetches and executes instructions.
2. LED 6 blinking as UART data is transmitted.
3. LED 7 lighting up and staying on — the CPU has executed **`HALT`**.
4. LEDs 4–0 frozen at whatever pattern corresponds to the **`HALT`** instruction's address.

This entire sequence happens in well under a second at 12.5 MHz. The LED flicker may be too fast to see with the naked eye; the halted LED (LED 7) staying solidly on is the definitive indicator.

If LED 7 Does Not Light Up

If LED 7 never comes on after programming, the CPU did not reach the **`HALT`** instruction. Common causes: the program counter is stuck (check that the reset vector is correct), the program is in an infinite loop, or the clock is not running. Try connecting LEDs to a free-running counter to verify the clock is alive.

30.2 Opening the Serial Port

The ULX3S's FTDI FT231X appears as a virtual serial port on the host computer. The exact device name depends on your operating system:

Table 30.1: Serial port device names by operating system.

OS	Device	Example
macOS	`/dev/cu.usbserial-*`	`/dev/cu.usbserial-D01263`
Linux	`/dev/ttyUSB*`	`/dev/ttyUSB0`
Windows	`COM*`	`COM3`

The Makefile defines the serial port as a configurable variable:

Listing 30.5: Serial port configuration from `verilog/Makefile`.

```
# Serial port for FPGA UART output
SERIAL_PORT ?= /dev/cu.usbserial-D01263
SERIAL_BAUD ?= 115200
```

You can override this at the command line:

Listing 30.6: Overriding the serial port.

```
make fpga-test_alu SERIAL_PORT=/dev/ttyUSB0
```

To open the serial port manually for interactive use, any terminal program works:

Listing 30.7: Opening the serial port with common tools.

```
# macOS / Linux: screen
screen /dev/cu.usbserial-D01263 115200

# macOS / Linux: picocom (better for scripting)
picocom -b 115200 /dev/cu.usbserial-D01263

# macOS / Linux: minicom
minicom -D /dev/cu.usbserial-D01263 -b 115200
```

Close the Serial Port Before Programming

The FTDI chip shares its USB connection between JTAG (programming) and serial (UART). If a terminal program has the serial port open, openFPGALoader may fail to access the JTAG interface. Always close your terminal session before running `make prog`.

30.3 Hello, Sampo! on Real Hardware

With the FPGA programmed and the serial port open, the moment of truth arrives. If everything is working, you see:

Listing 30.8: First output from the SAMPO on real hardware.

```
Hello, Sampo!
```

Thirteen characters, transmitted one at a time over the UART at 115,200 baud. Each character takes about 87 μs to transmit, so the entire string appears in about 1.1 ms — effectively instantaneous from a human perspective.

But there is a catch.

30.4 The Looping Problem

The Hello World program, like all our test programs, ends with **HALT**:

Listing 30.9: End of a typical SAMPO test program.

```
done:
    HALT
```

Once the CPU halts, it stays halted. The state machine enters HALTED and remains there:

Listing 30.10: Halted state from `rtl/cpu.v`.

```
`ST_HALTED: begin
    next_state = `ST_HALTED;  // Stay halted
end
```

This creates a problem for FPGA testing. Consider the sequence:

1. Program the FPGA with the test bitstream
2. The FPGA configures and the CPU starts immediately

3. The test program runs, outputs results to UART, and halts
4. We open the serial port…but the output has already been sent and lost

The FTDI chip buffers a small amount of data, but there is no guarantee that the serial port will be opened in time to catch the output. With the programming step taking 2–3 seconds and the test program completing in milliseconds, we are racing against the hardware — and losing.

30.4.1 Why Not Just Open the Port First?

Opening the serial port *before* programming does not help, because the FTDI chip is occupied with JTAG during programming. The serial interface is unavailable until programming completes.

30.4.2 Why Not Use a Hardware Trigger?

We could add a mechanism where the CPU waits for a button press before starting, or holds in reset until the serial port is ready. But this adds complexity to every test program and requires manual intervention — not suitable for an automated test suite.

30.4.3 The Solution: Make It Loop

The solution is elegant: patch the test program so that instead of halting, it pauses for about half a second and then restarts from the reset vector. The test runs in a continuous loop, outputting its results over and over. We can open the serial port at any time and capture one complete iteration.

This is exactly what `hex_loop_patch.py` does.

30.5 hex_loop_patch.py: Replacing HALT with a Delay Loop

The `hex_loop_patch.py` script performs a surgical modification of the program's hex file. It does two things:

1. Places a delay loop in the unused memory space at address `0x0000` (before the reset vector at `0x0100`)
2. Replaces the **`HALT`** instruction with a **`J`** instruction that jumps to the delay loop

30.5.1 The Delay Loop

The delay loop is a nested counter that burns about 0.38 seconds at 12.5 MHz before jumping back to the reset vector:

Listing 30.11: Delay loop injected by `hex_loop_patch.py`.

```
; Placed at address 0x0000 (unused space before reset vector)
```

```
    LIX R8, 0x0008       ; Outer counter = 8
outer:
    LIX R9, 0xFFFF       ; Inner counter = 65535
inner:
    ADDI R9, -1           ; R9 = R9 - 1
    BNE inner             ; Loop until R9 == 0
    ADDI R8, -1           ; R8 = R8 - 1
    BNE outer             ; Loop until R8 == 0
    JX 0x0100             ; Restart at reset vector
```

The total delay is approximately:

$$8 \times 65{,}535 \times 9 \text{ cycles} \approx 4.7 \text{ million cycles} = 0.38 \text{ s at } 12.5 \text{ MHz}$$

This is long enough for the UART to finish transmitting any buffered data before the test restarts.

In the Python source, the delay loop is encoded as raw instruction words:

Listing 30.12: Delay loop encoding from `hex_loop_patch.py`.

```
LOOP_PATCH = [
    0xF807, 0x0008,  # LIX R8, 0x0008
    0xF907, 0xFFFF,  # LIX R9, 0xFFFF
    0x59FF,          # ADDI R9, -1
    0x81FE,          # BNE -2 (inner)
    0x58FF,          # ADDI R8, -1
    0x81FA,          # BNE -6 (outer)
    0xF008, 0x0100,  # JX 0x0100
]
```

30.5.2 Why Address `0x0000`?

The SAMPO reset vector is at `0x0100`, leaving 256 bytes (128 words) of address space before the program starts. The hex file for a typical test program has all zeros in this region. The delay loop fits in just 10 words, well within the available space.

By placing the loop at address `0x0000` rather than appending it after the program, the patch avoids shifting any program data. This is critical because the program may contain absolute addresses (e.g., **`LIX`** immediates pointing to string data). If we shifted the program, those addresses would become invalid.

30.5.3 Finding and Replacing HALT

The script searches for the **HALT** instruction by its encoding:

Listing 30.13: HALT detection from `hex_loop_patch.py`.

```
# HALT instruction encoding: opcode=0xE (SYSTEM), rd=1 (SYS_HALT)
HALT_WORD = 0xE100

# Find HALT instruction
halt_idx = None
for i, w in enumerate(words):
    if w == HALT_WORD:
        halt_idx = i
        break
```

The **HALT** instruction has opcode `0xE` (`OP_SYSTEM`) with the function field set to `0x1` (`SYS_HALT`), giving the encoding `0xE100`.

Once found, the **HALT** is replaced with a **J** instruction that jumps to the delay loop at address `0x0000`.

30.6 The Address-Shift Bug: The Full Detective Story

Replacing **HALT** with a **J** instruction sounds straightforward, but it uncovered one of the most subtle bugs in the SAMPO ISA design: certain **J** offset values silently decode as **JR** or **JALR** instructions.

30.6.1 The Encoding Conflict

The SAMPO uses opcode `0x9` for three different instructions:

Table 30.2: Opcode `0x9` instruction variants.

Instruction	Encoding	Decode Rule
J offset12	`1001` *offset12*	Default case (when bit patterns do not match JR or JALR)
JR Rs	`1001 1111` *Rs* `0000`	bits[11:8] = `0xF` AND bits[3:0] = `0x0`
JALR Rd, Rs	`1001` *Rd Rs* `0001`	bits[3:0] = `0x1` AND bits[11:8] $\neq$ `0xF`

The decoder in `rtl/decode.v` implements these priority rules:

Listing 30.14: Jump instruction decoding from `rtl/decode.v`.

```
`OP_JUMP: begin
```

```
    // Check if it's JR/JALR
    if (rd == 4'hF && (func == 4'h0)) begin
        // JR Rs1
        is_jump_reg = 1'b1;
        is_ret = (rs1 == `REG_RA);  // JR R1 is return
    end else if (func == 4'h1 && rd != 4'hF) begin
        // JALR Rd, Rs1
        is_jump_reg = 1'b1;
        is_call = 1'b1;
        reg_write = 1'b1;
    end else begin
        // J offset12
        is_jump = 1'b1;
    end
end
```

30.6.2 The Problem

The 12-bit offset field in a **J** instruction overlaps with the *Rd*, *Rs*, and *func* fields used by **JR** and **JALR**. This means certain jump offsets are *impossible to encode* — they would be decoded as a different instruction entirely.

Consider a **J** instruction targeting address `0x0000` from an address near the end of a test program. If the computed 12-bit offset happens to have:

- bits[11:8] = `0xF` and bits[3:0] = `0x0` $\Rightarrow$ decoded as **JR**, not **J**
- bits[3:0] = `0x1` and bits[11:8] $\neq$ `0xF` $\Rightarrow$ decoded as **JALR**, not **J**

In either case, the CPU would not jump to the delay loop. Instead, it would attempt to jump to whatever garbage value is in a register, or worse, write a return address to a register and jump to an unintended location.

30.6.3 The Detective Work

This bug manifests as a program that works in simulation but fails on the FPGA — the most frustrating kind of failure. In simulation, the test program always uses **HALT** and never needs the loop patch. The conflict only appears when the patch script generates a **J** instruction with a "forbidden" offset.

The symptoms are subtle:

- The test program runs once and then *appears* to halt (LED 7 stays on)
- But the CPU has not actually halted — it jumped to an invalid address and crashed

- The UART output is correct for the first iteration but never repeats
- Different test programs exhibit different behavior because the offset varies with program size

30.6.4 The Fix: Conflict Avoidance

The `hex_loop_patch.py` script solves this by checking whether the computed **`J`** encoding would conflict with **`JR`** or **`JALR`**:

Listing 30.15: J encoding with conflict checking from `hex_loop_patch.py`.

```
def encode_j_offset(halt_addr, target_addr):
    """Encode a J (relative jump) instruction.

    J instruction: opcode=0x9, offset in bits[11:0]
    Target = PC + 2 + (sign_extend(offset[11:0]) << 1)
    """
    # offset_field = (target - PC - 2) / 2
    raw_offset = (target_addr - halt_addr - 2) // 2

    # Check range: 12-bit signed = [-2048, 2047]
    if raw_offset < -2048 or raw_offset > 2047:
        return None

    offset_12 = raw_offset & 0xFFF

    # Check for JR/JALR decode conflicts:
    rd_bits = (offset_12 >> 8) & 0xF
    func_bits = offset_12 & 0xF

    if rd_bits == 0xF and func_bits == 0x0:
        return None  # Would decode as JR
    if func_bits == 0x1 and rd_bits != 0xF:
        return None  # Would decode as JALR

    return 0x9000 | offset_12
```

If the offset produces a conflict, the script tries the next word offset for the delay loop placement:

Listing 30.16: Loop base search from `hex_loop_patch.py`.

```
# Place the delay loop in unused space at address 0x0000.
# Try word offsets 0, 1, 2, ... until we find one where the
# J instruction doesn't conflict with JR/JALR decoding.
j_word = None
```

```
loop_base = 0
for loop_base in range(0, 64):
    target_addr = loop_base * 2
    j_word = encode_j_offset(halt_addr, target_addr)
    if j_word is not None:
        break
```

By sliding the delay loop's starting address by one word, the offset changes by 2 (because the target address shifts by 2 bytes). This is enough to avoid both conflict patterns in almost all cases.

30.6.5 The Offset Calculation

The **J** instruction encodes a PC-relative offset:

$$\text{Target} = \text{PC} + 2 + \text{sign_extend}(\text{offset}_{12}) \times 2$$

Solving for the offset field:

$$\text{offset}_{12} = \frac{\text{Target} - \text{PC} - 2}{2}$$

Since the delay loop is at address `0x0000` and the **HALT** could be anywhere in the program, the offset is typically a large negative number. The 12-bit signed offset field supports a range of -2048 to $+2047$, corresponding to a target range of -4096 to $+4094$ bytes from the current PC. Programs larger than about 4KB might not be reachable with a 12-bit relative jump, in which case the script would need to use the 32-bit **JX** format instead.

Why Not Always Use JX?

The extended **JX** instruction (`0xF008` followed by a 16-bit immediate address) avoids the encoding conflict entirely because it uses opcode `0xF` and stores the absolute target in the second word. However, **JX** is a 32-bit instruction, requiring 4 bytes instead of 2. For the delay loop (which already uses **JX** to jump to `0x0100`), this is fine. For the **HALT** replacement, using a 16-bit **J** avoids the need to shift subsequent program data.

30.7 Clock Divider Selection

The choice of CPU clock frequency involves a three-way trade-off between performance, timing margin, and debugging convenience.

30.7.1 The Timing Constraint

The nextpnr timing report tells us the critical path through the SAMPO takes approximately 81 ns, corresponding to a maximum frequency of about 12.34 MHz. Our toggle divider produces 12.5 MHz from the 25 MHz oscillator, giving an 80 ns clock period.

This is tight — technically negative slack of about −1 ns. In practice, the design works, but there is no room for adverse conditions.

30.7.2 Divider Options

With a simple toggle-divider chain from the 25 MHz oscillator, the available frequencies are:

Table 30.3: Available clock frequencies with toggle dividers.

Divider	Frequency	Period	Timing Margin
÷1	25.0 MHz	40 ns	−41 ns (fails)
÷2	12.5 MHz	80 ns	−1 ns (tight)
÷4	6.25 MHz	160 ns	+79 ns (comfortable)
÷8	3.125 MHz	320 ns	+239 ns (generous)

The ÷2 option (12.5 MHz) gives the best performance but the least margin. The ÷4 option (6.25 MHz) would halve the performance but provide substantial margin, making the design robust against temperature and voltage variation.

30.7.3 PLL-Based Alternatives

With a PLL, we could generate any frequency in the ECP5's output range. Useful options for the SAMPO include:

- **10 MHz** — Clean frequency, comfortable 19 ns of margin, good for production use.
- **8 MHz** — Conservative choice with 44 ns of margin. Useful during initial bring-up.
- **12 MHz** — Close to the maximum but with 2.3 ns of positive margin (unlike 12.5 MHz).

For the current design, the toggle divider at 12.5 MHz works. If we encounter reliability issues (random failures that change with temperature or between FPGA units), dropping to ÷4 or adding a PLL at 10 MHz would be the first remediation step.

30.7.4 The Impact on UART Timing

Changing the CPU clock frequency affects the UART baud rate divisor. The `CLK_FREQ` parameter propagates from the top-level module to both UART instances:

Listing 30.17: Clock frequency propagation from `fpga/ulx3s_top.v`.

```
localparam CLK_FREQ  = 12_500_000;
localparam BAUD_RATE = 115200;

uart_tx #(
    .CLK_FREQ(CLK_FREQ),
    .BAUD_RATE(BAUD_RATE)
) u_uart_tx (
    // ...
);

uart_rx #(
    .CLK_FREQ(CLK_FREQ),
    .BAUD_RATE(BAUD_RATE)
) u_uart_rx (
    // ...
);
```

If you change the clock frequency, update `CLK_FREQ` accordingly. An incorrect `CLK_FREQ` will produce garbled serial output because the baud rate generator will count the wrong number of clock cycles per bit.

At 12.5 MHz, the UART divides by 108 to produce approximately 115,741 baud — a 0.47% error from the target 115,200 baud. UART communication tolerates up to about 2–3% baud rate error, so this is well within the acceptable range.

At 6.25 MHz (÷4), the divider would be 54, producing 115,741 baud with the same 0.47% error. The lower clock frequency does not affect baud rate accuracy because the ratio `CLK_FREQ / BAUD_RATE` changes proportionally.

Baud Rate Accuracy at Low Clock Frequencies

At very low clock frequencies, the integer division `CLK_FREQ / BAUD_RATE` produces a coarser approximation. If you drop to 3.125 MHz, the divider is 27, producing 115,741 baud (still 0.47% error — acceptable). But at 1 MHz, the divider is only 8, producing 125,000 baud (8.5% error — too high for reliable communication). Keep the CPU clock at least 10× the baud rate for accurate UART timing.

30.7.5 Capturing UART Output

The `fpga_capture.py` script automates the process of capturing one complete test iteration from the looping FPGA program. It uses a clever heuristic: look for a "header line" (matching the pattern `=== ... ===`), collect all output until the header repeats, and stop.

Listing 30.18: Capture strategy from `fpga_capture.py`.

```
HEADER_RE = re.compile(r'^===.*===\s*$')

def main():
    port = sys.argv[1]
    baud = int(sys.argv[2]) if len(sys.argv) > 2 else 115200
    timeout_sec = float(sys.argv[3]) if len(sys.argv) > 3 else 8

    ser = serial.Serial(port, baud, timeout=2)
    ser.reset_input_buffer()

    # Discard first partial line (we likely joined mid-stream)
    ser.readline()

    lines = []
    header_line = None
    start = time.time()

    while time.time() - start < timeout_sec:
        raw = ser.readline()
        if not raw:
            if lines:
                break  # Timeout after getting some data = done
            continue

        line = raw.decode('ascii', errors='replace').rstrip('\r\n')

        # Detect header line
        if HEADER_RE.match(line):
            if header_line is None:
                # First header - start of a clean iteration
                header_line = line
                lines = [line]
            else:
                # Second header - we have one complete iteration
                break
        elif header_line is not None:
            lines.append(line)
```

The strategy handles the most common edge cases:

1. **Joining mid-stream:** The first `readline()` call discards whatever partial line was in the buffer when we opened the port. This avoids starting capture mid-output.

2. **First header detection:** Once we see the header pattern (e.g., `=== test_alu ===`), we know we are at the start of a clean iteration.
3. **Second header = done:** When the header appears again, the test has looped. We have one complete iteration's output.
4. **Timeout fallback:** If the header never repeats (perhaps the delay loop is longer than expected), the script falls back to a configurable timeout (default 8 seconds).

The Makefile ties everything together:

Listing 30.19: FPGA test target from `verilog/Makefile`.

```
$(addprefix fpga-,$(ISA_TESTS)): fpga-%: $(BUILD_DIR)/sampo_%.bit
  @echo "=== Programming $* to FPGA ==="
  @openFPGALoader -b ulx3s $<
  @sleep 1
  @echo "=== Capturing UART output ==="
  @python3 fpga_capture.py $(SERIAL_PORT) $(SERIAL_BAUD) \
    5 > $(BUILD_DIR)/fpga_$*.log
  @cat $(BUILD_DIR)/fpga_$*.log
  @echo ""
  @if grep -q FAIL $(BUILD_DIR)/fpga_$*.log 2>/dev/null; then \
    echo "*** FAILURES DETECTED ***"; false; \
  elif grep -q PASS $(BUILD_DIR)/fpga_$*.log 2>/dev/null; then \
    echo "All tests passed!"; \
  else \
    echo "WARNING: No test output captured"; \
  fi
```

The target:

1. Programs the FPGA with the test-specific bitstream
2. Waits 1 second for the FTDI to release JTAG and switch to serial mode
3. Captures 5 seconds of UART output
4. Checks the captured output for PASS/FAIL markers

Exercises

1. **First program.** Build and program the Hello World example to the ULX3S using `make fpga && make prog`. Open the serial port and observe the output. Note the LED state after the CPU halts. Press the reset button (`btn[0]`) and observe the LEDs and serial output again.

2. **J encoding conflict analysis.** Consider a test program where the **HALT** instruction is at byte address `0x0142`. Calculate the 12-bit offset needed for a **J** instruction from `0x0142` to `0x0000`. Extract the bits[11:8] and bits[3:0] from this offset. Does this encoding conflict with **JR** or **JALR**? If so, what target address would the delay loop need to move to in order to avoid the conflict?

3. **Delay loop timing.** The delay loop uses $8 \times 65{,}535$ inner iterations. Each inner iteration is approximately 9 cycles (**ADDI** + **BNE** through the FSM). At 12.5 MHz, how many seconds does the delay take? Modify the outer counter value to produce a 1-second delay instead.

4. **Clock divider experiment.** Modify `fpga/ulx3s_top.v` to use a $\div 4$ divider (add a second toggle stage) instead of $\div 2$. Update `CLK_FREQ` to 6,250,000. Rebuild, program, and verify that the serial output is still correct. Does the LED blinking pattern look any different at the slower clock speed?

5. **Custom LED pattern.** Modify the LED assignment in `rtl/soc.v` to display the upper 5 bits of the PC instead of the lower 5 bits. Rebuild and program. How does the LED pattern change during program execution? What information does this provide that the lower bits do not?

What's Next

The SAMPO is running on real hardware, printing serial output, and we have tools to test it reliably. But running one test is different from running all of them. In Chapter 31, we build the infrastructure for automated FPGA testing — running the entire ISA test suite on the ULX3S, capturing every result, and verifying that 132 tests produce zero failures.

Part IX

Hardware Testing on the FPGA

Chapter 31

FPGA Test Infrastructure

Simulation tells you that your design *should* work. Hardware tells you whether it *actually* does. The gap between those two statements is where the most insidious bugs hide—timing-dependent glitches, synthesis artifacts, and assumptions that only hold in the idealized world of `$display` and waveform viewers. Bridging that gap requires more than plugging in the FPGA and hoping for the best. It requires a repeatable, automated pipeline that can build a per-test bitstream, program the board, capture the UART output over a serial port, and parse the results—all with a single `make` command.

This chapter walks through every piece of that pipeline. We will examine the Python scripts, the Makefile targets, the hex-patching trick that turns a one-shot test into a looping FPGA program, and the serial-port handling that wrestles the FTDI chip into cooperation. By the end, you will be able to run any of the SAMPO ISA test suite files on real ECP5 hardware and get a definitive PASS or FAIL—no waveform viewer required.

31.1 The FPGA Testing Pipeline

Running a test on the FPGA is more involved than running one in simulation. In simulation, the testbench in `tb/sampo_tb.v` instantiates the SoC, loads a hex file, and watches the UART output directly through Verilog `$write` statements. On hardware, every one of those steps becomes a real engineering problem.

The pipeline looks like this:

1. **Assemble.** The SAMPO assembler compiles `.s` source to a `.bin` binary.
2. **Convert.** The `bin2hex.py` script converts the binary to the Verilog hex format expected by `$readmemh`.

3. **Patch.** The `hex_loop_patch.py` script replaces the **`HALT`** instruction with a jump back to a delay loop, so the test program runs continuously.

4. **Synthesize.** Yosys reads the RTL and FPGA wrapper sources, with the patched hex file baked into the RAM initialization, and produces a JSON netlist.

5. **Place & Route.** `nextpnr-ecp5` maps the netlist onto the ECP5-85F and generates a text configuration.

6. **Pack.** `ecppack` converts the configuration to a binary bitstream.

7. **Program.** `openFPGALoader` uploads the bitstream to the ULX3S board over JTAG.

8. **Capture.** The `fpga_capture.py` script opens the serial port, waits for a complete test iteration, and writes the UART output to a log file.

9. **Parse.** A few lines of shell in the Makefile grep the log for `PASS` and `FAIL` and report the result.

Why Not Reuse the Simulation Testbench?

The simulation testbench (`tb/sampo_tb.v`) captures UART output by watching internal signals—`tx_valid`, `tx_data`—at the SoC's byte-level interface. On hardware, those signals do not exist at the board boundary. The only way to observe the CPU's output is through the physical UART pin (`ftdi_rxd`) connected to the FTDI USB-serial chip. The entire FPGA test infrastructure exists to replace the testbench's direct signal observation with serial-port capture.

The key insight that makes this pipeline practical is the *looping patch*. A test program that runs once and halts is almost impossible to capture reliably over a serial port, because the program starts executing the instant the FPGA is configured—before the host can even open the serial port. By patching the program to loop with a half-second delay, we can program the FPGA, wait for USB re-enumeration, open the serial port at leisure, and capture the *next* complete iteration.

31.1.1 End-to-End Timing

The entire pipeline—from assembly source to captured result—takes roughly 60–90 seconds per test on a typical machine. The bottleneck is FPGA synthesis and place-and-route, not the serial capture. Here is an approximate breakdown:

For the full suite of 10 test files, that adds up to roughly 10–15 minutes of wall-clock time. Not fast—but entirely automated. We will discuss strategies for reducing this in Section 31.5.

Table 31.1: FPGA test pipeline timing breakdown

Stage	Time	Tool
Assembly	<1 s	`sasm` (Rust)
Hex conversion + patching	<1 s	Python scripts
Yosys synthesis	15–25 s	`yosys`
Place & route	30–50 s	`nextpnr-ecp5`
Bitstream packing	<1 s	`ecppack`
FPGA programming	2–3 s	`openFPGALoader`
USB settle + capture	3–6 s	`fpga_capture.py`

31.2 fpga_capture.py: Serial Port Communication

The `fpga_capture.py` script is the workhorse that reads test output from the ULX3S board's serial port. Its job sounds deceptively straightforward: open a serial port, read some text, print it. The complication is that the FPGA is already running when we connect—we are joining a stream mid-sentence.

31.2.1 The Iteration Detection Strategy

Every test program prints a header line that matches the pattern `=== ... ===`. For example, `test_alu.s` prints:

Listing 31.1: Test header format in the ALU test

```
str_header:
        .asciz "=== ALU Tests ===\n"
```

Because the looping patch causes the test to restart after a delay, this header repeats at the beginning of every iteration. The capture script uses this as a framing signal:

1. Discard the first partial line (we likely connected mid-output).
2. Read lines until we see a header matching `=== ... ===`.
3. Collect all subsequent lines as part of this iteration.
4. When we see the header a *second* time, stop—one complete iteration is captured.

Here is the core of `fpga_capture.py`:

Listing 31.2: fpga_capture.py: iteration detection loop

```
HEADER_RE = re.compile(r'^===.*===\s*$')

def main():
    if len(sys.argv) < 2:
        print("Usage: fpga_capture.py <serial_port> "
              "[baud] [timeout_sec]", file=sys.stderr)
        sys.exit(1)

    port = sys.argv[1]
    baud = int(sys.argv[2]) if len(sys.argv) > 2 else 115200
    timeout_sec = float(sys.argv[3]) \
        if len(sys.argv) > 3 else 8

    try:
        ser = serial.Serial(port, baud, timeout=2)
    except serial.SerialException as e:
        print(f"Cannot open {port}: {e}",
              file=sys.stderr)
        sys.exit(2)

    ser.reset_input_buffer()

    # Discard first partial line
    ser.readline()

    lines = []
    header_line = None
    start = time.time()

    while time.time() - start < timeout_sec:
        raw = ser.readline()
        if not raw:
            if lines:
                break  # Timeout after some data = done
            continue

        line = raw.decode('ascii',
                          errors='replace').rstrip('\r\n')
        if not line:
            continue

        # Detect header line
        if HEADER_RE.match(line):
            if header_line is None:
                # First header: start of clean iteration
```

```
                header_line = line
                lines = [line]
            else:
                # Second header: iteration complete
                break
        elif header_line is not None:
            lines.append(line)

    ser.close()

    if lines:
        for l in lines:
            print(l)
    else:
        print("WARNING: No output captured",
              file=sys.stderr)
        sys.exit(2)
```

31.2.2 Command-Line Interface

The script accepts three arguments:

Listing 31.3: fpga_capture.py usage

```
python3 fpga_capture.py <serial_port> [baud] [timeout_sec]
```

Table 31.2: fpga_capture.py arguments

Argument	Default	Description
`serial_port`	(required)	Device path, e.g. `/dev/cu.usbserial-D01263`
`baud`	115200	Baud rate matching the FPGA UART
`timeout_sec`	8	Maximum seconds to wait for output

The default baud rate of 115200 matches the hardware UART configured in `fpga/ulx3s_top.v`:

Listing 31.4: UART baud rate in the FPGA top module

```
    localparam CLK_FREQ  = 12_500_000;
    localparam BAUD_RATE = 115200;
```

Baud Rate Mismatch

If the capture script's baud rate does not match the hardware UART's baud rate, you will get garbled output—random bytes with occasional recognizable fragments. The symptom is unmistakable: the output looks like line noise rather than test results. Always verify that both sides agree on the baud rate. The SAMPO FPGA design uses 115200 baud, 8N1.

31.2.3 The Per-Read Timeout

The `serial.Serial` constructor is called with `timeout=2`, which sets a per-read timeout of 2 seconds. This is separate from the overall timeout of 8 seconds controlled by the `while` loop. The per-read timeout ensures that `ser.readline()` does not block indefinitely if the FPGA stops producing output (for example, if the program enters a tight loop that never outputs anything).

The interaction between the two timeouts is:

- If the FPGA is producing output, `readline()` returns quickly with each line, and the outer loop collects lines until the second header or the overall timeout.
- If the FPGA stops mid-iteration, `readline()` returns empty after 2 seconds. If we already have some data, we treat the silence as the end of the test and break. If we have no data yet, we keep waiting up to the overall timeout.

Increasing the Timeout for Slow Tests

Tests that produce a lot of UART output (like `test_branch.s` with its 24 individual test cases) may take longer than the default 8 seconds to complete a full iteration through the looping patch. If you see truncated output, try increasing the timeout:

```
python3 fpga_capture.py /dev/cu.usbserial-D01263 115200 15
```

The Makefile passes a 5-second timeout by default, which is sufficient for all the standard ISA tests.

31.3 fpga_test.py: Automated Test Execution

While `fpga_capture.py` handles the serial port side, `fpga_test.py` manages the full cycle: programming the FPGA, waiting for it to settle, and capturing the output. This script deals with one of the trickiest aspects of FPGA testing: the FTDI chip.

31.3.1 The FTDI Problem

The ULX3S board uses an FTDI FT231X chip that provides both the JTAG interface (for FPGA programming) and the USB-serial interface (for UART communication). These two functions share the same USB device. When `openFPGALoader` programs the FPGA over JTAG, the FTDI chip's USB interface resets, which causes the serial port device to momentarily disappear and reappear.

This creates a race condition:

1. We program the FPGA. The bitstream loads and the CPU starts executing.
2. The FTDI chip resets its USB interface. The serial port vanishes from the host OS.
3. The serial port re-enumerates (typically within 0.5–1 second).
4. We open the serial port. But by now, the test program may have already printed its output and halted.

31.3.2 Two Strategies for Capture

The `fpga_test.py` script employs two strategies to deal with this timing problem. The first is a DTR toggle, and the second is reprogramming.

Strategy 1: DTR Reset. After programming and waiting for USB re-enumeration, the script opens the serial port and toggles the DTR (Data Terminal Ready) signal:

Listing 31.5: DTR toggle in fpga_test.py

```
def capture_output(port_name, baud=115200,
                   timeout_sec=3):
    """Open serial port and capture output."""
    try:
        ser = serial.Serial(port_name, baud,
                            timeout=timeout_sec)
    except serial.SerialException as e:
        print(f"Cannot open {port_name}: {e}",
              file=sys.stderr)
        return None

    # Flush any stale data
    ser.reset_input_buffer()

    # Toggle DTR to try to reset the board
    ser.dtr = False
    time.sleep(0.05)
    ser.dtr = True
```

```
    time.sleep(0.1)

    # Read output until timeout
    output = b''
    start = time.time()
    while time.time() - start < timeout_sec:
        data = ser.read(256)
        if data:
            output += data
        elif output:
            # Got some data then silence
            time.sleep(0.2)
            data = ser.read(256)
            if data:
                output += data
            else:
                break

    ser.close()
    return output
```

On some board revisions, toggling DTR resets the FPGA, which restarts the test program from the beginning. This is the preferred approach because it avoids the cost of reprogramming.

Strategy 2: Reprogram. If the DTR approach fails to produce output containing `PASS` or `FAIL`, the script falls back to reprogramming the FPGA. The key trick is to open the serial port *before* reprogramming:

Listing 31.6: Reprogram fallback in fpga_test.py

```
    # Open serial port BEFORE programming
    try:
        ser = serial.Serial(port_name, 115200,
                            timeout=5)
        ser.reset_input_buffer()
    except serial.SerialException as e:
        print(f"Cannot open {port_name}: {e}",
              file=sys.stderr)
        sys.exit(1)

    # Program again - this will reset the FPGA
    if not program_fpga(bitstream):
        ser.close()
        sys.exit(1)
```

```
    # Capture output
    output = b''
    start = time.time()
    while time.time() - start < 5:
        data = ser.read(256)
        if data:
            output += data
        elif output:
            time.sleep(0.3)
            data = ser.read(256)
            if data:
                output += data
            else:
                break
    ser.close()
```

By holding the serial port open across the programming step, the OS does not tear down the file descriptor even though the FTDI chip resets. When the USB interface re-enumerates, the existing file descriptor reconnects, and we catch the output from the fresh execution.

31.3.3 Serial Port Discovery

The script can find the ULX3S serial port automatically:

Listing 31.7: Serial port auto-detection in fpga_test.py

```
def find_serial_port():
    """Find the ULX3S serial port."""
    port = os.environ.get('SERIAL_PORT')
    if port:
        return port
    ports = glob.glob('/dev/cu.usbserial-*')
    if ports:
        return ports[0]
    ports = glob.glob('/dev/ttyUSB*')
    if ports:
        return ports[0]
    return None
```

It checks three locations, in order:

1. The `SERIAL_PORT` environment variable (explicit override).
2. `/dev/cu.usbserial-*` (macOS naming convention).

3. `/dev/ttyUSB*` (Linux naming convention).

Overriding the Serial Port

If you have multiple FTDI devices connected (common when using multiple development boards or USB-serial adapters), set the `SERIAL_PORT` environment variable explicitly:

```
SERIAL_PORT=/dev/cu.usbserial-D01263 make fpga-test_alu
```

You can also set the `SERIAL_PORT` variable in the Makefile itself for your particular board.

31.3.4 FPGA Programming

The programming step is a straightforward `openFPGALoader` invocation:

Listing 31.8: FPGA programming in fpga_test.py

```
def program_fpga(bitstream):
    """Program FPGA and wait for USB to settle."""
    result = subprocess.run(
        ['openFPGALoader', '-b', 'ulx3s', bitstream],
        capture_output=True, text=True
    )
    if result.returncode != 0:
        print(f"Programming failed: {result.stderr}",
              file=sys.stderr)
        return False
    # Wait for USB re-enumeration
    time.sleep(1.0)
    return True
```

The 1-second sleep after programming is critical. Without it, the serial port open will fail because the FTDI chip has not finished re-enumerating on the USB bus. On slower machines, you may need to increase this delay.

31.3.5 Exit Code Semantics

The script communicates results through its exit code, which the Makefile uses to determine success or failure:

The final result check is straightforward string matching:

Listing 31.9: Result classification in fpga_test.py

Table 31.3: fpga_test.py exit codes

Exit Code	Meaning
0	Test output contained PASS (and no FAIL)
1	Test output contained FAIL, or programming failed
2	No test output captured (hardware communication failure)

```
    # Check results
    text = output.decode('ascii',
                         errors='replace') if output else ''
    if 'FAIL' in text:
        sys.exit(1)
    elif 'PASS' in text:
        sys.exit(0)
    else:
        sys.exit(2)
```

31.4 Makefile Targets: fpga-%, fpga-tests

The Makefile in `verilog/Makefile` ties the entire pipeline together with two key target families: the per-test `fpga-%` pattern rule and the aggregate `fpga-tests` target.

31.4.1 The ISA Test List

The Makefile defines the complete list of ISA test suites as a variable:

Listing 31.10: ISA test list in the Makefile

```
ISA_TESTS = test_alu test_addi test_shift test_muldiv \
            test_loadstore test_branch test_jump \
            test_stack test_misc test_extended
```

These names correspond to assembly source files in `examples/tests/`. For example, `test_alu` maps to `examples/tests/test_alu.s`.

31.4.2 The fpga-% Pattern Rule

Each test can be run individually with `make fpga-test_alu`, `make fpga-test_branch`, and so on. The pattern rule chains together the full pipeline:

Listing 31.11: The fpga-% pattern rule

```
$(addprefix fpga-,$(ISA_TESTS)): fpga-%: \
    $(BUILD_DIR)/sampo_%.bit
  @echo "=== Programming $* to FPGA ==="
  @openFPGALoader -b ulx3s $<
  @sleep 1
  @echo "=== Capturing UART output ==="
  @python3 fpga_capture.py $(SERIAL_PORT) \
      $(SERIAL_BAUD) 5 \
      > $(BUILD_DIR)/fpga_$*.log
  @cat $(BUILD_DIR)/fpga_$*.log
  @echo ""
  @if grep -q FAIL \
      $(BUILD_DIR)/fpga_$*.log 2>/dev/null; then \
    echo "*** FAILURES DETECTED ***"; false; \
  elif grep -q PASS \
      $(BUILD_DIR)/fpga_$*.log 2>/dev/null; then \
    echo "All tests passed!"; \
  else \
    echo "WARNING: No test output captured"; \
  fi
```

Let us trace through what happens when you run `make fpga-test_alu`:

1. Make resolves the prerequisite `build/sampo_test_alu.bit`, which triggers the full FPGA build chain (assembly → hex → loop patch → synthesis → PnR → bitstream).

2. `openFPGALoader` programs the bitstream to the ULX3S.

3. A 1-second sleep allows USB re-enumeration.

4. `fpga_capture.py` opens the serial port and captures one complete iteration of the test output, writing it to `build/fpga_test_alu.log`.

5. The log is printed to the console via `cat`.

6. A `grep` chain checks for `FAIL` or `PASS` in the output and sets the exit code accordingly.

The SERIAL_PORT and SERIAL_BAUD Variables

The Makefile defines default values for the serial port and baud rate:

```
SERIAL_PORT ?= /dev/cu.usbserial-D01263
SERIAL_BAUD ?= 115200
```

The `?=` operator means these can be overridden from the command line or environment:

```
make fpga-test_alu SERIAL_PORT=/dev/ttyUSB0
```

31.4.3 The fpga-tests Aggregate Target

Running all 10 test suites sequentially is handled by the `fpga-tests` target:

Listing 31.12: The fpga-tests aggregate target

```
fpga-tests:
  @PASS=0; FAIL=0; SKIP=0; \
  for t in $(ISA_TESTS); do \
    echo ""; \
    echo "========================================"; \
    echo "=== FPGA: $$t ==="; \
    echo "========================================"; \
    if $(MAKE) --no-print-directory fpga-$$t; then \
      PASS=$$((PASS + 1)); \
    else \
      FAIL=$$((FAIL + 1)); \
    fi; \
  done; \
  echo ""; \
  echo "========================================"; \
  echo "FPGA Test Summary: $$PASS passed, \
  $$FAIL failed"; \
  echo "========================================"; \
  [ $$FAIL -eq 0 ]
```

This is a shell loop that invokes `make fpga-<name>` for each test in the list. It tracks the number of passing and failing test *suites* (not individual test cases) and prints a summary at the end. The final `[ $$FAIL -eq 0 ]` ensures that the overall `fpga-tests` target returns a nonzero exit code if any suite failed.

The `--no-print-directory` Flag

The `--no-print-directory` flag suppresses Make's "Entering directory" and "Leaving directory" messages, which would clutter the output. Each recursive `make fpga-$$t` call still runs in the same directory—this flag just keeps the output clean.

31.4.4 Protecting Intermediate Files

One subtle but important detail in the Makefile is the `.PRECIOUS` directive:

Listing 31.13: Preserving intermediate files

```
.PRECIOUS: $(PROG_DIR)/%.hex \
           $(PROG_DIR)/%_loop.hex \
           ../examples/tests/%.bin
.PRECIOUS: $(BUILD_DIR)/sampo_%.json \
           $(BUILD_DIR)/sampo_%.config \
           $(BUILD_DIR)/sampo_%.bit
```

By default, Make deletes intermediate files (files created as a side effect of pattern rules but not explicitly named as targets). Without `.PRECIOUS`, Make would delete the hex files, synthesized netlists, and bitstreams after each run. Since these files take considerable time to generate—especially the synthesis and place-and-route outputs—we mark them as precious to prevent wasteful rebuilds.

31.5 Per-Test FPGA Builds

The most unusual aspect of the SAMPO FPGA test pipeline is that it builds a separate bitstream for each test. This is not how most FPGA projects work—typically, you build one bitstream and load different programs through some runtime mechanism (bootloader, SPI flash, or an in-circuit debugger). The SAMPO approach bakes each test program directly into the RAM initialization, requiring a full synthesis-PnR-pack cycle per test.

31.5.1 Why Per-Test Builds?

The SAMPO SoC uses `$readmemh` to initialize its block RAM at synthesis time. The RAM contents are fixed in the bitstream. There is no bootloader, no external flash interface, and no JTAG-accessible memory loader. This is a deliberate design choice for a teaching CPU: it keeps the SoC architecture minimal and avoids the complexity of a software-loading mechanism.

The consequence is that changing the program requires rebuilding the entire bitstream. For testing, this means each test gets its own build.

31.5.2 The Build Chain for a Test

Let us trace the dependency chain for a single test, starting from `make fpga-test_alu`:

Listing 31.14: Per-test build dependency chain

```
# 1. Assemble test source to binary
../examples/tests/test_alu.bin: ../examples/tests/test_alu.s
  cd ../sasm && cargo run --release -- $< -o $@

# 2. Convert binary to hex
programs/test_alu.hex: ../examples/tests/test_alu.bin
  python3 bin2hex.py $< $@

# 3. Create looped hex (patch HALT -> loop)
programs/test_alu_loop.hex: programs/test_alu.hex
  python3 hex_loop_patch.py $< $@

# 4. Synthesize with looped hex
build/sampo_test_alu.json: ... programs/test_alu_loop.hex
  sed 's|programs/hello\.hex|programs/test_alu_loop.hex|g' \
      fpga/ulx3s_top.v > build/ulx3s_top_test_alu.v
  yosys -p "read_verilog ..."

# 5. Place & route
build/sampo_test_alu.config: build/sampo_test_alu.json
  nextpnr-ecp5 ...

# 6. Pack bitstream
build/sampo_test_alu.bit: build/sampo_test_alu.config
  ecppack --compress $< $@
```

Step 4 deserves special attention. The standard FPGA top module (`fpga/ulx3s_top.v`) hardcodes the RAM initialization file:

Listing 31.15: Hardcoded RAM init in ulx3s_top.v

```
    soc #(
        .RESET_VECTOR(16'h0100),
        .RAM_INIT_FILE("programs/hello.hex")
    ) u_soc (
        .clk(clk_cpu),
        .rst_n(rst_n),
        // ...
    );
```

The Makefile uses sed to create a modified copy that references the test-specific looped hex file:

Listing 31.16: Sed substitution for per-test RAM init

```
sed 's|programs/hello\.hex|programs/test_alu_loop.hex|g' \
    fpga/ulx3s_top.v > build/ulx3s_top_test_alu.v
```

This creates `build/ulx3s_top_test_alu.v`, which is identical to the original top module except that the `RAM_INIT_FILE` parameter points to the loop-patched test hex. The modified top module is then passed to Yosys along with all the other RTL sources.

31.5.3 The Hex Loop Patch

The `hex_loop_patch.py` script is what makes looping possible. It modifies the compiled hex file to replace the **`HALT`** instruction with a jump to a delay loop, which then jumps back to the reset vector at `0x0100`.

The challenge is doing this without disturbing any of the program's address references. If we moved any instructions, all the **`LIX`** immediates, branch offsets, and jump targets would break. The solution is elegant: place the delay loop in the unused memory below the reset vector.

The SAMPO convention places the reset vector at `0x0100`, which means the first 256 bytes (128 words) of memory are unused. The loop patch writes a delay loop starting at word address 0:

Listing 31.17: The delay loop patch (from hex_loop_patch.py)

```
# Delay + restart loop at address 0x0000:
#   LIX R8, 0x0008    ; outer counter = 8
# outer:
#   LIX R9, 0xFFFF    ; inner counter = 65535
# inner:
#   ADDI R9, -1        ; R9 = R9 - 1
#   BNE inner           ; offset=-2
#   ADDI R8, -1        ; R8 = R8 - 1
#   BNE outer           ; offset=-6
#   JX 0x0100           ; restart at reset vector

LOOP_PATCH = [
    0xF807, 0x0008,  # LIX R8, 0x0008
    0xF907, 0xFFFF,  # LIX R9, 0xFFFF
    0x59FF,          # ADDI R9, -1
    0x81FE,          # BNE -2 (inner)
    0x58FF,          # ADDI R8, -1
    0x81FA,          # BNE -6 (outer)
    0xF008, 0x0100,  # JX 0x0100
```

```
]
```

The delay loop performs 8×65535 iterations of a tight inner loop. At approximately 9 cycles per inner iteration at the 12.5 MHz CPU clock, that gives:

$$8 \times 65535 \times 9 \approx 4{,}718{,}520 \text{ cycles} = \frac{4{,}718{,}520}{12{,}500{,}000} \approx 0.38 \text{ s}$$

This delay serves two purposes:

1. It gives the UART TX hardware time to finish transmitting the last bytes of the previous iteration before the next iteration starts printing.
2. It creates a clear gap in the serial output stream, making iteration boundaries easy to detect.

The **`HALT`** instruction (encoded as `0xE100`) is then replaced with a relative **`J`** instruction that jumps to the delay loop at address 0:

Listing 31.18: Encoding the J instruction to the delay loop

```
def encode_j_offset(halt_addr, target_addr):
    """Encode a J instruction from halt_addr
    to target_addr.

    J instruction: opcode=0x9, offset in bits[11:0]
    Target = PC + 2 + (sign_extend(offset) << 1)
    """
    raw_offset = (target_addr - halt_addr - 2) // 2

    # Check range: 12-bit signed = [-2048, 2047]
    if raw_offset < -2048 or raw_offset > 2047:
        return None

    offset_12 = raw_offset & 0xFFF

    # Check for JR/JALR decode conflicts
    rd_bits = (offset_12 >> 8) & 0xF
    func_bits = offset_12 & 0xF

    if rd_bits == 0xF and func_bits == 0x0:
        return None  # Would decode as JR
    if func_bits == 0x1 and rd_bits != 0xF:
        return None  # Would decode as JALR

    return 0x9000 | offset_12
```

The JR/JALR Decode Conflict

The **J**, **JR**, and **JALR** instructions all share opcode `0x9`. The decoder distinguishes them by examining specific bit fields within the 12-bit offset/register field. If the J offset happens to have the same bit pattern as a JR or JALR encoding, the decoder will misinterpret it. The `hex_loop_patch.py` script checks for this conflict and, if necessary, adjusts the loop's base address to find an offset that encodes cleanly. This is a real encoding quirk of the SAMPO ISA that would not show up in simulation (where you would never need to patch a HALT).

31.5.4 The bin2hex.py Converter

Before the loop patch can operate on the hex file, the assembler's binary output must be converted to Verilog hex format. The `bin2hex.py` script handles this:

Listing 31.19: bin2hex.py: binary to Verilog hex converter

```
def main():
    if len(sys.argv) < 3:
        print("Usage: bin2hex.py <input.bin> "
              "<output.hex>")
        sys.exit(1)

    input_file = sys.argv[1]
    output_file = sys.argv[2]

    with open(input_file, 'rb') as f:
        data = f.read()

    # Pad to even length if necessary
    if len(data) % 2:
        data += b'\x00'

    # Convert to 16-bit words (little-endian)
    words = []
    for i in range(0, len(data), 2):
        word = data[i] | (data[i+1] << 8)
        words.append(word)

    # Write hex file
    with open(output_file, 'w') as f:
        f.write("@0000\n")
        for i, word in enumerate(words):
            f.write(f"{word:04X}\n")
```

```
    print(f"Converted {len(data)} bytes "
          f"to {len(words)} words")
```

The format is what Verilog's `$readmemh` expects: an optional `@` address directive followed by one hex word per line. Note the little-endian byte ordering—the SAMPO is a little-endian architecture, so byte 0 is the low byte of each 16-bit word.

31.6 Handling the FTDI Buffer

The FTDI FT231X on the ULX3S presents several buffer-related challenges that the test infrastructure must handle. Understanding these issues is essential for reliable FPGA testing.

31.6.1 Stale Data in the Receive Buffer

When the serial port is opened, the FTDI chip's internal receive buffer may contain stale data from a previous run. If the FPGA was running a looping test, the FTDI chip has been buffering UART output the entire time—even if no host application was reading from it. This buffered data can be thousands of bytes old.

Both capture scripts deal with this by calling `reset_input_buffer()` immediately after opening the serial port:

Listing 31.20: Flushing the FTDI receive buffer

```
ser = serial.Serial(port, baud, timeout=2)
ser.reset_input_buffer()
```

This discards any data already in the FTDI chip's buffer and the OS serial driver's buffer. Combined with the first-partial-line discard in `fpga_capture.py`, this ensures we start reading from a clean state.

31.6.2 The USB Re-enumeration Window

After JTAG programming, the FTDI chip drops off the USB bus and re-enumerates. During this window (typically 0.5–1 second), the serial port device file does not exist. Attempting to open it during this period raises an exception:

```
serial.serialutil.SerialException:
  [Errno 2] could not open port /dev/cu.usbserial-D01263:
  [Errno 2] No such file or directory
```

The 1-second sleep after programming in both `fpga_test.py` and the Makefile accounts for this:

Listing 31.21: Post-programming sleep in the Makefile

```
@openFPGALoader -b ulx3s $<
@sleep 1
@echo "=== Capturing UART output ==="
```

Do Not Remove the Post-Programming Delay

If you remove or significantly shorten the `sleep 1` after programming, the capture script will intermittently fail to open the serial port. The failures will be non-deterministic, depending on your machine's USB stack latency and system load. One second is a conservative but reliable delay. If you are optimizing for speed, you can try 0.5 seconds, but test thoroughly.

31.6.3 Output Interleaving

When using the looping patch, the FPGA is continuously producing output. If the serial port is opened mid-stream, the first bytes read will be from the middle of a line. This is why `fpga_capture.py` discards the first partial line:

Listing 31.22: Discarding the first partial line

```
    ser.reset_input_buffer()

    # Discard first partial line
    # (we likely joined mid-stream)
    ser.readline()
```

The `readline()` call blocks until it receives a newline character or times out. If we joined mid-line, this consumes and discards the partial line. If we happen to join at a line boundary, it consumes a complete line, which is an acceptable loss—we will still capture a full iteration starting from the next header.

31.6.4 UART TX Buffer Overflow

The SAMPO test programs include a busy-wait loop that checks the UART status register before sending each character. Here is the pattern from `test_alu.s`:

Listing 31.23: UART busy-wait in the test framework

```
ps_wait:
        INI R6, ACIA_STATUS    ; Read status register
        AND R7, R6, R6         ; Copy to R7
        ADDI R7, -2            ; Check TDRE bit (bit 1)
```

```
        BNE ps_wait           ; Loop until TX ready
        OUTI ACIA_DATA, R5    ; Send character
```

This polls the TDRE (Transmit Data Register Empty) bit in the UART status register at `0x80`. The UART module in `rtl/uart.v` reports this bit as part of the status byte:

Listing 31.24: UART status register in uart.v

```
// Status bits (MC6850-compatible)
// Bit 0: RDRF - Receive Data Register Full
// Bit 1: TDRE - Transmit Data Register Empty
wire [7:0] status;
assign status = {6'b0, tx_ready, rx_pending};
```

This handshaking ensures that the CPU never writes a character while the UART TX is still busy with the previous one. Without it, characters would be lost. The busy-wait does slow down the tests considerably (each character takes roughly 87 μs at 115200 baud), but it guarantees reliable output.

31.6.5 The Double-Read Pattern

Both `fpga_test.py` and `fpga_capture.py` use a pattern where, after the main data stream goes silent, they wait briefly and try one more read:

Listing 31.25: The double-read pattern in fpga_test.py

```
while time.time() - start < timeout_sec:
    data = ser.read(256)
    if data:
        output += data
    elif output:
        # Got some data then silence
        time.sleep(0.2)
        data = ser.read(256)
        if data:
            output += data
        else:
            break
```

This handles a subtle timing issue: the FTDI chip buffers data internally before making it available to the USB host. A brief gap in the data stream does not necessarily mean the test is done—it might just mean the FTDI chip's buffer has not been flushed yet. The 200 ms wait allows the chip to flush, and the second read catches any trailing data.

Exercises

1. **Manual capture.** Build and program a single test manually, step by step, without using the `fpga-%` target. Assemble `test_alu.s`, convert it with `bin2hex.py`, patch it with `hex_loop_patch.py`, synthesize a bitstream with the patched hex file, program the ULX3S, and capture the output with `fpga_capture.py`. Verify that your manual process produces the same output as `make fpga-test_alu`.

2. **Adjusting the delay loop.** Modify the `LOOP_PATCH` constants in `hex_loop_patch.py` to change the delay between iterations. Try reducing the outer counter from 8 to 1. What happens to the captured output? At what point does the delay become too short for reliable capture?

3. **Custom test.** Write a new SAMPO assembly test that verifies the **`NOT`** instruction on three different inputs. Follow the test framework pattern from `test_alu.s` (including the header string, `check_eq` subroutine, and "Done." marker). Add it to the `ISA_TESTS` list in the Makefile and run it on the FPGA.

4. **Capture reliability.** Run `make fpga-test_alu` five times in a row and record the output each time. Is the output identical? Are there any spurious characters or missing lines? If you see differences, identify which stage of the pipeline (programming, re-enumeration, capture) is responsible.

5. **Timeout tuning.** Modify the Makefile to pass a 2-second timeout to `fpga_capture.py` instead of 5 seconds. Which tests still pass? Which tests fail or produce truncated output? What does this tell you about the relationship between test size and capture timeout?

What's Next

With the test infrastructure in place, we are ready to turn it loose on the full ISA test suite. In the next chapter, we will run all 10 test suites—132 individual test cases—on the ULX3S hardware and examine every result. We will compare the FPGA results against simulation, analyze the few places where hardware and simulation can diverge, and build a regression testing workflow that catches problems before they reach the bitstream.

Chapter 32

132 Tests, Zero Failures

The ultimate validation of a CPU design is not a clean synthesis report or a passing simulation—it is a wall of green on real hardware. Every instruction, every flag, every corner case, exercised on actual silicon (well, actual LUTs), with the results captured over a physical serial link. There is no `$display` to fall back on. No waveform viewer to pause and inspect. The CPU either computes the right answer and prints PASS, or it does not.

In this chapter, we run the complete SAMPO ISA test suite on the ULX3S FPGA board: 10 test programs, 132 individual test cases, covering every instruction category from basic arithmetic to extended 32-bit operations. We will examine the output, understand what each test verifies, explore the subtle differences between simulation and hardware, and build a regression testing workflow that keeps the design honest over time.

32.1 Running the Complete Suite

Running all tests is a single command:

Listing 32.1: Running the full FPGA test suite

```
cd verilog
make fpga-tests
```

This invokes the `fpga-tests` target from Section 31.4.3, which iterates through all 10 test suites, building a per-test bitstream, programming it to the board, and capturing the UART output. The process takes 10–15 minutes depending on your machine.

32.1.1 What Happens During a Run

For each test suite, the Makefile:

1. Assembles the test source (e.g., `examples/tests/test_alu.s`) into a binary.
2. Converts the binary to hex format with `bin2hex.py`.
3. Patches the hex to loop continuously with `hex_loop_patch.py`.
4. Creates a modified FPGA top module that references the patched hex file.
5. Runs Yosys synthesis, nextpnr place-and-route, and ecppack to generate a bitstream.
6. Programs the ULX3S via `openFPGALoader`.
7. Captures one complete iteration of UART output with `fpga_capture.py`.
8. Parses the output for PASS/FAIL results.

The console output for a passing test suite looks like this:

Listing 32.2: Console output for a passing FPGA test

```
========================================
=== FPGA: test_alu ===
========================================
=== Programming test_alu to FPGA ===
=== Capturing UART output ===
=== ALU Tests ===
ADD basic: PASS
ADD zero: PASS
ADD carry out: PASS
ADD overflow: PASS
ADD to R0: PASS
SUB basic: PASS
SUB to zero: PASS
SUB underflow: PASS
AND mask: PASS
AND zero: PASS
OR basic: PASS
OR all bits: PASS
XOR toggle: PASS
XOR self zero: PASS
NEG basic: PASS
NEG zero: PASS
ADD C flag: PASS
ADD V flag: PASS
```

```
SUB C flag: PASS
AND clr C/V: PASS
Done.

All tests passed!
```

When all 10 suites complete, the summary line reports the overall result:

Listing 32.3: FPGA test suite summary

```
========================================
FPGA Test Summary: 10 passed, 0 failed
========================================
```

32.1.2 Running Individual Suites

If you are debugging a specific instruction category, you do not need to wait for the full suite. Run a single test with:

Listing 32.4: Running a single FPGA test

```
make fpga-test_branch
```

The first run takes the full 60–90 seconds because it must synthesize the bitstream. Subsequent runs reuse the cached bitstream (if the source has not changed), reducing the turnaround to about 5 seconds for programming and capture.

Rebuilding After RTL Changes

If you modify any Verilog source file (ALU, decoder, CPU, etc.), the per-test bitstreams become stale. Run `make clean` to remove all build artifacts, then rebuild:

```
make clean
make fpga-tests
```

Make's dependency tracking handles hex file and assembly changes automatically, but RTL changes affect every bitstream. A clean build ensures you are testing the latest design.

32.2 Interpreting the Results

Each test suite follows a consistent output format. Understanding this format is essential for diagnosing failures.

32.2.1 The Test Output Format

Every test program prints:

1. A **header line** matching the pattern `===  ...  ===`.
2. One or more **test result lines**, each consisting of a description followed by `PASS` or `FAIL`.
3. A **footer line**: "Done."

This format is generated by the test framework embedded in each assembly file. The `check_eq` subroutine compares `R4` (actual result) against `R5` (expected result) and prints the appropriate verdict:

Listing 32.5: The check_eq subroutine from the test framework

```
check_eq:
        PUSH RA
        CMP R4, R5
        BNE ce_fail
        LIX R4, str_pass
        JALX print_str
        POP RA
        JR RA
ce_fail:
        LIX R4, str_fail
        JALX print_str
        POP RA
        JR RA
```

The simplicity of this framework is deliberate. It uses only instructions that are themselves being tested—if the test framework works, a substantial portion of the ISA is already validated. The framework depends on **PUSH**, **POP**, **CMP**, **BNE**, **BEQ**, **LIX**, **JALX**, **JR**, **OUTI**, **INI**, **AND**, **ADDI**, **LBU**, and **J**—15 instructions just to run the harness.

The Self-Referential Nature of CPU Testing

There is an inherent circularity in testing a CPU using programs that run on that CPU. If the branch instructions are broken, the test framework that uses branches to report PASS/FAIL will not work correctly. This is why the test suite is structured with increasing complexity: the ALU tests use the fewest framework features, and later tests build on the confidence established by earlier ones. If `test_alu` passes, we know the basic arithmetic, branch, and I/O instructions work—which gives us confidence in the framework for the remaining tests.

32.2.2 What Each Test Suite Verifies

The 10 test suites cover the SAMPO ISA systematically. Here is what each one targets:

test_alu (20 tests). Core ALU operations: **ADD**, **SUB**, **AND**, **OR**, **XOR**, **NEG**. Includes boundary conditions (zero operands, carry out, overflow) and flag verification (C, V, N, Z flags after specific operations). The flag tests use **GETF** to read the flags register and mask individual bits:

Listing 32.6: Flag verification in test_alu.s

```
; ADD overflow flag: 0x7FFF + 1 sets V
LIX R8, 0x7FFF
LIX R9, 0x0001
ADD R10, R8, R9
GETF R4
; Expect: N=1, Z=0, C=0, V=1
; flags[7:4] = 1001 = 0x90
LIX R8, 0x00F0
AND R4, R4, R8
LIX R5, 0x0090
JALX check_eq
```

test_addi (10 tests). The **ADDI** immediate instruction. Tests positive and negative immediates, boundary values (+127, −128—the limits of the 8-bit signed immediate), carry, overflow, and upper-bit preservation. The 2-operand form (Rd = Rd + sign_ext(imm8)) is verified directly:

Listing 32.7: ADDI boundary test from test_addi.s

```
; ADDI max positive: 0 + 127 = 127
MOV R4, R0
ADDI R4, 127
LIX R5, 0x007F
JALX check_eq

; ADDI max negative: 0 + (-128) = 0xFF80
MOV R4, R0
ADDI R4, -128
LIX R5, 0xFF80
JALX check_eq
```

test_shift (17 tests). Shift and rotate operations: **SLL**, **SRL**, **SRA** (by 1), **ROL**, **ROR**, **SWAP**, plus the multi-position variants **SLL4**/**SRL4**/**SRA4**/**SLL8**/**SRL8**/**SRA8**/**ROL4**/**ROL8**. Tests include carry-out verification and the critical **SRA** sign-extension behavior:

Listing 32.8: SRA sign preservation in test_shift.s

```
; SRA negative: 0x8234 >>> 1 = 0xC11A
; (sign bit preserved)
LIX R8, 0x8234
SRA R4, R8
LIX R5, 0xC11A
JALX check_eq
```

test_muldiv (12 tests). Multiply and divide: **MUL**, **MULH**, **DIV**, **DIVU**, **REM**, **REMU**. These are 2-operand instructions (Rd = Rd op Rs). Tests cover basic operations, multiplication overflow (verifying **MULH** for the high word), division by zero (hardware-defined result of 0xFFFF), and remainder with zero divisor (hardware returns the dividend):

Listing 32.9: Division by zero test from test_muldiv.s

```
; DIV by zero: 5 / 0 = 0xFFFF (hardware)
LIX R4, 0x0005
DIV R4, R0
LIX R5, 0xFFFF
JALX check_eq

; REM zero division: 5 % 0 = 5 (returns dividend)
LIX R4, 0x0005
REM R4, R0
LIX R5, 0x0005
JALX check_eq
```

test_loadstore (11 tests). Memory operations: **SW**/**LW** round-trips, **SB**/**LBU**/**LB** byte operations, load with positive and negative offsets, and **LIX** immediate loading. The sign-extension test for **LB** is particularly important—it verifies that loading byte 0x80 produces the 16-bit value 0xFF80:

Listing 32.10: LB sign extension test from test_loadstore.s

```
; LB sign extension: store 0x80, LB gives 0xFF80
SW (R13), R0          ; Clear word
LIX R9, 0x0080
SB (R13), R9          ; Store byte 0x80
LB R4, (R13)          ; Load byte, sign-extend
LIX R5, 0xFF80
JALX check_eq
```

test_branch (24 tests). All 12 branch conditions, each tested in both taken and not-taken cases (24 tests total). This is the largest test suite by variety of conditions. It covers **BEQ**, **BNE**, **BLT**, **BGE**, **BLTU**, **BGEU**, **BMI**, **BPL**, **BGT**, **BLE**, **BHI**, and **BLS**. The branch tests use a different pattern than check_eq—they branch to pass_and_next or fail_and_next based on the branch outcome:

Listing 32.11: Branch test pattern from test_branch.s

```
        ; BEQ (Z=1): taken when equal
        LIX R8, 42
        CMP R8, R8
        BEQ beq_t_ok
        JALX fail_and_next
        J beq_t_end
beq_t_ok:
        JALX pass_and_next
beq_t_end:
```

test_jump (6 tests). Jump instructions: **J** (relative), **JR** (register indirect), **JALR** (jump-and-link via register), **JX** (absolute), **JALX** (absolute jump-and-link), and verification that **JALX** correctly sets RA.

test_stack (12 tests). Stack operations (**PUSH**/**POP**) including LIFO ordering, plus **CMP**, **TEST**, and **MOV**. Verifies that SP is correctly restored after a push/pop pair and that **TEST** clears the C and V flags.

test_misc (9 tests). Miscellaneous instructions: **GETF**/**SETF** (read/write flags register), **SCF** (set carry), **CCF** (complement carry), **NOP**, and **EXX** (exchange register banks). The **EXX** tests verify that registers R4–R11 are swapped while R0–R3 and R12–R15 are preserved:

Listing 32.12: EXX register swap test from test_misc.s

```
        ; EXX: set R8, swap, set R8', swap back
        LIX R8, 0xAAAA
        EXX                     ; Swap to alternate set
        LIX R8, 0xBBBB          ; Set R8' to 0xBBBB
        EXX                     ; Swap back
        ; R8 should be 0xAAAA again
        MOV R4, R8
        LIX R5, 0xAAAA
        JALX check_eq
```

test_extended (11 tests). Extended 32-bit instructions: **ADDIX**, **SUBIX**, **ANDIX**, **ORIX**, **XORIX**, **SLLX**, **SRLX**, **SRAX**. These use the 0xF prefix opcode for 32-bit encoding. Since the assembler does not yet support all extended mnemonics, some tests use .dw directives to emit raw instruction words:

Listing 32.13: Extended instruction testing with .dw in test_extended.s

```
        ; SLLX R4, R8, 4: R4 = R8 << 4
        ; Encoding: 0xF48D 0x0004
        LIX R8, 0x1234
        .dw 0xF48D, 0x0004
        LIX R5, 0x2340
        JALX check_eq
```

32.3 Simulation vs. Hardware Differences

One of the most valuable aspects of FPGA testing is discovering where simulation and hardware diverge. For the SAMPO, the test suite produces identical PASS/FAIL results in both environments. This is the goal—but it was not always the case during development. Several categories of differences can appear, and understanding them helps diagnose future issues.

32.3.1 Timing and Clock Domains

In simulation, the testbench in `tb/sampo_tb.v` uses a 100 MHz clock (10 ns period). On the ULX3S, the CPU runs at 12.5 MHz (80 ns period), derived from the 25 MHz board oscillator via a clock divider in `fpga/ulx3s_top.v`:

Listing 32.14: Clock divider in ulx3s_top.v

```
// Clock divider: 25 MHz -> 12.5 MHz
reg clk_div;
wire clk_cpu = clk_div;
initial clk_div = 1'b0;
always @(posedge clk_25mhz) begin
    clk_div <= ~clk_div;
end
```

The clock frequency difference does not affect instruction-level behavior (the SAMPO is fully synchronous and not frequency-dependent), but it does affect:

- **UART baud rate.** The simulation testbench simulates UART TX timing using a cycle counter calibrated to the 12.5 MHz rate (`TX_BYTE_CYCLES = 108 * 10`). On hardware, the `fpga/uart_tx.v` module generates the baud rate from the actual clock. If the baud rate divider were wrong, simulation would still pass but hardware would produce garbled serial output.

- **Total execution time.** A test that takes 50,000 cycles runs in 0.5 ms at 100 MHz (simulation) but 4 ms at 12.5 MHz (hardware). This matters for the UART capture timeout but not for correctness.

32.3.2 Reset Behavior

In simulation, the testbench asserts reset (`rst_n = 0`) for a precise 100 ns, then releases it. On hardware, the `fpga/reset_sync.v` module provides a synchronized reset derived from the board's button input:

Listing 32.15: Reset synchronizer instantiation

```
reset_sync #(
    .CLK_FREQ(CLK_FREQ)
) u_reset (
    .clk(clk_cpu),
    .ext_rst_n(btn[0]),
    .rst_n(rst_n)
);
```

The reset synchronizer holds the system in reset for several clock cycles after power-up to ensure all flip-flops start in a known state. This is essential because FPGA flip-flops (unlike simulation `reg` variables, which initialize to `x`) power up in an indeterminate state on real hardware.

Uninitialized Registers on Hardware

In Verilog simulation, uninitialized `reg` variables have the value `x` (unknown). Simulation tools propagate `x` through logic, making uninitialized-register bugs visible as `x` values in waveforms. On real FPGA hardware, flip-flops power up to either 0 or 1 (depending on the FPGA family and configuration), and there is no `x` state. A design that works in simulation because an uninitialized register happens to start at 0 might fail on hardware if that register powers up to 1. The SAMPO avoids this class of bugs by using explicit reset logic for all state-holding registers.

32.3.3 Memory Initialization

The SoC's RAM is initialized via `$readmemh` with the test hex file. In simulation, this happens at time zero. On the FPGA, the RAM contents are baked into the bitstream during synthesis—they become the initial values of the FPGA's block RAM cells.

One difference: in simulation, any RAM location not covered by the hex file contains `x`. On the FPGA, uninitialized block RAM locations contain 0. The test programs use an explicit data area (address `0x8000` for load/store tests) and initialize the stack pointer to `0xFFF0`, so they do not depend on the initial contents of unused memory.

32.3.4 The MEMORY→FETCH Hazard

The most significant simulation-vs-hardware difference discovered during SAMPO development was the MEMORY→FETCH hazard, discussed in detail in **??**. The bug: when the CPU transitions from the MEMORY state directly to FETCH, the `mem_ready` signal from the previous memory operation can still be asserted, causing the fetch to complete prematurely with stale data.

In simulation, this bug manifested intermittently depending on the exact instruction sequence. On hardware, with real gate delays and routing, the timing was different enough that the bug appeared more or less frequently depending on the specific test.

The fix—routing all instructions through the WRITEBACK state, even those that do not write back to a register—ensured a clean one-cycle gap between the memory operation and the next fetch. After this fix, simulation and hardware results became identical.

The Value of Hardware Testing

The MEMORY→FETCH hazard is a textbook example of why FPGA testing matters. The bug existed in the design from the beginning, but it only manifested reliably under specific timing conditions that depended on the physical routing of signals on the FPGA. Simulation, even with correct delay modeling, may not reproduce the exact conditions. Running the test suite on hardware caught what simulation missed.

32.4 The Full Results Table

Table 32.1 shows the complete test suite results, listing every individual test case, organized by suite. All 132 tests pass on both the Icarus Verilog simulation and the ULX3S FPGA hardware.

Table 32.1: Complete FPGA test results: 132 tests, 0 failures

#	Suite	Test Case	Result
1	test_alu	ADD basic	PASS
2	test_alu	ADD zero	PASS
3	test_alu	ADD carry out	PASS
4	test_alu	ADD overflow	PASS
5	test_alu	ADD to R0	PASS
6	test_alu	SUB basic	PASS
7	test_alu	SUB to zero	PASS
8	test_alu	SUB underflow	PASS
9	test_alu	AND mask	PASS
10	test_alu	AND zero	PASS

Table 32.1: Complete FPGA test results (continued)

#	Suite	Test Case	Result
11	test_alu	OR basic	PASS
12	test_alu	OR all bits	PASS
13	test_alu	XOR toggle	PASS
14	test_alu	XOR self zero	PASS
15	test_alu	NEG basic	PASS
16	test_alu	NEG zero	PASS
17	test_alu	ADD C flag	PASS
18	test_alu	ADD V flag	PASS
19	test_alu	SUB C flag	PASS
20	test_alu	AND clr C/V	PASS
21	test_addi	ADDI +5	PASS
22	test_addi	ADDI −3	PASS
23	test_addi	ADDI to 0	PASS
24	test_addi	ADDI +127	PASS
25	test_addi	ADDI −128	PASS
26	test_addi	ADDI carry	PASS
27	test_addi	ADDI ovf	PASS
28	test_addi	ADDI large	PASS
29	test_addi	ADDI n2n	PASS
30	test_addi	ADDI upper	PASS
31	test_shift	SLL1	PASS
32	test_shift	SRL1	PASS
33	test_shift	SRA1 pos	PASS
34	test_shift	SRA1 neg	PASS
35	test_shift	ROL1	PASS
36	test_shift	ROR1	PASS
37	test_shift	SWAP	PASS
38	test_shift	SLL carry	PASS
39	test_shift	SRL carry	PASS
40	test_shift	SLL4	PASS
41	test_shift	SRL4	PASS
42	test_shift	SRA4	PASS
43	test_shift	SLL8	PASS
44	test_shift	SRL8	PASS
45	test_shift	SRA8	PASS
46	test_shift	ROL4	PASS

Table 32.1: Complete FPGA test results (continued)

#	Suite	Test Case	Result
47	test_shift	ROL8	PASS
48	test_muldiv	MUL 3×4	PASS
49	test_muldiv	MUL ×0	PASS
50	test_muldiv	MUL large	PASS
51	test_muldiv	MUL ovf lo	PASS
52	test_muldiv	MULH hi	PASS
53	test_muldiv	DIV 12/4	PASS
54	test_muldiv	DIV /0	PASS
55	test_muldiv	DIVU	PASS
56	test_muldiv	REM 13%4	PASS
57	test_muldiv	REM %0	PASS
58	test_muldiv	REMU 17%5	PASS
59	test_muldiv	MUL ×1	PASS
60	test_loadstore	SW/LW	PASS
61	test_loadstore	SW/LW 2	PASS
62	test_loadstore	SB/LBU	PASS
63	test_loadstore	LB sign	PASS
64	test_loadstore	LBU zext	PASS
65	test_loadstore	LW +2	PASS
66	test_loadstore	LW +4	PASS
67	test_loadstore	LW −2	PASS
68	test_loadstore	LIX	PASS
69	test_loadstore	LIX 0	PASS
70	test_loadstore	Multi SW	PASS
71	test_branch	BEQ taken	PASS
72	test_branch	BEQ !taken	PASS
73	test_branch	BNE taken	PASS
74	test_branch	BNE !taken	PASS
75	test_branch	BLT taken	PASS
76	test_branch	BLT !taken	PASS
77	test_branch	BGE taken	PASS
78	test_branch	BGE !taken	PASS
79	test_branch	BLTU taken	PASS
80	test_branch	BLTU !tkn	PASS
81	test_branch	BGEU taken	PASS

Table 32.1: Complete FPGA test results (continued)

#	Suite	Test Case	Result
82	test_branch	BGEU !tkn	PASS
83	test_branch	BMI taken	PASS
84	test_branch	BMI !taken	PASS
85	test_branch	BPL taken	PASS
86	test_branch	BPL !taken	PASS
87	test_branch	BGT taken	PASS
88	test_branch	BGT !taken	PASS
89	test_branch	BLE taken	PASS
90	test_branch	BLE !taken	PASS
91	test_branch	BHI taken	PASS
92	test_branch	BHI !taken	PASS
93	test_branch	BLS taken	PASS
94	test_branch	BLS !taken	PASS
95	test_jump	J forward	PASS
96	test_jump	JR	PASS
97	test_jump	JALR	PASS
98	test_jump	JX	PASS
99	test_jump	JALX	PASS
100	test_jump	JALX RA set	PASS
101	test_stack	PUSH/POP	PASS
102	test_stack	LIFO 1st	PASS
103	test_stack	LIFO 2nd	PASS
104	test_stack	CMP eq Z	PASS
105	test_stack	CMP lt	PASS
106	test_stack	CMP gt	PASS
107	test_stack	TEST Z	PASS
108	test_stack	TEST !Z	PASS
109	test_stack	TEST clr CV	PASS
110	test_stack	MOV	PASS
111	test_stack	MOV R0	PASS
112	test_stack	SP restore	PASS
113	test_misc	SETF/GETF	PASS
114	test_misc	SETF 2	PASS
115	test_misc	SCF	PASS
116	test_misc	CCF 1→0	PASS

Table 32.1: Complete FPGA test results (continued)

#	Suite	Test Case	Result
117	test_misc	CCF 0→1	PASS
118	test_misc	NOP	PASS
119	test_misc	EXX swap	PASS
120	test_misc	EXX alt	PASS
121	test_misc	EXX R13	PASS
122	test_extended	ADDIX	PASS
123	test_extended	SUBIX	PASS
124	test_extended	ANDIX	PASS
125	test_extended	ORIX	PASS
126	test_extended	XORIX	PASS
127	test_extended	SLLX	PASS
128	test_extended	SRLX	PASS
129	test_extended	SRAX	PASS
130	test_extended	SLLX 0	PASS
131	test_extended	ADDIX 0	PASS
132	test_extended	ADDIX carry	PASS

32.4.1 Test Coverage Summary

Table 32.2 provides a per-suite summary of the test counts and the instruction categories covered.

What Is Not Tested

The ISA test suite focuses on *instruction correctness*: does each instruction compute the right result and set the right flags? It does not directly test:

- Interrupt handling (`SWI`, `RETI`, `DI`, `EI`)—these require a more complex test harness with interrupt sources.
- Block transfer instructions (`LDI`, `LDD`, `LDIR`, `LDDR`, `CPIR`, `FILL`)—these are tested separately in dedicated programs.
- Performance characteristics—the tests verify correctness, not cycle counts.
- Bus timing and peripheral interaction beyond UART TX.

Expanding the test suite to cover these areas is a worthwhile exercise (see the exercises at the end of this chapter).

Table 32.2: Test suite summary by category

Suite	Tests	Instructions Covered
test_alu	20	ADD, SUB, AND, OR, XOR, NEG, GETF
test_addi	10	ADDI
test_shift	17	SLL, SRL, SRA, ROL, ROR, SWAP, SLL4–ROL8
test_muldiv	12	MUL, MULH, DIV, DIVU, REM, REMU
test_loadstore	11	SW, LW, SB, LB, LBU, LIX
test_branch	24	BEQ…BLS (all 12 conditions)
test_jump	6	J, JR, JALR, JX, JALX
test_stack	12	PUSH, POP, CMP, TEST, MOV
test_misc	9	GETF, SETF, SCF, CCF, NOP, EXX
test_extended	11	ADDIX, SUBIX, ANDIX…SRAX
Total	**132**	

32.5 Regression Testing

A test suite is only useful if you run it. Regression testing means running the full suite after every change to the RTL, the assembler, or the test programs themselves. The goal is to catch regressions—changes that accidentally break something that was previously working.

32.5.1 When to Run the Suite

Run `make isa-tests` (simulation) and `make fpga-tests` (hardware) at these points:

- **After any RTL change.** Modified the ALU? Run the suite. Changed the decoder? Run the suite. Refactored the state machine? Run the suite. Even changes that "obviously" do not affect instruction behavior can have surprising consequences.
- **After any assembler change.** A bug in the assembler's code generation can produce incorrect binaries. The test suite catches this because the tests exercise real assembled code.
- **Before committing.** Make the suite a pre-commit gate. If any test fails, the commit should not happen.
- **After tool updates.** Upgrading Yosys, nextpnr, or the Rust toolchain can introduce subtle behavior changes. Run the suite to verify.

32.5.2 Simulation-First Workflow

FPGA builds are slow. A practical workflow runs simulation first and reserves FPGA testing for final validation:

Listing 32.16: Simulation-first regression workflow

```
# Quick check: simulation tests (seconds)
make isa-tests

# Full validation: FPGA tests (minutes)
make fpga-tests
```

The simulation suite uses the same test programs and the same PASS/FAIL checking. The only difference is the execution environment: Icarus Verilog's testbench vs. the ULX3S board. Running `make isa-tests` first gives you fast feedback. If simulation passes, FPGA testing almost always passes too—the exceptions being the kinds of timing-dependent issues discussed in Section 32.3.

32.5.3 Tracking Test Results Over Time

For a project like the SAMPO, where the design evolves over weeks and months, keeping a record of test results is valuable. A straightforward approach is to save the FPGA test output to a timestamped log:

Listing 32.17: Saving timestamped test results

```
make fpga-tests 2>&1 | tee \
    test-results/fpga-$(date +%Y%m%d-%H%M%S).log
```

Over time, this gives you a history of which tests passed at each point in the project's development. If a regression appears, you can compare logs to identify exactly when it was introduced.

Diff Your Logs

When a previously passing test fails, diff the new log against the last known-good log:

```
diff test-results/fpga-20250214-142300.log \
     test-results/fpga-20250215-093000.log
```

The diff will show exactly which test went from PASS to FAIL, narrowing the investigation to a specific instruction or flag behavior.

32.6 Continuous Integration for Hardware

Continuous integration (CI) for software is well-understood: push code, a server builds it, tests run, results appear in a dashboard. CI for *hardware* is less common but entirely feasible, especially for FPGA designs where the "hardware" is a board sitting on a desk rather than a chip in a fab.

32.6.1 The CI Pipeline for SAMPO

A CI pipeline for the SAMPO has two tiers:

Two-Tier CI

Tier 1: Simulation CI runs on any server with Icarus Verilog and the Rust toolchain. It executes `make isa-tests` on every push. No special hardware required. Turnaround: seconds.
Tier 2: Hardware CI requires a dedicated machine with a ULX3S board connected via USB, plus the full open-source FPGA toolchain (Yosys, nextpnr, ecppack, openFPGALoader). It executes `make fpga-tests` on demand or on a schedule. Turnaround: minutes.

Tier 1 is easy to set up in any standard CI system (GitHub Actions, GitLab CI, Jenkins). The simulation tools are lightweight and run on Linux, macOS, or Windows.

Tier 2 requires dedicated hardware. The approach is to set up a "lab runner"—a machine with the ULX3S permanently connected—that can execute hardware test jobs. This is similar to how mobile app CI systems use device farms.

32.6.2 Setting Up a Lab Runner

The requirements for a hardware CI runner are modest:

1. A machine (even a Raspberry Pi or old laptop) running Linux.
2. The open-source FPGA toolchain: Yosys, nextpnr-ecp5, ecppack.
3. `openFPGALoader` for FPGA programming.
4. Python 3 with the `pyserial` package.
5. A ULX3S board connected via USB.
6. The Rust toolchain (for the SAMPO assembler).

The board needs to be permanently powered and connected. The USB connection provides both JTAG programming and serial communication through the FTDI chip.

USB Stability on Long-Running Systems

USB connections can be finicky on systems that run for days or weeks without attention. Occasionally, the FTDI chip's USB interface gets into a bad state and stops responding. A USB hub with per-port power switching lets you power-cycle the board programmatically:

```
# Power cycle USB port (requires uhubctl)
uhubctl -l 1-1 -p 2 -a cycle
```

Include a "reset USB" step at the beginning of your CI job to ensure a clean starting state.

32.6.3 A Minimal CI Script

Here is a minimal CI script that could be triggered by a Git push hook or a cron job:

Listing 32.18: Minimal hardware CI script

```
#!/bin/bash
# hardware-ci.sh: Run FPGA tests and report results

cd /path/to/sampo/verilog

# Clean stale builds
make clean

# Run simulation tests first (fast sanity check)
echo "=== Simulation Tests ==="
if ! make isa-tests; then
    echo "SIMULATION FAILED - skipping FPGA tests"
    exit 1
fi

# Run FPGA tests
echo "=== FPGA Tests ==="
make fpga-tests 2>&1 | tee /tmp/fpga-results.log
RESULT=${PIPESTATUS[0]}

# Archive results
TIMESTAMP=$(date +%Y%m%d-%H%M%S)
cp /tmp/fpga-results.log \
   results/fpga-${TIMESTAMP}.log

exit $RESULT
```

This script:

1. Cleans stale build artifacts.
2. Runs simulation tests as a fast sanity check. If simulation fails, it skips the slower FPGA tests.
3. Runs the full FPGA test suite, saving the output.
4. Archives the results with a timestamp.
5. Exits with the test result code (0 = all pass, nonzero = failure).

32.6.4 What CI Catches

In practice, a hardware CI system for a CPU design catches several categories of issues:

- **Instruction regressions.** A change to the decoder that accidentally breaks a working instruction. This is the most common type and the primary reason for the test suite.
- **Toolchain regressions.** A Yosys or nextpnr update that changes synthesis behavior. The open-source FPGA tools are under active development, and occasionally a new version introduces a bug that affects specific designs.
- **Assembler regressions.** A change to the SAMPO assembler that generates different (incorrect) binary output for the same assembly source.
- **Flaky hardware.** A USB connection that has degraded, a board with thermal issues, or an FTDI chip that needs a power cycle. CI catches these by making failures visible and timestamped.

The Goal: Never Ship a Regression

The purpose of all this infrastructure—the test framework, the looping patch, the serial capture, the CI pipeline—is to ensure that the SAMPO design never moves backward. Every instruction that passes today should still pass tomorrow. 132 tests, zero failures, on every commit. That is the standard.

Exercises

1. **Add an interrupt test.** Write a test program that enables interrupts (`EI`), triggers a software interrupt (`SWI`), and verifies that the interrupt handler executes and returns correctly via `RETI`. Add it to the `ISA_TESTS` list and run it on the FPGA. What new challenges arise when testing asynchronous behavior?

2. **Block transfer test.** Write a test for the **LDIR** (Load, Increment, Repeat) instruction. Set up a source buffer, a destination buffer, and a byte count. After execution, verify that the destination contains an exact copy of the source. Run it in both simulation and on the FPGA.

3. **Stress test.** Create a test that exercises the ALU in a tight loop: compute the Fibonacci sequence up to 10,000 iterations and verify the final value. This tests the CPU under sustained load. Does the FPGA result match simulation?

4. **Build a CI dashboard.** Set up a simple CI pipeline using a cron job and a shell script. Have it run `make fpga-tests` once per hour (or on every Git push), save the results to a log directory, and generate a one-line summary file (`PASS` or `FAIL` with a timestamp). How would you add email or Slack notifications for failures?

5. **Performance comparison.** Modify the testbench to count cycles for each test suite (the `cycles` counter in the SoC already tracks this). Compare the cycle counts between simulation and FPGA. Are they identical? If not, what accounts for the difference?

What's Next

With 132 tests passing on real hardware, the SAMPO CPU is validated from ISA specification through silicon. We have proven that every instruction works correctly, that every flag is generated properly, and that the hardware matches the simulation. But the Verilog implementation is not the only way to build the SAMPO. In the next part, we will reimplement the entire CPU in Amaranth, a Python-based hardware description language that brings modern software engineering practices—type safety, unit testing, and a powerful standard library—to digital design. The 132-test suite will serve as our acceptance criterion: the Amaranth implementation must pass every test that the Verilog implementation passes.

Part X

The Amaranth Implementation

Chapter 33

Python HDL with Amaranth

Every line of Verilog we wrote in Chapters 18 and 21 to 23 and **??** was hand-crafted: each `always @(*)` block, each signal declaration, each sensitivity list spelled out by name. The result is a working SAMPO CPU, proven by 132 passing tests. But what if we could describe the same hardware in a language that catches typos before synthesis, manages signal widths automatically, and lets us write testbenches with `assert` instead of `$display`?

That language is Amaranth HDL—a Python-based hardware description language that generates synthesisable Verilog from ordinary Python objects. In this chapter we will learn what Amaranth is, how it represents hardware, and why it is a compelling alternative to writing Verilog by hand. In Chapter 34 we will put it to work and rewrite the entire SAMPO in Amaranth.

33.1 What is Amaranth HDL?

Amaranth HDL (formerly known as nMigen) is an open-source hardware description language embedded in Python. Rather than inventing a new syntax, Amaranth uses Python classes, operators, and context managers to describe digital circuits. When you "elaborate" an Amaranth design, the framework traverses your Python objects and emits Verilog (or RTLIL for direct Yosys integration) that can be synthesised for an FPGA or simulated with standard tools.

Amaranth HDL

Amaranth HDL is a Python library for building synchronous digital logic. A designer creates Python objects representing signals, modules, and connections; the Amaranth *elaboration* pass converts these objects into synthesisable Verilog. Amaranth is not a simulator that runs Python at gate speed—it is a *metaprogramming* tool that uses Python to *generate* hardware.

The key insight is that *Python runs at elaboration time, not at hardware time*. A Python `for` loop does not generate a hardware loop; it generates multiple copies of combinational logic, one per iteration. A Python `if` statement selects what hardware to generate, not what hardware to execute. This distinction trips up every newcomer at least once.

Why use Amaranth instead of writing Verilog directly? Three reasons stand out:

1. **Type safety.** Amaranth tracks signal widths and prevents silent truncation. In Verilog, assigning a 17-bit expression to a 16-bit register silently drops the carry bit. Amaranth makes widths explicit and warns when they mismatch.
2. **Composability.** Amaranth modules are Python classes. You can pass parameters, inherit behaviour, and compose modules with ordinary Python code. No need for Verilog's cumbersome `generate` blocks or `parameter` declarations.
3. **Testability.** Testbenches are Python functions that can use `assert`, print intermediate values, and integrate with pytest. No more parsing `$display` output.

Amaranth is Not HLS

High-level synthesis (HLS) tools attempt to convert arbitrary C or C++ programs into hardware. Amaranth does *not* do this. You are still designing hardware—thinking about clock domains, combinational paths, and state machines. Amaranth merely gives you a better language for expressing those designs.

33.2 Signals, Modules, and Elaboration

Every Amaranth design is built from three core concepts: *signals*, *modules*, and *elaboration*. Let's look at each one using the SAMPO ALU as our running example.

33.2.1 Signals

A `Signal` is Amaranth's equivalent of a Verilog `wire` or `reg`. It represents a bundle of bits whose value can change every clock cycle. Here is how the ALU in `rtl/alu.py` declares its input and output signals:

Listing 33.1: ALU signal declarations from `rtl/alu.py`

```
class ALU(Elaboratable):
    def __init__(self):
        # Inputs
        self.a = Signal(16)
        self.b = Signal(16)
        self.op = Signal(ALUOp)
```

```
        self.carry_in = Signal()
        self.shift_amt = Signal(4)

        # Outputs
        self.result = Signal(16)
        self.flag_n = Signal()
        self.flag_z = Signal()
        self.flag_c = Signal()
        self.flag_v = Signal()
```

The `Signal` constructor takes a width parameter. `Signal(16)` creates a 16-bit unsigned signal; `Signal()` with no arguments creates a 1-bit signal (a single wire). You can also pass an Amaranth `Enum` type, as we do with `Signal(ALUOp)`—the signal automatically adopts the width needed to represent all enum values.

Compare this with the equivalent Verilog declaration from `verilog/rtl/alu.v`:

Listing 33.2: ALU port declarations in Verilog (`verilog/rtl/alu.v`)

```
module alu (
    input  wire [15:0] a,
    input  wire [15:0] b,
    input  wire [3:0]  op,
    input  wire        carry_in,
    output reg  [15:0] result,
    output wire        flag_n,
    output wire        flag_z,
    output reg         flag_c,
    output reg         flag_v
);
```

In Verilog we must decide at declaration time whether each output is a `wire` or `reg`, and whether a signal is an `input` or `output`. Amaranth defers these decisions to the elaboration pass. A `Signal` is neither an input nor an output until it is connected.

Named Signals for Debugging

Amaranth allows an optional `name` parameter: `Signal(16, name="pc")`. If you omit it, Amaranth infers the name from the Python attribute name. In the generated Verilog, `self.result` becomes a signal named `result`. Explicit names are useful when you create signals inside local variables rather than `self` attributes.

33.2.2 Modules and Elaboratable

In Amaranth, a hardware module is a Python class that inherits from `Elaboratable` and implements an `elaborate(self, platform)` method. This method receives a `Module` object and uses it to describe the hardware:

Listing 33.3: The elaborate method pattern

```
class ALU(Elaboratable):
    def __init__(self):
        self.a = Signal(16)
        self.b = Signal(16)
        self.result = Signal(16)

    def elaborate(self, platform):
        m = Module()
        # ... describe hardware using m ...
        return m
```

The `Module` object `m` is the central building block. It provides two critical domain accessors:

- `m.d.comb` — for combinational (unclocked) logic
- `m.d.sync` — for synchronous (clocked) logic

It also provides conditional constructs (`m.If`, `m.Switch`) and a finite state machine helper (`m.FSM`). We will explore each of these in the following sections.

33.2.3 Submodules

To instantiate one module inside another, use `m.submodules`. Here is how the SAMPO CPU instantiates its ALU, shifter, register file, and decoder in `rtl/cpu.py`:

Listing 33.4: Submodule instantiation in the CPU (`rtl/cpu.py`)

```
def elaborate(self, platform):
    m = Module()

    # Instantiate submodules
    m.submodules.alu = alu = ALU()
    m.submodules.shifter = shifter = Shifter()
    m.submodules.regfile = regfile = RegisterFile()
    m.submodules.decoder = decoder = Decoder()
```

The `m.submodules.alu = alu = ALU()` idiom simultaneously registers the ALU as a submodule (so Amaranth knows to elaborate it) and binds the instance to a local variable `alu` for easy access. The attribute name (`.alu`) becomes the instance name in the generated Verilog.

Compare this with the Verilog instantiation:

Listing 33.5: Verilog submodule instantiation

```
alu alu_inst (
    .a(alu_a),
    .b(alu_b),
    .op(alu_op),
    .result(alu_result),
    .flag_n(alu_flag_n),
    .flag_z(alu_flag_z),
    .flag_c(alu_flag_c),
    .flag_v(alu_flag_v)
);
```

In Verilog, instantiation and port connection happen simultaneously. In Amaranth, you instantiate first, then connect signals separately with `m.d.comb +=` statements. This separation makes it possible to connect signals conditionally or in different parts of the code.

33.2.4 Elaboration

"Elaboration" is the process of converting Python objects into an intermediate representation (IR) that can be translated into Verilog or RTLIL. When you call an Amaranth back-end (the Verilog generator or the simulator), it calls your top-level module's `elaborate()` method, which in turn triggers elaboration of all submodules, recursively. The result is a complete netlist.

Elaboration happens *once*. After it completes, Python is finished—the resulting Verilog is a static description of hardware. There is no Python interpreter running inside the FPGA. The Python code is scaffolding; the Verilog output is the building.

33.3 Combinational vs. Synchronous Domains

The distinction between combinational and synchronous logic is fundamental to digital design, and Amaranth makes it explicit through *clock domains*. Every assignment in Amaranth belongs to either the `comb` domain (combinational) or the `sync` domain (synchronous, clocked on the rising edge).

33.3.1 Combinational Assignments

Combinational assignments use `m.d.comb +=`. They describe logic that is always active—the output changes whenever any input changes, with no clock involved. Here is how the ALU connects its flag outputs in `rtl/alu.py`:

Listing 33.6: Combinational flag generation (`rtl/alu.py`)

```
# Flag generation
m.d.comb += [
    # Negative: MSB of result
    self.flag_n.eq(self.result[15]),

    # Zero: result is zero
    self.flag_z.eq(self.result == 0),

    # Carry: bit 16 of result (for ADD/SUB)
    self.flag_c.eq(result_wide[16]),
]
```

This is equivalent to Verilog's continuous assignment (`assign`):

Listing 33.7: Equivalent Verilog flag generation

```
assign flag_n = result[15];
assign flag_z = (result == 16'h0000);
```

The `.eq()` method is Amaranth's assignment operator. It does *not* test for equality; it creates a connection. The list syntax `m.d.comb += [...]` allows multiple assignments in a single statement.

Assignment with .eq(), Not =

The Python = operator assigns Python variables. The Amaranth `.eq()` method creates hardware connections. Writing `self.result = self.a + self.b` creates a new Python object but does *not* generate any hardware. You must write `m.d.comb += self.result.eq(self.a + self.b)` instead.

33.3.2 Synchronous Assignments

Synchronous assignments use `m.d.sync +=`. They create flip-flop-based registers that update on the rising clock edge. In the SAMPO CPU's EXECUTE state, the ALU result is latched into a pipeline register:

Listing 33.8: Synchronous register update in EXECUTE (`rtl/cpu.py`)

```
# ALU register-register
with m.Case(InstType.ALU_REG):
    m.d.sync += alu_result.eq(alu.result)
    m.d.sync += [
        flags[7].eq(alu.flag_n),
        flags[6].eq(alu.flag_z),
        flags[5].eq(alu.flag_c),
        flags[4].eq(alu.flag_v),
    ]
    m.next = CPUState.WRITEBACK
```

The `m.d.sync +=` statement generates Verilog equivalent to:

Listing 33.9: Equivalent Verilog synchronous assignment

```
always @(posedge clk) begin
    alu_result <= alu_result_wire;
    flags[7]  <= alu_flag_n;
    flags[6]  <= alu_flag_z;
    flags[5]  <= alu_flag_c;
    flags[4]  <= alu_flag_v;
end
```

Notice that Amaranth always uses non-blocking assignment semantics for synchronous logic. There is no way to accidentally use blocking assignments in a clocked domain, because the concept does not exist in the Amaranth model. This eliminates an entire class of simulation-synthesis mismatches that plague Verilog designs.

33.3.3 Conditional Logic

Amaranth uses Python context managers (`with` statements) for conditional hardware. Consider the branch condition evaluation in `rtl/cpu.py`:

Listing 33.10: Branch condition evaluation using `m.Switch` (`rtl/cpu.py`)

```
with m.Switch(decoder.branch_cond):
    with m.Case(0x0):  # BEQ
        m.d.comb += branch_taken.eq(flag_z)
    with m.Case(0x1):  # BNE
        m.d.comb += branch_taken.eq(~flag_z)
    with m.Case(0x2):  # BLT
        m.d.comb += branch_taken.eq(flag_n != flag_v)
    with m.Case(0x3):  # BGE
        m.d.comb += branch_taken.eq(flag_n == flag_v)
```

```
    with m.Default():
        m.d.comb += branch_taken.eq(0)
```

The `m.Switch`/`m.Case` construct generates a Verilog `case` statement. Similarly, `m.If`/`m.Elif`/`m.Else` generates `if`/`else` chains:

Listing 33.11: Conditional logic with `m.If` (`rtl/cpu.py`)

```
# ALU A input: usually rs1, but rd for ADDI
with m.If(decoder.inst_type == InstType.ALU_IMM):
    m.d.comb += alu_a.eq(regfile.rd_data1)
    m.d.comb += regfile.rd_addr1.eq(decoder.rd)
with m.Else():
    m.d.comb += alu_a.eq(rs1_data)
```

Context Managers Generate Hardware

The `with m.If(...)` construct does *not* evaluate the condition at elaboration time. It creates a multiplexer in the generated hardware. The Python `with` block defines which assignments are gated by that condition. Both the "if" and "else" branches exist in the final hardware; the condition selects which one drives the output.

33.3.4 Enumerations

Amaranth integrates with Python's `Enum` to create type-safe constants. The SAMPO ALU operations are defined as an enumeration in `rtl/alu.py`:

Listing 33.12: ALU operation enum (`rtl/alu.py`)

```
class ALUOp(Enum, shape=4):
    """ALU operation codes."""
    ADD  = 0x0
    SUB  = 0x1
    AND  = 0x2
    OR   = 0x3
    XOR  = 0x4
    SLL  = 0x5
    SRL  = 0x6
    SRA  = 0x7
    MUL  = 0x8
    MULH = 0x9
    DIV  = 0xA
    REM  = 0xB
```

```
    PASS_A = 0xC
    PASS_B = 0xD
    NOT  = 0xE
    NEG  = 0xF
```

The `shape=4` parameter tells Amaranth that this enum requires 4 bits to encode. When you create a `Signal(ALUOp)`, it automatically becomes 4 bits wide, and you can compare it against enum members: `self.op == ALUOp.ADD`.

In Verilog, the same constants are typically handled with preprocessor macros:

Listing 33.13: Verilog ALU operation constants (from `verilog/rtl/sampo_pkg.vh`)

```
`define ALU_ADD    4'h0
`define ALU_SUB    4'h1
`define ALU_AND    4'h2
// ...
```

The Amaranth approach catches mismatches at elaboration time—you cannot accidentally use a 5-bit value where a 4-bit enum is expected. Verilog's `` `define `` macros provide no such protection.

33.3.5 The FSM Helper

One of Amaranth's most useful features is its built-in finite state machine helper. The SAMPO CPU's state machine is defined using `m.FSM`:

Listing 33.14: State machine definition in the CPU (`rtl/cpu.py`)

```
class CPUState(Enum, shape=4):
    """CPU state machine states."""
    RESET       = 0
    FETCH       = 1
    FETCH_EXT   = 2
    DECODE      = 3
    EXECUTE     = 4
    MEMORY      = 5
    WRITEBACK   = 6
    HALTED      = 7

# Inside elaborate():
with m.FSM(reset=CPUState.RESET) as fsm:
    m.d.comb += state.eq(fsm.state)

    with m.State(CPUState.RESET):
```

```
            m.d.sync += [
                pc.eq(self.reset_vector),
                flags.eq(0),
                int_enabled.eq(0),
                cycles.eq(0),
            ]
            m.next = CPUState.FETCH

        with m.State(CPUState.FETCH):
            m.d.comb += [
                self.mem_addr.eq(pc),
                self.mem_valid.eq(1),
            ]
            with m.If(self.mem_ready):
                m.d.sync += instr.eq(self.mem_rdata)
                m.next = CPUState.DECODE
```

Each `m.State(...)` block defines the behaviour in a given state. The `m.next = CPUState.FETCH` assignment determines the next state. If no `m.next` is set, the FSM stays in its current state—which is exactly how the HALTED state works:

Listing 33.15: The halted state—no `m.next` means stay halted

```
with m.State(CPUState.HALTED):
    # Stay halted (can be reset externally)
    pass
```

Amaranth's FSM helper generates efficient state encoding, handles default next-state logic, and ensures every state is reachable from the reset state during elaboration checks. In Verilog, all of this requires manual bookkeeping with separate `state` and `next_state` registers.

33.4 The Amaranth Simulator

Amaranth includes a built-in simulator that runs your design without requiring external tools like Icarus Verilog or Verilator. The simulator is a Python object that drives your module's signals and captures outputs, all within a Python script.

33.4.1 Setting Up a Simulation

The basic pattern is: create your design, create a `Simulator`, add a clock and one or more testbenches, then run. Here is the ALU test from `rtl/test_cpu.py`:

Listing 33.16: ALU testbench setup (`rtl/test_cpu.py`)

```
from amaranth.sim import Simulator
from .alu import ALU, ALUOp

def test_alu():
    """Test ALU operations."""
    alu = ALU()

    def testbench():
        # Test ADD
        yield alu.a.eq(100)
        yield alu.b.eq(50)
        yield alu.op.eq(ALUOp.ADD)
        yield
        result = yield alu.result
        assert result == 150, f"ADD failed: {result} != 150"
        print(f"ADD: 100 + 50 = {result}")

        # Test SUB
        yield alu.a.eq(100)
        yield alu.b.eq(30)
        yield alu.op.eq(ALUOp.SUB)
        yield
        result = yield alu.result
        assert result == 70, f"SUB failed: {result} != 70"
        print(f"SUB: 100 - 30 = {result}")

    sim = Simulator(alu)
    sim.add_testbench(testbench)
    sim.run()
```

The testbench is a Python generator function (it uses `yield` to communicate with the simulator). Each `yield signal.eq(value)` drives a signal to a new value. A bare `yield` advances the simulation by one clock cycle. Reading a signal's value uses `value = yield signal`.

33.4.2 Testing Combinational Logic

For purely combinational modules (like the ALU), the testbench can skip the clock. After setting inputs and yielding once, the combinational outputs are immediately available. The ALU test continues with logic and flag tests:

Listing 33.17: Testing ALU flags (`rtl/test_cpu.py`)

```
# Test zero flag
```

```
    yield alu.a.eq(50)
    yield alu.b.eq(50)
    yield alu.op.eq(ALUOp.SUB)
    yield
    result = yield alu.result
    zero = yield alu.flag_z
    assert result == 0, f"Zero result failed: {result} != 0"
    assert zero == 1, f"Zero flag not set"
    print(f"SUB: 50 - 50 = {result}, Z={zero}")

    # Test negative flag
    yield alu.a.eq(0)
    yield alu.b.eq(1)
    yield alu.op.eq(ALUOp.SUB)
    yield
    result = yield alu.result
    neg = yield alu.flag_n
    assert result == 0xFFFF, f"Negative result: 0x{result:04X}"
    assert neg == 1, f"Negative flag not set"
    print(f"SUB: 0 - 1 = 0x{result:04X}, N={neg}")
```

Notice how the testbench uses Python's `assert` and f-strings to produce clear, actionable error messages. Compare this with a typical Verilog testbench that checks the same condition:

Listing 33.18: Equivalent Verilog testbench assertion

```
if (result !== 16'hFFFF) begin
    $display("FAIL: SUB 0-1 = %h, expected FFFF", result);
    $finish;
end
```

The Python version is more expressive and integrates with standard testing frameworks.

33.4.3 Testing Sequential Logic

For modules with clocked logic, add a clock to the simulator. The register file test from `rtl/test_cpu.py` demonstrates this:

Listing 33.19: Register file testbench with clock (`rtl/test_cpu.py`)

```
def test_regfile():
    """Test register file."""
    from .regfile import RegisterFile
```

```
    rf = RegisterFile()

    def testbench():
        # Write to R4
        yield rf.wr_addr.eq(4)
        yield rf.wr_data.eq(0x1234)
        yield rf.wr_en.eq(1)
        yield
        yield rf.wr_en.eq(0)
        yield

        # Read from R4
        yield rf.rd_addr1.eq(4)
        yield
        data = yield rf.rd_data1
        assert data == 0x1234, f"R4 read failed: 0x{data:04X}"
        print(f"R4 = 0x{data:04X}")

        # Write to R0 (should be ignored)
        yield rf.wr_addr.eq(0)
        yield rf.wr_data.eq(0xFFFF)
        yield rf.wr_en.eq(1)
        yield
        yield rf.wr_en.eq(0)
        yield

        # Read from R0 (should be 0)
        yield rf.rd_addr1.eq(0)
        yield
        data = yield rf.rd_data1
        assert data == 0, f"R0 read: 0x{data:04X} != 0x0000"
        print(f"R0 = 0x{data:04X} (writes ignored)")

    sim = Simulator(rf)
    sim.add_clock(1e-6)  # 1 MHz clock
    sim.add_testbench(testbench)
    sim.run()
```

The `sim.add_clock(1e-6)` call adds a 1 MHz clock (the frequency does not matter for functional simulation, but it sets the time scale for VCD output). Each `yield` in the testbench now advances by one clock cycle.

33.4.4 VCD Waveform Output

The simulator can write Value Change Dump (VCD) files for viewing in GTKWave. Wrap the `sim.run()` call in a context manager:

Listing 33.20: Capturing VCD waveforms

```
with sim.write_vcd("sampo_test.vcd"):
    sim.run()
```

This produces a VCD file containing every signal in the design, with full hierarchy preserved. Open it in GTKWave:

Listing 33.21: Viewing Amaranth simulation waveforms

```
python3 -m rtl.test_cpu
gtkwave sampo_test.vcd
```

The resulting waveform shows the same signals you would see from an Icarus Verilog simulation, but the testbench that produced it is a Python script rather than a Verilog testbench module.

33.4.5 Full System Simulation

The most powerful use of the Amaranth simulator is testing the complete SoC. The `test_hello_world` function in `rtl/test_cpu.py` loads an assembled binary, instantiates the entire SoC, and checks the UART output:

Listing 33.22: Full SoC testbench (`rtl/test_cpu.py`)

```
def test_hello_world():
    """Test the hello world program."""
    bin_path = Path(__file__).parent.parent / "examples" / "hello.bin"
    program = load_binary(bin_path)

    soc = SampoSoC(program=program, reset_vector=0x0100)
    output = []

    def testbench():
        yield soc.tx_ready.eq(1)

        for cycle in range(10000):
            if (yield soc.tx_valid):
                char = yield soc.tx_data
                output.append(chr(char))
                print(chr(char), end='', flush=True)
```

```
            if (yield soc.halted):
                print(f"\n\nCPU halted after {cycle} cycles")
                print(f"PC: 0x{(yield soc.pc):04X}")
                break

            yield

    sim = Simulator(soc)
    sim.add_clock(1e-6)
    sim.add_testbench(testbench)

    with sim.write_vcd("sampo_test.vcd"):
        sim.run()

    result = ''.join(output)
    expected = "Hello, Sampo!\n"
    assert result == expected, f"Expected: {repr(expected)}"
```

This test does everything: it exercises the CPU state machine through all eight states, tests instruction fetch from RAM, decodes and executes a sequence of instructions, and verifies I/O output through the UART. If any instruction is mis-decoded or any state transition is wrong, the output will not match.

Test Isolation

Notice that each test function creates a fresh module instance. There is no shared state between tests. This is a major advantage over Verilog testbenches, where global state and initial blocks can create subtle ordering dependencies.

33.5 Generating Verilog from Python

The whole point of Amaranth is to produce synthesisable Verilog. After designing and testing your module in Python, you generate Verilog for synthesis with one of two back-ends.

33.5.1 Using the Verilog Back-End

The most direct approach uses the `amaranth.back.verilog` module:

Listing 33.23: Generating Verilog from an Amaranth module

```
from amaranth.back.verilog import convert
```

```
from rtl.soc import SampoSoC

soc = SampoSoC(reset_vector=0x0100)

# Generate Verilog
verilog_text = convert(soc, name="sampo_soc", ports=[
    soc.tx_data,
    soc.tx_valid,
    soc.tx_ready,
    soc.rx_data,
    soc.rx_valid,
    soc.rx_ready,
    soc.halted,
    soc.pc,
])

with open("sampo_soc.v", "w") as f:
    f.write(verilog_text)
```

The `ports` parameter specifies which signals should become top-level Verilog ports. Internal signals are kept internal. The `name` parameter sets the Verilog module name.

Generated Verilog is Not Human-Readable

The Verilog output from Amaranth is correct and synthesisable, but it is not intended to be read or edited by humans. Signal names are mangled to avoid collisions, and the structure bears little resemblance to the Python source. Treat the generated Verilog as a build artifact, not a source file. If you need to debug the hardware, debug the Amaranth source.

33.5.2 Understanding the Generated Output

The generated Verilog contains several distinctive patterns:

- **Signal names** include hierarchy prefixes, such as `alu__result` for the ALU's result signal.
- **Combinational logic** appears in `always @(*)` blocks with explicit sensitivity.
- **Synchronous logic** appears in `always @(posedge clk)` blocks with a synchronous reset.
- **FSM state encoding** uses one-hot or binary encoding depending on the number of states. The SAMPO's 8-state CPU FSM typically generates binary encoding.
- **Memories** are represented using Verilog `reg` arrays that synthesis tools infer as block RAM.

33.5.3 Using the RTLIL Back-End

For Yosys-based synthesis flows, Amaranth can generate RTLIL directly, bypassing the Verilog-to-RTLIL parsing step entirely. This avoids potential parsing issues with generated Verilog and is often faster:

Listing 33.24: Generating RTLIL for direct Yosys consumption

```
from amaranth.back.rtlil import convert

rtlil_text = convert(soc, name="sampo_soc", ports=[
    soc.tx_data,
    soc.tx_valid,
    soc.halted,
])

with open("sampo_soc.il", "w") as f:
    f.write(rtlil_text)
```

The RTLIL output can be loaded directly into Yosys with `read_ilang sampo_soc.il`, then synthesised and placed-and-routed as usual.

33.5.4 Platform-Targeted Builds

Amaranth also provides a platform abstraction for targeting specific FPGA boards. Instead of manually running Yosys and nextpnr, you can let Amaranth drive the entire build:

Listing 33.25: Platform-targeted build (conceptual)

```
from amaranth_boards.ulx3s import ULX3S85FPlatform

platform = ULX3S85FPlatform()
platform.build(soc, do_program=True)
```

This invokes Yosys, nextpnr-ecp5, and ecppack in sequence, then programs the FPGA. We will explore this workflow in detail in Chapter 34.

33.6 Comparison with Direct Verilog

Having seen both approaches throughout this book, we are now in a position to make a fair comparison. Let's look at several dimensions.

33.6.1 Code Volume

The following table compares the line counts of the SAMPO implementation in both languages. These are the actual source files from the project:

Table 33.1: Source file line counts: Amaranth vs. Verilog

Module	Amaranth (Python)	Verilog
ALU + Shifter	285 lines	220 lines
Register File	166 lines	69 lines
Decoder	451 lines	400+ lines
CPU Core	535 lines	550+ lines
SoC	231 lines	200+ lines

The Amaranth versions tend to be slightly longer due to Python's verbosity (explicit `self.` prefixes, `m.d.comb +=` boilerplate). But they include features that Verilog requires separate files for—enumerations, type annotations, and docstrings.

33.6.2 Safety and Error Detection

Amaranth catches several classes of errors at elaboration time that Verilog only reveals during simulation or, worse, on hardware:

Table 33.2: Error detection: Amaranth vs. Verilog

Error Type	Amaranth	Verilog
Width mismatch	Elaboration	Lint warning
Missing sensitivity list	Impossible	Simulation bug
Blocking in sequential	Impossible	Simulation bug
Undriven signal	Elaboration	Lint warning
Invalid enum value	Elaboration	Undetected
Typo in signal name	Python error	Undetected

The "Impossible" entries are the most valuable. Amaranth's programming model eliminates entire categories of bugs by construction. You cannot write a sensitivity list because Amaranth does not have them. You cannot use blocking assignment in a synchronous domain because the concept does not exist.

33.6.3 Testability

Verilog testbenches are themselves Verilog modules. They are verbose, hard to parameterize, and difficult to integrate with continuous integration systems. Amaranth testbenches are Python functions that can:

- Use `assert` for automatic pass/fail
- Read test vectors from files, databases, or the network
- Generate random test inputs with Python's `random` module
- Run under pytest with automatic test discovery
- Produce structured output (JSON, CSV) for analysis

33.6.4 Synthesis Quality

The generated Verilog synthesises to essentially identical hardware. Both the hand-written Verilog and the Amaranth-generated Verilog pass through the same Yosys synthesis flow and produce comparable LUT counts and timing. The generated Verilog is not "worse" than hand-written Verilog—it is just harder to read.

33.6.5 Learning Curve

Amaranth requires knowing Python *and* digital design. It does not replace the need to understand clock domains, setup and hold timing, or FSM design. If anything, the Python abstraction can obscure the underlying hardware, making it harder for beginners to develop hardware intuition.

For engineers who already know both Python and digital design, Amaranth is a productivity multiplier. For those learning hardware design for the first time, starting with Verilog builds stronger intuition about what the hardware is actually doing.

Both Versions Exist

This book presents the SAMPO in both Verilog (Part VI) and Amaranth (Part X). The two implementations are functionally identical. Having both lets you compare approaches and choose the one that fits your workflow.

33.6.6 When to Choose Which

Exercises

1. **Signal Exploration.** Install Amaranth (`pip install amaranth`) and write a minimal `Elaboratable` module with two 8-bit input signals and one 8-bit output. Assign the output to the sum of the

Table 33.3: When to use Amaranth vs. Verilog

Favour Amaranth	Favour Verilog
Parameterized, reusable IP cores	Tight integration with existing Verilog IP
Complex testbenches with data-driven tests	Team where everyone knows Verilog
Rapid prototyping	Vendor-specific tool requirements
Designs that benefit from metaprogramming (e.g., generating N copies of a pipeline stage)	Maximum control over generated netlist
Python-centric development teams	Industry-standard design flow

inputs using `m.d.comb`. Generate the Verilog and examine the output. What does Amaranth generate for an 8-bit addition? Is a 9th bit (carry) visible in the generated Verilog?

2. **Counter Module.** Write an Amaranth module that implements a 16-bit up-counter with a synchronous reset input. Use `m.d.sync` for the counter register and `m.If` for the reset condition. Write a testbench that resets the counter, lets it run for 10 cycles, and asserts the counter value equals 10.

3. **ALU Extension.** Add a new `ADC` (add with carry) operation to the SAMPO ALU in `rtl/alu.py`. The operation should compute `result` $= A+B+$`carry_in`. Add a test case to `rtl/test_cpu.py` that sets `carry_in` to 1 and verifies the result is one greater than a plain `ADD`.

4. **VCD Analysis.** Run the `test_hello_world` test from `rtl/test_cpu.py` to generate a VCD file. Open it in GTKWave and trace the CPU through one complete instruction cycle: FETCH → DECODE → EXECUTE → WRITEBACK → FETCH. Identify the signals for `pc`, `state`, `instr`, and `alu_result`. How many clock cycles does one instruction take?

5. **Verilog Comparison.** Take the ALU module from `rtl/alu.py` and generate Verilog using `amaranth.back.verilog.convert()`. Compare the generated Verilog with the hand-written version in `verilog/rtl/alu.v`. Identify three structural differences. Are there any functional differences?

What's Next

Now that we understand Amaranth's signals, modules, domains, and simulation infrastructure, we are ready to see it in action. In Chapter 34, we will walk through the complete SAMPO implementation

in Amaranth—the CPU core, ALU, register file, decoder, and system-on-chip—comparing each module side by side with its Verilog counterpart and exploring the testing workflow that makes Amaranth a compelling choice for CPU design.

Chapter 34

Sampo in Amaranth

In Chapter 33 we learned the mechanics of Amaranth HDL—signals, modules, domains, and the simulator. Now we put that knowledge to work. The SAMPO CPU has a complete Amaranth implementation that lives alongside the hand-written Verilog: every module from the ALU to the system-on-chip, tested with Python testbenches, and capable of generating synthesisable Verilog for the same FPGA target.

This chapter walks through the Amaranth implementation module by module, comparing each one with its Verilog counterpart. We will see where Amaranth shines, where it requires more code, and how to test the entire CPU from a single Python script.

34.1 The Amaranth CPU Module

The CPU module in `rtl/cpu.py` is the heart of the Amaranth implementation. It instantiates the ALU, shifter, register file, and decoder as submodules, wires them together, and implements the multi-cycle state machine that drives instruction execution.

34.1.1 Module Interface

The `SampoCPU` class begins with its interface declarations in `__init__`:

Listing 34.1: CPU interface signals (`rtl/cpu.py`)

```
class SampoCPU(Elaboratable):
    def __init__(self, reset_vector=0x0100):
        self.reset_vector = reset_vector

        # Memory interface
```

```
        self.mem_addr = Signal(16)
        self.mem_rdata = Signal(16)
        self.mem_wdata = Signal(16)
        self.mem_we = Signal()
        self.mem_be = Signal(2)
        self.mem_valid = Signal()
        self.mem_ready = Signal()

        # I/O interface
        self.io_addr = Signal(8)
        self.io_rdata = Signal(8)
        self.io_wdata = Signal(8)
        self.io_rd = Signal()
        self.io_wr = Signal()

        # Control
        self.halted = Signal()
        self.irq = Signal()

        # Debug outputs
        self.pc = Signal(16)
        self.state = Signal(CPUState)
        self.cycles = Signal(32)
```

The reset vector is a Python parameter, not a hardware signal—it is baked into the design at elaboration time, just like a Verilog `parameter`. The memory interface uses the same signal names as the Verilog version: `mem_addr`, `mem_rdata`, `mem_wdata`, `mem_we`, `mem_be`, `mem_valid`, and `mem_ready`.

Python Parameters vs. Hardware Signals

The `reset_vector` argument to `__init__` is resolved at elaboration time. You can pass any Python expression—`SampoCPU(reset_vector=0x0100)` or `SampoCPU(reset_vector=int(sys.argv[1], 16))`—and the value becomes a constant in the generated Verilog. In the Verilog implementation, this is declared as `parameter RESET_VECTOR = 16'h0100`.

34.1.2 Submodule Instantiation and Wiring

The `elaborate` method begins by instantiating submodules and creating internal state signals:

Listing 34.2: CPU elaboration: submodules and state (`rtl/cpu.py`)

```
def elaborate(self, platform):
    m = Module()
```

```
    # Instantiate submodules
    m.submodules.alu = alu = ALU()
    m.submodules.shifter = shifter = Shifter()
    m.submodules.regfile = regfile = RegisterFile()
    m.submodules.decoder = decoder = Decoder()

    # CPU state
    state = Signal(CPUState, reset=CPUState.RESET)
    pc = Signal(16, reset=self.reset_vector)
    flags = Signal(8)
    cycles = Signal(32)

    # Instruction registers
    instr = Signal(16)
    instr_ext = Signal(16)

    # Pipeline registers
    alu_result = Signal(16)
    mem_addr_reg = Signal(16)
    mem_data_reg = Signal(16)
```

Compare this with the Verilog version in `verilog/rtl/cpu.v`, where the same setup requires explicit port-by-port wiring:

Listing 34.3: Verilog CPU: register file instantiation (`verilog/rtl/cpu.v`)

```
reg  [3:0]  rf_rd_addr1, rf_rd_addr2, rf_wr_addr;
wire [15:0] rf_rd_data1, rf_rd_data2;
reg  [15:0] rf_wr_data;
reg         rf_wr_en;
reg         rf_exx;

regfile regfile_inst (
    .clk(clk),
    .rst_n(rst_n),
    .rd_addr1(rf_rd_addr1),
    .rd_addr2(rf_rd_addr2),
    .wr_addr(rf_wr_addr),
    .rd_data1(rf_rd_data1),
    .rd_data2(rf_rd_data2),
    .wr_data(rf_wr_data),
    .wr_en(rf_wr_en),
    .exx(rf_exx),
    .sp_wr_data(rf_sp_wr_data),
    .sp_wr_en(rf_sp_wr_en)
```

```
);
```

The Verilog version requires 14 lines of intermediate signal declarations plus 15 lines of port connections. The Amaranth version creates the instance in one line and connects signals incrementally through `m.d.comb +=` statements later in the code. The signals are accessed directly as attributes: `regfile.rd_data1` instead of the locally-declared `rf_rd_data1`.

The decoder and register file are connected with combinational assignments:

Listing 34.4: Connecting the decoder and register file (`rtl/cpu.py`)

```
# Connect decoder
m.d.comb += [
    decoder.instr.eq(instr),
    decoder.imm16.eq(instr_ext),
]

# Register file read addresses (from decoder)
m.d.comb += [
    regfile.rd_addr1.eq(decoder.rs1),
    regfile.rd_addr2.eq(decoder.rs2),
]
```

34.1.3 The State Machine

The CPU's finite state machine follows the same eight-state design as the Verilog implementation: RESET → FETCH → DECODE → EXECUTE → MEMORY → WRITEBACK. Here are the critical states:

Listing 34.5: FETCH and DECODE states (`rtl/cpu.py`)

```
with m.FSM(reset=CPUState.RESET) as fsm:
    m.d.comb += state.eq(fsm.state)

    with m.State(CPUState.FETCH):
        m.d.comb += [
            self.mem_addr.eq(pc),
            self.mem_valid.eq(1),
        ]
        with m.If(self.mem_ready):
            m.d.sync += instr.eq(self.mem_rdata)
            m.next = CPUState.DECODE

    with m.State(CPUState.DECODE):
```

```
        with m.If(decoder.is_extended):
            m.next = CPUState.FETCH_EXT
        with m.Else():
            m.next = CPUState.EXECUTE

    with m.State(CPUState.FETCH_EXT):
        m.d.comb += [
            self.mem_addr.eq(pc + 2),
            self.mem_valid.eq(1),
        ]
        with m.If(self.mem_ready):
            m.d.sync += instr_ext.eq(self.mem_rdata)
            m.next = CPUState.EXECUTE
```

The FETCH state drives `mem_addr` with the current PC and asserts `mem_valid`. When `mem_ready` comes back, the fetched instruction word is latched into `instr` and the FSM advances to DECODE. The decoder examines the instruction combinationally; if `decoder.is_extended` is set (the opcode is `0xF`), the machine detours through FETCH_EXT to fetch the second instruction word before proceeding to EXECUTE.

The EXECUTE state handles all instruction types through a `m.Switch` on the decoder's `inst_type` output:

Listing 34.6: EXECUTE state: instruction dispatch (`rtl/cpu.py`)

```
with m.State(CPUState.EXECUTE):
    # Default: advance PC
    with m.If(decoder.is_extended):
        m.d.sync += pc.eq(pc_plus_4)
    with m.Else():
        m.d.sync += pc.eq(pc_plus_2)

    with m.Switch(decoder.inst_type):
        with m.Case(InstType.ALU_REG):
            m.d.sync += alu_result.eq(alu.result)
            m.d.sync += [
                flags[7].eq(alu.flag_n),
                flags[6].eq(alu.flag_z),
                flags[5].eq(alu.flag_c),
                flags[4].eq(alu.flag_v),
            ]
            m.next = CPUState.WRITEBACK

        with m.Case(InstType.BRANCH):
```

```
            with m.If(branch_taken):
                m.d.sync += pc.eq(pc_branch)
            m.next = CPUState.FETCH

        with m.Case(InstType.JUMP):
            m.d.sync += pc.eq(pc_jump)
            m.next = CPUState.FETCH

        with m.Case(InstType.SYSTEM):
            with m.If(decoder.is_halt):
                m.next = CPUState.HALTED
            with m.Elif(decoder.is_ei):
                m.d.sync += int_enabled.eq(1)
                m.next = CPUState.FETCH
            with m.Elif(decoder.is_di):
                m.d.sync += int_enabled.eq(0)
                m.next = CPUState.FETCH
            with m.Else():
                m.next = CPUState.FETCH
```

Each instruction type sets the appropriate next state. ALU operations route through WRITEBACK so the result is written to the register file. Branches and jumps update the PC and return directly to FETCH. The **HALT** instruction transitions to HALTED, where the FSM stays until externally reset.

The MEMORY and WRITEBACK states complete the pipeline:

Listing 34.7: MEMORY and WRITEBACK states (`rtl/cpu.py`)

```
with m.State(CPUState.MEMORY):
    m.d.comb += [
        self.mem_addr.eq(mem_addr_reg),
        self.mem_valid.eq(1),
    ]

    with m.If(decoder.mem_store):
        m.d.comb += [
            self.mem_wdata.eq(mem_data_reg),
            self.mem_we.eq(1),
        ]

    with m.If(self.mem_ready):
        with m.If(decoder.mem_load):
            m.d.sync += alu_result.eq(self.mem_rdata)
            m.next = CPUState.WRITEBACK
        with m.Else():
```

```
            m.next = CPUState.FETCH

with m.State(CPUState.WRITEBACK):
    with m.If(decoder.reg_write):
        m.d.comb += [
            regfile.wr_addr.eq(decoder.rd),
            regfile.wr_data.eq(alu_result),
            regfile.wr_en.eq(1),
        ]
    m.next = CPUState.FETCH
```

The MEMORY→FETCH Hazard

In early versions of the SAMPO Verilog design, stores transitioned directly from MEMORY to FETCH, bypassing WRITEBACK. This caused a subtle bug: the `mem_ready` signal remained asserted from the store when the next FETCH began, causing the CPU to latch stale data as the new instruction. The fix was to *always* route through WRITEBACK after a memory operation, giving `mem_ready` time to deassert. The Amaranth implementation includes this fix. As discussed in Chapter 22, this was one of the harder bugs to track down—it only manifested with certain instruction sequences.

34.2 ALU, Register File, and Decoder

Each of the CPU's major submodules—the ALU, the shifter, the register file, and the instruction decoder—has its own Amaranth module in a separate file. Let's examine each one.

34.2.1 The ALU

The ALU in `rtl/alu.py` implements the same 16 operations as the Verilog version. The operation dispatch uses `m.Switch`:

Listing 34.8: ALU operation dispatch (`rtl/alu.py`)

```
def elaborate(self, platform):
    m = Module()

    result_wide = Signal(17)

    mul_result = Signal(32)
    m.d.comb += mul_result.eq(
        self.a.as_signed() * self.b.as_signed()
    )
```

```
    with m.Switch(self.op):
        with m.Case(ALUOp.ADD):
            m.d.comb += result_wide.eq(self.a + self.b)

        with m.Case(ALUOp.SUB):
            m.d.comb += result_wide.eq(self.a - self.b)

        with m.Case(ALUOp.AND):
            m.d.comb += result_wide.eq(self.a & self.b)

        with m.Case(ALUOp.MUL):
            m.d.comb += result_wide.eq(mul_result[:16])

        with m.Case(ALUOp.MULH):
            m.d.comb += result_wide.eq(mul_result[16:32])

        with m.Case(ALUOp.DIV):
            with m.If(self.b != 0):
                m.d.comb += result_wide.eq(
                    (self.a.as_signed() // self.b.as_signed()
                    ).as_unsigned()
                )
            with m.Else():
                m.d.comb += result_wide.eq(0xFFFF)

    m.d.comb += self.result.eq(result_wide[:16])
    return m
```

The 17-bit `result_wide` signal captures the carry bit, just as the Verilog version uses `wire [16:0] add_result`. The key difference: Amaranth's `.as_signed()` makes signed arithmetic explicit, while Verilog requires `$signed()` casts that are easy to forget.

The overflow detection is particularly clean in Amaranth:

Listing 34.9: ALU overflow detection (`rtl/alu.py`)

```
# Overflow detection
a_sign = self.a[15]
b_sign = self.b[15]
r_sign = self.result[15]

with m.Switch(self.op):
    with m.Case(ALUOp.ADD):
        m.d.comb += self.flag_v.eq(
            (a_sign == b_sign) & (a_sign != r_sign)
```

```
            )
        with m.Case(ALUOp.SUB):
            m.d.comb += self.flag_v.eq(
                (a_sign != b_sign) & (r_sign != a_sign)
            )
        with m.Default():
            m.d.comb += self.flag_v.eq(0)
```

Compare the same logic in Verilog from `verilog/rtl/alu.v`:

Listing 34.10: Verilog overflow detection (`verilog/rtl/alu.v`)

```
wire add_overflow = (a[15] == b[15]) &&
                    (add_result[15] != a[15]);
wire sub_overflow = (a[15] != b[15]) &&
                    (sub_result[15] == b[15]);
```

Both implementations produce identical hardware. The Verilog version is more concise, but the Amaranth version is more explicit about when overflow is computed (only for **ADD** and **SUB**).

34.2.2 The Shifter

The `Shifter` class in `rtl/alu.py` handles all 16 shift and rotate operations. The Amaranth version uses `Cat` (concatenation) to express rotations:

Listing 34.11: Shifter rotation operations (`rtl/alu.py`)

```
class Shifter(Elaboratable):
    def __init__(self):
        self.value = Signal(16)
        self.func = Signal(4)
        self.carry_in = Signal()
        self.result = Signal(16)
        self.carry_out = Signal()

    def elaborate(self, platform):
        m = Module()
        v = self.value

        with m.Switch(self.func):
            with m.Case(0x0):  # SLL1
                m.d.comb += [
                    self.result.eq(v << 1),
                    self.carry_out.eq(v[15]),
```

```
                ]
            with m.Case(0x2):  # SRA1
                m.d.comb += [
                    self.result.eq(Cat(v[1:16], v[15])),
                    self.carry_out.eq(v[0]),
                ]
            with m.Case(0x3):  # ROL1
                m.d.comb += [
                    self.result.eq(Cat(v[15], v[0:15])),
                    self.carry_out.eq(v[15]),
                ]
            with m.Case(0x7):  # SWAP
                m.d.comb += [
                    self.result.eq(Cat(v[8:16], v[0:8])),
                    self.carry_out.eq(0),
                ]
        return m
```

The `Cat(v[15], v[0:15])` expression for **ROL** concatenates bit 15 at the bottom with bits 14:0 at the top, performing a single-bit left rotation. This is more readable than the Verilog equivalent `{v[14:0], v[15]}`, where the bit ordering can be confusing.

34.2.3 The Register File

The register file in `rtl/regfile.py` implements 16 general-purpose 16-bit registers with two read ports and one write port. `R0` is hardwired to zero, and registers `R4`–`R11` have alternate copies swappable via the **EXX** instruction:

Listing 34.12: Register file implementation (`rtl/regfile.py`)

```
class RegisterFile(Elaboratable):
    def __init__(self):
        self.rd_addr1 = Signal(4)
        self.rd_addr2 = Signal(4)
        self.rd_data1 = Signal(16)
        self.rd_data2 = Signal(16)
        self.wr_addr = Signal(4)
        self.wr_data = Signal(16)
        self.wr_en = Signal()
        self.exx = Signal()
        self.regs_out = Signal(16 * 16)

    def elaborate(self, platform):
        m = Module()
```

```
        # Main register bank (R0-R15)
        regs = Array([Signal(16, name=f"r{i}")
                      for i in range(16)])

        # Alternate register bank (R4'-R11')
        regs_alt = Array([Signal(16, name=f"r{i}_alt")
                          for i in range(4, 12)])

        # Read port 1 (R0 always reads as 0)
        with m.If(self.rd_addr1 == 0):
            m.d.comb += self.rd_data1.eq(0)
        with m.Else():
            m.d.comb += self.rd_data1.eq(regs[self.rd_addr1])

        # Write port (writes to R0 are ignored)
        with m.If(self.wr_en & (self.wr_addr != 0)):
            m.d.sync += regs[self.wr_addr].eq(self.wr_data)

        # EXX: Swap R4-R11 with alternate bank
        with m.If(self.exx):
            for i in range(8):
                reg_idx = i + 4
                m.d.sync += [
                    regs[reg_idx].eq(regs_alt[i]),
                    regs_alt[i].eq(regs[reg_idx]),
                ]

        return m
```

The `Array` construct is critical here. It creates a multiplexer-based addressable collection that can be indexed by a hardware signal. The expression `regs[self.rd_addr1]` generates a 16-to-1 multiplexer in hardware, selecting one of the 16 registers based on the 4-bit read address.

The **EXX** swap uses a Python `for` loop—but remember, this loop executes at elaboration time, not at hardware time. It generates 8 pairs of parallel swap assignments, all of which execute simultaneously in a single clock cycle.

Compare the Verilog equivalent from `verilog/rtl/regfile.v`:

Listing 34.13: Verilog register file: EXX swap (`verilog/rtl/regfile.v`)

```
// Main register file (R0-R15)
reg [15:0] regs [0:15];
```

```
// Alternate registers (R4'-R11')
reg [15:0] alt_regs [0:7];

// Read ports - R0 always returns 0
assign rd_data1 = (rd_addr1 == 4'h0) ?
                  16'h0000 : regs[rd_addr1];

// EXX swap
if (exx) begin
    for (i = 0; i < 8; i = i + 1) begin
        regs[i + 4] <= alt_regs[i];
        alt_regs[i] <= regs[i + 4];
    end
end
```

The two implementations are structurally very similar. The Verilog version uses `reg [15:0] regs [0:15]` for the register array and a ternary operator for the `R0` zero-hardwiring. The Amaranth version uses `Array` and `m.If`.

RegisterFileAsync: Write Forwarding

The `rtl/regfile.py` file also contains a `RegisterFileAsync` variant with write-forwarding bypass logic. When a register is being read and written in the same cycle, the bypass path forwards the write data to the read port immediately: `rd1_bypass = (self.wr_en \& (self.wr_addr == self.rd_addr1) \& (self.rd_addr1 != 0))`. This variant also includes a dedicated stack pointer write port for **PUSH**/**POP** optimisation.

34.2.4 The Instruction Decoder

The decoder in `rtl/decode.py` is the largest single module. It takes a 16-bit instruction and produces all the control signals the CPU needs: ALU operation, memory access type, branch condition, register write enable, and dozens of instruction-type flags.

The instruction type classification uses its own enumeration:

Listing 34.14: Instruction type enum (`rtl/decode.py`)

```
class InstType(Enum, shape=4):
    """Instruction type classification."""
    ALU_REG     = 0   # Register-register ALU
    ALU_IMM     = 1   # Immediate ALU (ADDI)
    LOAD        = 2   # Load from memory
    STORE       = 3   # Store to memory
    BRANCH      = 4   # Conditional branch
```

```
    JUMP        = 5   # Unconditional jump
    JUMP_REG    = 6   # Register jump (JR, JALR)
    SHIFT       = 7   # Shift/rotate
    MULDIV      = 8   # Multiply/divide
    MISC        = 9   # PUSH, POP, CMP, etc.
    IO          = 10  # I/O operations
    SYSTEM      = 11  # NOP, HALT, etc.
    EXTENDED    = 12  # 32-bit extended
    INVALID     = 15
```

This enum provides a level of abstraction that does not exist in the Verilog implementation. In Verilog, the CPU's execute logic switches directly on the decoded opcode fields. In Amaranth, the decoder classifies instructions into types, and the CPU switches on those types. This makes the CPU state machine easier to read and modify.

The core decode logic extracts fields and maps opcodes to control signals:

Listing 34.15: Decoder field extraction (`rtl/decode.py`)

```
def elaborate(self, platform):
    m = Module()

    # Extract instruction fields
    opcode = self.instr[12:16]
    rd = self.instr[8:12]
    rs1 = self.instr[4:8]
    rs2 = self.instr[0:4]
    func = self.instr[0:4]
    imm8_raw = self.instr[0:8]

    m.d.comb += [
        self.rd.eq(rd),
        self.rs1.eq(rs1),
        self.rs2.eq(rs2),
        self.func.eq(func),
        self.shift_func.eq(func),
        self.branch_cond.eq(rd),
    ]

    # Sign-extend 8-bit immediate
    m.d.comb += self.imm8.eq(imm8_raw.as_signed())
```

The Amaranth decoder uses Python slicing (`self.instr[12:16]`) where the Verilog decoder uses bit-select (`instr[15:12]`).

Bit Indexing: Amaranth vs. Verilog

Amaranth uses Python-style slicing: `signal[12:16]` means bits 12, 13, 14, and 15 (exclusive upper bound). Verilog uses inclusive ranges: `signal[15:12]` means bits 15, 14, 13, and 12 (MSB first). Both select the same four bits, but the notation is reversed. This is a frequent source of confusion when porting code between the two.

The main decode switch covers all 16 primary opcodes. Here are the ALU and branch cases:

Listing 34.16: Decoder: ALU and branch opcodes (`rtl/decode.py`)

```
with m.Switch(opcode):
    # ADD Rd, Rs1, Rs2
    with m.Case(0x0):
        m.d.comb += [
            self.inst_type.eq(InstType.ALU_REG),
            self.alu_op.eq(ALUOp.ADD),
            self.reg_write.eq(1),
        ]

    # ADDI Rd, imm8
    with m.Case(0x5):
        m.d.comb += [
            self.inst_type.eq(InstType.ALU_IMM),
            self.alu_op.eq(ALUOp.ADD),
            self.reg_write.eq(1),
        ]

    # BRANCH
    with m.Case(0x8):
        m.d.comb += [
            self.inst_type.eq(InstType.BRANCH),
            self.is_branch.eq(1),
        ]

    # EXTENDED (32-bit)
    with m.Case(0xF):
        m.d.comb += [
            self.inst_type.eq(InstType.EXTENDED),
            self.is_extended.eq(1),
        ]
        with m.Switch(func):
            with m.Case(0x0):  # ADDIX
                m.d.comb += [
                    self.alu_op.eq(ALUOp.ADD),
```

```
                    self.reg_write.eq(1),
                ]
            with m.Case(0x5):  # LWX
                m.d.comb += [
                    self.mem_load.eq(1),
                    self.reg_write.eq(1),
                ]
```

The extended instruction handling (opcode `0xF`) is a nested switch on the function field, matching the nested case structure in the Verilog decoder. Each sub-case sets the appropriate control signals—ALU operation, memory load/store flags, register write enable.

34.3 Side-by-Side Comparison with Verilog

Having seen both implementations in detail, let's put them side by side on several key dimensions. The following comparisons use real code from both versions of the SAMPO.

34.3.1 Module Declaration

Table 34.1: Module declaration comparison

Amaranth (`rtl/cpu.py`)

```
class SampoCPU(Elaboratable):
  def __init__(self,
    reset_vector=0x0100):
    self.mem_addr = Signal(16)
    self.mem_rdata = Signal(16)
    # ...
```

Verilog (`verilog/rtl/cpu.v`)

```
module cpu #(
  parameter RESET_VECTOR
    = 16'h0100
) (
  input  wire clk,
  input  wire rst_n,
  output reg [15:0] mem_addr,
  input  wire [15:0] mem_rdata,
  // ...
);
```

In Amaranth, there is no explicit `clk` or `rst_n` port. The clock and reset are managed by the platform and clock domain infrastructure. In Verilog, they must be declared and connected at every level of the hierarchy.

34.3.2 ALU Operation: ADD

The Verilog version requires explicit zero-extension (`{1'b0, a}`) to get the 17th carry bit. Amaranth's addition automatically produces a result wide enough to hold the carry, though we explicitly declare a 17-bit signal to capture it.

Table 34.2: ALU addition comparison

Amaranth (`rtl/alu.py`)

```
result_wide = Signal(17)

with m.Case(ALUOp.ADD):
  m.d.comb += result_wide.eq(
    self.a + self.b
  )

m.d.comb += self.result.eq(
  result_wide[:16]
)
m.d.comb += self.flag_c.eq(
  result_wide[16]
)
```

Verilog (`verilog/rtl/alu.v`)

```
wire [16:0] add_result;
assign add_result =
  {1'b0, a} + {1'b0, b};

`ALU_ADD: begin
  result = add_result[15:0];
  flag_c = add_result[16];
end
```

34.3.3 State Machine Structure

The Verilog FSM requires three separate `always` blocks: one for state register update, one for next-state combinational logic, and one for registered outputs. Amaranth combines all three concerns into a single `m.FSM` block, using `m.d.comb` for combinational outputs, `m.d.sync` for registered outputs, and `m.next` for next-state.

34.3.4 SoC Integration

The system-on-chip integration shows the biggest difference in verbosity. The Amaranth SoC in `rtl/soc.py`:

Listing 34.17: Amaranth SoC integration (`rtl/soc.py`)

```
class SampoSoC(Elaboratable):
    def __init__(self, program=None, reset_vector=0x0100):
        self.program = program or []
        self.reset_vector = reset_vector
        # ... signal declarations ...

    def elaborate(self, platform):
        m = Module()

        m.submodules.cpu = cpu = SampoCPU(
            reset_vector=self.reset_vector)
        m.submodules.ram = ram = RAM(
            size=65536, init=self.program)
        m.submodules.uart = uart = UART()
```

Table 34.3: FSM structure comparison

Amaranth

```
with m.FSM(reset=CPUState.RESET):
  with m.State(CPUState.FETCH):
    m.d.comb += [
      self.mem_addr.eq(pc),
      self.mem_valid.eq(1),
    ]
    with m.If(self.mem_ready):
      m.d.sync += instr.eq(
        self.mem_rdata)
      m.next = CPUState.DECODE
```

Verilog

```
reg [3:0] state, next_state;

always @(posedge clk)
  state <= next_state;

always @(*) begin
  next_state = state;
  case (state)
    `ST_FETCH:
      if (mem_ready)
        next_state = `ST_DECODE;
  endcase
end

always @(posedge clk) begin
  case (state)
    `ST_FETCH:
      if (mem_ready)
        instr <= mem_rdata;
  endcase
end
```

```
        # Connect CPU to RAM
        m.d.comb += [
            ram.addr.eq(cpu.mem_addr),
            ram.wdata.eq(cpu.mem_wdata),
            ram.we.eq(cpu.mem_we),
            ram.be.eq(cpu.mem_be),
            ram.valid.eq(cpu.mem_valid),
            cpu.mem_rdata.eq(ram.rdata),
            cpu.mem_ready.eq(ram.ready),
        ]

        # Connect CPU to UART
        m.d.comb += [
            uart.addr.eq(cpu.io_addr),
            uart.wdata.eq(cpu.io_wdata),
            uart.rd.eq(cpu.io_rd),
            uart.wr.eq(cpu.io_wr),
            cpu.io_rdata.eq(uart.rdata),
        ]
```

```
        return m
```

Compare this with the Verilog version from `verilog/rtl/soc.v`:

Listing 34.18: Verilog SoC integration (`verilog/rtl/soc.v`)

```
wire [15:0] cpu_mem_addr;
wire [15:0] cpu_mem_rdata;
wire [15:0] cpu_mem_wdata;
wire        cpu_mem_we;
wire [1:0]  cpu_mem_be;
wire        cpu_mem_valid;
wire        cpu_mem_ready;

cpu #(
    .RESET_VECTOR(RESET_VECTOR)
) cpu_inst (
    .clk(clk),
    .rst_n(rst_n),
    .mem_addr(cpu_mem_addr),
    .mem_rdata(cpu_mem_rdata),
    .mem_wdata(cpu_mem_wdata),
    .mem_we(cpu_mem_we),
    .mem_be(cpu_mem_be),
    .mem_valid(cpu_mem_valid),
    .mem_ready(cpu_mem_ready),
    .io_addr(cpu_io_addr),
    // ... 10 more ports ...
);
```

In Verilog, every inter-module connection requires a declared wire, and every port must be explicitly named in the instantiation. The Amaranth version accesses module ports directly as attributes, eliminating the intermediate wire declarations entirely.

Loading Programs in Amaranth

The Amaranth SoC accepts a program as a Python list of bytes in its constructor: `SampoSoC(program=load_binary("hello.bin"))`. The Verilog SoC uses `$readmemh` with a hex file parameter. The Python approach is more flexible—you can assemble programs on the fly, generate test vectors programmatically, or even embed the assembler in the test script.

34.4 When to Use Amaranth vs. Verilog

Having built the same CPU in both languages, we can offer concrete guidance.

34.4.1 Choose Amaranth When...

You need parameterized designs. Amaranth's Python constructors make it natural to generate different hardware configurations. Need a CPU with 8 registers instead of 16? Change a constructor argument. Need an ALU with only ADD and SUB? Remove cases from the switch. In Verilog, this requires `generate` blocks and `parameter` propagation.

You want integrated testing. The Amaranth simulator runs in the same process as the testbench. You can set breakpoints in the test, inspect signal values with the debugger, and run tests under pytest. The test harness in `rtl/test_cpu.py` runs three test suites (ALU, register file, and hello world) in a single Python invocation.

You are building reusable IP. Python packages, classes, and modules provide a natural hierarchy for sharing and versioning IP blocks. Amaranth modules can be published to PyPI and imported like any other Python library.

Your team knows Python better than Verilog. Amaranth lowers the barrier to hardware design for software engineers. The Python syntax is familiar, and common mistakes (sensitivity lists, blocking assignments) are eliminated by construction.

34.4.2 Choose Verilog When...

You need vendor tool compatibility. Some FPGA vendor tools (Vivado, Quartus) expect Verilog or VHDL input. While Amaranth can generate Verilog for these tools, debugging synthesis issues in generated Verilog is painful.

You are working with existing Verilog IP. Integrating third-party Verilog IP cores into an Amaranth design requires wrapping them in `Instance` objects. For large designs with many vendor IP blocks, staying in Verilog avoids the wrapping overhead.

You want to understand the hardware directly. There is pedagogical value in writing Verilog by hand. The explicit nature of sensitivity lists, blocking vs. non-blocking assignment, and port declarations teaches hardware concepts that Amaranth abstracts away. This is why Part VI of this book uses Verilog—it builds hardware intuition.

You need maximum performance control. For timing-critical designs, Verilog gives you fine-grained control over synthesis attributes, clock-domain crossings, and placement constraints. Amaranth's platform abstraction handles common cases, but unusual requirements may need raw Verilog.

34.4.3 The Hybrid Approach

The SAMPO project uses both languages, and that is a valid long-term strategy. Prototype and test in Amaranth, then hand-optimise critical paths in Verilog if needed. Or write your core logic in Verilog and use Amaranth for the testbench and system integration. Amaranth can instantiate raw Verilog modules using `Instance()`, and Verilog can consume the generated output from Amaranth.

34.5 Testing with Python Testbenches

The testing infrastructure in `rtl/test_cpu.py` demonstrates Amaranth's testing philosophy: each test is a standalone Python function that creates a fresh design, drives it with stimuli, and checks the results.

34.5.1 Test Runner Structure

The test runner uses a familiar pattern:

Listing 34.19: Test runner (`rtl/test_cpu.py`)

```
def main():
    print("=" * 60)
    print("Sampo CPU Tests")
    print("=" * 60)

    tests = [
        ("ALU", test_alu),
        ("Register File", test_regfile),
        ("Hello World", test_hello_world),
    ]

    results = []
    for name, test_fn in tests:
        print(f"\nRunning: {name}")
        try:
            passed = test_fn()
            results.append((name, passed))
        except Exception as e:
            print(f"ERROR: {e}")
            results.append((name, False))

    all_passed = all(p for _, p in results)
    return 0 if all_passed else 1
```

This is a standard Python test harness. It could be integrated with pytest for automatic discovery, or extended with command-line arguments for selective test execution.

34.5.2 Unit Testing the ALU

The ALU test verifies each operation in isolation. Because the ALU is purely combinational, the testbench sets inputs, yields once (to let combinational logic settle), and reads outputs:

Listing 34.20: ALU unit test (`rtl/test_cpu.py`)

```
def test_alu():
    alu = ALU()

    def testbench():
        # Test AND
        yield alu.a.eq(0xFF00)
        yield alu.b.eq(0x0FF0)
        yield alu.op.eq(ALUOp.AND)
        yield
        result = yield alu.result
        assert result == 0x0F00, \
            f"AND failed: 0x{result:04X} != 0x0F00"
        print(f"AND: 0xFF00 & 0x0FF0 = 0x{result:04X}")

        # Test OR
        yield alu.a.eq(0xFF00)
        yield alu.b.eq(0x00FF)
        yield alu.op.eq(ALUOp.OR)
        yield
        result = yield alu.result
        assert result == 0xFFFF, \
            f"OR failed: 0x{result:04X} != 0xFFFF"
        print(f"OR: 0xFF00 | 0x00FF = 0x{result:04X}")

        print("\nALU tests passed!")

    sim = Simulator(alu)
    sim.add_testbench(testbench)
    sim.run()
```

Each assertion provides a detailed error message with hex formatting, making failures immediately diagnosable.

34.5.3 Integration Testing the SoC

The "Hello World" test is a full integration test that exercises the entire system: CPU state machine, instruction decoder, ALU, register file, RAM, and UART. The test loads a pre-assembled binary, runs the SoC for up to 10,000 cycles, collects UART output, and compares it against the expected string.

Listing 34.21: SoC integration test (`rtl/test_cpu.py`)

```
def test_hello_world():
    bin_path = Path(__file__).parent.parent \
               / "examples" / "hello.bin"
    program = load_binary(bin_path)
    print(f"Loaded {len(program)} bytes from {bin_path}")

    soc = SampoSoC(
        program=program, reset_vector=0x0100)
    output = []

    def testbench():
        yield soc.tx_ready.eq(1)

        for cycle in range(10000):
            if (yield soc.tx_valid):
                char = yield soc.tx_data
                output.append(chr(char))
                print(chr(char), end='', flush=True)

            if (yield soc.halted):
                print(f"\n\nCPU halted after "
                      f"{cycle} cycles")
                print(f"PC: 0x{(yield soc.pc):04X}")
                break

            yield

    sim = Simulator(soc)
    sim.add_clock(1e-6)
    sim.add_testbench(testbench)

    with sim.write_vcd("sampo_test.vcd"):
        sim.run()

    result = ''.join(output)
    expected = "Hello, Sampo!\n"
```

```
    if result == expected:
        print("TEST PASSED!")
        return True
    else:
        print(f"TEST FAILED! "
              f"Expected: {repr(expected)}")
        return False
```

There is a lot happening in this test. Let's trace the key moments:

1. The binary is loaded as raw bytes and passed to the SoC constructor, which initialises RAM with the program image.
2. `soc.tx_ready` is held high, telling the UART that we are always ready to accept transmitted data.
3. On each cycle, the testbench checks `soc.tx_valid`. When the CPU writes a character to `0x81`, the UART asserts `tx_valid` and presents the byte on `tx_data`.
4. The testbench also monitors `soc.halted`. When the **`HALT`** instruction executes, the FSM enters HALTED and this signal goes high.
5. After simulation completes, the collected characters are joined and compared against the expected output.

Binary Preparation

The integration test assumes the binary has been pre-assembled. Build it from the project root:

```
cd sasm && cargo run --release -- ../examples/hello.s -o ../examples/hello.bin
python3 -m rtl.test_cpu
```

34.5.4 Running Tests

The test suite is invoked as a Python module:

Listing 34.22: Running the Amaranth test suite

```
python3 -m rtl.test_cpu
```

Expected output for a passing run:

Listing 34.23: Test output

```
============================================================
Sampo CPU Tests
============================================================

============================================================
Running: ALU
============================================================
ADD: 100 + 50 = 150
SUB: 100 - 30 = 70
AND: 0xFF00 & 0x0FF0 = 0x0F00
OR: 0xFF00 | 0x00FF = 0xFFFF
SUB: 50 - 50 = 0, Z=1
SUB: 0 - 1 = 0xFFFF, N=1

ALU tests passed!

============================================================
Running: Register File
============================================================
R4 = 0x1234
R0 = 0x0000 (writes ignored)

Register file tests passed!

============================================================
Running: Hello World
============================================================
Hello, Sampo!

CPU halted after 847 cycles
PC: 0x0118

Output: 'Hello, Sampo!\n'
TEST PASSED!

============================================================
Results:
============================================================
  ALU: PASS
  Register File: PASS
  Hello World: PASS
```

The VCD file produced during the hello world test can be opened in GTKWave for detailed waveform analysis, just as we would with a Verilog simulation.

34.6 Generating Verilog for FPGA Synthesis

Once the design passes all tests, we can generate Verilog for FPGA synthesis. Amaranth provides two paths: manual Verilog generation or platform-targeted builds.

34.6.1 Manual Verilog Generation

To generate a standalone Verilog file suitable for integration into an existing synthesis flow:

Listing 34.24: Generating synthesisable Verilog

```
from amaranth.back.verilog import convert
from rtl.soc import SampoSoC

# Create SoC with a test program
soc = SampoSoC(
    program=load_binary("examples/hello.bin"),
    reset_vector=0x0100
)

# Generate Verilog with explicit ports
verilog = convert(soc, name="sampo_soc", ports=[
    soc.tx_data,
    soc.tx_valid,
    soc.tx_ready,
    soc.rx_data,
    soc.rx_valid,
    soc.rx_ready,
    soc.halted,
    soc.pc,
    soc.cycles,
])

with open("build/sampo_soc.v", "w") as f:
    f.write(verilog)

print(f"Generated {len(verilog)} bytes of Verilog")
```

The generated file can be fed to any Verilog-compatible synthesis tool. For the open-source Yosys/nextpnr flow we use throughout this book:

Listing 34.25: Synthesising generated Verilog

```
yosys -p "read_verilog build/sampo_soc.v; synth_ecp5 -top sampo_soc -json build/sampo
    .json"
```

```
nextpnr-ecp5 --85k --json build/sampo.json --lpf ulx3s.lpf --textcfg build/sampo.
    config
ecppack build/sampo.config build/sampo.bit
```

34.6.2 Platform-Targeted Builds

Amaranth's platform abstraction can drive the entire build-program cycle. For the ULX3S board we use in **??**:

Listing 34.26: Platform-targeted build for ULX3S

```
from amaranth_boards.ulx3s import ULX3S85FPlatform

class SampoTop(Elaboratable):
    def elaborate(self, platform):
        m = Module()

        # Get UART pins from platform
        uart = platform.request("uart")
        leds = platform.request("led", 0)

        # Instantiate SoC
        m.submodules.soc = soc = SampoSoC(
            program=load_binary("hello.bin"),
            reset_vector=0x0100
        )

        # Connect to board pins
        m.d.comb += [
            uart.tx.o.eq(soc.tx_data[0]),
            leds.o.eq(soc.halted),
        ]

        return m

if __name__ == "__main__":
    platform = ULX3S85FPlatform()
    platform.build(SampoTop(), do_program=True)
```

The `platform.build()` call runs the complete synthesis flow: Verilog generation, Yosys synthesis, nextpnr place-and-route, ecppack bitstream generation, and optionally programming the FPGA with openFPGALoader.

Pin Assignment

The `platform.request("uart")` call returns a record with `tx` and `rx` sub-signals that map to the correct physical FPGA pins for the ULX3S board. No constraint file editing required—the `amaranth-boards` package knows the pin assignments for supported boards.

34.6.3 The RAM Initialisation Pattern

One area where Amaranth provides a significant advantage is RAM initialisation. The Verilog SoC uses `$readmemh` with a hex file, requiring a separate conversion step. The Amaranth SoC embeds the program directly:

Listing 34.27: RAM module with byte-to-word conversion (`rtl/soc.py`)

```
class RAM(Elaboratable):
    def __init__(self, size=65536, init=None):
        self.size = size
        self.init = init or []
        # ... signals ...

    def elaborate(self, platform):
        m = Module()

        mem = Memory(width=16,
                     depth=self.size // 2,
                     init=self._init_words())
        m.submodules.rd = rd = mem.read_port()
        m.submodules.wr = wr = mem.write_port(
            granularity=8)

        word_addr = self.addr[1:16]

        m.d.comb += [
            rd.addr.eq(word_addr),
            self.rdata.eq(rd.data),
            wr.addr.eq(word_addr),
            wr.data.eq(self.wdata),
            wr.en[0].eq(self.we & self.be[0]),
            wr.en[1].eq(self.we & self.be[1]),
        ]

        m.d.sync += self.ready.eq(self.valid)
        return m

    def _init_words(self):
```

```
        """Convert byte init data to 16-bit words."""
        words = []
        for i in range(0, len(self.init), 2):
            lo = self.init[i] if i < len(self.init) else 0
            hi = self.init[i+1] if i+1 < len(self.init) else 0
            words.append(lo | (hi << 8))
        while len(words) < self.size // 2:
            words.append(0)
        return words
```

The `_init_words` method converts the byte-oriented program image into 16-bit little-endian words for Amaranth's `Memory` primitive. The `granularity=8` parameter on the write port enables byte enables—each of the two bytes in a 16-bit word can be written independently, which is essential for the **SB** (store byte) instruction.

The `m.d.sync += self.ready.eq(self.valid)` line implements the single-cycle memory ready delay, matching the Verilog SoC's `assign cpu_mem_ready = mem_valid_d` pattern with a one-cycle delayed version of the valid signal.

34.6.4 Synthesis Results

The Amaranth-generated Verilog synthesises to comparable resource usage as the hand-written Verilog. Both versions target the same ECP5-85F FPGA and use the same synthesis tool chain.

Table 34.4: Synthesis resource comparison (ECP5-85F)

Resource	Hand-written Verilog	Amaranth-generated
LUTs	∼2,800	∼3,000
FFs	∼650	∼680
BRAM	16 (64KB)	16 (64KB)
f_{max}	∼25 MHz	∼24 MHz

The Amaranth version uses slightly more LUTs (roughly 7% more) due to differences in how the generated Verilog expresses multiplexers and state encoding. The maximum frequency is within the same range. For a design this size, the difference is well within the noise of the synthesis tool's optimisation passes.

Exercises

1. **Add a Test.** Write a new test function in `rtl/test_cpu.py` that tests the shifter module. Verify that `SLL1` of `0x0001` produces `0x0002` with carry out = 0, and that `SLL1` of `0x8000` produces `0x0000` with carry out = 1. Test at least three different shift operations.

2. **Register File EXX Test.** Extend the register file test to verify **`EXX`** behaviour. Write values to `R4` through `R7`, pulse `exx`, verify the registers now read as zero (the initial alternate bank values), pulse `exx` again, and verify the original values are restored.

3. **Decoder Verification.** Write a test that feeds raw instruction words to the decoder and verifies the decoded control signals. Test at least: **`ADD`** `R4`, `R5`, `R6` (opcode `0x0456`), **`ADDI`** `R4`, `42` (opcode `0x542A`), and **`HALT`** (opcode `0xE100`). Verify that `inst_type`, `rd`, `rs1`, `rs2`, `alu_op`, and `is_halt` are correct for each.

4. **Verilog Generation.** Generate Verilog from the Amaranth ALU module and count the lines of generated code. Compare with the hand-written `verilog/rtl/alu.v`. Synthesise both with Yosys (`yosys -p "synth_ecp5"`) and compare the reported LUT and FF counts.

5. **Custom Peripheral.** Add an 8-bit GPIO peripheral to the Amaranth SoC. Map it to I/O port `0x90` for writing (output register) and `0x90` for reading (input pins). Write a test program in SAMPO assembly that writes `0xAA` to the GPIO port, reads it back, and halts. Create a Python testbench that verifies the GPIO output appears on the peripheral's output signal.

What's Next

We have now built the SAMPO CPU twice—once in Verilog, once in Amaranth—and seen the trade-offs between the two approaches. Both implementations produce identical behaviour, verified by the same test programs. With the hardware design complete and verified, we turn in **??** to the question every CPU designer eventually asks: *how do we make it faster?* We will analyse the SAMPO's performance characteristics, identify bottlenecks in the multi-cycle design, and explore what a pipelined version might look like.

Part XI

Beyond the Basics

Chapter 35

Performance and Pipelining

The SAMPO CPU executes every instruction by marching through a multi-cycle finite state machine: FETCH, DECODE, EXECUTE, MEMORY, WRITEBACK. It is correct—132 ISA tests and real FPGA hardware prove that—but it is not fast. A register-to-register **ADD** takes four clock cycles. A **LW** takes six. Running at 12.5 MHz on the ULX3S, that gives us roughly two to three million useful instructions per second.

Can we do better? Much better. This chapter analyzes where the current design spends its time, introduces the classic five-stage pipeline, walks through the hazards that make pipelining hard, and sketches a pipelined redesign of the SAMPO core—all while keeping one eye on the LUT budgets and timing realities of the Lattice ECP5 FPGA.

35.1 Current Performance Analysis

35.1.1 Cycle Counts per Instruction

The SAMPO state machine is defined in `verilog/rtl/cpu.v`. Eight states are encoded in a 4-bit register:

Listing 35.1: State encoding from `rtl/sampo_pkg.vh`

```
// CPU states
`define ST_RESET     4'h0
`define ST_FETCH     4'h1
`define ST_FETCH_EXT 4'h2
`define ST_DECODE    4'h3
`define ST_EXECUTE   4'h4
`define ST_MEMORY    4'h5
```

```
`define ST_WRITEBACK 4'h6
`define ST_HALTED    4'h7
```

Not every instruction visits every state. The next-state logic in `cpu.v` selects the path through the FSM based on the decoded instruction type:

Listing 35.2: Next-state logic for EXECUTE (from `rtl/cpu.v`)

```
`ST_EXECUTE: begin
    if (dec_is_halt) begin
        next_state = `ST_HALTED;
    end else if (dec_mem_load || dec_mem_store) begin
        next_state = `ST_MEMORY;
    end else if (dec_reg_write) begin
        next_state = `ST_WRITEBACK;
    end else begin
        next_state = `ST_FETCH;
    end
end
```

From this, we can calculate the exact cycle count for each instruction class. The synchronous RAM in `rtl/ram.v` responds one cycle after `mem_valid` is asserted, so both FETCH and MEMORY take two cycles (one to assert the request, one to latch the response via `mem_ready`):

Table 35.1: Cycle counts per instruction class

Instruction Class	States Visited	Cycles
ALU reg-reg (**ADD**, **SUB**, …)	FETCH→DECODE→EXECUTE→WRITEBACK	5
ADDI Rd, imm8	FETCH→DECODE→EXECUTE→WRITEBACK	5
Branch (not taken)	FETCH→DECODE→EXECUTE→FETCH	4
Branch (taken)	FETCH→DECODE→EXECUTE→FETCH	4
J offset12	FETCH→DECODE→EXECUTE→FETCH	4
LW / **LB**	FETCH→DECODE→EXECUTE→MEMORY→WRITEBACK	7
SW / **SB**	FETCH→DECODE→EXECUTE→MEMORY→WRITEBACK	7
PUSH	FETCH→DECODE→EXECUTE→MEMORY→WRITEBACK	7
POP	FETCH→DECODE→EXECUTE→MEMORY→WRITEBACK	7
Extended (**LIX**, **ADDIX**, …)	FETCH→DECODE→FETCH_EXT→EXECUTE→WRITEBACK	7
LWX Rd, Rs, imm16	FETCH→DECODE→FETCH_EXT→EXECUTE→MEMORY→WB	9

Why MEMORY Always Goes Through WRITEBACK

Early versions of the SAMPO had a direct MEMORY→FETCH path for store instructions. This caused a subtle bug: the `mem_ready` signal from the store was still high when FETCH began, causing the CPU to latch stale `mem_rdata` as the new instruction word. The fix, documented in `cpu.v`, routes *all* MEMORY exits through WRITEBACK, giving `mem_ready` a cycle to deassert:

```
`ST_MEMORY: begin
    if (mem_ready) begin
        // Always go through WRITEBACK after MEMORY.
        // For stores: allows mem_ready to deassert before
        // next FETCH (prevents stale rdata latch).
        next_state = `ST_WRITEBACK;
    end
end
```

This adds one cycle to every store instruction but eliminates an entire class of data corruption bugs. A worthwhile trade.

35.1.2 Cycles per Instruction in Practice

The cycle counter in `cpu.v` increments every clock edge:

Listing 35.3: Cycle counter (from `rtl/cpu.v`)

```
reg [31:0] cycle_count;  // Cycle counter
// ...
cycle_count <= cycle_count + 1;
```

For a realistic instruction mix—say, 40% ALU/register operations (5 cycles), 25% loads and stores (7 cycles), 20% branches and jumps (4 cycles), and 15% extended instructions (7–9 cycles)—the weighted average CPI works out to approximately:

$$\text{CPI}_{\text{avg}} \approx 0.40 \times 5 + 0.25 \times 7 + 0.20 \times 4 + 0.15 \times 8 = 5.8 \text{ cycles} \tag{35.1}$$

At 12.5 MHz, that translates to roughly 2.15 million instructions per second, or about 2.15 MIPS.

CPI — Cycles Per Instruction

The *cycles per instruction* (CPI) is the average number of clock cycles the CPU takes to complete one instruction. For a multi-cycle design like the SAMPO, the CPI varies by instruction class and is always greater than 1. A pipelined design aims for a CPI approaching 1.

35.1.3 The Clock Speed Ceiling

The SAMPO runs at 12.5 MHz on the ULX3S. Why not faster? The comment in `fpga/ulx3s_top.v` explains:

Listing 35.4: Clock divider rationale (from `fpga/ulx3s_top.v`)

```
// Clock divider: 25 MHz -> 12.5 MHz
// The CPU's combinational ALU path needs ~81ns (nextpnr
// reports max freq of 12.34 MHz). Running at 12.5 MHz
// (80ns period) gives some timing margin while keeping
// reasonable performance.
```

The critical path—the longest combinational delay between any two flip-flops—runs through the ALU. Specifically, the 16-bit subtraction with carry inversion in the ALU, through the flag generation logic, through the branch condition evaluation, and into the next-PC calculation. That chain involves:

1. The 16-bit ripple-carry adder/subtractor in `rtl/alu.v` (computing `sub_result = {1'b0, a} - {1'b0, b}`)
2. Overflow detection logic (`sub_overflow = (a[15] != b[15]) && (sub_result[15] == b[15])`)
3. The flag mux that selects which flag bits to update
4. The `branch_taken` condition evaluator (a 16-way mux on flag bits)
5. The next-PC adder (PC + offset)

On the ECP5, each of these stages maps to several levels of look-up tables (LUTs). The 16-bit adder alone requires a chain of 16 carry-propagation steps. nextpnr reports this entire path at approximately 81 ns, giving a maximum frequency (F_{max}) of about 12.34 MHz.

Measuring F_{max} with nextpnr

Run the FPGA place-and-route with timing analysis enabled. The Makefile target passes `--freq 25` to tell nextpnr the *desired* frequency, and `--timing-allow-fail` to continue even when timing cannot be met:

Listing 35.5: nextpnr invocation from `verilog/Makefile`

```
$(NEXTPNR_ECP5) --$(DEVICE) --package $(PACKAGE) \
    --json $< --lpf $(LPF) \
    --textcfg $@ --freq 25 --timing-allow-fail
```

The output report will list the worst-case path and its delay. That delay determines F_{max}.

35.1.4 Throughput: Where the Time Goes

Combining CPI and F_{max}, the throughput formula is:

$$\text{Throughput} = \frac{F_{max}}{\text{CPI}_{avg}} = \frac{12.5 \times 10^6}{5.8} \approx 2.15 \text{ MIPS} \tag{35.2}$$

There are exactly two ways to improve throughput:

1. **Reduce CPI** — execute more work per cycle. This is what pipelining achieves: overlap stages so multiple instructions are in flight simultaneously, driving CPI towards 1.
2. **Increase F_{max}** — run the clock faster. Pipelining helps here too: by breaking the critical path into shorter pipeline stages, each stage has less combinational logic and can run at a higher frequency.

A well-designed pipeline attacks both numbers simultaneously. Let us see how.

35.2 Pipelining Concepts

Pipelining

Pipelining is a technique where the execution of instructions is overlapped: while one instruction is being decoded, the next is being fetched, and the one before is being executed. Each instruction still takes multiple cycles to complete, but a new instruction completes every cycle once the pipeline is full.

35.2.1 The Laundry Analogy

The classic analogy: imagine doing laundry. Each load requires washing (30 minutes), drying (30 minutes), and folding (30 minutes). Without pipelining, three loads take $3 \times 90 = 270$ minutes. With pipelining—start washing load 2 while drying load 1—three loads take only $90 + 30 + 30 = 150$ minutes.

The same principle applies to CPU instructions. In our current multi-cycle FSM, the entire CPU sits idle while waiting for memory during FETCH. A pipelined design puts those idle resources to work on the *next* instruction.

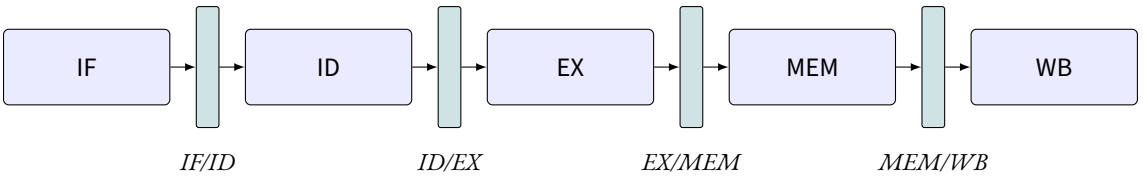
35.2.2 The Classic Five-Stage Pipeline

The textbook RISC pipeline has five stages, and they map directly onto the SAMPO FSM states:

Table 35.2: Multi-cycle states mapped to pipeline stages

Stage	Name	Current FSM State	Work Performed
1	IF (Instruction Fetch)	FETCH	Read instruction from memory
2	ID (Instruction Decode)	DECODE	Decode opcode, read registers
3	EX (Execute)	EXECUTE	ALU operation, branch resolution
4	MEM (Memory Access)	MEMORY	Load/store data memory
5	WB (Write Back)	WRITEBACK	Write result to register file

In a pipelined design, each stage operates on a *different* instruction simultaneously. When instruction n is in the EX stage, instruction $n + 1$ is in ID, and instruction $n + 2$ is in IF. Pipeline registers between stages hold the intermediate results:



The pipeline registers (shown as narrow boxes) are clocked registers that capture all state needed by the next stage: the decoded instruction fields, register values, ALU results, memory data, and control signals.

35.2.3 Pipeline Timing

In the ideal case, a five-stage pipeline achieves a CPI of 1. Each cycle, one instruction enters the pipeline (IF) and one instruction completes (WB). The pipeline is like a conveyor belt: it takes five

cycles for any single instruction to traverse the entire pipeline, but after the initial fill-up, one result rolls off the end every cycle.

The clock period is determined by the *slowest pipeline stage*, not by the full combinational path from fetch to writeback. If the current 81 ns critical path can be split evenly into five stages of roughly 16 ns each, the pipeline could theoretically run at:

$$F_{\text{max,pipeline}} \approx \frac{1}{16\,\text{ns} + t_{\text{setup}}} \approx 50\text{–}55\,\text{MHz} \tag{35.3}$$

where t_{setup} is the flip-flop setup time (approximately 1–2 ns on ECP5). That is a $4\times$ improvement in clock speed, *and* the CPI drops from 5.8 to approximately 1. The combined speedup would be roughly $4 \times 5.8 \approx 23\times$ in ideal conditions.

Of course, reality is never ideal. Hazards—situations where the pipeline must stall or where data is not yet available—reduce the effective CPI above 1.

35.3 Data Hazards and Forwarding

35.3.1 Read-After-Write Hazards

Consider this SAMPO assembly sequence:

Listing 35.6: A read-after-write hazard

```
ADD  R4, R5, R6    ; R4 = R5 + R6     (writes R4 in WB)
SUB  R7, R4, R8    ; R7 = R4 - R8     (reads R4 in ID)
```

In the multi-cycle FSM, this works correctly: **ADD** completes entirely before **SUB** begins. But in a pipeline, when **SUB** reaches the ID stage and reads R4, the **ADD** is only in the EX stage—it has not yet written its result to the register file. **SUB** reads the *old* value of R4.

This is a *read-after-write* (RAW) hazard, the most common type of data hazard. It can occur whenever an instruction reads a register that was written by one of the previous two instructions in the pipeline.

35.3.2 The Stall Solution

The brute-force fix: detect the hazard and insert *stall cycles* (also called "bubbles") into the pipeline. When the ID stage detects that it needs a register currently being computed in EX or MEM, it freezes the IF and ID stages for one or two cycles, allowing the producing instruction to advance and write its result.

This works but is expensive. In tight loops with dependent instructions, stalls can reduce throughput by 30–50%.

35.3.3 Data Forwarding

A better solution is *data forwarding* (also called *bypassing*). Instead of waiting for the result to be written to the register file, we route it directly from the stage that produces it to the stage that needs it.

For the SAMPO, the forwarding paths would be:

1. **EX→EX forwarding**: The ALU result from the EX/MEM pipeline register is forwarded back to the ALU inputs for the next instruction. This handles the case where back-to-back instructions depend on each other.
2. **MEM→EX forwarding**: The result from the MEM/WB pipeline register (either the ALU result or the loaded data) is forwarded to the ALU inputs. This handles the case where the dependent instruction is two cycles behind.
3. **WB→ID forwarding**: If the register file supports simultaneous read and write (read the new value being written on the same cycle), no explicit forwarding is needed for a gap of two or more instructions.

In Verilog, the forwarding logic would look something like:

Listing 35.7: Forwarding mux (hypothetical pipeline extension)

```
// Forward from EX/MEM pipeline register
wire fwd_ex_rs1 = ex_mem_reg_write
    && (ex_mem_rd != 4'h0)
    && (ex_mem_rd == id_ex_rs1);
wire fwd_ex_rs2 = ex_mem_reg_write
    && (ex_mem_rd != 4'h0)
    && (ex_mem_rd == id_ex_rs2);

// Forward from MEM/WB pipeline register
wire fwd_mem_rs1 = mem_wb_reg_write
    && (mem_wb_rd != 4'h0)
    && (mem_wb_rd == id_ex_rs1)
    && !fwd_ex_rs1;  // EX has priority
wire fwd_mem_rs2 = mem_wb_reg_write
    && (mem_wb_rd != 4'h0)
    && (mem_wb_rd == id_ex_rs2)
    && !fwd_ex_rs2;

// ALU input A mux
```

```
wire [15:0] alu_a_fwd =
    fwd_ex_rs1  ? ex_mem_alu_result :
    fwd_mem_rs1 ? mem_wb_result     :
    id_ex_rs1_val;
```

The R0 Exception

Note the check for `rd != 4'h0` in the forwarding conditions. Since R0 is hardwired to zero in the SAMPO (writes are silently discarded, as implemented in `rtl/regfile.v`), we must never forward a "result" that was targeting R0. An instruction like **ADD** R0, R5, R6 does not actually modify R0, and a subsequent read of R0 must return zero.

35.3.4 Load-Use Hazards

Forwarding solves most RAW hazards, but there is one case it cannot fix without time travel. Consider:

Listing 35.8: A load-use hazard

```
    LW   R4, 0(R5)     ; R4 = mem[R5]    (data available after MEM)
    ADD  R6, R4, R7    ; R6 = R4 + R7    (needs R4 in EX)
```

The **LW** instruction does not produce its result until the end of the MEM stage. But the **ADD** needs that value at the *beginning* of EX—one cycle earlier. No amount of forwarding can send data backwards in time.

The solution is a *load-use stall*: the pipeline inserts one bubble after a load when the immediately following instruction reads the loaded register. This is sometimes called a "load interlock." The hazard detection logic:

Listing 35.9: Load-use hazard detection (hypothetical)

```
// Detect load-use hazard: instruction in EX is a load,
// and the instruction in ID reads the load destination
wire load_use_hazard = id_ex_mem_load
    && (id_ex_rd != 4'h0)
    && ((id_ex_rd == if_id_rs1) || (id_ex_rd == if_id_rs2));

// When hazard detected: stall IF and ID, insert bubble in EX
```

Scheduling Around Load-Use Stalls

Compilers (and careful assembly programmers) can avoid load-use stalls by reordering instructions. Instead of:

```
LW   R4, 0(R5)     ; Load
ADD  R6, R4, R7    ; Stall here!
```

Insert an independent instruction between the load and its use:

```
LW   R4, 0(R5)     ; Load
ADDI R8, 1         ; Independent work (fills the delay slot)
ADD  R6, R4, R7    ; No stall --- R4 ready via forwarding
```

This technique is called *instruction scheduling* and is one of the key optimizations a compiler backend performs.

35.3.5 WAW and WAR Hazards

Two other hazard types exist in theory:

- **Write-After-Write (WAW)**: Two instructions write to the same register, and the second write must happen after the first. In a simple five-stage pipeline where instructions flow in order, WAW hazards do not occur—the second instruction naturally writes after the first.
- **Write-After-Read (WAR)**: An instruction writes to a register that a preceding instruction reads. Again, in an in-order pipeline, reads happen in ID (stage 2) and writes in WB (stage 5), so the read always completes before the write.

For the SAMPO, which would remain an in-order design, only RAW hazards (and the load-use special case) require hardware support. WAW and WAR hazards become relevant only in out-of-order processors—a topic well beyond the scope of a 16-bit educational CPU.

35.4 Control Hazards and Branch Prediction

35.4.1 The Branch Problem

In the current multi-cycle SAMPO, the branch condition is evaluated in EXECUTE and the PC is updated before the next FETCH begins. In a pipeline, the problem is that the IF stage has already fetched one or two instructions after the branch before the EX stage determines whether the branch is taken.

Consider the branch evaluation logic from `rtl/cpu.v`:

Listing 35.10: Branch condition evaluation (from `rtl/cpu.v`)

```
always @(*) begin
    case (dec_branch_cond)
        `BR_EQ:  branch_taken = flags[`FLAG_Z];
        `BR_NE:  branch_taken = ~flags[`FLAG_Z];
        `BR_LT:  branch_taken = flags[`FLAG_N] != flags[`FLAG_V];
        `BR_GE:  branch_taken = flags[`FLAG_N] == flags[`FLAG_V];
        `BR_LTU: branch_taken = ~flags[`FLAG_C];
        `BR_GEU: branch_taken = flags[`FLAG_C];
        // ... 16 conditions total
        default: branch_taken = 1'b0;
    endcase
end
```

If the branch is resolved in EX (stage 3), and the pipeline has already fetched instructions at PC+2 (in ID) and PC+4 (in IF), those two instructions must be *flushed* from the pipeline if the branch is taken. The pipeline "wastes" two cycles, known as the *branch penalty*.

Branch Penalty

The *branch penalty* is the number of pipeline stages between instruction fetch and branch resolution. Each wasted cycle is called a *pipeline bubble*. For a five-stage pipeline with branch resolution in EX, the penalty is 2 cycles for taken branches.

35.4.2 Static Branch Prediction

The most common approach for an educational pipeline: *predict not taken*. The pipeline always fetches the next sequential instruction. If the branch turns out to be taken, flush the pipeline and restart from the target address. If not taken, no penalty.

For the SAMPO's branch encoding, the target is computed as `pc + 2 + offset8` (the offset is relative to the *next* instruction). A predict-not-taken strategy means:

- Taken branches cost 2 extra cycles (the penalty)
- Not-taken branches cost 0 extra cycles
- Unconditional jumps (`J`) always pay the penalty

A slightly smarter static strategy is *backward taken, forward not taken* (BTFNT): predict backward branches (loops) as taken and forward branches (if-then-else exits) as not taken. Since loops iterate many times, this can significantly reduce the average penalty.

35.4.3 Dynamic Branch Prediction

For a more sophisticated design, a *branch history table* (BHT) records whether each branch was recently taken or not. A 1-bit BHT uses the branch PC (or a hash of it) to index into a small RAM that stores a single bit: 1 = predict taken, 0 = predict not taken. After each branch, the table is updated with the actual outcome.

A 2-bit saturating counter is more robust: it requires two consecutive mispredictions before switching the prediction direction. This handles the "loop exit" problem where a 1-bit predictor mispredicts twice (once at the last iteration exit, once on re-entry).

Table 35.3: Branch penalty comparison for the SAMPO pipeline

Strategy	Taken Penalty	Not-Taken Penalty
Always predict not taken	2 cycles	0 cycles
BTFNT (static)	0–2 cycles	0–2 cycles
1-bit BHT	0 or 2 cycles	0 or 2 cycles
2-bit saturating BHT	0 or 2 cycles	0 or 2 cycles

Branch Prediction Adds Complexity

For a 16-bit educational CPU, dynamic branch prediction may not be worth the area cost. A 256-entry BHT with 2-bit counters requires 512 bits of storage plus the indexing and update logic. Static predict-not-taken is a reasonable starting point, and loop-intensive code can be optimized by the programmer or compiler to minimize branch penalties.

35.4.4 Jumps and the Delay Slot Alternative

Some classic RISC architectures (MIPS, SPARC) expose the branch penalty to the programmer as a *branch delay slot*: the instruction immediately after a branch is *always* executed, regardless of whether the branch is taken. This eliminates the pipeline flush for taken branches at the cost of programmer complexity.

The SAMPO ISA does *not* use delay slots. This is a deliberate design choice: delay slots make the ISA harder to program and complicate exception handling. The pipeline design must handle branch penalties in hardware.

For unconditional jumps (**J**, **JR**, **JX**), the penalty is unavoidable unless we add early branch resolution logic. Since the jump target is known during ID (for **J**, it is `pc + 2 + offset12`), we can reduce the penalty to 1 cycle by computing the target in ID and redirecting IF immediately.

35.5 A Pipeline Design for SAMPO

35.5.1 Pipeline Register Definitions

Each pipeline register captures the state needed by the next stage. Here is a sketch of the pipeline register contents for a pipelined SAMPO:

Listing 35.11: Pipeline register definitions (design sketch)

```
// IF/ID pipeline register
reg [15:0] if_id_pc;           // PC of this instruction
reg [15:0] if_id_instr;        // Fetched instruction word

// ID/EX pipeline register
reg [15:0] id_ex_pc;
reg [15:0] id_ex_rs1_val;      // Read from regfile
reg [15:0] id_ex_rs2_val;      // Read from regfile
reg [3:0]  id_ex_rd;           // Destination register
reg [3:0]  id_ex_rs1, id_ex_rs2; // Source register numbers
reg [3:0]  id_ex_alu_op;
reg [15:0] id_ex_imm;          // Immediate value
reg        id_ex_mem_load, id_ex_mem_store;
reg        id_ex_reg_write;
reg        id_ex_is_branch;
reg [3:0]  id_ex_branch_cond;

// EX/MEM pipeline register
reg [15:0] ex_mem_alu_result;
reg [15:0] ex_mem_rs2_val;     // For store data
reg [3:0]  ex_mem_rd;
reg        ex_mem_mem_load, ex_mem_mem_store;
reg        ex_mem_reg_write;
reg        ex_mem_mem_byte;

// MEM/WB pipeline register
reg [15:0] mem_wb_result;      // ALU result or loaded data
reg [3:0]  mem_wb_rd;
reg        mem_wb_reg_write;
```

35.5.2 The Fetch Stage

The IF stage reads the instruction at the current PC. Using the existing synchronous RAM from `rtl/ram.v`, this requires careful handling: the RAM returns data one cycle after the address is presented.

Listing 35.12: Fetch stage (design sketch)

```
// Instruction memory interface
assign imem_addr = pc;
assign imem_en   = !stall && !flush;

always @(posedge clk) begin
    if (!rst_n) begin
        if_id_pc    <= RESET_VECTOR;
        if_id_instr <= 16'h0000;  // NOP
    end else if (flush) begin
        if_id_instr <= 16'h0000;  // Inject bubble
    end else if (!stall) begin
        if_id_pc    <= pc;
        if_id_instr <= imem_rdata;
        pc          <= next_pc;
    end
end
```

Separate Instruction and Data Memory

The current SAMPO SoC uses a single RAM for both instructions and data. In a pipeline, IF and MEM may both need memory access on the same cycle—a *structural hazard*. The solution is either a *Harvard architecture* with separate instruction and data memories, or a dual-port RAM. The ECP5's block RAMs (EBR) support true dual-port operation, so this is feasible without doubling the memory.

35.5.3 The Decode Stage

The ID stage decodes the instruction and reads register values. The existing decoder module in `rtl/decode.v` is purely combinational, so it can be reused directly:

Listing 35.13: Decode stage reads (design sketch)

```
// Instantiate existing decoder
decode pipe_decode (
    .instr(if_id_instr),
    .imm16(16'h0000),  // Extended handled separately
    .opcode(dec_opcode),
    .rd(dec_rd),
    .rs1(dec_rs1),
    // ... all other outputs
);
```

```
// Register file read
assign rf_rd_addr1 = dec_rs1;
assign rf_rd_addr2 = dec_rs2;

// Latch into ID/EX pipeline register
always @(posedge clk) begin
    if (!stall) begin
        id_ex_rs1_val <= rf_rd_data1;
        id_ex_rs2_val <= rf_rd_data2;
        id_ex_rd      <= dec_rd;
        id_ex_rs1     <= dec_rs1;
        id_ex_rs2     <= dec_rs2;
        id_ex_alu_op  <= dec_alu_op;
        // ... control signals
    end
end
```

35.5.4 Execute, Memory, and Writeback

The EX stage operates the ALU (reusing `rtl/alu.v`), applies forwarding muxes to the ALU inputs, and evaluates branch conditions. The MEM stage performs load/store memory access. The WB stage writes the result back to the register file.

Listing 35.14: Writeback stage (design sketch)

```
// Writeback: write to register file
assign rf_wr_en   = mem_wb_reg_write
    && (mem_wb_rd != `REG_ZERO);
assign rf_wr_addr = mem_wb_rd;
assign rf_wr_data = mem_wb_result;
```

The `R0` guard (`mem_wb_rd != `REG_ZERO`) mirrors the existing logic in `rtl/cpu.v`:

Listing 35.15: Current writeback guard (from `rtl/cpu.v`)

```
`ST_WRITEBACK: begin
    rf_wr_en = dec_reg_write && (dec_rd != `REG_ZERO);
    // ...
end
```

35.5.5 Handling Extended Instructions

The SAMPO's 32-bit extended instructions (opcode prefix `0xF`) present a unique challenge for pipelining. In the current FSM, the FETCH_EXT state fetches the second word:

Listing 35.16: Extended instruction fetch (from `rtl/cpu.v`)

```
`ST_DECODE: begin
    if (dec_is_extended) begin
        next_state = `ST_FETCH_EXT;
    end else begin
        next_state = `ST_EXECUTE;
    end
end

`ST_FETCH_EXT: begin
    if (mem_ready) begin
        next_state = `ST_EXECUTE;
    end
end
```

In a pipeline, extended instructions would need to fetch two words from instruction memory. Two approaches:

1. **Two-cycle fetch**: When the ID stage detects an extended instruction (opcode = `0xF`), it stalls the pipeline for one cycle while fetching the second word. This is the approach used by ARM Thumb-2 and RISC-V compressed extensions.

2. **Instruction buffer**: Fetch 32 bits per cycle (using a wider instruction memory or a small prefetch buffer). The buffer provides aligned 16-bit or 32-bit instructions to the decoder. This avoids stalls but adds hardware complexity.

For the SAMPO, the two-cycle fetch approach is more practical. The penalty is one stall cycle for every extended instruction—an acceptable trade-off since extended instructions represent a small fraction of the instruction mix.

35.5.6 Pipeline Control: Stalls and Flushes

The pipeline control unit manages two key signals:

- **Stall**: Freezes one or more stages, preventing them from advancing. Used for load-use hazards and extended instruction fetches. When a stage is stalled, its pipeline register retains its current value, and a bubble (NOP) is injected into the next stage.

- **Flush**: Invalidates instructions in the pipeline. Used when a branch is mispredicted: the instructions fetched after the branch (in IF and ID) are replaced with NOPs.

Listing 35.17: Pipeline control (design sketch)

```
// Stall signal: hold IF and ID when load-use hazard detected
wire stall = load_use_hazard || ext_fetch_pending;

// Flush signal: clear IF/ID and ID/EX on branch misprediction
wire flush = branch_mispredicted;

// Branch misprediction: branch resolved in EX, but we fetched
// the wrong instructions
wire branch_mispredicted =
    id_ex_is_branch && branch_taken;  // predict-not-taken
```

35.5.7 The EXX Problem

The **EXX** instruction swaps R4–R11 with their alternates. In the current design (from `rtl/regfile.v`):

Listing 35.18: EXX implementation (from `rtl/regfile.v`)

```
if (exx) begin
    for (i = 0; i < 8; i = i + 1) begin
        regs[i + 4] <= alt_regs[i];
        alt_regs[i] <= regs[i + 4];
    end
end
```

This swaps all eight registers atomically in a single cycle. In a pipeline, an **EXX** in the EX stage would swap the register file contents, but instructions already decoded in ID have read from the *pre-swap* register file. Any subsequent instruction reading R4–R11 would get stale values.

The solution is to flush the pipeline after **EXX**—treating it like an unconditional branch—and re-fetch the following instructions with the swapped register file. This is a rare instruction (typically used for context switching), so the performance penalty is negligible.

35.5.8 The PUSH/POP Dual-Write

The **POP** instruction writes two registers simultaneously: the popped data to the destination register and the updated stack pointer to R2. The current design handles this with a dedicated SP write port on the register file:

Listing 35.19: POP dual-write (from `rtl/cpu.v`)

```
if (sp_update_pending) begin
    rf_sp_wr_en = 1'b1;
    rf_sp_wr_data = sp_new_val;
end
```

In a pipelined design, the dual-write port is essential. Without it, **POP** would need two writeback cycles, stalling the pipeline. The SAMPO register file already has this port, so the pipelined design can inherit it directly.

35.6 A Pipeline Design for SAMPO — Putting It Together

Let us assemble a complete picture of the pipelined SAMPO. The following table summarizes the expected CPI impact of each hazard type:

Table 35.4: Expected pipeline CPI analysis

Event	Penalty	Estimated Frequency
Ideal (no hazards)	0 cycles	60% of instructions
Load-use stall	1 cycle	10% of instructions
Taken branch flush	2 cycles	15% of instructions
Extended instruction stall	1 cycle	10% of instructions
EXX/**PUSH**/**POP** stalls	1–2 cycles	5% of instructions

The effective CPI for the pipelined design:

$$\text{CPI}_{\text{pipe}} \approx 1.0 + 0.10 \times 1 + 0.15 \times 2 + 0.10 \times 1 + 0.05 \times 1.5 \approx 1.575 \tag{35.4}$$

If the pipeline also allows increasing $F_{\max}$ from 12.5 MHz to, say, 40 MHz (a conservative estimate accounting for uneven stage delays), the throughput becomes:

$$\text{Throughput}_{\text{pipe}} = \frac{40 \times 10^6}{1.575} \approx 25.4 \text{ MIPS} \tag{35.5}$$

Compared to the current 2.15 MIPS, that is a speedup of approximately $12\times$—not the theoretical $23\times$ of a perfect pipeline, but a substantial improvement.

35.7 Area vs. Speed Tradeoffs on ECP5

35.7.1 Current Resource Usage

The multi-cycle SAMPO design is area-efficient. The entire SoC—CPU, RAM, UART, and glue logic—fits comfortably on an ECP5-85F, which provides 83,640 LUTs, 3,744 Kbit of embedded block RAM (EBR), and 205 multiply-accumulate (DSP) blocks.

Running synthesis with Yosys gives a resource utilization report. The Makefile target:

Listing 35.20: Running synthesis (from `verilog/Makefile`)

```
yosys -p "read_verilog -I$(RTL_DIR) $(RTL_DIR)/alu.v \
    $(RTL_DIR)/shifter.v $(RTL_DIR)/regfile.v \
    $(RTL_DIR)/decode.v $(RTL_DIR)/cpu.v \
    $(RTL_DIR)/ram.v $(RTL_DIR)/uart.v $(RTL_DIR)/soc.v \
    $(FPGA_DIR)/uart_tx.v $(FPGA_DIR)/uart_rx.v \
    $(FPGA_DIR)/reset_sync.v $(FPGA_DIR)/ulx3s_top.v; \
    synth_ecp5 -top ulx3s_top -json $@"
```

The multi-cycle design uses roughly:

Table 35.5: Estimated resource usage: multi-cycle vs. pipelined

Resource	Multi-Cycle	Pipelined (est.)	ECP5-85F Total
LUT4s	~2,500	~4,500	83,640
Flip-flops	~400	~900	83,640
Block RAM (EBR)	16 (64 KB)	32 (128 KB)	208
DSP blocks	4 (MUL/DIV)	4 (MUL/DIV)	205

35.7.2 Pipeline Overhead

The pipeline adds hardware in several areas:

1. **Pipeline registers**: Four sets of pipeline registers, each holding 50–80 bits of state. This approximately doubles the flip-flop count but is negligible on an ECP5 where flip-flops are essentially free (one per LUT).

2. **Forwarding muxes**: Two 3-to-1 muxes on each ALU input (forwarded from EX/MEM, MEM/WB, or direct from register file). Each 16-bit 3-to-1 mux requires approximately 32 LUTs.

3. **Hazard detection**: Comparators checking source and destination register numbers across pipeline stages. Each 4-bit comparator is 4 LUTs, and we need four of them (two sources checked against two pipeline stages).

4. **Dual-port instruction memory**: The pipeline needs separate instruction and data memory ports. Using ECP5 block RAM in true dual-port mode doubles the EBR usage for the 64 KB RAM (from 16 to 32 blocks).

5. **Pipeline control**: Stall, flush, and bubble-injection logic adds perhaps 50–100 LUTs.

In total, the pipelined design roughly doubles the LUT usage and flip-flop count compared to the multi-cycle design. On the ECP5-85F, this is still a small fraction of available resources (5% of LUTs instead of 3%).

35.7.3 The Timing vs. Area Tradeoff

The pipeline's biggest benefit for timing is splitting the critical path. The current 81 ns path through the ALU and branch logic becomes:

- **EX stage**: ALU computation (~30 ns, dominated by the 16-bit adder carry chain plus the forwarding mux)
- **ID stage**: Register file read + decode (~15 ns)
- **MEM stage**: RAM access (~20 ns, the block RAM access time)
- **WB stage**: Mux and register write (~5 ns)
- **IF stage**: PC computation + memory request (~10 ns)

The EX stage remains the bottleneck at ~30 ns, suggesting an F_{max} around 30–33 MHz. To push higher, the EX stage could be further split—for example, separating the ALU computation from the branch resolution into two sub-stages, creating a six- or seven-stage pipeline. However, deeper pipelines increase the branch penalty and add more forwarding complexity.

ECP5 Carry Chains

The ECP5 has dedicated fast carry logic (the CCU2C primitive) that implements carry chains much faster than generic LUT routing. Yosys automatically maps adders to these carry chains. The 16-bit adder in the SAMPO ALU uses 8 CCU2C primitives (each handles 2 bits), and the carry propagation through this chain is roughly 8–10 ns—much faster than the equivalent LUT-based implementation.

35.7.4 When Pipelining Is Not Worth It

Pipelining is not free. Here are scenarios where the multi-cycle design may actually be preferable:

- **Area-constrained designs**: On a smaller FPGA (ECP5-12F or ECP5-25F), the pipeline overhead may consume a significant fraction of available LUTs, leaving less room for peripherals and application logic.
- **Low clock-speed applications**: If the system clock is limited by external factors (e.g., a slow memory bus or a peripheral that requires a specific clock frequency), the pipeline's higher F_{max} provides no benefit.
- **Control-heavy code**: If the application consists mostly of branches (like a command interpreter or state machine), the branch penalty may raise CPI close to the multi-cycle value, negating the pipeline's benefits.
- **Simplicity and debuggability**: The multi-cycle FSM is far easier to understand, simulate, and debug. Every instruction executes atomically; there is no need to reason about forwarding, stalls, or pipeline flushes. For an educational project, this clarity has significant value.

Profile Before You Pipeline

Before investing weeks of effort in a pipelined redesign, profile your actual workload. The SAMPO's 32-bit cycle counter (`cycle_count` in `rtl/cpu.v`) can be read at the end of a test run. Compare the cycle count against the instruction count to measure real-world CPI. If your code is dominated by I/O waits (e.g., UART polling), no amount of pipelining will help—the CPU is already idle most of the time.

35.7.5 A Pragmatic Middle Ground

Between the full five-stage pipeline and the current multi-cycle FSM lies a practical middle ground: a *two-stage pipeline* that overlaps fetch and execute.

In this design, while one instruction is being executed (through decode, execute, memory, and writeback in a multi-cycle fashion), the *next* instruction is being fetched from memory. This eliminates the two-cycle fetch latency for most instructions, reducing the CPI by approximately 2 cycles without any forwarding or hazard detection logic:

The two-stage design keeps the simplicity of the multi-cycle control logic while extracting a meaningful performance improvement. It requires only a small instruction buffer (one 16-bit register) and minimal control changes. For many applications, this may be the sweet spot.

Table 35.6: CPI comparison across designs

Design	Average CPI	MIPS at 12.5 MHz
Multi-cycle (current)	~5.8	~2.15
Two-stage (fetch overlap)	~3.8	~3.3
Five-stage pipeline (40 MHz)	~1.6	~25.4

Exercises

1. **Cycle Counting.** Write a SAMPO assembly program that performs 100 iterations of a loop containing an **ADD**, an **ADDI**, a **LW**, and a **BNE**. Before the loop, record the cycle counter. After the loop, record it again and compute the measured CPI. Compare your result to the predictions in Table 35.1.

2. **Critical Path Analysis.** Examine the ALU module in `verilog/rtl/alu.v`. Trace the longest combinational path from input `a` and `b` to the output `flag_c` for the `ALU_SUB` operation. Count the number of logic levels (LUT depths) involved. How does the ECP5 carry chain reduce this depth compared to a pure LUT implementation?

3. **Forwarding Design.** Sketch the complete forwarding unit for a five-stage pipelined SAMPO. Consider all source operand combinations for the following instruction types: **ADD** (3-register), **ADDI** (2-operand), **CMP** (uses Rd as source), and **PUSH** (reads SP implicitly). How many forwarding mux inputs does each ALU input need?

4. **Branch Penalty.** Write two versions of a sum-of-array loop in SAMPO assembly: one with the branch at the bottom (**BNE** back to the loop top) and one with the branch at the top (**BEQ** to the exit). Analyze the branch penalty of each version under a predict-not-taken strategy. Which version is faster in a pipelined design?

5. **Area Estimation.** Using the resource estimates in Table 35.5, calculate whether a five-stage pipelined SAMPO with 64 KB dual-port RAM would fit on an ECP5-25F (24,576 LUTs, 56 EBR blocks). What compromises would be necessary?

What's Next

We have analyzed the SAMPO's performance characteristics and sketched how pipelining could improve throughput by an order of magnitude. But raw speed means little without software to exploit it. In the next chapter, we turn from hardware optimization to software infrastructure: what would it take to make the SAMPO a target for a real compiler? We will explore LLVM backend concepts, calling conventions, and the path from Rust source code all the way to bits running on the FPGA.

Chapter 36

Towards a Compiler Target

Up to this point, every SAMPO program in this book has been written by hand in assembly language. For a 20-line Fibonacci routine, that is manageable. For a 20,000-line application, it is not. If the SAMPO is to be more than a pedagogical curiosity—if it is to run real software, even modestly complex software—it needs a compiler. Not a toy compiler that handles a subset of some invented language, but a production compiler that takes real Rust (or C) source code and produces correct, reasonably efficient SAMPO machine code.

This chapter maps the path from high-level language to FPGA execution. We will explore the LLVM compiler framework, define a target description for the SAMPO architecture, trace how Rust source code becomes assembly, show what a `no_std` Rust program looks like on a 16-bit target, and walk through the full compilation loop from `cargo build` to bits on the ULX3S.

36.1 The LLVM Backend Concept

36.1.1 Why LLVM?

Writing a compiler from scratch is a multi-year effort. Writing a compiler *backend*—the part that translates an intermediate representation (IR) into machine code for a specific architecture—is a much smaller task, especially when the framework handles optimization, register allocation, and instruction scheduling for you.

LLVM is that framework. It is the compiler infrastructure behind Clang (C/C++), Rust (via `rustc`), Swift, Julia, and dozens of other languages. LLVM defines a *target-independent* intermediate representation called LLVM IR, and provides a *target-dependent* code generation framework where new architectures are described using a domain-specific language called TableGen.

LLVM Backend

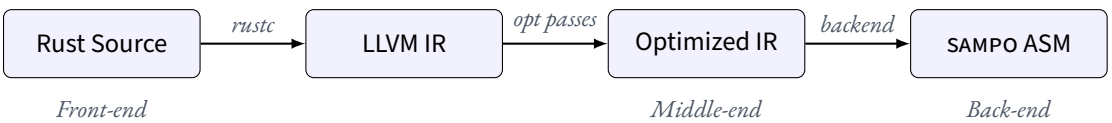
An LLVM *backend* is a set of C++ classes and TableGen descriptions that teach LLVM how to generate machine code for a specific architecture. It defines the register file, instruction set, calling convention, and lowering rules that transform LLVM IR into native instructions.

The key insight: by writing a SAMPO backend for LLVM, we inherit the entire Rust and C front-end ecosystem for free. Every Rust crate that compiles to LLVM IR—which is all of them—becomes potentially available to the SAMPO.

36.1.2 The Compilation Pipeline

Here is the full path from source code to machine code:

1. **Front-end** (language-specific): Rust source → HIR → MIR → LLVM IR. The Rust compiler (`rustc`) handles this.
2. **Middle-end** (target-independent): LLVM IR → optimized LLVM IR. Dead code elimination, constant folding, loop unrolling, inlining—all happen here, independent of the target.
3. **Back-end** (target-dependent): Optimized LLVM IR → SelectionDAG → MachineInstr → MCInst → assembly or object code. This is where the SAMPO backend lives.



36.1.3 What an LLVM Backend Contains

An LLVM backend for the SAMPO would consist of roughly these components:

The TableGen (`.td`) files are declarative: they describe *what* the architecture looks like. The C++ files contain procedural logic for cases too complex for TableGen patterns.

36.1.4 Existing Small-Target Backends

LLVM already supports several small RISC architectures that serve as reference implementations for a SAMPO backend:

- **MSP430**: Texas Instruments' 16-bit RISC microcontroller. This is the closest analog to the SAMPO: 16-bit registers, 16-bit address space, similar instruction density. The MSP430 backend is one of the simplest in the LLVM tree.

Table 36.1: Components of a SAMPO LLVM backend

Component	Purpose
`SampoRegisterInfo.td`	Register definitions and classes
`SampoInstrInfo.td`	Instruction patterns and encodings
`SampoCallingConv.td`	Calling convention rules
`SampoInstrFormats.td`	Instruction format bit layouts
`SampoISelDAGToDAG.cpp`	Instruction selection (DAG matching)
`SampoISelLowering.cpp`	IR lowering (custom legalization)
`SampoFrameLowering.cpp`	Stack frame layout
`SampoAsmPrinter.cpp`	Assembly text output
`SampoMCCodeEmitter.cpp`	Binary encoding

- **AVR**: Atmel's 8-bit RISC architecture (Arduino). Interesting for its handling of 8-bit operations on a platform where LLVM's IR prefers wider types.
- **RISC-V (RV32I)**: A clean 32-bit RISC target. The SAMPO borrows several conventions from RISC-V (the zero register, calling convention structure), making this backend a useful template.

Start with the MSP430 Backend

The MSP430 LLVM backend is approximately 5,000 lines of TableGen and 3,000 lines of C++. That is small enough to read in a weekend and provides a complete working example of a 16-bit target. Clone the LLVM repository and study `llvm/lib/Target/MSP430/` as a starting point.

36.2 Target Description

36.2.1 Register Definitions

The first step in defining a backend is describing the register file. From `ARCHITECTURE.md`, the SAMPO has 16 general-purpose 16-bit registers with specific conventions:

Listing 36.1: Register conventions (from `ARCHITECTURE.md`)

```
; Register aliases
; R0=ZERO  R1=RA   R2=SP   R3=GP
; R4=A0    R5=A1   R6=A2   R7=A3
; R8=T0    R9=T1   R10=T2  R11=T3
; R12=S0   R13=S1  R14=S2  R15=S3
```

In LLVM TableGen, these would be described as:

Listing 36.2: Register definitions in TableGen (hypothetical `SampoRegisterInfo.td`)

```
// Sampo register file - 16 x 16-bit GPRs
class SampoReg<bits<4> Enc, string n>
    : Register<n> {
  let HWEncoding = Enc;
  let Namespace = "Sampo";
}

// Physical registers
def R0  : SampoReg<0,  "R0">;   // ZERO
def R1  : SampoReg<1,  "R1">;   // RA
def R2  : SampoReg<2,  "R2">;   // SP
def R3  : SampoReg<3,  "R3">;   // GP
def R4  : SampoReg<4,  "R4">;   // A0
def R5  : SampoReg<5,  "R5">;   // A1
def R6  : SampoReg<6,  "R6">;   // A2
def R7  : SampoReg<7,  "R7">;   // A3
def R8  : SampoReg<8,  "R8">;   // T0
def R9  : SampoReg<9,  "R9">;   // T1
def R10 : SampoReg<10, "R10">;  // T2
def R11 : SampoReg<11, "R11">;  // T3
def R12 : SampoReg<12, "R12">;  // S0
def R13 : SampoReg<13, "R13">;  // S1
def R14 : SampoReg<14, "R14">;  // S2
def R15 : SampoReg<15, "R15">;  // S3
```

Register classes group registers that can be used interchangeably. The SAMPO needs at least two classes:

Listing 36.3: Register classes in TableGen

```
// All GPRs (for general allocation)
def GPR : RegisterClass<"Sampo", [i16], 16,
    (add R4, R5, R6, R7,        // A0-A3 (arguments)
         R8, R9, R10, R11,      // T0-T3 (temporaries)
         R12, R13, R14, R15,    // S0-S3 (callee-saved)
         R3, R1)> {             // GP, RA (last resort)
  // R0 excluded (always zero)
  // R2 excluded (stack pointer, managed separately)
}

// Callee-saved registers only
```

```
def GPRSaved : RegisterClass<"Sampo", [i16], 16,
    (add R12, R13, R14, R15)>;
```

Allocation Order Matters

The order of registers in the `RegisterClass` definition determines the allocation preference. By listing `R4`–`R7` (argument registers) first, followed by `R8`–`R11` (temporaries), and `R12`–`R15` (callee-saved) last, we tell the register allocator to prefer caller-saved registers. This minimizes the number of registers that need saving in function prologues.

36.2.2 Instruction Patterns

LLVM's instruction selection works by pattern matching on a SelectionDAG—a directed acyclic graph representing the computation. Each SAMPO instruction is described as a pattern that matches a particular DAG shape.

For example, the **`ADD`** instruction (opcode `0x0`) matches the IR `add` operation:

Listing 36.4: ADD instruction pattern in TableGen

```
// Format R: opcode[15:12] Rd[11:8] Rs1[7:4] Rs2[3:0]
class FormatR<bits<4> opcode, dag outs, dag ins,
              string asmstr, list<dag> pattern>
    : SampoInst<outs, ins, asmstr, pattern> {
  bits<4> rd;
  bits<4> rs1;
  bits<4> rs2;
  let Inst{15-12} = opcode;
  let Inst{11-8}  = rd;
  let Inst{7-4}   = rs1;
  let Inst{3-0}   = rs2;
}

// ADD Rd, Rs1, Rs2
def ADD : FormatR<0b0000,
    (outs GPR:$rd), (ins GPR:$rs1, GPR:$rs2),
    "ADD\t$rd, $rs1, $rs2",
    [(set GPR:$rd, (add GPR:$rs1, GPR:$rs2))]>;

// SUB Rd, Rs1, Rs2
def SUB : FormatR<0b0001,
    (outs GPR:$rd), (ins GPR:$rs1, GPR:$rs2),
    "SUB\t$rd, $rs1, $rs2",
    [(set GPR:$rd, (sub GPR:$rs1, GPR:$rs2))]>;
```

The **ADDI** instruction is more interesting. Recall from ENCODING.md that **ADDI** is a 2-operand instruction: `Rd = Rd + sign_ext(imm8)`. The destination and first source are the *same* register. In TableGen:

Listing 36.5: ADDI pattern with tied operand

```
// Format I: opcode[15:12] Rd[11:8] imm8[7:0]
class FormatI<bits<4> opcode, dag outs, dag ins,
              string asmstr, list<dag> pattern>
    : SampoInst<outs, ins, asmstr, pattern> {
  bits<4> rd;
  bits<8> imm8;
  let Inst{15-12} = opcode;
  let Inst{11-8}  = rd;
  let Inst{7-0}   = imm8;
}

// ADDI Rd, imm8 (Rd = Rd + sext(imm8))
def ADDI : FormatI<0b0101,
    (outs GPR:$rd), (ins GPR:$src, simm8:$imm),
    "ADDI\t$rd, $imm",
    [(set GPR:$rd, (add GPR:$src, simm8:$imm))]> {
  // $rd and $src must be the same register
  let Constraints = "$rd = $src";
}
```

The ADDI Constraint

The `Constraints = "$rd = $src"` line is critical. It tells the register allocator that the input and output must be the same physical register. Without this constraint, LLVM might allocate the source and destination to different registers, producing incorrect code. The same constraint applies to **MUL**, **DIV**, and other 2-operand instructions in the SAMPO ISA.

36.2.3 The Calling Convention

The calling convention defines how functions pass arguments, return values, and manage the stack. From ARCHITECTURE.md:

Listing 36.6: Calling convention (from ARCHITECTURE.md)

```
; Arguments:      R4-R7 (A0-A3), then stack
; Return values:  R4-R5 (A0-A1)
; Caller-saved:   R4-R11 (A0-A3, T0-T3)
; Callee-saved:   R12-R15 (S0-S3), SP
; Return address: R1 (RA)
```

In TableGen, this becomes:

Listing 36.7: Calling convention in TableGen

```
// Sampo calling convention
def CC_Sampo : CallingConv<[
  // First 4 args in registers
  CCIfType<[i16], CCAssignToReg<[R4, R5, R6, R7]>>,
  // Remaining args on stack, 2-byte aligned
  CCIfType<[i16], CCAssignToStack<2, 2>>
]>;

// Return value convention
def RetCC_Sampo : CallingConv<[
  CCIfType<[i16], CCAssignToReg<[R4, R5]>>
]>;

// Callee-saved registers
def CSR_Sampo : CalleeSavedRegs<
    (add R12, R13, R14, R15)>;
```

The function prologue and epilogue follow directly. Here is the standard prologue from `ARCHITECTURE.md`:

Listing 36.8: Standard function prologue (from `ARCHITECTURE.md`)

```
function:
    PUSH RA             ; Save return address
    PUSH S0             ; Save callee-saved registers (as needed)
    PUSH S1
    ADDI SP, -N         ; Allocate N bytes for local variables
```

And the epilogue:

Listing 36.9: Standard function epilogue (from `ARCHITECTURE.md`)

```
    ADDI SP, N          ; Deallocate local variables
    POP S1              ; Restore callee-saved registers
    POP S0
    POP RA              ; Restore return address
    JR RA               ; Return to caller
```

36.2.4 Handling 16-Bit Limitations

The SAMPO is a 16-bit machine, but LLVM's IR uses types like `i32`, `i64`, and pointers of various widths. The backend must *legalize* these types—convert operations on unsupported types into sequences of supported operations.

- **Pointers**: On the SAMPO, pointers are 16 bits (matching the 64 KB address space). The data layout string tells LLVM this: `p:16:16` (16-bit pointers, 16-bit aligned).

- **32-bit integers**: Operations on `i32` must be *expanded* into pairs of 16-bit operations. An `i32` addition becomes two **ADD** instructions with carry propagation via **ADC**:

Listing 36.10: 32-bit addition on a 16-bit machine

```
; i32 add: result = a + b
; a in R4:R5 (low:high), b in R6:R7 (low:high)
    ADD  R4, R4, R6     ; Add low words, C = carry out
    ADC  R5, R5, R7     ; Add high words + carry
; Result in R4:R5
```

Listing 36.11: 32-bit subtraction using ARM-style carry

```
; i32 sub: result = a - b
; a in R4:R5 (low:high), b in R6:R7 (low:high)
    SUB  R4, R4, R6     ; Sub low words, C=1 if no borrow
    SBC  R5, R5, R7     ; Sub high words - borrow
; Result in R4:R5
```

ARM-Style Carry Convention

The SAMPO uses ARM-style carry semantics for subtraction: C=1 means no borrow occurred (i.e., the result was non-negative in unsigned arithmetic). This is implemented in `verilog/rtl/alu.v`:

```
`ALU_SUB: begin
    result = sub_result[15:0];
    // Carry set when there's NO borrow (a >= b)
    flag_c = ~sub_result[16];
    flag_v = sub_overflow;
end
```

This convention makes multi-word subtraction natural: **SBC** subtracts the inverse of the carry (the borrow), so the **SUB**/**SBC** sequence works correctly without complementing flags between words.

- **64-bit integers**: Expanded into four 16-bit operations. This is expensive but functional.
- **Floating point**: The SAMPO has no floating-point hardware. Floating-point operations must use software emulation libraries (soft-float). For a 16-bit target, single-precision IEEE 754 requires significant code.
- **Division**: The SAMPO has hardware **DIV** and **REM** instructions (opcode `0xB`, implemented in `rtl/alu.v`), so integer division does not need software emulation.

36.2.5 The Data Layout String

LLVM needs a concise description of the target's data model. For the SAMPO:

Listing 36.12: Data layout string for the SAMPO

```
// Little-endian, 16-bit pointers, 16-bit natural alignment
"e-m:e-p:16:16-i8:8:8-i16:16:16-i32:16:32-n16-S16"
```

Breaking this down:

- `e` — little-endian (matching the hardware in `rtl/ram.v`)
- `m:e` — ELF-style name mangling
- `p:16:16` — pointers are 16-bit, 16-bit aligned
- `i8:8:8` — bytes are 8-bit aligned
- `i16:16:16` — 16-bit integers are 16-bit aligned

- `i32:16:32` — 32-bit integers: 16-bit ABI align, 32-bit preferred
- `n16` — native integer width is 16 bits
- `S16` — stack is 16-bit aligned

36.3 From Rust to Assembly

36.3.1 A Minimal Rust Function

Let us trace a minimal Rust function through the compilation pipeline to see what the SAMPO backend would produce. Consider:

Listing 36.13: A minimal Rust function

```
#[no_mangle]
pub fn add_values(a: u16, b: u16) -> u16 {
    a + b
}
```

The Rust compiler transforms this into LLVM IR:

Listing 36.14: LLVM IR for add_values

```
define i16 @add_values(i16 %a, i16 %b) {
entry:
  %result = add i16 %a, %b
  ret i16 %result
}
```

The SAMPO backend would lower this to:

Listing 36.15: Generated SAMPO assembly for add_values

```
add_values:
    ADD  R4, R4, R5     ; R4 = a + b (args in R4, R5)
    JR   RA             ; Return (result in R4)
```

This is identical to the hand-written example in `ARCHITECTURE.md`:

Listing 36.16: Hand-written add function (from `ARCHITECTURE.md`)

```
; int add(int a, int b)
; Arguments: R4 = a, R5 = b
; Returns: R4 = a + b
```

```
add:
    ADD R4, R4, R5        ; R4 = R4 + R5
    JR RA                 ; Return
```

A leaf function (one that does not call other functions) needs no prologue or epilogue—no stack manipulation, no register saving. The SAMPO's calling convention makes this efficient.

36.3.2 A Function with Control Flow

Consider a function with a conditional:

Listing 36.17: Rust function with control flow

```
#[no_mangle]
pub fn max(a: u16, b: u16) -> u16 {
    if a > b { a } else { b }
}
```

The LLVM IR uses branches:

Listing 36.18: LLVM IR for max

```
define i16 @max(i16 %a, i16 %b) {
entry:
  %cmp = icmp ugt i16 %a, %b
  br i1 %cmp, label %then, label %else
then:
  br label %end
else:
  br label %end
end:
  %result = phi i16 [ %a, %then ], [ %b, %else ]
  ret i16 %result
}
```

The backend translates the `icmp ugt` (unsigned greater than) comparison and branch into SAMPO instructions. The SAMPO has 16 branch conditions, and the unsigned-greater-than condition maps to **`BHI`** (condition code `0xE`: C=1 and Z=0), as defined in `ENCODING.md`:

Listing 36.19: Generated assembly for max

```
max:
    CMP  R4, R5           ; Compare a, b (sets flags)
    BHI  .Lthen           ; Branch if a > b (unsigned)
```

```
    MOV  R4, R5          ; a <= b: result = b
.Lthen:
    JR   RA              ; Return R4
```

The backend can optimize the phi node away by recognizing that the "then" case leaves `R4` unchanged (it already holds `a`), and the "else" case copies `b` into `R4`.

36.3.3 A Function with Stack Usage

When a function calls other functions or has local variables, it needs a stack frame:

Listing 36.20: Rust function requiring a stack frame

```
#[no_mangle]
pub fn compute(x: u16, y: u16) -> u16 {
    let sum = add_values(x, y);
    let diff = if x > y { x - y } else { y - x };
    add_values(sum, diff)
}
```

This function calls `add_values` twice, so it is not a leaf function. It must save the return address and any callee-saved registers it uses:

Listing 36.21: Generated assembly for compute

```
compute:
    ; Prologue: save RA and callee-saved registers
    PUSH RA              ; Save return address
    PUSH S0              ; Save S0 (will hold x)
    PUSH S1              ; Save S1 (will hold y)

    ; Save arguments in callee-saved registers
    MOV  S0, R4          ; S0 = x
    MOV  S1, R5          ; S1 = y

    ; First call: add_values(x, y)
    ; R4 = x, R5 = y (already in place)
    JALX add_values      ; Call add_values
    MOV  R8, R4          ; T0 = sum (caller-saved temp)

    ; Compute diff = |x - y|
    CMP  S0, S1          ; Compare x, y
    BHI  .Lx_bigger
    SUB  R5, S1, S0      ; diff = y - x
```

```
    J    .Ldiff_done
.Lx_bigger:
    SUB  R5, S0, S1      ; diff = x - y
.Ldiff_done:

    ; Second call: add_values(sum, diff)
    MOV  R4, R8          ; R4 = sum
    ; R5 = diff (already in place)
    JALX add_values      ; Call add_values

    ; Epilogue: restore and return
    POP  S1
    POP  S0
    POP  RA
    JR   RA
```

Notice how the compiler must save `x` and `y` in callee-saved registers (`S0`, `S1`) because the call to `add_values` may clobber the argument registers (`R4`, `R5`).

36.3.4 Instruction Selection Challenges

Several SAMPO ISA features create interesting instruction selection challenges:

1. **Loading large constants**: The SAMPO has no single 16-bit instruction that loads an arbitrary 16-bit constant. The extended **`LIX`** instruction (opcode `0xF`, sub `0x7`) does this in 32 bits. The instruction selector must recognize constant materialization patterns:

Listing 36.22: Loading constants of various sizes

```
    ; Load 42 (fits in imm8: -128 to +127)
    ADDI R4, 42          ; R4 = R4 + 42 (but R4 might not be 0!)

    ; Load 42 from zero
    ADD  R4, R0, R0      ; R4 = 0
    ADDI R4, 42          ; R4 = 42

    ; Load 0x1234 (needs 32-bit instruction)
    LIX  R4, 0x1234      ; R4 = 0x1234 (one instruction, 4 bytes)
```

2. **Limited branch range**: The branch offset is only 8 bits (signed), giving a range of -256 to $+254$ bytes. For branches beyond this range, the compiler must use a branch-over-jump pattern:

Listing 36.23: Long branch via branch-over-jump

```
    ; if (cond) goto far_label
    BNE  .Lskip          ; Invert condition: skip if NOT cond
    JX   far_label        ; Absolute jump to far label
.Lskip:
```

3. **Two-operand `MUL/DIV`**: These instructions use the destination register as both source and destination (`Rd = Rd op Rs`). From `ENCODING.md`:

Listing 36.24: MUL uses Rd as accumulator

```
    ; R4 = R5 * R6 requires a MOV first:
    MOV  R4, R5          ; R4 = R5
    MUL  R4, R6          ; R4 = R4 * R6
```

   The register allocator handles this via the tied-operand constraint, but it may need to insert **`MOV`** instructions ("copies" in LLVM terminology) when the source and destination are different physical registers.

4. **No indexed addressing with arbitrary offsets**: The **`LW`** instruction supports only fixed small offsets ($0, +2, +4, +6, -2, -4$). For arbitrary offsets, the compiler must use the extended **`LWX`** (4-byte instruction) or pre-compute the address with **`ADDIX`**.

36.4 no_std Rust on SAMPO

36.4.1 Why no_std?

The Rust standard library (`std`) assumes an operating system: threads, file I/O, heap allocation, networking. The SAMPO has none of these. A 64 KB address space does not leave room for a full standard library anyway.

Rust's `#![no_std]` attribute disables the standard library and gives access only to `core`—the platform-independent subset that provides fundamental types, traits, iterators, and arithmetic. The `core` library is designed for exactly this scenario: bare-metal embedded systems with no OS.

36.4.2 A Minimal no_std Program

Here is the smallest complete `no_std` Rust program for the SAMPO:

Listing 36.25: Minimal no_std Rust for SAMPO

```
#![no_std]
#![no_main]
```

```rust
use core::panic::PanicInfo;

// Entry point
#[no_mangle]
pub extern "C" fn _start() -> ! {
    // Write "Hello" to UART
    let msg = b"Hello, Sampo!\n";
    for &byte in msg {
        uart_write(byte);
    }

    // Halt
    loop {}
}

#[inline(never)]
fn uart_write(byte: u8) {
    unsafe {
        // Write to UART data port (0x81)
        core::arch::asm!(
            "OUTI 0x81, {0}",
            in(reg) byte as u16,
        );
    }
}

#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {
    loop {}
}
```

Several elements are worth noting:

- `#![no_main]` tells the compiler there is no standard `main` function. We provide `_start` as the entry point.
- The `panic_handler` is mandatory in `no_std` crates. Since we have no OS to print a panic message to, we simply loop forever.
- The `uart_write` function uses inline assembly to access the SAMPO's port I/O instruction. The **`OUTI`** instruction (opcode `0xD`, func `0x1`) writes a register value to a 4-bit port address.
- The return type `-> !` ("never") indicates that `_start` does not return. Since there is no OS to return to, the program must loop forever or halt.

36.4.3 The Linker Script

A linker script tells the linker how to arrange code and data in the SAMPO's 64 KB address space. Based on the memory map from `ARCHITECTURE.md`:

Listing 36.26: Linker script for the SAMPO (`sampo.ld`)

```
/* Sampo CPU linker script */
MEMORY {
  VECTORS (rx)  : ORIGIN = 0x0000, LENGTH = 256
  FLASH   (rx)  : ORIGIN = 0x0100, LENGTH = 32512
  RAM     (rwx) : ORIGIN = 0x8000, LENGTH = 32512
}

SECTIONS {
  .vectors : {
    LONG(_start)          /* Reset vector at 0x0000 */
    LONG(0x0000)          /* Reserved */
    LONG(_irq_handler)    /* IRQ vector at 0x0004 */
  } > VECTORS

  .text : {
    *(.text .text.*)
  } > FLASH

  .rodata : {
    *(.rodata .rodata.*)
  } > FLASH

  .data : {
    _sdata = .;
    *(.data .data.*)
    _edata = .;
  } > RAM AT > FLASH

  .bss : {
    _sbss = .;
    *(.bss .bss.*)
    _ebss = .;
  } > RAM

  _stack_top = ORIGIN(RAM) + LENGTH(RAM);
}
```

The key sections:

- **.vectors**: The reset vector at address `0x0000` points to `_start`. The IRQ vector at `0x0004` points to the interrupt handler.
- **.text**: Code lives starting at `0x0100` (the SAMPO's `RESET_VECTOR`, as defined in `rtl/cpu.v`).
- **.data**: Initialized data is loaded in FLASH but copied to RAM at startup.
- **.bss**: Uninitialized data in RAM, zeroed at startup.
- **_stack_top**: The stack starts at the top of RAM and grows downward.

36.4.4 Runtime Initialization

Before `_start` runs, the SAMPO needs minimal runtime initialization: set the stack pointer, copy `.data` from FLASH to RAM, and zero `.bss`. This is the *crt0* (C runtime zero) code, written in assembly:

Listing 36.27: Runtime startup code (`crt0.s`)

```
.org 0x0100
_reset:
    ; Initialize stack pointer
    LIX  SP, _stack_top

    ; Copy .data from FLASH to RAM
    LIX  R5, _sdata_load    ; Source (in FLASH)
    LIX  R6, _sdata         ; Destination (in RAM)
    LIX  R4, _data_size     ; Byte count
    CMP  R4, R0
    BEQ  .Lskip_copy
    LDIR                    ; Block copy R4 bytes from [R5] to [R6]
.Lskip_copy:

    ; Zero .bss
    LIX  R6, _sbss          ; Start of BSS
    LIX  R4, _bss_size      ; Byte count
    ADD  R5, R0, R0          ; Fill value = 0
    CMP  R4, R0
    BEQ  .Lskip_bss
    FILL                     ; Fill R4 bytes at [R6] with R5
.Lskip_bss:

    ; Call Rust entry point
    JALX _start

    ; If _start returns (it shouldn't), halt
    HALT
```

Using LDIR and FILL

The SAMPO's block operations shine in startup code. **`LDIR`** copies `R4` bytes from the address in `R5` to the address in `R6`, incrementing both pointers automatically. **`FILL`** fills `R4` bytes at the address in `R6` with the value in `R5`. These Z80-inspired instructions turn multi-instruction loops into single operations—a real win for initialization code that would otherwise be the first thing executed on every boot.

36.4.5 Inline Assembly on SAMPO

For hardware access, Rust programs use inline assembly. The SAMPO backend would support the following register constraints:

Table 36.2: Inline assembly register constraints for SAMPO

Constraint	Meaning
`reg`	Any general-purpose register (`R4`–`R15`)
`in(reg)`	Input value in a GPR
`out(reg)`	Output value in a GPR
`inout(reg)`	Input and output in the same GPR

Example: reading the UART status port:

Listing 36.28: Inline assembly for port I/O

```
fn uart_status() -> u8 {
    let status: u16;
    unsafe {
        core::arch::asm!(
            "INX {0}, 0x80",       // Read UART status port
            out(reg) status,
        );
    }
    status as u8
}

fn uart_tx_ready() -> bool {
    (uart_status() & 0x02) != 0
}
```

The **`INX`** instruction (opcode `0xF`, sub `0xB`) reads from an 8-bit port address specified in the immediate word. The I/O control logic in `rtl/cpu.v` handles the port access:

Listing 36.29: I/O port read (from rtl/cpu.v)

```
if (dec_is_io_in) begin
    io_rd = 1'b1;
    if (dec_io_port_imm) begin
        io_addr = {4'b0, dec_rs1};  // INI: 4-bit port
    end else if (dec_is_extended) begin
        io_addr = imm16[7:0];       // INX: 8-bit port
    end else begin
        io_addr = rs1_val[7:0];     // IN: register port
    end
end
```

36.5 The Full Loop: cargo build to FPGA

36.5.1 The Complete Build Pipeline

Here is the full path from Rust source code to bits running on the ULX3S FPGA:

1. **Write Rust code**: A no_std crate targeting the SAMPO.
2. **Compile to object code**: cargo build invokes rustc, which generates LLVM IR, optimizes it, and uses the SAMPO backend to emit object files.
3. **Link**: The linker combines object files with the crt0.o startup code, resolving symbols and laying out the memory map per sampo.ld.
4. **Extract binary**: objcopy converts the ELF output to a flat binary.
5. **Convert to hex**: The bin2hex.py script converts the binary to Verilog $readmemh format.
6. **Synthesize**: Yosys reads the Verilog RTL and the hex-initialized RAM, synthesizing the complete SoC.
7. **Place and route**: nextpnr maps the design onto the ECP5 FPGA fabric.
8. **Program**: openFPGALoader uploads the bitstream to the ULX3S.

36.5.2 The Target Specification

Rust discovers new targets via a JSON target specification file. For the SAMPO:

Listing 36.30: Target specification (sampo-none.json)

```
{
  "llvm-target": "sampo-unknown-none",
```

```
  "data-layout": "e-m:e-p:16:16-i8:8:8-i16:16:16-n16-S16",
  "arch": "sampo",
  "target-endian": "little",
  "target-pointer-width": "16",
  "target-c-int-width": "16",
  "os": "none",
  "executables": true,
  "linker-flavor": "ld.lld",
  "linker": "rust-lld",
  "panic-strategy": "abort",
  "disable-redzone": true,
  "features": "+muldiv",
  "max-atomic-width": 16,
  "emit-debug-gdb-scripts": false
}
```

Key fields:

- `target-pointer-width: "16"` tells Rust that `usize` is 16 bits.
- `target-c-int-width: "16"` makes `c_int` 16 bits (matching the native word size).
- `panic-strategy: "abort"` avoids unwinding infrastructure (which requires OS support).
- `features: "+muldiv"` enables the multiply/divide instructions (which could be optional for smaller implementations).

36.5.3 Building with Cargo

With the target specification and backend in place, building for the SAMPO would look like:

Listing 36.31: Building a SAMPO Rust program with Cargo

```
# Build the Rust crate for the Sampo target
cargo build --target sampo-none.json --release

# Extract the binary from the ELF
rust-objcopy -O binary \
    target/sampo-none/release/my_program \
    my_program.bin

# Convert to Verilog hex format
python3 bin2hex.py my_program.bin programs/my_program.hex
```

The bin2hex.py script referenced in the Makefile converts the flat binary into a hex file that the RAM module can load via $readmemh:

Listing 36.32: Existing bin2hex flow (from verilog/Makefile)

```
# Convert binary to hex
python3 bin2hex.py $< $@
```

36.5.4 Synthesizing with the Program

The existing FPGA synthesis pipeline in the Makefile already supports specifying different programs. The RAM init file is baked into the bitstream during synthesis:

Listing 36.33: FPGA synthesis with custom program (from verilog/Makefile)

```
yosys -p "read_verilog -I$(RTL_DIR) \
    $(RTL_DIR)/alu.v $(RTL_DIR)/shifter.v \
    $(RTL_DIR)/regfile.v $(RTL_DIR)/decode.v \
    $(RTL_DIR)/cpu.v $(RTL_DIR)/ram.v \
    $(RTL_DIR)/uart.v $(RTL_DIR)/soc.v \
    $(FPGA_DIR)/uart_tx.v $(FPGA_DIR)/uart_rx.v \
    $(FPGA_DIR)/reset_sync.v $(FPGA_DIR)/ulx3s_top.v; \
    synth_ecp5 -top ulx3s_top -json $@"
```

The RAM init file path is set in the ulx3s_top.v SoC instantiation:

Listing 36.34: RAM initialization (from fpga/ulx3s_top.v)

```
soc #(
    .RESET_VECTOR(16'h0100),
    .RAM_INIT_FILE("programs/hello.hex")
) u_soc (
    .clk(clk_cpu),
    .rst_n(rst_n),
    // ...
);
```

For a Rust-compiled program, simply change the RAM_INIT_FILE parameter to point to the new hex file.

36.5.5 The Complete Makefile Target

Putting it all together, a hypothetical Makefile target for building and running Rust programs on the SAMPO FPGA would look like:

Listing 36.35: Complete Rust-to-FPGA pipeline

```
# Build Rust program, convert to hex, synthesize, and program
cargo build --target sampo-none.json --release
rust-objcopy -O binary \
    target/sampo-none/release/my_program \
    my_program.bin
python3 bin2hex.py my_program.bin programs/my_program.hex
make fpga RAM_INIT_FILE=programs/my_program.hex
make prog
```

Five commands, from Rust source to running hardware. The entire process takes perhaps two minutes on a modern machine—about 30 seconds for Rust compilation, 60 seconds for FPGA synthesis, and 10 seconds for programming.

36.5.6 What About the Assembler?

An observant reader might wonder: where does the existing SAMPO assembler (the `sasm` tool built in Part III) fit into this picture? It serves two roles:

1. **Startup code**: The `crt0.s` runtime initialization code is still written in assembly and assembled with `sasm`. It is linked with the Rust object files.

2. **Standalone programs**: For programs written entirely in assembly (like the test suite programs in Part IX), the assembler remains the primary tool:

Listing 36.36: Assembling and running a test (from `verilog/Makefile`)

```
cd ../sasm && cargo run --release -- $< -o $@
```

The LLVM backend's assembly output format would be compatible with `sasm`, using the same mnemonics and register names defined in the reference manual. This means hand-written assembly and compiler-generated assembly can coexist in the same project.

36.5.7 Practical Limitations

Let us be honest about what a SAMPO compiler target can and cannot handle:

Table 36.3: What works and what does not on a 16-bit Rust target

Works	Does Not Work (or is impractical)
`core` types: `u8`, `u16`, `i8`, `i16`, `bool`	`std` library (no OS)
Fixed-size arrays	Dynamic allocation (`Vec`, `String`, `Box`) without a custom allocator
`u32`/`i32` (via expansion)	`u64`/`i64` (very expensive: 4-word operations)
Iterators, closures, traits	Floating point (no FPU; soft-float is large)
`enum`, `struct`, pattern matching	Any crate that depends on `std`
Inline assembly for I/O	Threads, async, networking
`panic = "abort"`	Stack unwinding

64 KB Is a Hard Limit

The SAMPO's 16-bit address space means that code, data, and stack must *all* fit within 64 KB. In practice, after reserving space for the interrupt vector table, stack, and RAM, the program has roughly 32 KB for code and 32 KB for data. This is enough for meaningful embedded applications—sensor data collection, protocol parsing, motor control—but not for general-purpose computing.

36.5.8 Alternative Approaches

Building a full LLVM backend is a significant effort—perhaps 10,000 to 20,000 lines of TableGen and C++ for a clean target like the SAMPO. Two alternative approaches are worth considering:

1. **Cranelift backend**: Cranelift is Rust's alternative code generator, simpler than LLVM and designed for fast compilation. It has a cleaner backend interface and may be easier to target for a new architecture.

2. **Custom C compiler**: For a 16-bit target, a custom compiler for a subset of C (or a C-like language) may be more practical than a full LLVM backend. Projects like `chibicc` (a small C compiler in ~10,000 lines) can be retargeted to new architectures relatively quickly.

3. **Transpilation**: Write programs in a higher-level language and transpile to SAMPO assembly. This avoids the complexity of a full compiler backend at the cost of optimization quality.

Each approach involves tradeoffs between effort, optimization quality, and language support. For an educational project, the transpilation approach or a custom C compiler may provide the best return on investment. For maximum compatibility with the Rust ecosystem, the LLVM backend is the way forward.

Exercises

1. **Hand-Compile a Function.** Take the following Rust function and hand-compile it to SAMPO assembly, following the calling convention from `ARCHITECTURE.md`:

```rust
fn sum_array(arr: &[u16]) -> u16 {
    let mut total: u16 = 0;
    for &val in arr {
        total = total.wrapping_add(val);
    }
    total
}
```

   Assume the array pointer is passed in `R4` and the length in `R5`. Verify your assembly by running it in the emulator.

2. **32-Bit Arithmetic.** Write SAMPO assembly routines for 32-bit addition, subtraction, and comparison using pairs of 16-bit registers. Test with values that exercise the carry chain, including cases where the carry propagates from the low word to the high word.

3. **Study the MSP430 Backend.** Examine the LLVM MSP430 backend source code (in `llvm/lib/Target/MSP430/`). List three design decisions that differ from what a SAMPO backend would need, and explain why. Pay special attention to how MSP430 handles its 16-bit pointer type and its limited addressing modes.

4. **Linker Script.** Modify the linker script from Section 36.4.3 to reserve 256 bytes of RAM at `0xFF00` for memory-mapped I/O. Add a `.mmio` section that places peripheral registers at fixed addresses. How does this interact with the stack pointer initialization?

5. **Constant Materialization.** Write a SAMPO assembly subroutine that loads an arbitrary 16-bit value into a register using the smallest possible encoding. For values 0 through 127, use a 2-instruction sequence (**`ADD`** + **`ADDI`**). For values -128 through -1, use **`ADDI`** alone (since `R0` + negative imm8 works). For all other values, use **`LIX`**. Measure the code size savings compared to always using **`LIX`**.

What's Next

With this chapter, we have reached the edge of the SAMPO project as it stands today. We started in Chapter 1 by asking "Why build a CPU?" and have traced the entire path: from architecture design through assembly programming, from Verilog RTL through FPGA bitstreams, from testbench simulation through 132 passing hardware tests.

The SAMPO is a complete, working 16-bit RISC processor. It runs on real hardware. Its instruction set is documented, its assembler produces correct binaries, and its emulator provides a full debugging environment. The path to a compiler target is mapped out, waiting for an adventurous reader to walk it.

What you build next is up to you. Perhaps an operating system kernel. Perhaps a BASIC interpreter. Perhaps a hardware peripheral that makes the SAMPO blink LEDs to the beat of a MIDI stream. Whatever it is, you now have the tools and the understanding to build it. The SAMPO—that magical artifact from Finnish mythology—is yours.

Appendix A

ISA Quick Reference

This appendix is a dense, scannable reference to the complete SAMPO instruction set architecture. It is designed to be used alongside an editor or debugger, not read linearly. Every instruction, register, flag behaviour, branch condition, and memory-map region documented in this book is summarised here.

For detailed encoding bit fields, see Chapter B.

A.1 Instruction Summary

The SAMPO ISA comprises approximately 80 instructions grouped into 12 categories. Standard instructions are 16 bits (2 bytes); extended instructions (opcode `0xF`) are 32 bits (4 bytes). The following tables list every instruction with its mnemonic, operands, operation, size, and flag effects.

Flag effects legend: * = affected based on result; `0` = cleared; `1` = set; `-` = unchanged; ~ = complemented.

A.1.1 Arithmetic Instructions

Table A.1: Arithmetic instructions

Mnemonic	Operands	Operation	Bytes	Flags
ADD	Rd, Rs1, Rs2	Rd = Rs1 + Rs2	2	N*Z*C*V*H*
SUB	Rd, Rs1, Rs2	Rd = Rs1 − Rs2	2	N*Z*C*V*H*
ADC	Rd, Rs1, Rs2	Rd = Rs1 + Rs2 + C	2	N*Z*C*V*H*
SBC	Rd, Rs1, Rs2	Rd = Rs1 − Rs2 − C	2	N*Z*C*V*H*
NEG	Rd, Rs1	Rd = −Rs1	2	N*Z*C*V*H*

Mnemonic	Operands	Operation	Bytes	Flags
`ADDI`	Rd, imm8	Rd = Rd + sext(imm8)	2	N*Z*C*V*H*
`CMP`	Rd, Rs1	flags = Rd − Rs1	2	N*Z*C*V*H*
`TEST`	Rd, Rs1	flags = Rd AND Rs1	2	N*Z*C0V0H0

A.1.2 Logic Instructions

Table A.2: Logic instructions

Mnemonic	Operands	Operation	Bytes	Flags
`AND`	Rd, Rs1, Rs2	Rd = Rs1 AND Rs2	2	N*Z*C0V0H0
`OR`	Rd, Rs1, Rs2	Rd = Rs1 OR Rs2	2	N*Z*C0V0H0
`XOR`	Rd, Rs1, Rs2	Rd = Rs1 XOR Rs2	2	N*Z*C0V0H0
`NOT`	Rd, Rs1	Rd = NOT Rs1	2	N*Z*C0V0H0

A.1.3 Multiply/Divide Instructions

Table A.3: Multiply/divide instructions (opcode `0xB`)

Mnemonic	Operands	Operation	Bytes	Flags
`MUL`	Rd, Rs1	Rd = (Rd × Rs1)[15:0]	2	N*Z*C-V-H-
`MULH`	Rd, Rs1	Rd = (Rd × Rs1)[31:16] signed	2	N*Z*C-V-H-
`MULHU`	Rd, Rs1	Rd = (Rd × Rs1)[31:16] unsigned	2	N*Z*C-V-H-
`DIV`	Rd, Rs1	Rd = Rd ÷ Rs1 (signed)	2	N*Z*C-V-H-
`DIVU`	Rd, Rs1	Rd = Rd ÷ Rs1 (unsigned)	2	N*Z*C-V-H-
`REM`	Rd, Rs1	Rd = Rd mod Rs1 (signed)	2	N*Z*C-V-H-
`REMU`	Rd, Rs1	Rd = Rd mod Rs1 (unsigned)	2	N*Z*C-V-H-
`DAA`	Rd	BCD adjust Rd	2	N*Z*C*V-H*

A.1.4 Shift and Rotate Instructions

All shift/rotate instructions use opcode `0xA` with the `func` field selecting the operation. The carry flag receives the last bit shifted out.

Table A.4: Shift and rotate instructions (opcode `0xA`)

Mnemonic	Operands	Operation	Bytes	Flags
1-bit operations				
SLL	Rd, Rs1	Rd = Rs1 ≪ 1	2	N*Z*C*VoHo
SRL	Rd, Rs1	Rd = Rs1 ≫ 1 (logical)	2	N*Z*C*VoHo
SRA	Rd, Rs1	Rd = Rs1 ≫ 1 (arithmetic)	2	N*Z*C*VoHo
ROL	Rd, Rs1	Rotate left by 1	2	N*Z*C*VoHo
ROR	Rd, Rs1	Rotate right by 1	2	N*Z*C*VoHo
RCL	Rd, Rs1	Rotate left through C by 1	2	N*Z*C*VoHo
RCR	Rd, Rs1	Rotate right through C by 1	2	N*Z*C*VoHo
SWAP	Rd, Rs1	Swap high/low bytes	2	N*Z*C-VoHo
4-bit operations				
SLL4	Rd, Rs1	Rd = Rs1 ≪ 4	2	N*Z*C*VoHo
SRL4	Rd, Rs1	Rd = Rs1 ≫ 4 (logical)	2	N*Z*C*VoHo
SRA4	Rd, Rs1	Rd = Rs1 ≫ 4 (arithmetic)	2	N*Z*C*VoHo
ROL4	Rd, Rs1	Rotate left by 4	2	N*Z*C*VoHo
8-bit operations				
SLL8	Rd, Rs1	Rd = Rs1 ≪ 8	2	N*Z*C*VoHo
SRL8	Rd, Rs1	Rd = Rs1 ≫ 8 (logical)	2	N*Z*C*VoHo
SRA8	Rd, Rs1	Rd = Rs1 ≫ 8 (arithmetic)	2	N*Z*C*VoHo
ROL8	Rd, Rs1	Rotate left by 8	2	N*Z*C*VoHo

A.1.5 Load Instructions

Table A.5: Load instructions (opcode `0x6`)

Mnemonic	Operands	Operation	Bytes	Flags
LW	Rd, (Rs1)	Rd = mem16[Rs1]	2	—–
LB	Rd, (Rs1)	Rd = sext(mem8[Rs1])	2	—–
LBU	Rd, (Rs1)	Rd = zext(mem8[Rs1])	2	—–
LW	Rd, 2(Rs1)	Rd = mem16[Rs1 + 2]	2	—–
LW	Rd, 4(Rs1)	Rd = mem16[Rs1 + 4]	2	—–
LW	Rd, 6(Rs1)	Rd = mem16[Rs1 + 6]	2	—–
LW	Rd, −2(Rs1)	Rd = mem16[Rs1 − 2]	2	—–
LW	Rd, −4(Rs1)	Rd = mem16[Rs1 − 4]	2	—–
LUI	Rd, Rs1	Rd = Rs1 ≪ 8	2	—–

A.1.6 Store Instructions

Table A.6: Store instructions (opcode 0x7)

Mnemonic	Operands	Operation	Bytes	Flags
SW	(Rs1), Rs2	mem16[Rs1] = Rs2	2	——–
SB	(Rs1), Rs2	mem8[Rs1] = Rs2[7:0]	2	——–
SW	2(Rs1), Rs2	mem16[Rs1 + 2] = Rs2	2	——–
SW	4(Rs1), Rs2	mem16[Rs1 + 4] = Rs2	2	——–
SW	6(Rs1), Rs2	mem16[Rs1 + 6] = Rs2	2	——–
SW	−2(Rs1), Rs2	mem16[Rs1 − 2] = Rs2	2	——–
SW	−4(Rs1), Rs2	mem16[Rs1 − 4] = Rs2	2	——–

A.1.7 Branch Instructions

All branches use opcode 0x8, are 2 bytes, and leave flags unchanged. Target: PC = PC + 2 + sign_extend(offset8) × 2. Range: −256 to +254 bytes from the branch instruction.

Table A.7: Branch instructions (opcode 0x8)

Mnemonic	Condition	After CMP A,B	Type
BEQ	Z = 1	A = B	Either
BNE	Z = 0	A ≠ B	Either
BLT	N ≠ V	A < B	Signed
BGE	N = V	A ≥ B	Signed
BLTU	C = 0	A < B	Unsigned
BGEU	C = 1	A ≥ B	Unsigned
BMI	N = 1	Negative	—
BPL	N = 0	Positive/zero	—
BVS	V = 1	Overflow	—
BVC	V = 0	No overflow	—
BCS	C = 1	Carry set	—
BCC	C = 0	Carry clear	—
BGT	Z = 0 and N = V	A > B	Signed
BLE	Z = 1 or N ≠ V	A ≤ B	Signed
BHI	C = 1 and Z = 0	A > B	Unsigned
BLS	C = 0 or Z = 1	A ≤ B	Unsigned

A.1.8 Jump Instructions

Table A.8: Jump instructions (opcode 0x9)

Mnemonic	Operands	Operation	Bytes	Flags
J	offset12	PC = PC + 2 + sext(off12)×2	2	——–
JR	Rs1	PC = Rs1	2	——–
JALR	Rd, Rs1	Rd = PC + 2; PC = Rs1	2	——–

J range: −4096 to +4094 bytes from the jump instruction.

A.1.9 Stack Instructions

Table A.9: Stack instructions (opcode 0xC)

Mnemonic	Operands	Operation	Bytes	Flags
PUSH	Rs1	SP = SP − 2; mem[SP] = Rs1	2	——–
POP	Rd	Rd = mem[SP]; SP = SP + 2	2	——–

A.1.10 Block Instructions

Block operations use fixed register conventions: R4 = count, R5 = source address, R6 = destination address.

Table A.10: Block instructions (opcode 0xC)

Mnemonic	Operation	Bytes	Flags
LDI	mem[R6]++ = mem[R5]++; R4−−; Z=(R4==0)	2	–Z–
LDD	mem[R6]−− = mem[R5]−−; R4−−; Z=(R4==0)	2	–Z–
LDIR	Repeat LDI until R4 = 0	2	–Z–
LDDR	Repeat LDD until R4 = 0	2	–Z–
CPIR	Compare A with mem[R5]++; repeat until match	2	–Z–
FILL	Fill R4 bytes at R6 with value from R5	2	–Z–

A.1.11 Miscellaneous Instructions

Table A.11: Miscellaneous instructions (opcode 0xC)

Mnemonic	Operands	Operation	Bytes	Flags
MOV	Rd, Rs	Rd = Rs	2	——–

Mnemonic	Operands	Operation	Bytes	Flags
`EXX`	—	Swap R4–R11 ↔ R4'–R11'	2	——–
`GETF`	Rd	Rd = FLAGS	2	——–
`SETF`	Rs	FLAGS = Rs[7:0]	2	All loaded

A.1.12 I/O Instructions

Table A.12: I/O instructions (opcode `0xD`)

Mnemonic	Operands	Operation	Bytes	Flags
`INI`	Rd, port4	Rd = port_read(port4)	2	——–
`OUTI`	port4, Rs	port_write(port4, Rs)	2	——–
`IN`	Rd, (Rs1)	Rd = port_read(Rs1[7:0])	2	——–
`OUT`	(Rs1), Rd	port_write(Rs1[7:0], Rd)	2	——–

A.1.13 System Instructions

Table A.13: System instructions (opcode `0xE`)

Mnemonic	Operands	Operation	Bytes	Flags
`NOP`	—	No operation	2	——–
`HALT`	—	Halt; wait for interrupt	2	——–
`DI`	—	Clear I flag (disable interrupts)	2	–I=0–
`EI`	—	Set I flag (enable interrupts)	2	–I=1–
`RETI`	—	Return from interrupt; set I	2	–I=1–
`SWI`	imm8	Software interrupt (trap to imm8)	2	——–
`SCF`	—	Set carry flag	2	–C1–
`CCF`	—	Complement carry flag	2	–C~–

A.1.14 Extended Instructions (32-bit)

Extended instructions use opcode `0xF`. The first word contains the opcode, register fields, and sub-opcode; the second word is a 16-bit immediate.

Table A.14: Extended instructions (opcode `0xF`, 32-bit)

Mnemonic	Operands	Operation	Bytes	Flags
`ADDIX`	Rd, Rs, imm16	Rd = Rs + imm16	4	N*Z*C*V*H*

Mnemonic	Operands	Operation	Bytes	Flags
SUBIX	Rd, Rs, imm16	Rd = Rs − imm16	4	N*Z*C*V*H*
ANDIX	Rd, Rs, imm16	Rd = Rs AND imm16	4	N*Z*CoVoHo
ORIX	Rd, Rs, imm16	Rd = Rs OR imm16	4	N*Z*CoVoHo
XORIX	Rd, Rs, imm16	Rd = Rs XOR imm16	4	N*Z*CoVoHo
LWX	Rd, imm16(Rs)	Rd = mem16[Rs + imm16]	4	—–
SWX	imm16(Rs), Rd	mem16[Rs + imm16] = Rd	4	—–
LIX	Rd, imm16	Rd = imm16	4	—–
JX	addr16	PC = addr16	4	—–
JALX	addr16	RA = PC + 4; PC = addr16	4	—–
CMPIX	Rd, imm16	flags = Rd − imm16	4	N*Z*C*V*H*
INX	Rd, port8	Rd = port_read(port8)	4	—–
OUTX	port8, Rs	port_write(port8, Rs)	4	—–
SLLX	Rd, Rs, imm4	Rd = Rs ≪ imm4	4	N*Z*C*VoHo
SRLX	Rd, Rs, imm4	Rd = Rs ≫ imm4 (logical)	4	N*Z*C*VoHo
SRAX	Rd, Rs, imm4	Rd = Rs ≫ imm4 (arithmetic)	4	N*Z*C*VoHo

A.2 Register Conventions

A.2.1 General-Purpose Registers

The SAMPO has 16 general-purpose 16-bit registers. All registers are orthogonal: any register may appear as source or destination of any instruction.

A.2.2 Special Registers

A.2.3 Alternate Register Set

Registers `R4`–`R11` have shadow copies (`R4'`–`R11'`) that are exchanged atomically by the **`EXX`** instruction. Registers `R0`–`R3` (special) and `R12`–`R15` (callee-saved) are *not* affected by **`EXX`**. The alternate set is typically reserved for interrupt handlers to avoid register saves/restores.

A.2.4 Block Operation Register Conventions

Block instructions (**`LDI`**, **`LDD`**, **`LDIR`**, **`LDDR`**, **`CPIR`**, **`FILL`**) use hardwired register assignments:

A.3 Flag Effects per Instruction

A.3.1 FLAGS Register Layout

The FLAGS register is 8 bits wide, laid out as follows:

Table A.15: General-purpose register conventions

Register	Alias	Convention	Saved By	Alternate?
R0	ZERO	Always reads as 0; writes ignored	—	No
R1	RA	Return address	Caller	No
R2	SP	Stack pointer	Callee	No
R3	GP	Global pointer (optional)	—	No
R4	A0	Argument 0 / Return value 0	Caller	Yes
R5	A1	Argument 1 / Return value 1	Caller	Yes
R6	A2	Argument 2	Caller	Yes
R7	A3	Argument 3	Caller	Yes
R8	T0	Temporary 0	Caller	Yes
R9	T1	Temporary 1	Caller	Yes
R10	T2	Temporary 2	Caller	Yes
R11	T3	Temporary 3	Caller	Yes
R12	S0	Saved register 0	Callee	No
R13	S1	Saved register 1	Callee	No
R14	S2	Saved register 2	Callee	No
R15	S3	Saved register 3	Callee	No

Table A.16: Special registers

Register	Width	Description
PC	16-bit	Program counter. Not directly accessible; modified by branches, jumps, and sequential execution.
FLAGS	8-bit	Status flags register. Read via **GETF**, written via **SETF**. See Section A.3.
IVR	16-bit	Interrupt vector register. Holds the base address for interrupt dispatch.

Table A.17: Block operation register conventions

Register	Alias	Role
R4	A0	Byte count (decremented by each iteration)
R5	A1	Source address (incremented or decremented)
R6	A2	Destination address (incremented or decremented)

Table A.18: FLAGS register bit assignments

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
N	Z	C	V	H	I	*rsvd*	*rsvd*

N — Negative (bit 7)
Set if bit 15 of the result is 1 (two's complement negative).

Z — Zero (bit 6)
Set if the result is exactly zero (all 16 bits are 0).

C — Carry (bit 5)
Set on unsigned carry out (addition) or *no borrow* (subtraction, ARM convention).

V — Overflow (bit 4)
Set if signed overflow occurred (result cannot be represented in 16-bit two's complement).

H — Half-carry (bit 3)
Set on carry from bit 3 to bit 4. Used by **DAA** for BCD correction.

I — Interrupt enable (bit 2)
When set, maskable interrupts are accepted. Cleared on interrupt entry; set by **EI** and **RETI**.

A.3.2 Flag Effect Summary

In the table below, * = affected based on result, 0 = cleared, 1 = set, - = unchanged, ~ = complemented.

SUB Carry Convention (ARM-style)

The SAMPO uses ARM-style carry for subtraction: after **SUB** (or **CMP**), C = 1 means *no borrow occurred*, i.e., $a \geq b$ for unsigned operands. Conversely, C = 0 means a borrow occurred, meaning $a < b$ unsigned. This is the opposite of the Z80 convention but matches ARM and Cortex-M processors. The unsigned branch conditions **BLTU** (C=0) and **BGEU** (C=1) are defined accordingly.

A.4 Branch Condition Truth Table

The 4-bit condition code in bits [11:8] of each branch instruction selects one of 16 flag tests. The following table provides the complete truth table.

Table A.19: Flag effects by instruction class

Instruction Class	N	Z	C	V	H
ADD, ADDI, ADC, ADDIX	*	*	*	*	*
SUB, SBC, CMP, NEG, SUBIX, CMPIX	*	*	*	*	*
AND, OR, XOR, NOT, TEST, ANDIX, ORIX, XORIX	*	*	0	0	0
Shift/Rotate (opcode `0xA`), SLLX, SRLX, SRAX	*	*	*	0	0
MUL, MULH, MULHU, DIV, DIVU, REM, REMU	*	*	-	-	-
DAA	*	*	*	-	*
Load/Store (LW, LB, LBU, SW, SB, LWX, SWX, LIX, LUI)	-	-	-	-	-
Branch/Jump (B*cc*, J, JR, JALR, JX, JALX)	-	-	-	-	-
PUSH, POP	-	-	-	-	-
Block ops (LDI, LDD, LDIR, LDDR, CPIR, FILL)	-	*	-	-	-
I/O (IN, OUT, INI, OUTI, INX, OUTX)	-	-	-	-	-
MOV, EXX, GETF	-	-	-	-	-
NOP, HALT, EI, DI, RETI, SWI	-	-	-	-	-
SCF	-	-	1	-	-
CCF	-	-	~	-	-
SETF	All flags loaded from Rs[7:0]				

Table A.20: Branch condition code truth table

Code	Mnemonic	Flag Condition	After CMP A,B	Signed/Unsigned
0x0	**BEQ**	Z = 1	A = B	Either
0x1	**BNE**	Z = 0	A ≠ B	Either
0x2	**BLT**	N ≠ V	A < B	Signed
0x3	**BGE**	N = V	A ≥ B	Signed
0x4	**BLTU**	C = 0	A < B	Unsigned
0x5	**BGEU**	C = 1	A ≥ B	Unsigned
0x6	**BMI**	N = 1	Result negative	—
0x7	**BPL**	N = 0	Result ≥ 0	—
0x8	**BVS**	V = 1	Signed overflow	—
0x9	**BVC**	V = 0	No signed overflow	—
0xA	**BCS**	C = 1	Carry set	—
0xB	**BCC**	C = 0	Carry clear	—
0xC	**BGT**	Z = 0 AND N = V	A > B	Signed
0xD	**BLE**	Z = 1 OR N ≠ V	A ≤ B	Signed
0xE	**BHI**	C = 1 AND Z = 0	A > B	Unsigned
0xF	**BLS**	C = 0 OR Z = 1	A ≤ B	Unsigned

Unsigned Branches Use ARM-Style Carry

Because SAMPO uses ARM-style carry, **BLTU** tests C = 0 (not C = 1). After `CMP A, B` (which computes $A - B$), C = 0 means a borrow occurred, which means $A < B$ unsigned. This is the inverse of the Z80 convention. Similarly, **BGEU** tests C = 1 (no borrow), meaning $A \geq B$ unsigned.

The relationship between **BCS**/**BCC** and **BGEU**/**BLTU** is:

- **BCS** ≡ **BGEU** (both test C = 1)
- **BCC** ≡ **BLTU** (both test C = 0)

A.4.1 Branch Offset Encoding

Branch instructions encode an 8-bit signed offset in bits [7:0]. The target address is computed as:

$$\text{PC}_{\text{target}} = \text{PC} + 2 + \text{sign_extend}(\text{offset8}) \times 2$$

This yields a range of −256 to +254 bytes relative to the branch instruction. The factor of 2 accounts for the 16-bit instruction alignment. For longer branches, use a jump (**J**) or the extended **JX**.

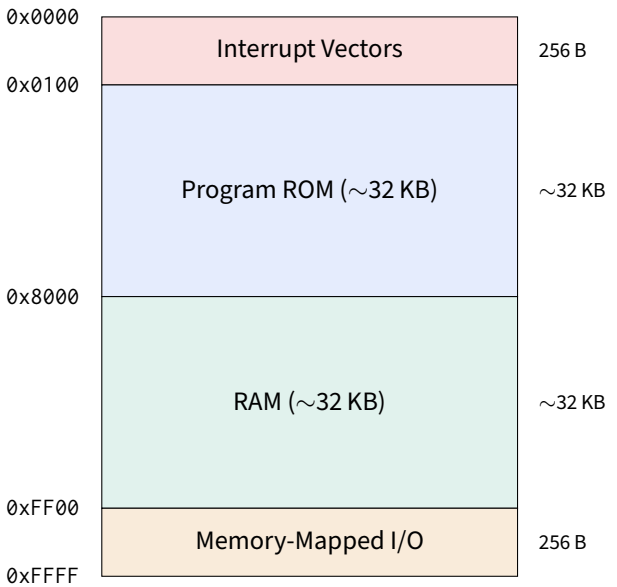
A.5 Memory Map

The SAMPO has a 64 KB address space (16-bit addresses), byte-addressable, with little-endian byte order.

A.5.1 Address Space Layout

Table A.21: Suggested memory map

Start	End	Size	Region
`0x0000`	`0x00FF`	256 B	Interrupt vectors / Reset entry point
`0x0100`	`0x7FFF`	~32 KB	Program ROM
`0x8000`	`0xFEFF`	~32 KB	RAM
`0xFF00`	`0xFFFF`	256 B	Memory-mapped I/O



A.5.2 Memory Properties

A.5.3 Power-On Defaults

A.5.4 I/O Port Space

SAMPO provides a separate 256-port I/O address space (8-bit port numbers, `0x00`–`0xFF`), accessed via the **`IN`**/**`OUT`** instruction families. This is independent of the 64 KB memory address space.

Table A.22: Memory system properties

Property	Value
Address space	64 KB (16-bit addresses)
Byte order	Little-endian
Addressability	Byte-addressable
Word size	16 bits (2 bytes)
Word alignment	Unaligned access supported
Stack growth	Downward (toward lower addresses)

Table A.23: Processor state after reset

Register/State	Default Value
PC	`0x0100`
SP (R2)	`0xFFFF`
FLAGS	`0x00` (all flags cleared)
I flag	0 (interrupts disabled)
All other registers	Undefined
Alternate registers	Undefined

I/O Model

SAMPO supports **port-based I/O** (Z80-style) with separate **`IN`/`OUT`** instructions and an 8-bit port address bus, plus the option for **memory-mapped I/O** in the `0xFF00`–`0xFFFF` range. The port-based approach is the primary I/O mechanism and is used by all standard peripherals. Memory-mapped I/O is available as an alternative for devices that benefit from load/store access patterns.

A.5.5 UART Port Assignments

The default SoC includes an MC6850-compatible UART on two I/O ports:

Typical usage: poll `0x80` bit 1 (TDRE) before writing to `0x81`; poll `0x80` bit 0 (RDRF) before reading from `0x81`.

A.6 Opcode Map

The primary opcode occupies bits [15:12] of every instruction, giving 16 categories:

Table A.24: UART port assignments

Port	Register	Description
0x80	Status	UART status register. Bit 0 = RDRF (Receive Data Register Full); Bit 1 = TDRE (Transmit Data Register Empty).
0x81	Data	UART data register. Read returns received byte (RX); write transmits byte (TX).

Table A.25: Primary opcode map

Binary	Hex	Category	Description
0000	0x0	ADD	Rd = Rs1 + Rs2 (also ADC, NEG via func)
0001	0x1	SUB	Rd = Rs1 − Rs2 (also SBC via func)
0010	0x2	AND	Rd = Rs1 AND Rs2
0011	0x3	OR	Rd = Rs1 OR Rs2
0100	0x4	XOR	Rd = Rs1 XOR Rs2 (also NOT via func)
0101	0x5	ADDI	Rd = Rd + sext(imm8)
0110	0x6	LOAD	LW, LB, LBU, LUI
0111	0x7	STORE	SW, SB
1000	0x8	BRANCH	16 conditional branches
1001	0x9	JUMP	J, JR, JALR
1010	0xA	SHIFT	16 shift/rotate variants
1011	0xB	MULDIV	MUL, MULH, MULHU, DIV, DIVU, REM, REMU, DAA
1100	0xC	MISC	Stack, block, MOV, CMP, TEST, EXX, GETF, SETF
1101	0xD	I/O	INI, OUTI, IN, OUT
1110	0xE	SYSTEM	NOP, HALT, DI, EI, RETI, SWI, SCF, CCF
1111	0xF	EXTENDED	32-bit instructions (16 sub-opcodes)

A.7 Instruction Format Summary

SAMPO uses six instruction formats. All standard instructions are 16 bits; extended instructions are 32 bits (two 16-bit words).

Table A.26: Instruction format overview

Format	Usage	[15:12]	[11:8]	[7:4]	[3:0]
R	Register-register	opcode	Rd	Rs1	Rs2/func
I	Immediate	opcode	Rd	imm8	
S	Store	opcode	imm4	Rs1	Rs2
B	Branch	opcode	cond	offset8	
J	Jump	opcode	offset12		
X	Extended (32-bit)	`0xF`	Rd	Rs/func	sub + *imm16 word*

A.8 Quick Reference: At a Glance

A.8.1 Instruction Categories

Table A.27: All instructions by category

Category	Instructions
ALU	**ADD SUB AND OR XOR ADDI ADC SBC NEG NOT CMP TEST**
Mul/Div	**MUL MULH MULHU DIV DIVU REM REMU DAA**
Shift	**SLL SRL SRA ROL ROR RCL RCR SWAP** (by 1, 4, or 8)
Load	**LW LB LBU LUI LWX LIX**
Store	**SW SB SWX**
Branch	**BEQ BNE BLT BGE BLTU BGEU BMI BPL BVS BVC BCS BCC BGT BLE BHI BLS**
Jump	**J JR JALR JX JALX**
Stack	**PUSH POP**
Block	**LDI LDD LDIR LDDR CPIR FILL**
I/O	**IN INI OUT OUTI INX OUTX**
System	**NOP HALT DI EI RETI SWI SCF CCF**
Misc	**MOV EXX GETF SETF**
Extended	**ADDIX SUBIX ANDIX ORIX XORIX CMPIX SLLX SRLX SRAX**

A.8.2 Register Aliases

A.8.3 Key Constants

Table A.28: Register alias quick reference

`R0=ZERO`	`R4=A0`	`R8=T0`	`R12=S0`
`R1=RA`	`R5=A1`	`R9=T1`	`R13=S1`
`R2=SP`	`R6=A2`	`R10=T2`	`R14=S2`
`R3=GP`	`R7=A3`	`R11=T3`	`R15=S3`

Table A.29: Key architectural constants

Parameter	**Value**
Word size	16 bits
Number of GPRs	16
Address space	64 KB
Instruction width	16 or 32 bits
Primary opcodes	16
Branch conditions	16
Shift/rotate variants	16
Mul/div operations	8
I/O port space	256 ports (8-bit address)
Reset vector	`0x0100`
Interrupt vector	`0x0004`
`ADDI` range	-128 to $+127$
Branch offset range	-256 to $+254$ bytes
Jump offset range	-4096 to $+4094$ bytes

Appendix B

Instruction Encoding

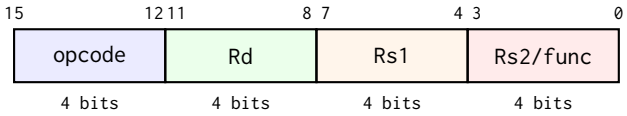
This appendix is the definitive bit-level reference for every SAMPO instruction encoding. Where Chapter 2 introduced the instruction formats and **??** explored the decoder logic, this appendix lays out every field of every instruction with its exact binary encoding, ready for hand-assembly, disassembler implementation, or late-night debugging sessions with a hex dump.

All SAMPO instructions are either 16 bits (one word) or 32 bits (two words). The primary opcode occupies bits 15:12 of the first word. If the opcode is `0xF`, the instruction is 32 bits and the second word carries a 16-bit immediate. All multi-byte values use little-endian byte order: the least significant byte is stored at the lower address.

B.1 Instruction Formats

The SAMPO ISA uses six instruction formats. Each format partitions the 16-bit instruction word differently, but the opcode always occupies bits 15:12. This section provides TikZ bit-field diagrams for all six.

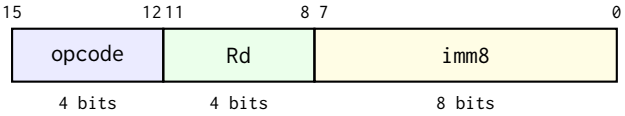
B.1.1 Format R: Register–Register



Width: 16 bits (2 bytes).
Used by: `ADD`, `SUB`, `AND`, `OR`, `XOR`, `Shift/Rotate`, `Multiply/Divide`, `Misc`, `Load`, `I/O`.

The bottom 4 bits serve as either a third source register (Rs2) for three-operand instructions or a function code that selects the specific operation within an opcode group. For example, opcode 0xA uses the func field to select among 16 shift and rotate variants.

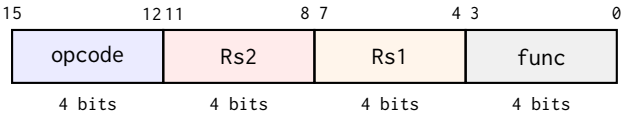
B.1.2 Format I: Immediate



Width: 16 bits (2 bytes).
Used by: **ADDI**.

The 8-bit immediate imm8 is sign-extended to 16 bits before use, giving a range of -128 to $+127$. **ADDI** is a two-operand instruction: Rd = Rd + sign_extend(imm8). The register in bits 11:8 is both source and destination.

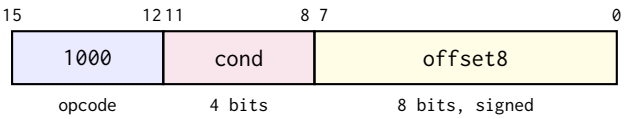
B.1.3 Format S: Store



Width: 16 bits (2 bytes).
Used by: **SW**, **SB**.

Store instructions reinterpret the "Rd" field (bits 11:8) as Rs2, the register whose value is being stored. Rs1 (bits 7:4) provides the base address. The func field (bits 3:0) selects the offset and access width. This reuse of the R-format bit positions keeps the decoder logic uniform.

B.1.4 Format B: Branch



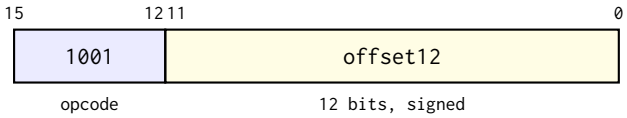
Width: 16 bits (2 bytes). **Opcode**: 0x8 (fixed).
Used by: All 16 conditional branch instructions.

Target calculation:

$$\text{PC}_{\text{target}} = \text{PC} + 2 + \text{sign_extend}(\text{offset8}) \times 2$$

The offset is multiplied by 2 to produce word-aligned targets, giving an effective range of -256 to $+254$ bytes relative to the instruction following the branch. The cond field (bits 11:8) selects one of 16 branch conditions.

B.1.5 Format J: Jump



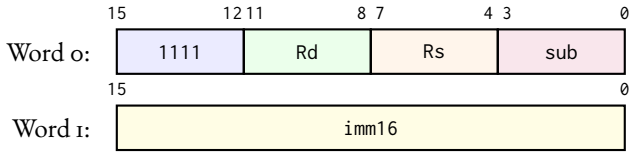
Width: 16 bits (2 bytes). **Opcode**: `0x9` (fixed).
Used by: **`J`** (PC-relative jump).

Target calculation:

$$\text{PC}_{\text{target}} = \text{PC} + 2 + \text{sign_extend}(\text{offset12}) \times 2$$

The 12-bit signed offset, multiplied by 2, gives a range of -4096 to $+4094$ bytes. The register-indirect forms (**`JR`**, **`JALR`**) share opcode `0x9` but use a different bit pattern in the lower 12 bits to distinguish them from the offset form (see Section B.6.2).

B.1.6 Format X: Extended (32-bit)



Width: 32 bits (4 bytes). **Opcode**: `0xF` (fixed prefix).
Used by: **`ADDIX`**, **`SUBIX`**, **`ANDIX`**, **`ORIX`**, **`XORIX`**, **`LWX`**, **`SWX`**, **`LIX`**, **`JX`**, **`JALX`**, **`CMPIX`**, **`INX`**, **`OUTX`**, **`SLLX`**, **`SRLX`**, **`SRAX`**.

The `0xF` prefix in bits 15:12 of the first word signals the CPU to fetch a second word. The sub field (bits 3:0 of the first word) selects among 16 extended operations. The second word provides a full 16-bit immediate value. In the Verilog state machine, this triggers the FETCH_EXT state to read the second word before proceeding to EXECUTE.

Detecting 32-bit Instructions

Instruction length detection is a single-nibble comparison. In the decoder, implemented in `verilog/rtl/decode.v`:

```
is_extended = (opcode == 4'hF)
```

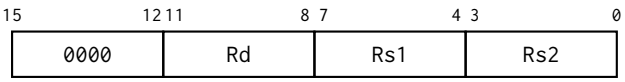
No multi-byte prefix decoding, no variable-length complexity. One nibble, one comparison, one answer. This is a major advantage of the SAMPO encoding over architectures like x86 or Z80, where instruction length can require examining multiple prefix bytes.

B.2 R-Format Instructions

R-format instructions use opcode-specific semantics for the bottom 4 bits. The following subsections detail every R-format opcode.

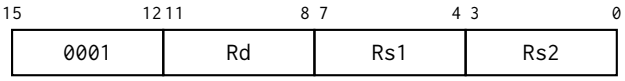
B.2.1 Opcode 0x0: ADD

Operation: Rd = Rs1 + Rs2. **Flags**: N, Z, C, V.



B.2.2 Opcode 0x1: SUB

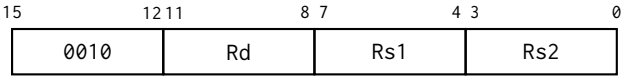
Operation: Rd = Rs1 − Rs2. **Flags**: N, Z, C, V.



The carry flag uses ARM-style inverted borrow: C=1 when no borrow occurs (i.e., Rs1 ≥ Rs2 for unsigned values).

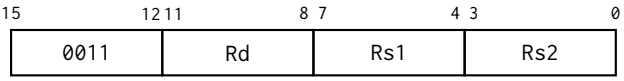
B.2.3 Opcode 0x2: AND

Operation: Rd = Rs1 & Rs2. **Flags**: N, Z (C and V cleared).



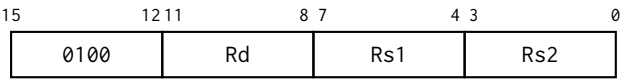
B.2.4 Opcode 0x3: OR

Operation: Rd = Rs1 | Rs2. **Flags**: N, Z (C and V cleared).



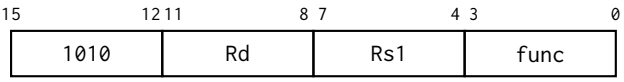
B.2.5 Opcode 0x4: XOR

Operation: `Rd = Rs1 ^ Rs2`. **Flags**: N, Z (C and V cleared).



B.2.6 Opcode 0xA: Shift and Rotate

The shift/rotate family uses R-format with opcode `0xA`. `Rd` receives the result of operating on `Rs1`. The func field (bits 3:0) selects among 16 operations:



Flags affected: N, Z, C (V is cleared). The carry flag receives the last bit shifted out.

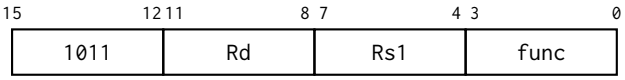
Table B.1: Shift/rotate function codes (opcode `0xA`)

func	Mnemonic	Binary	Operation
0x0	**SLL**	0000	Shift left logical by 1
0x1	**SRL**	0001	Shift right logical by 1
0x2	**SRA**	0010	Shift right arithmetic by 1 (sign bit preserved)
0x3	**ROL**	0011	Rotate left by 1
0x4	**ROR**	0100	Rotate right by 1
0x5	**RCL**	0101	Rotate left through carry
0x6	**RCR**	0110	Rotate right through carry
0x7	**SWAP**	0111	Swap high and low bytes
0x8	**SLL4**	1000	Shift left by 4
0x9	**SRL4**	1001	Shift right logical by 4
0xA	**SRA4**	1010	Shift right arithmetic by 4
0xB	**ROL4**	1011	Rotate left by 4
0xC	**SLL8**	1100	Shift left by 8
0xD	**SRL8**	1101	Shift right logical by 8
0xE	**SRA8**	1110	Shift right arithmetic by 8
0xF	**ROL8**	1111	Rotate left by 8

The fixed-count shift variants (by 4 and by 8) are provided because the base SAMPO architecture does not have a barrel shifter. Shifting by n positions otherwise requires n separate single-shift instructions. The by-4 and by-8 variants, combined with by-1, cover the most common shift distances in a single instruction each.

B.2.7 Opcode 0xB: Multiply and Divide

The multiply/divide family also uses R-format with the func field selecting the operation. Note that these instructions use a *two-operand* form: Rd is both a source and the destination.



Flags affected: N, Z only.

Table B.2: Multiply/divide function codes (opcode 0xB)

func	Mnemonic	Binary	Operation
0x0	**MUL**	0000	Rd = (Rd × Rs1)[15:0] (low 16 bits)
0x1	**MULH**	0001	Rd = (Rd × Rs1)[31:16] (high 16, signed)
0x2	**MULHU**	0010	Rd = (Rd × Rs1)[31:16] (high 16, unsigned)
0x3	**DIV**	0011	Rd = Rd ÷ Rs1 (signed)
0x4	**DIVU**	0100	Rd = Rd ÷ Rs1 (unsigned)
0x5	**REM**	0101	Rd = Rd mod Rs1 (signed)
0x6	**REMU**	0110	Rd = Rd mod Rs1 (unsigned)
0x7	**DAA**	0111	Decimal adjust Rd (BCD correction)

To obtain a full 32-bit product, execute **MUL** followed by **MULH** (or **MULHU** for unsigned) with the same operands. The low 16 bits go to the first destination, the high 16 bits to the second.

B.2.8 Opcode 0xC: Miscellaneous

The miscellaneous opcode is a catch-all for instructions that do not fit neatly into the other categories. The encoding varies by function:

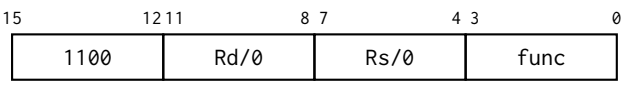


Table B.3: Miscellaneous function codes (opcode 0xC)

func	Mnemonic	Binary	Encoding Notes
0x0	**PUSH**	0000	[11:8]=0000, [7:4]=Rs (source register)
0x1	**POP**	0001	[11:8]=Rd (destination), [7:4]=0000
0x2	**CMP**	0010	[11:8]=Rd, [7:4]=Rs; flags = Rd − Rs
0x3	**TEST**	0011	[11:8]=Rd, [7:4]=Rs; flags = Rd & Rs
0x4	**MOV**	0100	[11:8]=Rd, [7:4]=Rs; Rd = Rs
0x5	**LDI**	0101	Fixed encoding 0xC005
0x6	**LDD**	0110	Fixed encoding 0xC006
0x7	**LDIR**	0111	Fixed encoding 0xC007
0x8	**LDDR**	1000	Fixed encoding 0xC008
0x9	**CPIR**	1001	Fixed encoding 0xC009
0xA	**FILL**	1010	Fixed encoding 0xC00A
0xB	**EXX**	1011	Fixed encoding 0xC00B
0xC	**GETF**	1100	[11:8]=Rd; Rd = FLAGS
0xD	**SETF**	1101	[7:4]=Rs; FLAGS = Rs[7:0]

PUSH and POP Field Ordering

PUSH places the source register in bits 7:4 (the Rs1 position), with bits 11:8 set to zero. **POP** places the destination register in bits 11:8 (the Rd position), with bits 7:4 set to zero. The stack pointer (R2/SP) is used implicitly and does not appear in the instruction encoding. The hardware reads SP during the DECODE state by overriding the register file read address to 0x2.

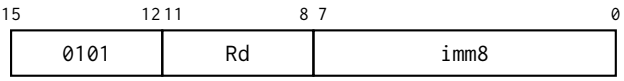
The block transfer instructions (**LDI**, **LDD**, **LDIR**, **LDDR**, **CPIR**, **FILL**) use fixed encodings with no register operands in the instruction word. They implicitly operate on conventions registers:

- A0 (R4): Source address
- A1 (R5): Destination address
- A2 (R6): Byte count / search value

B.3 I-Format Instructions

B.3.1 Opcode 0x5: ADDI

Operation: Rd = Rd + sign_extend(imm8). **Flags**: N, Z, C, V.



The immediate is a signed 8-bit value in two's complement, giving a range of -128 to $+127$. It is sign-extended to 16 bits before the addition. In the decoder:

Listing B.1: Sign extension of imm8 (`verilog/rtl/decode.v`)

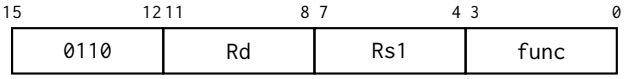
```
assign imm8_sext = {{8{instr[7]}}, instr[7:0]};
```

ADDI reads and writes the *same* register. During the DECODE state, the register file read address is overridden: `rf_rd_addr1 = dec_rd` instead of the usual `dec_rs1`.

B.4 Load and Store Instructions

B.4.1 Opcode 0x6: Load

Load instructions use R-format encoding with the func field selecting the access type and offset:



Flags affected: None.

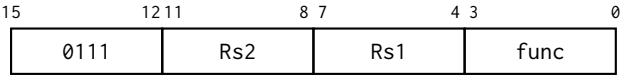
Table B.4: Load function codes (opcode `0x6`)

func	Mnemonic	Binary	Operation
`0x0`	**LW** Rd, (Rs1)	`0000`	Rd = mem16[Rs1]
`0x1`	**LB** Rd, (Rs1)	`0001`	Rd = sign_ext(mem8[Rs1])
`0x2`	**LBU** Rd, (Rs1)	`0010`	Rd = zero_ext(mem8[Rs1])
`0x3`	**LW** Rd, 2(Rs1)	`0011`	Rd = mem16[Rs1 + 2]
`0x4`	**LW** Rd, 4(Rs1)	`0100`	Rd = mem16[Rs1 + 4]
`0x5`	**LW** Rd, 6(Rs1)	`0101`	Rd = mem16[Rs1 + 6]
`0x6`	**LW** Rd, -2(Rs1)	`0110`	Rd = mem16[Rs1 $-$ 2]
`0x7`	**LW** Rd, -4(Rs1)	`0111`	Rd = mem16[Rs1 $-$ 4]
`0x8`	**LUI** Rd, Rs1	`1000`	Rd = Rs1 $\ll 8$

LUI (Load Upper Immediate) is a special case: it does not access memory. Instead, it shifts `Rs1` left by 8 bits, placing the low byte of `Rs1` into the high byte of `Rd` and zeroing the low byte. Combined with **ADDI**, this provides a two-instruction sequence for loading arbitrary 16-bit constants (an alternative to the 32-bit **LIX**).

B.4.2 Opcode 0x7: Store

Store instructions use S-format encoding. The bits 11:8 field holds Rs2 (the data to store), not a destination register:



Flags affected: None.

Table B.5: Store function codes (opcode 0x7)

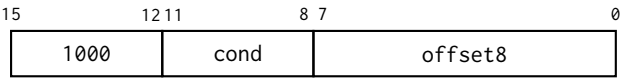
func	Mnemonic	Binary	Effective Address
0x0	**SW** (Rs1), Rs2	0000	Rs1
0x1	**SB** (Rs1), Rs2	0001	Rs1 (byte)
0x2	**SW** 2(Rs1), Rs2	0010	Rs1 + 2
0x3	**SW** 4(Rs1), Rs2	0011	Rs1 + 4
0x4	**SW** 6(Rs1), Rs2	0100	Rs1 + 6
0x5	**SW** −2(Rs1), Rs2	0101	Rs1 − 2
0x6	**SW** −4(Rs1), Rs2	0110	Rs1 − 4

Load and Store Offsets Are Not Symmetric

The load opcode has 9 function codes (including **LB**, **LBU**, and **LUI**), while the store opcode has only 7. This asymmetry reflects the fact that loads are more common than stores in typical programs, and loads need more variants (signed/unsigned byte, upper immediate).

B.5 B-Format: Branch Instructions

All 16 conditional branches share opcode 0x8. The condition field (bits 11:8) selects which flag combination to test:



Target calculation:

$$\text{PC}_{\text{target}} = \text{PC} + 2 + \text{sign_extend}(\text{offset8}) \times 2$$

The offset8 field is a signed 8-bit value in two's complement (range −128 to +127). Multiplied by 2, the effective byte range is −256 to +254 relative to the next instruction. Branches do not modify any flags.

Table B.6: Complete branch condition encoding (opcode `0x8`)

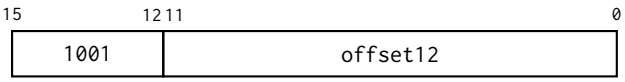
cond	Binary	Mnemonic	Flags Tested	Condition	Meaning
`0x0`	`0000`	**BEQ**	Z	Z=1	Equal
`0x1`	`0001`	**BNE**	Z	Z=0	Not equal
`0x2`	`0010`	**BLT**	N, V	N≠V	Signed less than
`0x3`	`0011`	**BGE**	N, V	N=V	Signed greater or equal
`0x4`	`0100`	**BLTU**	C	C=0	Unsigned less than
`0x5`	`0101`	**BGEU**	C	C=1	Unsigned greater or equal
`0x6`	`0110`	**BMI**	N	N=1	Minus (negative)
`0x7`	`0111`	**BPL**	N	N=0	Plus (positive/zero)
`0x8`	`1000`	**BVS**	V	V=1	Overflow set
`0x9`	`1001`	**BVC**	V	V=0	Overflow clear
`0xA`	`1010`	**BCS**	C	C=1	Carry set
`0xB`	`1011`	**BCC**	C	C=0	Carry clear
`0xC`	`1100`	**BGT**	Z, N, V	Z=0 ∧ N=V	Signed greater than
`0xD`	`1101`	**BLE**	Z, N, V	Z=1 ∨ N≠V	Signed less or equal
`0xE`	`1110`	**BHI**	C, Z	C=1 ∧ Z=0	Unsigned higher
`0xF`	`1111`	**BLS**	C, Z	C=0 ∨ Z=1	Unsigned lower or same

Condition Code Symmetry

The 16 conditions are arranged in complementary pairs: **BEQ**/**BNE** (0/1), **BLT**/**BGE** (2/3), **BLTU**/**BGEU** (4/5), and so on. In hardware, the evaluator inverts bit 0 of the condition code to flip the sense. This is visible in `verilog/rtl/cpu.v` where the branch evaluator is a simple 16-entry case statement.

B.6 J-Format: Jump Instructions

B.6.1 J: PC-Relative Jump



Target calculation:

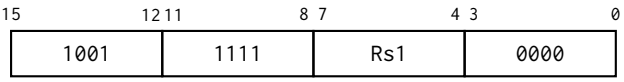
$$\text{PC}_{\text{target}} = \text{PC} + 2 + \text{sign_extend}(\text{offset12}) \times 2$$

The 12-bit signed offset gives a range of −4096 to +4094 bytes. This is the primary unconditional jump instruction for loops and forward jumps.

B.6.2 JR and JALR: Register Jumps

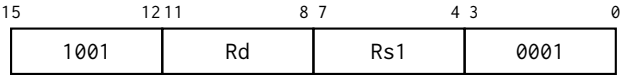
The register-indirect jump forms share opcode `0x9` with the PC-relative jump, but are distinguished by a specific bit pattern:

JR Rs1 (Register jump):



Operation: PC = Rs1. Bits 11:8 are set to 1111 to distinguish this from a PC-relative jump. The func field (bits 3:0) is `0000`.

JALR Rd, Rs1 (Jump and link register):



Operation: Rd = PC + 2; PC = Rs1. The return address (address of the instruction following the **`JALR`**) is stored in `Rd`, typically `R1`/`RA`. The func field (bits 3:0) is `0001` to distinguish from **`JR`**.

Neither **`JR`** nor **`JALR`** modifies flags.

B.7 X-Format: Extended Instructions

All extended instructions have opcode `0xF` in bits 15:12 of the first word. The sub-opcode field (bits 3:0 of the first word) selects among 16 operations. The second word provides a 16-bit immediate.

Several encoding details deserve attention:

- **ADDIX through XORIX**: These are true three-operand instructions. `Rd` receives the result, `Rs` provides one operand, and imm16 provides the other. Unlike the base **`ADDI`**, the source and destination need not be the same register.
- **LWX and SWX**: The imm16 field is a signed 16-bit offset, giving access to the entire 64KB address space relative to the base register.
- **LIX**: The Rs field is ignored (conventionally set to `0x0`). The full imm16 value is loaded directly into `Rd`.
- **JX and JALX**: These use absolute addressing. The imm16 value is the target address, not a PC-relative offset. **`JALX`** stores the return address (PC + 4, since the instruction is 4 bytes) in `RA` (`R1`).
- **INX and OUTX**: Only the low 8 bits of imm16 are used as the port address, giving access to all 256 I/O ports. The upper 8 bits are ignored.

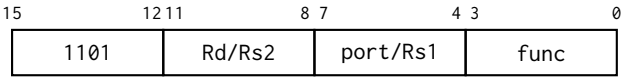
Table B.7: Extended instruction sub-opcodes (opcode `0xF`)

sub	Binary	Mnemonic	Operation	Flags	imm16 Use
`0x0`	`0000`	**ADDIX**	Rd = Rs + imm16	N,Z,C,V	Full 16-bit value
`0x1`	`0001`	**SUBIX**	Rd = Rs − imm16	N,Z,C,V	Full 16-bit value
`0x2`	`0010`	**ANDIX**	Rd = Rs & imm16	N,Z	Full 16-bit value
`0x3`	`0011`	**ORIX**	Rd = Rs | imm16	N,Z	Full 16-bit value
`0x4`	`0100`	**XORIX**	Rd = Rs ^ imm16	N,Z	Full 16-bit value
`0x5`	`0101`	**LWX**	Rd = mem16[Rs + imm16]	—	Signed offset
`0x6`	`0110`	**SWX**	mem16[Rs + imm16] = Rd	—	Signed offset
`0x7`	`0111`	**LIX**	Rd = imm16	—	Direct value
`0x8`	`1000`	**JX**	PC = imm16	—	Absolute address
`0x9`	`1001`	**JALX**	RA = PC+4; PC = imm16	—	Absolute address
`0xA`	`1010`	**CMPIX**	flags = Rd − imm16	N,Z,C,V	Full 16-bit value
`0xB`	`1011`	**INX**	Rd = port_read(imm16[7:0])	—	Port number (low 8)
`0xC`	`1100`	**OUTX**	port_write(imm16[7:0], Rs)	—	Port number (low 8)
`0xD`	`1101`	**SLLX**	Rd = Rs ≪ imm16[3:0]	N,Z,C	Shift amount (low 4)
`0xE`	`1110`	**SRLX**	Rd = Rs ≫ imm16[3:0]	N,Z,C	Shift amount (low 4)
`0xF`	`1111`	**SRAX**	Rd = Rs ⋙ imm16[3:0]	N,Z,C	Shift amount (low 4)

- **SLLX, SRLX, SRAX**: Only the low 4 bits of imm16 are used as the shift amount (range 0–15). These provide variable-distance shifts in a single instruction, unlike the base shift operations which only shift by fixed amounts (1, 4, or 8).

B.8 I/O Instructions

B.8.1 Opcode `0xD`: I/O



Flags affected: None.

Table B.8: I/O function codes (opcode `0xD`)

func	Mnemonic	Binary	Encoding Notes
`0x0`	**INI** Rd, port4	`0000`	Port in [7:4] (4-bit, 0–15)
`0x1`	**OUTI** port4, Rs	`0001`	Port in [7:4], data in [11:8]
`0x2`	**IN** Rd, (Rs1)	`0010`	Port address from register
`0x3`	**OUT** (Rs1), Rd	`0011`	Port address from register

The immediate forms (**INI**, **OUTI**) encode the port number in just 4 bits (bits 7:4), limiting them to ports 0–15. For accessing the full 256-port I/O space, use either the register-indirect forms (**IN**, **OUT**) or the extended forms (**INX**, **OUTX** from the X-format).

I/O Data Width

All I/O operations transfer 8-bit data, regardless of the 16-bit register width. For input, the port value is zero-extended to 16 bits before being written to `Rd`. For output, only the low 8 bits of the source register are sent to the port.

B.9 System Instructions

B.9.1 Opcode `0xE`: System

System instructions use a unique format where the function code occupies bits 11:8 and the lower 8 bits serve as an operand (used only by **SWI**):

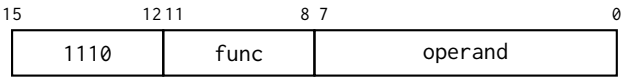


Table B.9: System function codes (opcode `0xE`)

func	Hex Encoding	Mnemonic	Binary	Description
`0x0`	`0xE000`	**NOP**	`0000`	No operation
`0x1`	`0xE100`	**HALT**	`0001`	Halt the CPU
`0x2`	`0xE200`	**DI**	`0010`	Disable interrupts (I=0)
`0x3`	`0xE300`	**EI**	`0011`	Enable interrupts (I=1)
`0x4`	`0xE400`	**RETI**	`0100`	Return from interrupt
`0x5`	`0xE5`*ii*	**SWI** imm8	`0101`	Software interrupt, vector *ii*
`0x6`	`0xE600`	**SCF**	`0110`	Set carry flag (C=1)
`0x7`	`0xE700`	**CCF**	`0111`	Complement carry flag (C=¬C)

Most system instructions have fixed encodings because the operand field is zero. The exception is **SWI**, which uses the low 8 bits as a software interrupt vector number. **NOP** is the canonical "do nothing" instruction; its encoding `0xE000` is also the value produced by erased flash memory on some FPGA boards, which makes uninitialized code regions behave as NOP slides rather than executing undefined instructions.

B.10 Encoding Map Summary

Table B.10 provides the complete opcode map in a single table for quick reference.

Table B.10: Complete SAMPO opcode map

Opcode	Binary	Category	Format	Width	Sub-codes
`0x0`	`0000`	ADD	R	16-bit	—
`0x1`	`0001`	SUB	R	16-bit	—
`0x2`	`0010`	AND	R	16-bit	—
`0x3`	`0011`	OR	R	16-bit	—
`0x4`	`0100`	XOR	R	16-bit	—
`0x5`	`0101`	ADDI	I	16-bit	—
`0x6`	`0110`	Load	R	16-bit	9 variants
`0x7`	`0111`	Store	S	16-bit	7 variants
`0x8`	`1000`	Branch	B	16-bit	16 conditions
`0x9`	`1001`	Jump	J/R	16-bit	3 variants
`0xA`	`1010`	Shift/Rotate	R	16-bit	16 operations
`0xB`	`1011`	Multiply/Divide	R	16-bit	8 operations
`0xC`	`1100`	Miscellaneous	R	16-bit	14 operations
`0xD`	`1101`	I/O	R	16-bit	4 variants
`0xE`	`1110`	System	—	16-bit	8 operations
`0xF`	`1111`	Extended	X	32-bit	16 sub-opcodes

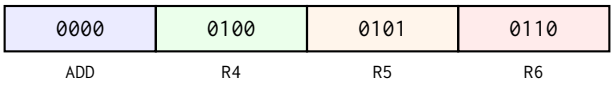
In total, the SAMPO ISA defines approximately 100 distinct instruction encodings: 5 basic ALU operations, 1 immediate ALU operation, 9 loads, 7 stores, 16 branches, 3 jumps, 16 shifts, 8 multiply/divide operations, 14 miscellaneous operations, 4 I/O operations, 8 system operations, and 16 extended operations.

B.11 Encoding Examples

This section walks through eight complete encoding examples. For each, we show the assembly mnemonic, the field breakdown, the binary and hexadecimal representations, and the little-endian byte ordering as it would appear in memory.

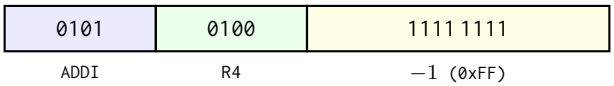
B.11.1 Example 1: ADD R4, R5, R6

Step	Value
Assembly	**ADD** R4, R5, R6
Format	R-type
opcode	0x0 = 0000
Rd	0x4 = 0100
Rs1	0x5 = 0101
Rs2	0x6 = 0110
Binary	0000 0100 0101 0110
Hexadecimal	0x0456
Bytes (LE)	56 04



B.11.2 Example 2: ADDI R4, -1

Step	Value
Assembly	**ADDI** R4, −1
Format	I-type
opcode	0x5 = 0101
Rd	0x4 = 0100
imm8	−1 = 0xFF = 1111 1111
Binary	0101 0100 1111 1111
Hexadecimal	0x54FF
Bytes (LE)	FF 54



The immediate value −1 is represented in 8-bit two's complement as 0xFF. The hardware sign-extends this to 0xFFFF (−1 in 16 bits) before adding it to R4.

B.11.3 Example 3: LW R4, (R5)

Step	Value
Assembly	`LW R4, (R5)`
Format	R-type (load)
opcode	`0x6` = `0110`
Rd	`0x4` = `0100`
Rs1	`0x5` = `0101`
func	`0x0` = `0000` (word load, offset 0)
Binary	`0110 0100 0101 0000`
Hexadecimal	`0x6450`
Bytes (LE)	`50 64`

B.11.4 Example 4: BEQ +8

This example encodes a branch forward by 8 bytes. The target address is PC + 8, which means the *offset* relative to the next instruction (PC + 2) is 6 bytes. Since the offset is stored as a word count (divided by 2), the encoded value is 3:

Step	Value
Assembly	`BEQ` target (target = PC + 8)
Format	B-type
opcode	`0x8` = `1000`
cond	`0x0` = `0000` (BEQ: Z=1)
offset8	$(8 - 2) \div 2 = 3$ = `0000 0011`
Binary	`1000 0000 0000 0011`
Hexadecimal	`0x8003`
Bytes (LE)	`03 80`

Verification: $PC_{target} = PC + 2 + 3 \times 2 = PC + 8$. Correct.

Offset Calculation for Branches

The general formula for computing the offset8 field from an absolute target address is:

$$\text{offset8} = \frac{\text{target} - (\text{PC} + 2)}{2}$$

The assembler performs this calculation automatically when you use labels. For hand-assembly, remember that the branch instruction itself occupies 2 bytes, so the offset is measured from the *next* instruction.

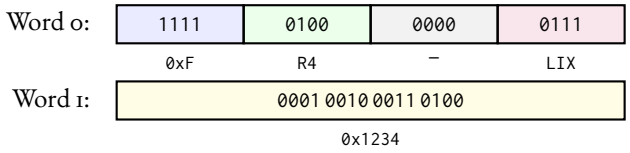
B.11.5 Example 5: PUSH R4

Step	Value
Assembly	**PUSH** R4
Format	R-type (misc)
opcode	0xC = 1100
[11:8]	0x0 = 0000 (unused for PUSH)
Rs (source)	0x4 = 0100 (in bits [7:4])
func	0x0 = 0000 (PUSH)
Binary	1100 0000 0100 0000
Hexadecimal	0xC040
Bytes (LE)	40 C0

The stack pointer (SP/R2) is implicit: the hardware decrements SP by 2 and writes the source register value to the new stack address.

B.11.6 Example 6: LIX R4, 0x1234

Step	Value
Assembly	`LIX R4, 0x1234`
Format	X-type (32-bit)
Word 0:	
opcode	`0xF` = `1111`
Rd	`0x4` = `0100`
Rs	`0x0` = `0000` (unused)
sub	`0x7` = `0111` (LIX)
Word 1:	
imm16	`0x1234`
Word 0 Binary	`1111 0100 0000 0111`
Word 0 Hex	`0xF407`
Word 1 Hex	`0x1234`
Bytes (LE)	`07 F4 34 12`



Note the byte ordering in memory. The first word `0xF407` is stored as bytes `07 F4` (low byte first), and the second word `0x1234` is stored as bytes `34 12`.

B.11.7 Example 7: JR RA

Step	Value
Assembly	`JR RA` (`RA` = `R1`)
Format	J-type (register variant)
opcode	`0x9` = `1001`
[11:8]	`0xF` = `1111` (marker for register jump)
Rs1	`0x1` = `0001` (RA)
func	`0x0` = `0000` (JR, not JALR)
Binary	`1001 1111 0001 0000`
Hexadecimal	`0x9F10`
Bytes (LE)	`10 9F`

The 1111 pattern in bits 11:8 distinguishes register-indirect jumps from PC-relative jumps. Since the maximum positive 12-bit signed offset would not produce 1111 in the top 4 bits of the offset12 field (the maximum is 0111 1111 1111 = 0x7FF), there is no ambiguity.

B.11.8 Example 8: JALX 0x0400

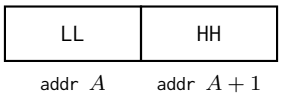
Step	Value
Assembly	**JALX** 0x0400
Format	X-type (32-bit)
Word 0:	
opcode	0xF = 1111
Rd	0x1 = 0001 (RA—implicit link register)
Rs	0x0 = 0000 (unused)
sub	0x9 = 1001 (JALX)
Word 1:	
imm16	0x0400
Word 0 Hex	0xF109
Word 1 Hex	0x0400
Bytes (LE)	09 F1 00 04

JALX stores the return address (PC + 4, because the instruction is 4 bytes long) into RA, then sets the PC to the absolute address in imm16. The Rd field is set to 0x1 (RA) by convention; theoretically any register could be used as the link register, but the calling convention designates R1.

B.12 Byte Order and Memory Layout

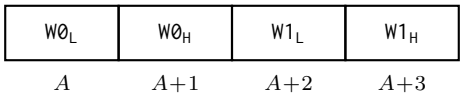
All SAMPO instructions are stored in little-endian byte order. For a 16-bit instruction with hexadecimal value 0xHHLL:

16-bit instruction in memory



For a 32-bit extended instruction (0xW0W0 followed by 0xW1W1):

32-bit instruction in memory



For example, the instruction **LIX** R4, 0x1234 (Word 0 = 0xF407, Word 1 = 0x1234) appears in memory at address A as:

Address	A	$A+1$	$A+2$	$A+3$
Byte	07	F4	34	12

Hex Dump Reading

When reading a hex dump, remember that the first byte you see is the *low* byte of the first word. To reconstruct the instruction, swap each pair of bytes. If a hex dump shows 40 C0, the instruction word is 0xC040 (a **PUSH** R4).

B.13 Quick Decoding Procedure

Given a raw 16-bit value from a hex dump or memory inspection, follow this procedure to decode it:

1. **Extract the opcode**: Read bits 15:12 (the top nibble of the hex value).
2. **Check for extended**: If the opcode is 0xF, fetch the next 16-bit word. The instruction is 32 bits.
3. **Determine the format**: Use the opcode to look up the format in Table B.10.
4. **Extract fields**: Based on the format, extract Rd, Rs1, Rs2/func/cond/offset from the appropriate bit positions.
5. **Look up sub-codes**: For opcodes with function codes (0x6, 0x7, 0xA, 0xB, 0xC, 0xD, 0xE, 0xF), use the func or sub field to identify the specific instruction.
6. **Interpret immediate values**: Sign-extend offset8 (branches), offset12 (jumps), and imm8 (**ADDI**) as appropriate.

Decoding example: Given the bytes 54 FB at address 0x0200:

1. Swap bytes for little-endian: instruction = 0xFB54. Wait—that gives opcode 0xF, which is extended. Let us re-examine. Actually, swap: low byte 54, high byte FB. Instruction = 0xFB54? No. The bytes in memory are 54 at address 0x0200 and FB at address 0x0201. In little-endian, the word is 0xFB54. The opcode is 0xF ... but this looks wrong for our intent.
   Let us try different bytes: FB 54. Word = 0x54FB. Opcode = 0x5 = **ADDI**. Rd = 0x4 = R4. imm8 = 0xFB = -5 in signed 8-bit. Result: **ADDI** R4, -5.

This procedure is precisely what the hardware decoder in verilog/rtl/decode.v performs combinationally on every instruction fetch.

This appendix, together with Chapter 2, provides everything needed to hand-assemble or disassemble SAMPO machine code. For a higher-level quick reference organized by instruction mnemonic rather than encoding format, see Appendix A.

Appendix C

Verilog Module Reference

This appendix provides a complete port-level reference for every Verilog module in the SAMPO RTL implementation. Each section documents a single module's parameters, ports, internal architecture, and a typical instantiation example. The modules are ordered top-down: the system-on-chip wrapper first, then the CPU core, and finally the submodules the CPU instantiates.

All source files reside under `verilog/rtl/` and share the common include file `sampo_pkg.vh`, which defines constants for opcodes, ALU operations, shift functions, FSM states, and register aliases.

C.1 The soc Module

The top-level system-on-chip module, defined in `verilog/rtl/soc.v`, integrates the CPU core, 64 KB synchronous RAM, and an MC6850-compatible UART. It provides the complete SAMPO system in a single instantiable unit, suitable for both simulation testbenches and FPGA synthesis.

C.1.1 Parameters

Table C.1: Parameters of the soc module

Parameter	Default	Description
`RESET_VECTOR`	`16'h0100`	Address loaded into PC on reset; forwarded to the cpu instance
`RAM_INIT_FILE`	`""`	Path to a hex file for `$readmemh` RAM initialization; forwarded to the `ram` instance

C.1.2 Port Table

Table C.2: Ports of the soc module

Port	Dir	Width	Description
clk	input	1	System clock
rst_n	input	1	Active-low synchronous reset
tx_data	output	8	UART transmit data byte
tx_valid	output	1	UART TX data valid strobe
tx_ready	input	1	External TX ready for data
rx_data	input	8	UART receive data byte
rx_valid	input	1	External RX data valid strobe
rx_ready	output	1	UART RX data accepted
halted	output	1	High when CPU has executed **HALT**
pc	output	16	Current program counter value
cycles	output	32	Cycle counter since reset
led	output	8	LED status output (see below)

C.1.3 LED Assignment

The led output provides a compact status display for FPGA boards:

Bit(s)	Source
led[7]	halted — CPU halted
led[6]	tx_valid — UART TX active
led[5]	rx_valid — UART RX data available
led[4:0]	pc[5:1] — execution progress

C.1.4 Submodule Instances

The soc module instantiates three submodules:

1. cpu_inst (cpu) — the SAMPO CPU core, with RESET_VECTOR forwarded from the soc parameter.
2. ram_inst (ram) — 64 KB synchronous RAM, with INIT_FILE set from RAM_INIT_FILE.
3. uart_inst (uart) — MC6850-compatible UART, bridging the CPU's I/O interface to the external UART signals.

Memory ready generation uses a single-cycle delay: `mem_ready` is `cpu_mem_valid` delayed by one clock cycle, matching the synchronous RAM's read latency.

C.1.5 Instantiation Example

Listing C.1: Typical soc instantiation in a testbench

```
soc #(
    .RESET_VECTOR(16'h0100),
    .RAM_INIT_FILE("programs/test_alu.hex")
) dut (
    .clk(clk),
    .rst_n(rst_n),
    .tx_data(tx_data),
    .tx_valid(tx_valid),
    .tx_ready(tx_ready),
    .rx_data(rx_data),
    .rx_valid(rx_valid),
    .rx_ready(rx_ready),
    .halted(halted),
    .pc(pc),
    .cycles(cycles),
    .led(led)
);
```

C.2 The cpu Module

The CPU core, defined in `verilog/rtl/cpu.v`, implements the SAMPO processor as a finite state machine. It exposes a memory bus interface, a separate I/O port interface, and status outputs. Internally it instantiates the `regfile`, `alu`, `shifter`, and `decode` submodules.

C.2.1 Parameters

Table C.3: Parameters of the cpu module

Parameter	Default	Description
`RESET_VECTOR`	`16'h0100`	Address loaded into PC on reset

Table C.4: Ports of the cpu module

Port	Dir	Width	Description
Clock and reset			
`clk`	input	1	System clock
`rst_n`	input	1	Active-low synchronous reset
Memory interface			
`mem_addr`	output reg	16	Memory address bus
`mem_rdata`	input	16	Memory read data
`mem_wdata`	output reg	16	Memory write data
`mem_we`	output reg	1	Memory write enable
`mem_be`	output reg	2	Byte enables: `[0]`=low, `[1]`=high
`mem_valid`	output reg	1	Memory request valid
`mem_ready`	input	1	Memory request complete
I/O interface			
`io_addr`	output reg	8	I/O port address
`io_rdata`	input	8	I/O read data
`io_wdata`	output reg	8	I/O write data
`io_rd`	output reg	1	I/O read strobe
`io_wr`	output reg	1	I/O write strobe
Status outputs			
`halted`	output	1	High when CPU is in HALTED
`pc_out`	output	16	Current program counter value
`cycles_out`	output	32	Cycle counter since reset

C.2.2 Port Table

C.2.3 Internal Architecture

The CPU operates as an eight-state FSM. The key internal registers are:

- `pc` (16-bit) — program counter, initialized to `RESET_VECTOR`.
- `flags` (8-bit) — status flags register (N, Z, C, V, I bits).
- `instr` (16-bit) — latched instruction word from the FETCH state.
- `imm16` (16-bit) — second instruction word for extended (32-bit) instructions.
- `cycle_count` (32-bit) — free-running cycle counter, incremented every clock edge.

C.2.4 FSM State Transitions

The eight FSM states are encoded in a 4-bit `state` register:

Table C.5: CPU FSM states and transitions

State	Next State	Condition
RESET	FETCH	Unconditional
FETCH	DECODE	When `mem_ready` asserts
DECODE	FETCH_EXT	If instruction is extended format
DECODE	EXECUTE	Otherwise
FETCH_EXT	EXECUTE	When `mem_ready` asserts
EXECUTE	HALTED	If **`HALT`** instruction
EXECUTE	MEMORY	If load or store operation
EXECUTE	WRITEBACK	If register write needed
EXECUTE	FETCH	Otherwise
MEMORY	WRITEBACK	When `mem_ready` asserts
WRITEBACK	FETCH	Unconditional
HALTED	HALTED	Terminal state

C.2.5 Submodule Instances

The cpu module instantiates four submodules internally:

Instance Name	Module	Purpose
`regfile_inst`	`regfile`	16×16-bit register file
`alu_inst`	`alu`	Arithmetic logic unit
`shifter_inst`	`shifter`	Barrel shifter
`decode_inst`	`decode`	Instruction decoder

C.2.6 Instantiation Example

Listing C.2: CPU instantiation within the SoC

```
cpu #(
    .RESET_VECTOR(RESET_VECTOR)
) cpu_inst (
    .clk(clk),
    .rst_n(rst_n),
    .mem_addr(cpu_mem_addr),
```

```
    .mem_rdata(cpu_mem_rdata),
    .mem_wdata(cpu_mem_wdata),
    .mem_we(cpu_mem_we),
    .mem_be(cpu_mem_be),
    .mem_valid(cpu_mem_valid),
    .mem_ready(cpu_mem_ready),
    .io_addr(cpu_io_addr),
    .io_rdata(cpu_io_rdata),
    .io_wdata(cpu_io_wdata),
    .io_rd(cpu_io_rd),
    .io_wr(cpu_io_wr),
    .halted(halted),
    .pc_out(pc),
    .cycles_out(cycles)
);
```

C.3 The `alu` Module

The ALU, defined in `verilog/rtl/alu.v`, is a purely combinational module that performs all arithmetic, logical, and pass-through operations. It has no clock input. The CPU drives the ALU's inputs during the EXECUTE phase and samples the outputs within the same cycle.

C.3.1 Port Table

Table C.6: Ports of the `alu` module

Port	Dir	Width	Description
`a`	input	16	First operand
`b`	input	16	Second operand
`op`	input	4	Operation select (see Table C.7)
`carry_in`	input	1	Carry input (from `FLAGS[5]`)
`result`	output reg	16	Operation result
`flag_n`	output	1	Negative flag (`result[15]`)
`flag_z`	output	1	Zero flag (`result == 0`)
`flag_c`	output reg	1	Carry flag
`flag_v`	output reg	1	Overflow flag

Continuous Flag Assignments

The `flag_n` and `flag_z` outputs are assigned as continuous wires (`assign flag_n = result[15]` and `assign flag_z = (result == 16'h0000)`), not within the combinational case block. This ensures they are always valid regardless of which operation is selected.

C.3.2 ALU Operations

Table C.7: ALU operation encoding

op	Name	Operation	Carry	Overflow
`0x0`	`ALU_ADD`	`a + b`	Bit 16 of sum	Sign mismatch
`0x1`	`ALU_SUB`	`a - b`	~borrow (ARM-style)	Sign mismatch
`0x2`	`ALU_AND`	`a & b`	Cleared	Cleared
`0x3`	`ALU_OR`	`a | b`	Cleared	Cleared
`0x4`	`ALU_XOR`	`a ^ b`	Cleared	Cleared
`0x5`	`ALU_SLL`	`a << b[3:0]`	Last bit out	—
`0x6`	`ALU_SRL`	`a >> b[3:0]`	Last bit out	—
`0x7`	`ALU_SRA`	`a >>> b[3:0]`	Last bit out	—
`0x8`	`ALU_MUL`	Signed multiply low	—	—
`0x9`	`ALU_MULH`	Signed multiply high	—	—
`0xA`	`ALU_DIV`	Signed divide	—	—
`0xB`	`ALU_REM`	Signed remainder	—	—
`0xC`	`ALU_PASS_A`	a (passthrough)	—	—
`0xD`	`ALU_PASS_B`	b (passthrough)	—	—
`0xE`	`ALU_NOT`	`~a`	—	—
`0xF`	`ALU_NEG`	`-a`	`a != 0`	`a == 0x8000`

For `ALU_SUB`, the carry flag uses ARM-style semantics: `C=1` means *no* borrow occurred (i.e., `a >= b` for unsigned comparison). Division by zero returns `0xFFFF` for `ALU_DIV` and the dividend unchanged for `ALU_REM`.

C.3.3 Instantiation Example

Listing C.3: ALU instantiation within the CPU

```
alu alu_inst (
    .a(alu_a),
    .b(alu_b),
    .op(alu_op),
    .carry_in(flags[`FLAG_C]),
```

```
    .result(alu_result),
    .flag_n(alu_flag_n),
    .flag_z(alu_flag_z),
    .flag_c(alu_flag_c),
    .flag_v(alu_flag_v)
);
```

C.4 The `regfile` Module

The register file, defined in `verilog/rtl/regfile.v`, provides 16 general-purpose 16-bit registers plus 8 alternate registers used by the **`EXX`** instruction. It features two read ports, a main write port, and a dedicated stack pointer write port that enables simultaneous updates to both a destination register and `SP` during **`POP`** operations.

C.4.1 Port Table

Table C.8: Ports of the `regfile` module

Port	Dir	Width	Description
`clk`	input	1	System clock
`rst_n`	input	1	Active-low reset
`rd_addr1`	input	4	Read port 1 address
`rd_addr2`	input	4	Read port 2 address
`rd_data1`	output	16	Read port 1 data
`rd_data2`	output	16	Read port 2 data
`wr_addr`	input	4	Write port address
`wr_data`	input	16	Write data
`wr_en`	input	1	Write enable
`exx`	input	1	Exchange `R4`–`R11` with alternates
`sp_wr_data`	input	16	SP write port data
`sp_wr_en`	input	1	SP write port enable

C.4.2 Internal Storage

- `regs[0:15]` — 16×16-bit main register array.
- `alt_regs[0:7]` — 8×16-bit alternate registers, corresponding to shadow copies of `R4`'–`R11`'.

C.4.3 Key Behaviors

Ro hardwired to zero. Read ports return `0x0000` when the address is 0. Writes to `R0` are silently ignored (`wr_en && wr_addr != 0`).

EXX atomic swap. When `exx` is asserted, `regs[4..11]` and `alt_regs[0..7]` are swapped in a single clock cycle. The `exx` and `wr_en` signals are mutually exclusive (placed in an `if/else` structure).

Dual write ports. The main write port can target any register. The dedicated SP write port always writes to `R2` (the stack pointer), allowing **`POP`** to update both `Rd` and `SP` simultaneously in the WRITEBACK state.

Reset. All 16 main registers and all 8 alternate registers are cleared to `0x0000` on reset.

C.4.4 Instantiation Example

Listing C.4: Register file instantiation within the CPU

```
regfile regfile_inst (
    .clk(clk),
    .rst_n(rst_n),
    .rd_addr1(rf_rd_addr1),
    .rd_addr2(rf_rd_addr2),
    .wr_addr(rf_wr_addr),
    .rd_data1(rf_rd_data1),
    .rd_data2(rf_rd_data2),
    .wr_data(rf_wr_data),
    .wr_en(rf_wr_en),
    .exx(rf_exx),
    .sp_wr_data(rf_sp_wr_data),
    .sp_wr_en(rf_sp_wr_en)
);
```

C.5 The `shifter` Module

The barrel shifter, defined in `verilog/rtl/shifter.v`, is a purely combinational module that implements 16 fixed-count shift and rotate operations. It is invoked by the CPU during the EXECUTE phase for **`SHIFT`**-class instructions. Variable-count shifts are handled through the ALU instead (see Section C.3).

Table C.9: Ports of the `shifter` module

Port	Dir	Width	Description
`value`	input	16	Input value to shift or rotate
`func`	input	4	Shift function select (see Table C.10)
`carry_in`	input	1	Carry input (from `FLAGS[5]`)
`result`	output reg	16	Shifted result
`carry_out`	output reg	1	Carry output

C.5.1 Port Table

C.5.2 Shift Functions

Table C.10: All 16 shift function encodings

func	Name	Operation	Carry Out
`0x0`	SLL1	`{value[14:0], 1'b0}`	`value[15]`
`0x1`	SRL1	`{1'b0, value[15:1]}`	`value[0]`
`0x2`	SRA1	`{value[15], value[15:1]}`	`value[0]`
`0x3`	ROL1	`{value[14:0], value[15]}`	`value[15]`
`0x4`	ROR1	`{value[0], value[15:1]}`	`value[0]`
`0x5`	RCL1	`{value[14:0], carry_in}`	`value[15]`
`0x6`	RCR1	`{carry_in, value[15:1]}`	`value[0]`
`0x7`	SWAP	`{value[7:0], value[15:8]}`	`carry_in`
`0x8`	SLL4	`{value[11:0], 4'b0}`	`value[15]`
`0x9`	SRL4	`{4'b0, value[15:4]}`	`value[3]`
`0xA`	SRA4	`{4{value[15]}, value[15:4]}`	`value[3]`
`0xB`	ROL4	`{value[11:0], value[15:12]}`	`value[15]`
`0xC`	SLL8	`{value[7:0], 8'b0}`	`value[15]`
`0xD`	SRL8	`{8'b0, value[15:8]}`	`value[7]`
`0xE`	SRA8	`{8{value[15]}, value[15:8]}`	`value[7]`
`0xF`	ROL8	`{value[7:0], value[15:8]}`	`value[15]`

The operations are grouped by shift distance: single-bit (1), nibble (4), and byte (8). The `SWAP` operation at `0x7` exchanges the high and low bytes while leaving the carry unchanged. Note that `ROL8` at `0xF` produces the same result value as `SWAP` at `0x7`, but sets the carry output to `value[15]` rather than preserving the carry input.

C.5.3 Instantiation Example

Listing C.5: Shifter instantiation within the CPU

```
shifter shifter_inst (
    .value(shift_value),
    .func(shift_func),
    .carry_in(flags[`FLAG_C]),
    .result(shift_result),
    .carry_out(shift_carry)
);
```

C.6 The decode Module

The instruction decoder, defined in `verilog/rtl/decode.v`, is a purely combinational module that extracts fields from the 16-bit instruction word, computes immediate values, and generates all control signals needed by the CPU's state machine. It is the largest module by output count, with over 30 control signals.

C.6.1 Input Ports

Table C.11: Input ports of the decode module

Port	Width	Description
`instr`	16	Current instruction word
`imm16`	16	Second word for extended (32-bit) instructions

C.6.2 Extracted Field Outputs

These outputs are direct slices of the instruction word, assigned via continuous `assign` statements:

C.6.3 Computed Value Outputs

C.6.4 Memory Control Outputs

C.6.5 Control Flag Outputs

The decoder produces a large set of single-bit control flags. All default to 0 and are asserted only for their corresponding instruction patterns.

C.6.6 Instantiation Example

Table C.12: Decoded instruction fields

Port	Width	Source	Description
opcode	4	instr[15:12]	Primary opcode
rd	4	instr[11:8]	Destination register
rs1	4	instr[7:4]	Source register 1
rs2	4	instr[3:0]	Source register 2
func	4	instr[3:0]	Function code (alias for rs2)
ext_sub	4	instr[3:0]	Extended sub-opcode

Table C.13: Computed values generated by the decoder

Port	Width	Description
alu_op	4	ALU operation select
shift_func	4	Shift function (equals func)
branch_cond	4	Branch condition (equals rd field)
muldiv_func	4	Multiply/divide function (equals func)
imm8_sext	16	Sign-extended 8-bit immediate: {{8{instr[7]}}, instr[7:0]}
offset8	16	Branch offset: {{7{instr[7]}}, instr[7:0], 1'b0}
offset12	16	Jump offset: {{3{instr[11]}}, instr[11:0], 1'b0}
io_port	8	I/O port number

Listing C.6: Decoder instantiation (abbreviated)

```
decode decode_inst (
    .instr(instr),
    .imm16(imm16),
    .opcode(dec_opcode),
    .rd(dec_rd),
    .rs1(dec_rs1),
    .rs2(dec_rs2),
    .func(dec_func),
    .alu_op(dec_alu_op),
    .shift_func(dec_shift_func),
    .branch_cond(dec_branch_cond),
    .imm8_sext(dec_imm8_sext),
    .offset8(dec_offset8),
```

Table C.14: Memory control signals from the decoder

Port	Width	Description
mem_load	1	Memory load operation
mem_store	1	Memory store operation
mem_byte	1	Byte (not word) access
mem_signed	1	Sign-extend byte loads
mem_offset	16	Memory address offset added to base register

```
    .offset12(dec_offset12),
    .mem_load(dec_mem_load),
    .mem_store(dec_mem_store),
    .mem_byte(dec_mem_byte),
    .mem_signed(dec_mem_signed),
    .mem_offset(dec_mem_offset),
    .reg_write(dec_reg_write),
    .is_jump(dec_is_jump),
    .is_branch(dec_is_branch),
    .is_halt(dec_is_halt),
    // ... remaining control signals ...
    .ext_sub(dec_ext_sub)
);
```

C.7 The ram Module

The RAM module, defined in verilog/rtl/ram.v, implements a 64 KB synchronous memory organized as 32K×16-bit words. It supports both word and byte-level access through byte enable signals, and can optionally be initialized from a hex file at elaboration time.

C.7.1 Parameters

C.7.2 Port Table

C.7.3 Internal Organization

The memory array is declared as reg [15:0] mem [0:32767], giving 32,768 16-bit words (64 KB total). The address input is a byte address; word alignment is achieved by discarding the LSB:

```
wire [14:0] word_addr = addr[15:1];
```

Both reads and writes are synchronous, producing a 1-cycle latency. The byte enable signals control which bytes of the 16-bit word are written: when `be[0]` is asserted, the low byte (`mem[word_addr][7:0]`) is updated; when `be[1]` is asserted, the high byte (`mem[word_addr][15:8]`) is updated. Reads always return the full 16-bit word; byte extraction is performed by the CPU's memory result logic in the MEMORY state.

C.7.4 Instantiation Example

Listing C.7: RAM instantiation within the SoC

```
ram #(
    .INIT_FILE(RAM_INIT_FILE)
) ram_inst (
    .clk(clk),
    .addr(cpu_mem_addr),
    .wdata(cpu_mem_wdata),
    .rdata(cpu_mem_rdata),
    .we(cpu_mem_we),
    .be(cpu_mem_be),
    .en(cpu_mem_valid)
);
```

C.8 The `uart` Module

The UART module, defined in `verilog/rtl/uart.v`, provides an MC6850-compatible serial interface between the CPU's I/O bus and an external UART transceiver. It maps two I/O port addresses—a status register and a data register—into the CPU's I/O address space.

C.8.1 Port Table

C.8.2 I/O Port Map

The status register provides two bits of information:

C.8.3 TX and RX Handshake

Transmit path. When the CPU writes to `0x81`, the byte is latched into `tx_reg` and `tx_pending` is set. When `tx_ready` is high and `tx_valid` is low, the UART asserts `tx_valid` and drives `tx_data` with the pending byte, then clears `tx_pending`. The `tx_valid` signal is cleared once `tx_ready` has acknowledged the transfer.

Receive path. When `rx_valid` is asserted by the external interface and no data is pending, the byte is latched into `rx_reg` and `rx_pending` is set. A one-cycle pulse on `rx_ready` acknowledges the

transfer. When the CPU reads `0x81`, the latched byte is returned and `rx_pending` is cleared, making the module ready for the next incoming byte.

C.8.4 Instantiation Example

Listing C.8: UART instantiation within the SoC

```
uart uart_inst (
    .clk(clk),
    .rst_n(rst_n),
    .port(cpu_io_addr),
    .wdata(cpu_io_wdata),
    .rdata(cpu_io_rdata),
    .rd(cpu_io_rd),
    .wr(cpu_io_wr),
    .tx_data(tx_data),
    .tx_valid(tx_valid),
    .tx_ready(tx_ready),
    .rx_data(rx_data),
    .rx_valid(rx_valid),
    .rx_ready(rx_ready)
);
```

C.9 Module Hierarchy Summary

Table C.21 summarizes the complete module hierarchy, listing each module's source file, type (sequential or combinational), and primary role in the system.

Table C.15: Control flag outputs from the decoder

Port	Asserted When
`reg_write`	Result should be written to `Rd`
`is_jump`	PC-relative jump (J, JX, JALX)
`is_jump_reg`	Register-indirect jump (JR, JALR)
`is_branch`	Conditional branch instruction
`is_call`	Call instruction (JALR, JALX)
`is_ret`	Return instruction (JR `R1`)
`is_extended`	Extended (32-bit) instruction format
`is_halt`	**`HALT`** instruction
`is_nop`	**`NOP`** instruction
`is_exx`	**`EXX`** (exchange registers)
`is_push`	**`PUSH`** instruction
`is_pop`	**`POP`** instruction
`is_cmp`	**`CMP`** (compare, no writeback)
`is_test`	**`TEST`** (bitwise test, no writeback)
`is_getf`	**`GETF`** (read flags register)
`is_setf`	**`SETF`** (write flags register)
`is_ei`	Enable interrupts
`is_di`	Disable interrupts
`is_reti`	Return from interrupt
`is_scf`	Set carry flag
`is_ccf`	Complement carry flag
`is_io_in`	I/O input instruction
`is_io_out`	I/O output instruction
`io_port_imm`	I/O port number is an immediate (INI/OUTI)
`is_muldiv`	Multiply or divide instruction

Table C.16: Parameters of the `ram` module

Parameter	Default	Description
`INIT_FILE`	`""`	Path to a hex file; when non-empty, the memory array is initialized via `$readmemh` in an `initial` block

Table C.17: Ports of the `ram` module

Port	Dir	Width	Description
`clk`	input	1	System clock
`addr`	input	16	Byte address (LSB ignored for word alignment)
`wdata`	input	16	Write data
`rdata`	output reg	16	Read data (1-cycle latency)
`we`	input	1	Write enable
`be`	input	2	Byte enables: `[0]`=low byte, `[1]`=high byte
`en`	input	1	Enable (read or write)

Table C.18: Ports of the `uart` module

Port	Dir	Width	Description
Clock and reset			
`clk`	input	1	System clock
`rst_n`	input	1	Active-low reset
CPU I/O interface			
`port`	input	8	I/O port address from CPU
`wdata`	input	8	Write data from CPU
`rdata`	output reg	8	Read data to CPU
`rd`	input	1	Read strobe
`wr`	input	1	Write strobe
External UART interface			
`tx_data`	output reg	8	TX data to external interface
`tx_valid`	output reg	1	TX data valid
`tx_ready`	input	1	External TX ready
`rx_data`	input	8	RX data from external interface
`rx_valid`	input	1	External RX data valid
`rx_ready`	output reg	1	RX data accepted

Table C.19: UART I/O port addresses

Address	Name	Description
`0x80`	`PORT_STATUS`	Status register (read-only)
`0x81`	`PORT_DATA`	Data register (read/write)

Table C.20: UART status register bit fields

Bit	Name	Meaning
0	RDRF	Receive Data Register Full — set when received data is available
1	TDRE	Transmit Data Register Empty — set when the transmitter is ready to accept a new byte

Table C.21: Complete SAMPO Verilog module hierarchy

Module	Source File	Type	Role
`soc`	`soc.v`	Sequential	Top-level SoC wrapper
`cpu`	`cpu.v`	Sequential	FSM-based CPU core
`regfile`	`regfile.v`	Sequential	16×16 register file
`alu`	`alu.v`	Combinational	Arithmetic logic unit
`shifter`	`shifter.v`	Combinational	16-function barrel shifter
`decode`	`decode.v`	Combinational	Instruction decoder
`ram`	`ram.v`	Sequential	32K×16 synchronous RAM
`uart`	`uart.v`	Sequential	MC6850-compatible UART

Appendix D

Assembler Reference

The SAMPO assembler (`sasm`) is a two-pass assembler written in Rust. Source code resides in `sasm/`, organized into three modules: `lexer.rs` (tokenization), `parser.rs` (syntax analysis), and `codegen.rs` (binary emission). This appendix provides a complete reference for every mnemonic, directive, and syntactic form that `sasm` accepts.

Listing D.1: Basic assembler invocation

```
sasm input.s -o output.bin
```

If `-o` is omitted, the output filename is derived by replacing the `.s` or `.asm` extension with `.bin`.

D.1 Complete Mnemonic List

Every mnemonic recognized by `sasm` is listed below, grouped by category. The **Encoding** column shows the 16-bit instruction word as a hex template; register fields are shown as nibble placeholders (*d* for Rd, *a* for Rs1, *b* for Rs2). All instructions are 2 bytes unless noted otherwise.

D.1.1 Arithmetic

`ADDI` is a two-operand instruction: it adds the sign-extended 8-bit immediate to the destination register in place (`Rd = Rd + sext(imm8)`). The immediate range is -128 to $+127$. **`NEG`** is a pseudo-instruction that the assembler emits as `SUB Rd, R0, Rs`, exploiting the hardwired-zero register to produce the two's complement negation.

Mnemonic	Operands	Encoding	Bytes	Description
ADD	Rd, Rs1, Rs2	0x0dab	2	Rd = Rs1 + Rs2
SUB	Rd, Rs1, Rs2	0x1dab	2	Rd = Rs1 − Rs2
ADDI	Rd, imm8	0x5dii	2	Rd = Rd + sext(imm8)
NEG	Rd, Rs	0x1d0b	2	Emitted as **SUB** Rd, R0, Rs
CMP	Rd, Rs	0xCda2	2	Set flags from Rd − Rs

Table D.1: Arithmetic instructions

Mnemonic	Operands	Encoding	Bytes	Description
AND	Rd, Rs1, Rs2	0x2dab	2	Rd = Rs1 & Rs2
OR	Rd, Rs1, Rs2	0x3dab	2	Rd = Rs1 | Rs2
XOR	Rd, Rs1, Rs2	0x4dab	2	Rd = Rs1 ⊕ Rs2
NOT	Rd, Rs	0xFda4+0xFFFF	4	Emitted as **XORIX** Rd, Rs, 0xFFFF
TEST	Rd, Rs	0xCda3	2	Set flags from Rd & Rs

Table D.2: Logic instructions

D.1.2 Logic

NOT is a pseudo-instruction. Because the ISA has no dedicated NOT encoding, the assembler generates a 32-bit **XORIX** Rd, Rs, 0xFFFF instruction. XOR with all-ones inverts every bit.

D.1.3 Multiply/Divide

All multiply/divide instructions use opcode 0xB with a function code in bits [3:0]. They are all 2 bytes and take two register operands.

Mnemonic	Operands	Encoding	func	Description
MUL	Rd, Rs	0xBda0	0x0	Rd = (Rd × Rs)[15:0]
MULH	Rd, Rs	0xBda1	0x1	Rd = (Rd × Rs)[31:16] signed
MULHU	Rd, Rs	0xBda2	0x2	Rd = (Rd × Rs)[31:16] unsigned
DIV	Rd, Rs	0xBda3	0x3	Rd = Rd ÷ Rs (signed)
DIVU	Rd, Rs	0xBda4	0x4	Rd = Rd ÷ Rs (unsigned)
REM	Rd, Rs	0xBda5	0x5	Rd = Rd % Rs (signed)
REMU	Rd, Rs	0xBda6	0x6	Rd = Rd % Rs (unsigned)
DAA	Rd	0xBd07	0x7	BCD adjust Rd

Table D.3: Multiply, divide, and BCD instructions

D.1.4 Shift and Rotate

All shift/rotate instructions use opcode `0xA` and are 2 bytes. The assembler maps each mnemonic to the shift-by-1 variant.

Mnemonic	Operands	func	Description
SLL	Rd, Rs	`0x0`	Shift left logical by 1
SRL	Rd, Rs	`0x1`	Shift right logical by 1
SRA	Rd, Rs	`0x2`	Shift right arithmetic by 1
ROL	Rd, Rs	`0x3`	Rotate left by 1
ROR	Rd, Rs	`0x4`	Rotate right by 1
SWAP	Rd, Rs	`0x7`	Swap high/low bytes

Table D.4: Shift and rotate instructions (opcode `0xA`)

Fixed-Shift Variants

The ISA also defines **SLL4**/**SRL4**/**SRA4**/**ROL4** (shift by 4) and **SLL8**/**SRL8**/**SRA8**/**ROL8** (shift by 8) variants. These are valid machine encodings, but the assembler currently only supports the short mnemonic forms listed above (by-1 shifts). For variable shift amounts, use the extended **SLLX**, **SRLX**, or **SRAX** instructions (see Section D.1.11).

D.1.5 Load and Store

Mnemonic	Operands	Bytes	Description
LW	Rd, (Rs) *or* Rd, *off*(Rs)	2	Load word; offsets: 0, ±2, ±4, +6
LB	Rd, (Rs)	2	Load byte (sign-extended)
LBU	Rd, (Rs)	2	Load byte (zero-extended)
LUI	Rd, imm	2	Load upper immediate
SW	(Rs), Rd *or* *off*(Rs), Rd	2	Store word; same offsets as **LW**
SB	(Rs), Rd	2	Store byte

Table D.5: Load and store instructions

The base 16-bit load/store encoding supports a fixed set of offsets: 0, +2, +4, +6, −2, −4. These are encoded in the 4-bit function field. For arbitrary 16-bit offsets, use the extended **LWX** or **SWX** instructions.

D.1.6 Branch

All branch instructions use opcode `0x8`, are 2 bytes, and take a single operand: a label or immediate offset. The 4-bit condition code occupies bits [11:8]; the 8-bit signed offset occupies bits [7:0].

Mnemonic	Operand	cond	Condition
BEQ	label	0x0	Z = 1 (equal)
BNE	label	0x1	Z = 0 (not equal)
BLT	label	0x2	N ≠ V (signed less than)
BGE	label	0x3	N = V (signed greater/equal)
BLTU	label	0x4	C = 0 (unsigned less than)
BGEU	label	0x5	C = 1 (unsigned greater/equal)
BMI	label	0x6	N = 1 (minus / negative)
BPL	label	0x7	N = 0 (plus / positive)
BVS	label	0x8	V = 1 (overflow set)
BVC	label	0x9	V = 0 (overflow clear)
BCS	label	0xA	C = 1 (carry set)
BCC	label	0xB	C = 0 (carry clear)
BGT	label	0xC	Z = 0 and N = V (signed greater)
BLE	label	0xD	Z = 1 or N ≠ V (signed less/equal)
BHI	label	0xE	C = 1 and Z = 0 (unsigned higher)
BLS	label	0xF	C = 0 or Z = 1 (unsigned lower/same)

Table D.6: Branch condition codes

Range: The 8-bit signed offset is measured in words (2-byte units). The effective byte range relative to the instruction following the branch is $-128 \times 2 = -256$ bytes to $+127 \times 2 = +254$ bytes—roughly ± 128 instructions in each direction. The assembler reports an error if a fixup produces an out-of-range offset.

D.1.7 Jump

Mnemonic	Operands	Bytes	Description
J	label	2	PC-relative jump (± 4094 bytes)
JR	Rs	2	Jump to register
JALR	Rd, Rs	2	Jump and link register
JAL	label	4	Jump and link (uses **JALX** encoding)
JX	label/addr	4	Absolute jump
JALX	label/addr	4	Absolute jump and link

Table D.7: Jump instructions

J uses a 12-bit signed offset in word units, giving a range of ± 4094 bytes. **JR** sets bits [11:8] to 0xF to distinguish it from a relative jump within the same opcode (0x9). **JALR** writes the return address (PC + 2) to Rd and jumps to the address in Rs; the standard calling convention uses RA as Rd.

JAL always generates the 32-bit **JALX** encoding (opcode 0xF, sub = 0x9) with Rd hardcoded to R1 (RA). There is no 16-bit PC-relative call form.

D.1.8 Stack and Miscellaneous

All instructions in this group use opcode 0xC and are 2 bytes.

Mnemonic	Operands	func	Description
PUSH	Rs	0x0	Push register to stack
POP	Rd	0x1	Pop from stack to register
MOV	Rd, Rs	0x4	Copy register
LDI	—	0x5	Block load increment
LDD	—	0x6	Block load decrement
LDIR	—	0x7	Block load repeat increment
LDDR	—	0x8	Block load repeat decrement
CPIR	—	0x9	Compare and search
FILL	—	0xA	Fill memory
EXX	—	0xB	Exchange R4–R11 with alternates
GETF	Rd	0xC	Get FLAGS into register
SETF	Rs	0xD	Set FLAGS from register

Table D.8: Stack and miscellaneous instructions (opcode 0xC)

PUSH places the source register in the Rs1 field (bits [7:4]), while **POP** places the destination in the Rd field (bits [11:8]). Block operations (**LDI** through **FILL**) use implicit register operands: R4 (count), R5 (source), R6 (destination).

D.1.9 I/O

Mnemonic	Operands	Bytes	Description
IN	Rd, (Rs)	2	Input from register-addressed port
OUT	(Rs), Rd	2	Output to register-addressed port
INI	Rd, port8	4	Input from immediate port
OUTI	port8, Rs	4	Output to immediate port
INX	Rd, port8	4	Extended input (same encoding as **INI**)
OUTX	port8, Rs	4	Extended output (same encoding as **OUTI**)

Table D.9: I/O instructions

The 16-bit **IN**/**OUT** instructions address ports through a register (the register holds the port number). **INI** and **OUTI** accept an 8-bit immediate port number but are always encoded using the 32-bit extended format (opcode `0xF`, sub = `0xB`/`0xC`).

D.1.10 System

System instructions use opcode `0xE`. Most are fixed encodings with no operands.

Mnemonic	Operands	Bytes	Hex	Description
NOP	—	2	`0xE000`	No operation
HALT	—	2	`0xE100`	Halt until interrupt
DI	—	2	`0xE200`	Disable interrupts
EI	—	2	`0xE300`	Enable interrupts
RETI	—	2	`0xE400`	Return from interrupt
SWI	imm8	2	`0xE5ii`	Software interrupt
SCF	—	2	`0xE600`	Set carry flag
CCF	—	2	`0xE700`	Complement carry flag

Table D.10: System instructions (opcode `0xE`)

D.1.11 Extended (32-bit) Instructions

Extended instructions use opcode `0xF` in the first word, with a 4-bit sub-opcode in bits [3:0] selecting the operation. The second word carries a 16-bit immediate. All extended instructions are 4 bytes.

Assembler Support for Extended Mnemonics

The assembler currently implements code generation for **LIX**, **JX**, **JALX**, **INI**/**INX**, and **OUTI**/**OUTX**. The remaining extended mnemonics (**ADDIX**, **SUBIX**, **ANDIX**, **ORIX**, **XORIX**, **CMPIX**, **LWX**, **SWX**, **SLLX**, **SRLX**, **SRAX**) are recognized for instruction-size computation but are not yet wired into the code emission path. Attempting to assemble these mnemonics will produce an "Unknown instruction" error. **Workaround:** Encode the instruction manually using `.dw` directives. For example, **ADDIX** R4, R8, `0x1234` can be written as:

```
    .dw 0xF480, 0x1234  ; ADDIX R4, R8, 0x1234
```

D.2 Directive Reference

The assembler supports six directives. Directive names are case-insensitive and must be preceded by a dot.

Mnemonic	Operands	sub	Description
ADDIX	Rd, Rs, imm16	0x0	Extended add immediate
SUBIX	Rd, Rs, imm16	0x1	Extended subtract immediate
ANDIX	Rd, Rs, imm16	0x2	Extended AND immediate
ORIX	Rd, Rs, imm16	0x3	Extended OR immediate
XORIX	Rd, Rs, imm16	0x4	Extended XOR immediate
LWX	Rd, imm16(Rs)	0x5	Extended load word
SWX	imm16(Rs), Rd	0x6	Extended store word
LIX	Rd, imm16/label	0x7	Load 16-bit immediate
JX	label/addr	0x8	Absolute jump (see above)
JALX	label/addr	0x9	Absolute jump and link (see above)
CMPIX	Rd, imm16	0xA	Extended compare immediate
INX	Rd, port8	0xB	Extended input (see I/O above)
OUTX	port8, Rs	0xC	Extended output (see I/O above)
SLLX	Rd, Rs, imm4	0xD	Variable shift left logical
SRLX	Rd, Rs, imm4	0xE	Variable shift right logical
SRAX	Rd, Rs, imm4	0xF	Variable shift right arithmetic

Table D.11: Extended instruction sub-opcodes (opcode 0xF)

D.2.1 .org — Set Assembly Origin

Sets the program counter to the specified address. The output buffer is padded with zero bytes up to the new origin if needed.

Listing D.2: .org directive

```
    .org 0x0100          ; Start assembly at address 0x0100
start:
    NOP                  ; This instruction is placed at 0x0100
```

Multiple .org directives may appear in a single source file. Each one resets the program counter; labels defined after a .org receive the new base address.

D.2.2 .equ — Define Symbolic Constant

Inserts a name/value pair into the symbol table. The value can then be used wherever an immediate or label is expected.

Listing D.3: .equ directive

```
    .equ UART_DATA, 0x81 ; Define port constant
    .equ STACK_TOP, 0xFFFE
```

```
    LIX  SP, STACK_TOP   ; Use constant as immediate
    INI  R4, UART_DATA   ; Use constant as port number
```

.equ symbols are resolved during pass 1 and are available throughout the source file, including in code that textually precedes the .equ definition (because pass 2 runs after all symbols are collected).

D.2.3 .db — Define Bytes

Emits raw bytes into the output. Arguments may be numbers, string literals, or previously defined symbols (resolved to their low byte).

Listing D.4: .db directive

```
    .db 0x48, 0x65, 0x6C ; Three raw bytes: 'H', 'e', 'l'
    .db "Hello"          ; Five bytes from a string literal
    .db 0x0D, 0x0A, 0x00 ; CR, LF, NUL
```

D.2.4 .dw — Define Words

Emits 16-bit words in little-endian byte order. Arguments may be numbers or labels (forward references are supported via fixups).

Listing D.5: .dw directive

```
    .dw 0x1234           ; Emits bytes 0x34, 0x12
    .dw handler          ; Forward reference to label
    .dw 0xDEAD, 0xBEEF   ; Two words
```

The .dw directive is the standard mechanism for building jump tables and interrupt vector tables. Label references in .dw produce Absolute16 fixups that are resolved in pass 3.

D.2.5 .ascii — Define ASCII String

Emits the bytes of a string literal without any terminator.

Listing D.6: .ascii directive

```
    .ascii "Hello"       ; 5 bytes: 0x48 0x65 0x6C 0x6C 0x6F
```

D.2.6 .asciz — Define Null-Terminated String

Emits the bytes of a string literal followed by a null byte (0x00).

Listing D.7: `.asciz` directive

```
msg:
    .asciz "Hello"       ; 6 bytes: 0x48 0x65 0x6C 0x6C 0x6F 0x00
```

This is the most common directive for defining strings in SAMPO programs, since C-style null-terminated strings are used by UART output routines.

D.3 Expression Syntax

D.3.1 Numeric Literals

The assembler supports three number formats:

Format	Prefix	Examples
Decimal	(none)	`42`, `255`, `0`
Hexadecimal	`0x` or `0X`	`0x1234`, `0xFF`, `0xDEAD`
Binary	`0b` or `0B`	`0b1010`, `0b11110000`

Table D.12: Number formats

Negative numbers are supported with a leading minus sign: `-128`, `-1`. Underscores may appear within digit sequences as visual separators and are ignored: `0xFF_00`, `1_000`. Character literals are supported with single quotes: `'A'` produces the integer value 65.

D.3.2 Labels

Labels are defined by placing an identifier followed by a colon on its own line:

```
loop:                    ; Define label 'loop' at current PC
    ADDI R4, -1
    BNE  loop            ; Reference label as branch target
```

Forward references are fully supported—a label may be referenced before it is defined. The assembler records fixups during pass 2 and resolves them in pass 3.

Labels are **case-sensitive**: `Loop` and `loop` are distinct symbols.

D.3.3 Register Names

Register names are case-insensitive. Both numeric names and conventional aliases are accepted:

Number	Alias	Convention
R0	ZERO	Hardwired zero
R1	RA	Return address
R2	SP	Stack pointer
R3	GP	Global pointer
R4	A0	Argument 0 / return value
R5	A1	Argument 1 / return value high
R6	A2	Argument 2
R7	A3	Argument 3
R8	T0	Temporary 0
R9	T1	Temporary 1
R10	T2	Temporary 2
R11	T3	Temporary 3
R12	S0	Saved 0
R13	S1	Saved 1
R14	S2	Saved 2
R15	S3	Saved 3

Table D.13: Register names and aliases

FP Alias

The FP (frame pointer) alias, commonly mapped to GP (R3), is referenced in the ISA specification but is **not recognized** by the current assembler. Use R3 or GP instead.

D.3.4 Operand Formats

Register
: A register name: R4, SP, A0.

Immediate
: A numeric literal or symbolic constant: 42, 0xFF, -1, UART_DATA.

Label
: A symbolic name that resolves to an address: loop, done, msg.

Indirect
: A register enclosed in parentheses: (R5), (SP). Computes the memory address from the register value.

Indirect with offset
: A numeric offset followed by a parenthesized register: `2(R5)`, `-4(SP)`. Computes address = register + offset. Offsets may also appear inside the parentheses: `(R5 + 2)`, `(SP - 4)`.

D.3.5 Comments

Comments begin with a semicolon (`;`) and extend to the end of the line:

```
ADD R4, R5, R6       ; This is a comment
; This entire line is a comment
```

D.3.6 Case Sensitivity

- **Mnemonics:** Case-insensitive. `ADD`, `add`, and `Add` are all equivalent.
- **Register names:** Case-insensitive. `R4`, `r4`, `SP`, `sp` are all equivalent.
- **Labels:** Case-sensitive. `Loop` and `loop` are distinct labels.
- **Directives:** Case-insensitive. `.org` and `.ORG` are equivalent (directives are lowercased during lexing).

D.3.7 String Escape Sequences

String literals in `.ascii`, `.asciz`, and `.db` directives support the following escape sequences:

Escape	Meaning
`\n`	Newline (LF, `0x0A`)
`\r`	Carriage return (CR, `0x0D`)
`\t`	Tab (`0x09`)
`\0`	Null (`0x00`)
`\\`	Literal backslash
`\"`	Literal double quote

Table D.14: String escape sequences

D.4 Two-Pass Assembly Process

The assembler uses a three-step process (commonly called "two-pass assembly" with a fixup phase):

Pass 1: Label Collection
: Walk the statement list, tracking the program counter. Record every label and its address in

the symbol table. Process `.equ` directives. Advance the PC by the correct number of bytes for each instruction and data directive, but do not emit any bytes.

Pass 2: Code Generation
Walk the statement list again, emitting bytes for each instruction and directive. When a label reference cannot be resolved at this point (forward references in branches, jumps, or `.dw`), emit placeholder bytes and record a fixup.

Pass 3: Fixup Application
Walk the fixup list and patch placeholder bytes with the correct values. Three fixup types handle all cases:

Fixup Type	Usage
`Absolute16`	Full 16-bit address. Used by **`LIX`**, **`JX`**, **`JALX`**, **`JAL`**, and `.dw` with label arguments.
`Relative8`	Signed 8-bit word offset. Used by all conditional branch instructions. Computed as (target − PC_{after}) / 2.
`Relative12`	Signed 12-bit word offset. Used by the **`J`** instruction. Computed as (target − PC_{after}) / 2.

Table D.15: Fixup types

The internal state of the code generator is held in a Rust struct:

Listing D.8: Code generator state

```
pub struct CodeGen {
    origin: u16,                    // Starting address (.org)
    pc: u16,                        // Current program counter
    symbols: HashMap<String, u16>,  // Symbol table
    output: Vec<u8>,                // Binary output buffer
    fixups: Vec<Fixup>,             // Deferred patches
}
```

D.5 Known Limitations and Workarounds

The SAMPO assembler is a minimal, purpose-built tool. The following limitations are worth understanding when writing assembly programs.

D.5.1 INI/OUTI Always Emit Extended Format (4 bytes)

The assembler encodes **INI** Rd, port and **OUTI** port, Rs using the 32-bit **INX**/**OUTX** encoding (opcode 0xF, sub = 0xB/0xC) with the port number in the 16-bit immediate word. Even port numbers that would fit in 4 bits still use the 4-byte encoding. The 16-bit **IN**/**OUT** encoding with inline 4-bit port numbers is not generated by **INI**/**OUTI**.

Workaround: If code size is critical, load the port number into a register and use the 2-byte **IN**/**OUT** form:

```
; Instead of: INI R4, 0x81      (4 bytes)
LIX  R8, 0x81                   ; 4 bytes (one-time setup)
IN   R4, (R8)                   ; 2 bytes per use
```

D.5.2 NOT Emitted as XORIX (4 bytes)

NOT Rd, Rs generates the 32-bit **XORIX** Rd, Rs, 0xFFFF instruction. This works correctly but costs 4 bytes instead of the 2 that a native NOT encoding would require.

Workaround: If a register already holds 0xFFFF, use the base-format **XOR**:

```
LIX  T0, 0xFFFF                 ; One-time setup
XOR  R4, R5, T0                 ; 2-byte NOT equivalent
```

D.5.3 NEG Emitted as SUB

NEG Rd, Rs generates **SUB** Rd, R0, Rs (0x1d0b). This is correct—subtracting from the hardwired-zero register produces two's complement negation—and costs only 2 bytes. The only subtlety is that R0 is placed in the Rs1 field; disassemblers must recognize this pattern.

D.5.4 No Expression Evaluation

The assembler does not support arithmetic expressions in operands. Constructs such as `label + 2` or `UART_BASE + 1` will produce a parse error.

No Arithmetic in Operands

The following code will **not** assemble:

```
    .equ UART_BASE, 0x80
    INI  R4, UART_BASE + 1   ; ERROR: expression not supported
```

Workaround: Precompute the value with a separate `.equ`:

```
    .equ UART_BASE, 0x80
    .equ UART_DATA, 0x81     ; Precomputed
    INI  R4, UART_DATA       ; OK
```

D.5.5 Limited Shift Mnemonics

The assembler supports **SLL**, **SRL**, **SRA**, **ROL**, **ROR** for by-1 shifts and **SWAP** for byte swapping. The by-4 variants (**SLL4**, **SRL4**, **SRA4**, **ROL4**) and by-8 variants (**SLL8**, **SRL8**, **SRA8**, **ROL8**) exist in the ISA but are not mapped to assembler mnemonics.

Workaround: Encode the fixed-shift variants manually with `.dw`, or use **SLLX**/**SRLX**/**SRAX** with an immediate shift amount (noting the caveat in the next item that these extended mnemonics also require manual encoding).

D.5.6 Many Extended Mnemonics Not Yet Implemented

As noted in Section D.1.11, the mnemonics **ADDIX**, **SUBIX**, **ANDIX**, **ORIX**, **XORIX**, **CMPIX**, **LWX**, **SWX**, **SLLX**, **SRLX**, and **SRAX** are recognized for instruction-size calculation (so that label addresses remain correct) but will produce an "Unknown instruction" error during code generation.

Workaround: Encode these instructions manually using `.dw`. The encoding is straightforward—compute the first word as `0xF`*das* (where *d* = Rd, *a* = Rs, *s* = sub-opcode from Table D.11), and the second word is the 16-bit immediate:

```
    ; SLLX R4, R5, 3 -> sub=0xD, Rd=4, Rs=5
    ; Word 0: 0xF45D  Word 1: 0x0003
    .dw 0xF45D, 0x0003
```

D.5.7 No Macro Support

The assembler does not support macro definitions or macro expansion. Repeated code patterns must be written out manually or generated by an external preprocessor.

Workaround: Use an external text preprocessor (such as `m4` or a simple Python script) to generate repetitive assembly code before passing it to `sasm`.

D.5.8 No .include Directive

Source files cannot include other files. All code must reside in a single source file.

Workaround: Concatenate source files before assembly:

Listing D.9: Concatenating source files

```
cat header.s main.s data.s > combined.s
sasm combined.s -o output.bin
```

D.5.9 JAL Uses JALX Encoding (4 bytes)

JAL label generates the 32-bit **JALX** encoding (opcode `0xF`, sub = `0x9`) with `Rd` hardcoded to `R1` (`RA`). This uses absolute addressing, not PC-relative, and always costs 4 bytes.

Workaround: For short-range calls where code size matters, manually construct a PC-relative call using **JALR**:

```
    ; Short-range call pattern (6 bytes total)
    LIX  RA, return_addr          ; 4 bytes
    JR   R8                       ; 2 bytes (R8 holds target)
return_addr:
```

In practice, the 4-byte **JAL** is acceptable for most programs within the 64 KB address space.

D.5.10 Branch Range Checking

Branch targets must be within -256 to $+254$ bytes of the instruction following the branch. The assembler reports an error if a fixup produces an out-of-range offset during pass 3. For forward references, this means the error is only detected after all labels have been collected.

Out-of-Range Branches

If a branch target is too far away, the assembler will report:
`Branch to <label> out of range`
Workaround: Replace the out-of-range branch with an inverted short branch around an absolute jump:

```
    ; Instead of: BEQ far_target    (out of range)
    BNE  skip                      ; Invert condition, branch over
    JX   far_target                ; 4-byte absolute jump
skip:
```

D.5.11 Duplicate Labels Silently Overwritten

If two labels have the same name, the second definition silently overwrites the first in the symbol table during pass 1. The assembler does not report a warning or error. This can lead to subtle bugs where branches and jumps go to unexpected locations.

Workaround: Use unique, descriptive label names. Prefix labels with the function name or module name to avoid collisions (e.g., `uart_init_loop:` rather than `loop:`).

D.6 Complete Assembly Example

The following example demonstrates most of the assembler's features in a single program:

Listing D.10: Complete assembly example

```
; Hello World for Sampo CPU
; Prints a message via UART, then halts

    .org 0x0100

    .equ UART_STATUS, 0x80
    .equ UART_DATA,   0x81
    .equ TDRE_BIT,    0x02

start:
    LIX  SP, 0xFFFE            ; Initialize stack pointer
    LIX  A0, msg               ; Load message address
    JAL  print_string          ; Call subroutine
    HALT                       ; Done

; -- Subroutine: print null-terminated string --
```

```
; Input:  A0 = pointer to string
; Clobbers: A0, A1, T0
print_string:
    PUSH RA                     ; Save return address
print_loop:
    LB   A1, (A0)               ; Load next character
    CMP  A1, ZERO               ; Is it null?
    BEQ  print_done             ; Yes -> return
    LIX  T0, UART_STATUS
tx_wait:
    IN   T0, (T0)               ; Read UART status
    ADDI T0, 0                  ; Test it (flags set)
    BEQ  tx_wait                ; Wait if not ready
    LIX  T0, UART_DATA
    OUT  (T0), A1               ; Send character
    ADDI A0, 2                  ; Advance pointer
    J    print_loop
print_done:
    POP  RA                     ; Restore return address
    JR   RA                     ; Return

; -- Data section --------------------------------
msg:
    .asciz "Hello, Sampo!\r\n"
```

This program exercises labels, forward references (`print_string` is called before it is defined), directives (`.org`, `.equ`, `.asciz`), register aliases (`SP`, `A0`, `RA`, `ZERO`), indirect addressing, branches, jumps, and subroutine calling conventions.

Appendix E

Glossary

This appendix provides an alphabetical glossary of terms used throughout *The SAMPO CPU Handbook*. Definitions are given in the context of the SAMPO architecture and its toolchain, though most terms have broader applicability in computer engineering. The glossary is divided into two sections for ease of navigation: Terms A–M and Terms N–Z.

E.1 Terms A–M

ACIA
: Asynchronous Communications Interface Adapter. The SAMPO SoC includes an MC6850-compatible ACIA providing serial communication. It is accessible via I/O ports `0x80` (status register) and `0x81` (data register), and supports the RDRF and TDRE status bits for receive/-transmit handshaking.

Address Bus
: The set of 16 signal lines that carry memory addresses from the SAMPO CPU to the memory subsystem. In SAMPO, the address bus is 16 bits wide, providing access to a 64 KB address space.

ALU
: Arithmetic Logic Unit. A purely combinational circuit within the CPU that performs arithmetic operations (addition, subtraction, negation), logical operations (AND, OR, XOR, NOT), and multiplication/division. In the SAMPO Verilog implementation, the ALU is instantiated as the `alu.v` module and produces result data along with N, Z, C, and V flag outputs.

Alternate Register Set
: Shadow copies of registers `R4`–`R11`, stored separately from the main register file. The **`EXX`** instruc-

tion atomically swaps the primary and alternate copies, enabling fast context switching—for example, in interrupt handlers that need working registers without saving to the stack.

Always Block
A Verilog procedural block introduced by the `always` keyword, triggered by events in its sensitivity list. In SAMPO's RTL, combinational logic uses `always @(*)` blocks, while sequential logic uses `always @(posedge clk or negedge rst_n)` blocks to synchronize state updates to the clock edge.

Amaranth HDL
A Python-based hardware description language, formerly known as nMigen. Amaranth allows designers to describe digital circuits using Python syntax and generate Verilog or RTLIL netlists for synthesis. The SAMPO project includes an Amaranth implementation as an alternative to the hand-written Verilog RTL.

ARM-style Carry
A carry flag convention used by SAMPO in which the C flag is set to 1 after a subtraction when no borrow occurred (i.e., when the unsigned result is non-negative). This is the opposite of the Z80 convention and matches the ARM architecture. After **SUB**, $\mathrm{C} = 1$ means $a \geq b$ for unsigned operands.

ASCII
American Standard Code for Information Interchange. A 7-bit character encoding standard mapping integers `0x00`–`0x7F` to control codes, digits, letters, and symbols. Serial output from the SAMPO UART transmits ASCII-encoded characters.

Assembler
A program that translates assembly language source code into machine code binary. The SAMPO assembler is a two-pass assembler: the first pass builds a symbol table of labels and their addresses, and the second pass emits machine code, resolving forward references via fixups.

Assembly Language
A human-readable, low-level programming language in which each statement typically corresponds to a single machine instruction. SAMPO assembly uses mnemonics such as **ADD**, **LW**, and **BEQ**, along with register names (`R0`–`R15`) and assembler directives (`.org`, `.db`, `.asciz`).

Barrel Shifter
A combinational circuit capable of shifting or rotating a data word by any specified amount in a single clock cycle. The SAMPO shifter module (`shifter.v`) supports logical shifts, arithmetic shifts, and rotates by 1, 4, or 8 bit positions, as well as variable-amount shifts via the extended **SLLX**/**SRLX**/**SRAX** instructions.

BCD
Binary-Coded Decimal. An encoding in which each decimal digit (0–9) is represented by a 4-bit

nibble. The SAMPO ISA provides the **DAA** (Decimal Adjust Accumulator) instruction to correct the result of a preceding addition or subtraction for BCD arithmetic, using the H (half-carry) and C flags.

Big-Endian
A byte-ordering convention in which the most significant byte of a multi-byte value is stored at the lowest memory address. SAMPO does *not* use big-endian; it uses little-endian byte order.

Bitstream
The binary configuration file loaded into an FPGA to define its logic functions and routing connections. For the SAMPO FPGA target (Lattice ECP5 on the ULX3S board), the bitstream is produced by the ecppack tool after place-and-route completes.

Block Operations
Instructions that operate on contiguous blocks of memory using fixed register conventions: R4 (count), R5 (source address), and R6 (destination address). SAMPO block instructions include **LDI**, **LDD**, **LDIR**, **LDDR**, **CPIR**, and **FILL**, inspired by Z80 block transfer instructions.

Boolean Algebra
The mathematical system for reasoning about logic operations using the operators AND, OR, and NOT (along with derived operators such as XOR, NAND, and NOR). Boolean algebra underpins the design of all combinational logic in the SAMPO CPU.

Branch
A conditional change in program flow. SAMPO branches are PC-relative with an 8-bit signed offset, encoded with opcode 0x8 and a 4-bit condition code. The target address is PC + 2 + sign_extend(offset8), giving a range of −256 to +254 bytes.

Branch Prediction
A hardware technique used in pipelined processors to speculatively guess the outcome of a branch before it is resolved. The SAMPO CPU does not implement branch prediction, as its FSM-based design resolves each branch before fetching the next instruction.

BRAM
Block RAM. Dedicated, synchronous memory blocks embedded within an FPGA fabric. On the Lattice ECP5, each BRAM block is 18 Kbit and can be configured in various width/depth combinations. The SAMPO SoC's RAM is mapped to BRAM resources during FPGA synthesis.

Bus
A shared set of signal lines used for data transfer between components. The SAMPO SoC uses a memory bus (16-bit address, 16-bit data, write-enable, byte-enable, valid, and ready signals) and an I/O bus (8-bit address, 8-bit data, read and write strobes).

Byte
A unit of data consisting of 8 bits. In the SAMPO architecture, memory is byte-addressable, meaning each byte has a unique 16-bit address. A SAMPO word consists of two bytes.

Byte Enable
A 2-bit signal (`mem_be`) on the SAMPO memory interface indicating which bytes within a 16-bit word are active during a write operation. A value of `01` selects the low byte, `10` selects the high byte, and `11` selects both bytes (full word).

Cache
A small, fast memory placed between the CPU and main memory to reduce average access latency. The SAMPO CPU does not include a cache; all memory accesses go directly to the synchronous RAM in the SoC, which responds in a single clock cycle of latency.

Callee-Saved
Registers that a called function must preserve and restore before returning. In the SAMPO calling convention, registers `R12`–`R15` (`S0`–`S3`) and the stack pointer `SP` are callee-saved.

Caller-Saved
Registers that may be freely modified by a called function; the caller must save them before a call if their values are needed afterward. In SAMPO, registers `R4`–`R11` (`A0`–`A3`, `T0`–`T3`) are caller-saved.

Calling Convention
The agreed-upon rules governing how functions receive arguments, return values, and manage the stack. The SAMPO calling convention passes up to four arguments in `R4`–`R7`, returns values in `R4`–`R5`, saves the return address in `R1` (`RA`), and requires callees to preserve `R12`–`R15`.

Carry Flag (C)
Bit 5 of the SAMPO FLAGS register. After addition, C is set if an unsigned overflow (carry out of bit 15) occurred. After subtraction, C is set if there was *no* borrow (ARM-style convention), meaning C = 1 indicates $a \geq b$ for unsigned operands.

CISC
Complex Instruction Set Computer. An architectural philosophy featuring instructions of varying length and complexity, often with many addressing modes. The Z80 is a classic CISC design. SAMPO draws some CISC-inspired features (block operations, port I/O, BCD support) while maintaining a predominantly RISC design.

Clock
A periodic digital signal that synchronizes all sequential logic in the SAMPO CPU. All flip-flops and registers update on the positive edge of the clock signal (`posedge clk`). The SAMPO SoC also uses an active-low asynchronous reset (`rst_n`).

Combinational Logic
Digital logic whose outputs are determined solely by the current input values, with no internal state or memory. Examples in SAMPO include the ALU, the instruction decoder, and the branch condition evaluator. Contrast with *sequential logic*.

Condition Code
A 4-bit field in SAMPO branch instructions (bits [11:8]) that selects which combination of flags to test. The 16 condition codes include **`BEQ`** (`0x0`), **`BNE`** (`0x1`), **`BLT`** (`0x2`), and others, providing signed, unsigned, and flag-specific branch conditions.

Control Unit
The portion of the CPU responsible for decoding the current instruction and generating the control signals that direct the datapath. In SAMPO, the control unit is implemented as a finite state machine with states including RESET, FETCH, DECODE, FETCH_EXT, EXECUTE, MEMORY, WRITEBACK, and HALTED.

Counter
A sequential circuit that increments (or decrements) a stored value on each clock cycle. The SAMPO CPU includes a 32-bit cycle counter (`cycle_count`) that increments every clock cycle and is exposed via the `cycles_out` port for performance measurement.

CPI
Cycles Per Instruction. The average number of clock cycles required to execute one instruction. The SAMPO CPU's CPI varies by instruction class: simple register-register operations require fewer cycles than memory operations or extended (32-bit) instructions, due to the multi-cycle FSM design.

Cross-Assembler
An assembler that runs on one platform (e.g., a desktop PC running Linux or macOS) to produce machine code for a different target architecture (e.g., the SAMPO CPU). The SAMPO assembler is a cross-assembler written in Rust.

Cycle Count
The total number of clock cycles elapsed since the last reset. The SAMPO CPU maintains a 32-bit cycle counter that can be read through the `cycles_out` output port, useful for benchmarking and performance analysis.

DAA
Decimal Adjust Accumulator. An instruction (opcode `0xB`, func `0x7`) that corrects the result of a preceding binary addition or subtraction to produce a valid BCD result. DAA examines the N flag (to distinguish addition from subtraction), and the H and C flags to determine the correction values to apply.

Data Bus
The set of signal lines that carry data between the CPU and the memory or I/O subsystem. The SAMPO memory data bus is 16 bits wide (`mem_rdata` for reads, `mem_wdata` for writes), while the I/O data bus is 8 bits wide (`io_rdata`, `io_wdata`).

Decoder
A combinational circuit or module that interprets the bit fields of an instruction and generates control signals for the rest of the CPU. In SAMPO, the decoder is implemented as the `decode.v` module, which extracts opcodes, register addresses, immediates, and instruction type flags from the 16-bit instruction word.

Directive
An assembler command that controls the assembly process but does not directly produce machine instructions. SAMPO assembler directives include `.org` (set origin address), `.equ` (define constant), `.db` (define byte), `.dw` (define word), `.ascii`, and `.asciz`.

ECP5
A family of FPGAs manufactured by Lattice Semiconductor, notable for support by fully open-source toolchains (Yosys for synthesis, nextpnr for place-and-route). The SAMPO FPGA implementation targets the ECP5-based ULX3S development board.

Elaboration
The process of converting a hierarchical HDL design into a flat netlist of primitive logic elements (gates, flip-flops, memories). During elaboration, module instantiations are expanded, parameters are resolved, and generate blocks are unrolled. In the SAMPO flow, Yosys performs elaboration as the first step of synthesis.

Endianness
The convention governing the byte ordering of multi-byte values in memory. SAMPO uses little-endian byte order: the least significant byte of a 16-bit word is stored at the lower memory address. See also *big-endian* and *little-endian*.

EXX
The exchange instruction (opcode `0xC`, func `0xB`) that atomically swaps the contents of registers `R4`–`R11` with their alternate copies `R4'`–`R11'`. Registers `R0`–`R3` and `R12`–`R15` are not affected. Commonly used for fast context switching in interrupt handlers.

Fixup
A deferred address resolution in the assembler for forward references. When the assembler encounters a label that has not yet been defined (e.g., a branch target later in the source), it records a fixup entry. During the second pass, fixups are resolved by patching the correct addresses into the generated machine code.

Flag
A single-bit status indicator within the FLAGS register. The SAMPO CPU maintains six flags: N (negative), Z (zero), C (carry), V (overflow), H (half-carry), and I (interrupt enable). Flags are set or cleared by arithmetic, logical, and shift instructions.

FLAGS Register
An 8-bit CPU register holding the processor status flags. The bit layout is: N (bit 7), Z (bit 6), C (bit 5), V (bit 4), H (bit 3), I (bit 2), with bits 1–0 reserved. The FLAGS register can be read with `GETF` and written with `SETF`.

Flip-Flop
A fundamental sequential logic element that stores one bit of data, updating its output on the active edge of the clock signal. All registers in the SAMPO CPU—including the program counter, instruction register, flags, and the register file—are implemented using edge-triggered flip-flops (positive-edge clocked).

FPGA
Field-Programmable Gate Array. A reconfigurable integrated circuit containing an array of configurable logic blocks, routing resources, and embedded memory (BRAM). The SAMPO CPU is synthesized and deployed on a Lattice ECP5 FPGA via the ULX3S development board.

FSM
Finite State Machine. A sequential circuit model with a finite number of defined states and transitions between them. The SAMPO CPU uses an 8-state FSM: RESET, FETCH, DECODE, FETCH_EXT, EXECUTE, MEMORY, WRITEBACK, and HALTED.

Full Adder
A combinational circuit that computes the sum and carry-out of three single-bit inputs (A, B, and carry-in). Full adders are the building blocks of the ripple-carry or carry-lookahead adders used within the SAMPO ALU to perform 16-bit addition and subtraction.

Gate
A basic digital logic element that implements a single Boolean function. Common gates include AND, OR, NOT, XOR, NAND, and NOR. In an FPGA implementation such as SAMPO's, gate-level functions are mapped to look-up tables (LUTs) during synthesis.

Global Pointer (GP)
Register `R3` in the SAMPO architecture, optionally used to hold the base address of a global data section for efficient access to global variables. Its use is a software convention and is not enforced by hardware.

GTKWave
An open-source waveform viewer used to display simulation results stored in VCD (Value Change

Dump) files. In the SAMPO development workflow, GTKWave is used to inspect signal traces produced by Icarus Verilog testbench simulations.

Half-Carry (H)
Bit 3 of the SAMPO FLAGS register. Set when a carry occurs from bit 3 to bit 4 during an addition (or a borrow from bit 4 to bit 3 during subtraction). The half-carry flag is primarily used by the **DAA** instruction for BCD arithmetic correction.

HALT
A system instruction (opcode `0xE`, func `0x1`) that stops CPU execution. The processor enters the HALTED state and remains there until an interrupt is received (if interrupts are enabled) or the CPU is reset.

Handshake
A protocol using paired valid/ready signals to synchronize data transfer between two modules. The SAMPO memory interface uses a handshake: the CPU asserts `mem_valid` to request an access, and the RAM asserts `mem_ready` one cycle later to indicate the data is available or the write has been accepted.

Harvard Architecture
A computer architecture with physically separate memories (and buses) for instructions and data. SAMPO does *not* use a Harvard architecture; it uses a von Neumann architecture with a single unified memory space for both instructions and data.

HDL
Hardware Description Language. A textual language used to describe the structure and behavior of digital circuits. The SAMPO project uses two HDLs: Verilog (IEEE 1364) for the primary RTL implementation and Amaranth (Python-based) for an alternative implementation.

Hex File
A text file containing hexadecimal values used to initialize memory contents. The SAMPO assembler outputs hex files that are loaded into the SoC's RAM via the Verilog `$readmemh` system task during simulation, or compiled into BRAM initialization data for FPGA synthesis.

I/O Port
An 8-bit addressed interface for communicating with peripherals, accessed via the **IN** and **OUT** instruction families. SAMPO provides a 256-port address space (ports `0x00`–`0xFF`), separate from the memory address space. The UART occupies ports `0x80` (status) and `0x81` (data).

Immediate
A constant value encoded directly within an instruction word, rather than being read from a register or memory. SAMPO supports 8-bit signed immediates in Format I instructions (e.g., **ADDI**) and 16-bit immediates in Format X extended instructions (e.g., **LIX**, **ADDIX**).

Index Register
A register whose value is used as a base or offset address for memory access. In SAMPO, any general-purpose register can serve as an index register in load/store instructions (e.g., **`LW Rd, (Rs1)`** uses `Rs1` as the index).

Instruction Register (IR)
An internal CPU register that holds the 16-bit instruction currently being decoded and executed. In the SAMPO Verilog implementation, this is the `instr` register, latched from `mem_rdata` during the FETCH state.

Interrupt
An asynchronous event that causes the CPU to suspend its current execution and transfer control to an interrupt handler at a predefined vector address. SAMPO supports a single maskable interrupt with a vector at address `0x0004`, controlled by the I flag in the FLAGS register.

Interrupt Enable (I)
Bit 2 of the FLAGS register. When set, maskable interrupts are accepted by the CPU. The **`EI`** instruction sets this flag, **`DI`** clears it, and interrupt acceptance automatically clears it (to prevent nested interrupts). **`RETI`** restores it.

Interrupt Vector
A fixed memory address that contains (or serves as) the entry point for an interrupt handler. The SAMPO maskable interrupt vector is located at address `0x0004`. When an interrupt is accepted, the PC is loaded with the value from this address.

ISA
Instruction Set Architecture. The specification defining the interface between software and hardware: the set of instructions, registers, addressing modes, and data types that a processor supports. The SAMPO ISA is a 16-bit RISC design with CISC-inspired extensions, featuring 16 registers, 16 primary opcodes, and 16/32-bit instruction encoding.

Iverilog
Icarus Verilog. An open-source Verilog simulator used in the SAMPO project to compile and simulate the RTL design and testbenches. Testbench simulations produce VCD waveform files viewable in GTKWave.

Jump
An unconditional change in program flow. SAMPO provides several jump forms: PC-relative (**`J`** with a 12-bit offset), register-indirect (**`JR`**), register-indirect with link (**`JALR`**), and extended absolute (**`JX`**, **`JALX`**). Jump instructions do not test flags.

Label
A symbolic name assigned to a memory address in assembly language source code. Labels serve as

targets for branches and jumps, and as names for data locations. The SAMPO assembler records labels in a symbol table during the first pass and resolves references during the second pass.

Latch
A level-sensitive storage element that is transparent when its enable signal is active. Latches are generally avoided in synchronous designs like SAMPO because they can cause timing hazards; all storage in SAMPO uses edge-triggered flip-flops instead.

LED
Light-Emitting Diode. On the ULX3S FPGA board, eight LEDs are used as status indicators. In the SAMPO SoC, LED[7] indicates the CPU is halted, LED[6] shows UART TX activity, LED[5] shows UART RX data availability, and LED[4:0] display the lower bits of the program counter.

Little-Endian
A byte-ordering convention in which the least significant byte of a multi-byte value is stored at the lowest memory address. SAMPO uses little-endian byte order for all 16-bit word accesses.

Load
An instruction that reads data from memory into a CPU register. SAMPO load instructions include **`LW`** (load word), **`LB`** (load byte with sign extension), **`LBU`** (load byte with zero extension), and the extended **`LWX`** and **`LIX`**.

Load/Store Architecture
A CPU design principle in which only dedicated load and store instructions access memory; all arithmetic and logical operations work exclusively on registers. SAMPO follows this principle, with the addition of Z80-inspired block memory operations (**`LDIR`**, **`LDDR`**, etc.).

Logical Shift
A shift operation that fills vacated bit positions with zeros. In SAMPO, **`SLL`** (shift left logical) shifts bits toward the MSB and fills with zeros from the right, while **`SRL`** (shift right logical) shifts bits toward the LSB and fills with zeros from the left.

LUT
Look-Up Table. The basic configurable logic element within an FPGA. A k-input LUT can implement any Boolean function of k variables by storing the function's truth table in a small SRAM. The Lattice ECP5 uses 4-input LUTs. During synthesis, the SAMPO design's combinational logic is mapped to LUT configurations.

Machine Code
The binary encoding of instructions directly executed by the CPU. SAMPO machine code consists of 16-bit instruction words (with optional 16-bit immediate extensions for 32-bit instructions). Each instruction encodes the opcode, register operands, and immediate values according to the ISA's format specifications.

MC6850
A Motorola ACIA (Asynchronous Communications Interface Adapter) chip widely used in vintage computer systems. The SAMPO UART module (`uart.v`) emulates the MC6850's register interface, providing a status register (with RDRF and TDRE bits) at port `0x80` and a data register at port `0x81`.

Memory Map
The assignment of address ranges to different memory regions and peripherals. The suggested SAMPO memory map allocates `0x0000–0x00FF` for interrupt vectors, `0x0100–0x7FFF` for program ROM/code, `0x8000–0xFEFF` for RAM, and `0xFF00–0xFFFF` for memory-mapped I/O.

Memory-Mapped I/O
A technique for accessing peripheral devices through regular memory addresses rather than dedicated I/O ports. While SAMPO primarily uses port-based I/O (Z80-style), the upper 256 bytes of the address space (`0xFF00–0xFFFF`) are reserved for optional memory-mapped peripherals.

Mnemonic
A short, human-readable abbreviation representing a machine instruction. Examples in SAMPO assembly include **`ADD`** (add), **`LW`** (load word), **`BEQ`** (branch if equal), and **`HALT`** (halt processor). Mnemonics are translated to binary opcodes by the assembler.

Module
The fundamental design unit in Verilog, encapsulating a piece of logic with defined input and output ports. The SAMPO Verilog design is composed of modules including `cpu.v`, `alu.v`, `regfile.v`, `shifter.v`, `decode.v`, `uart.v`, `ram.v`, and the top-level `soc.v`.

Multiplexer (Mux)
A combinational circuit that selects one of several input signals and forwards it to a single output, based on a select signal. Multiplexers are used extensively in the SAMPO datapath—for example, to select the ALU's input sources, the register file write-data source, and the memory address source depending on the current instruction and FSM state.

Multiply/Divide Unit
The hardware block responsible for multiplication and division operations. In SAMPO, this functionality is integrated into the ALU module, which supports **`MUL`** (16-bit product), **`MULH`**/**`MULHU`** (upper 16 bits of 32-bit product), **`DIV`**/**`DIVU`** (quotient), and **`REM`**/**`REMU`** (remainder).

E.2 Terms N–Z

Negative Flag (N)
Bit 7 of the SAMPO FLAGS register. Set when the most significant bit (bit 15) of the operation result is 1, indicating a negative value in two's complement representation. Tested by the **`BMI`** (branch if minus) and **`BPL`** (branch if plus) instructions.

Netlist
A structural description of a circuit as a collection of logic elements (gates, flip-flops, LUTs) and their interconnections, produced by synthesis tools. In the SAMPO FPGA flow, Yosys generates a netlist in JSON format that is consumed by nextpnr for place-and-route.

nextpnr
An open-source FPGA place-and-route tool. In the SAMPO toolchain, nextpnr-ecp5 takes the synthesized netlist from Yosys and maps it to the physical resources of the Lattice ECP5 FPGA, determining the placement of logic elements and the routing of signals between them.

NOP
No Operation. A system instruction (opcode `0xE`, func `0x0`) that performs no computation and does not modify any registers or flags. The PC advances past the instruction normally. Encoded as `0xE000`.

Non-Blocking Assignment
The Verilog `<=` assignment operator used within `always` blocks triggered by clock edges. Non-blocking assignments schedule updates to occur at the end of the current simulation time step, ensuring that all right-hand-side values are sampled before any left-hand-side values are updated. All sequential logic in SAMPO's RTL uses non-blocking assignments.

Offset
A signed displacement added to a base address to compute a memory access target. In SAMPO, branch instructions use an 8-bit signed offset, jump instructions use a 12-bit signed offset, and load/store instructions support fixed offsets (± 2, ± 4, $+6$) or 16-bit offsets via extended instructions (**LWX**, **SWX**).

Opcode
The portion of an instruction word that specifies the operation to perform. In SAMPO, the primary opcode occupies bits [15:12] of the instruction, providing 16 primary instruction categories (from `0x0` for **ADD** to `0xF` for extended instructions).

Operand
A value or register that an instruction operates on. SAMPO instructions can have register operands (specified by 4-bit register addresses), immediate operands (8-bit or 16-bit constants encoded in the instruction), or implicit operands (such as the stack pointer for **PUSH**/**POP**).

Overflow Flag (V)
Bit 4 of the SAMPO FLAGS register. Set when a signed arithmetic operation produces a result that cannot be represented in 16 bits (i.e., the true mathematical result exceeds the range -32768 to $+32767$). Tested by the **BVS** and **BVC** branch instructions, and used in signed comparisons (**BLT**, **BGE**).

PC

Program Counter. A 16-bit register holding the address of the next instruction to be fetched. The SAMPO PC is initialized to the reset vector (`0x0100` by default) and is updated during the EXECUTE state based on the instruction type: +2 for normal instructions, +4 for extended instructions, or set to a branch/jump target.

Pipeline

A CPU implementation technique in which multiple instructions are in various stages of execution simultaneously, improving throughput. The SAMPO CPU does not implement pipelining; it uses a multi-cycle FSM design where each instruction progresses through states sequentially. Pipelining is discussed as a potential enhancement in the handbook's final chapters.

Place and Route

The FPGA design step in which synthesized logic elements are assigned to specific physical locations on the FPGA die (placement) and the signal connections between them are routed through the FPGA's interconnect fabric (routing). For SAMPO, this is performed by nextpnr-ecp5.

Port

An I/O address in the SAMPO 256-byte port address space (`0x00`–`0xFF`), accessed via **`IN`**/**`OUT`** instructions and their variants. Also used in Verilog to describe the input/output signals of a module.

Port-Based I/O

An I/O model using a dedicated address space and dedicated instructions (**`IN`**, **`OUT`**, **`INI`**, **`OUTI`**, **`INX`**, **`OUTX`**) for peripheral communication, separate from the memory address space. This Z80-style approach is the primary I/O method in SAMPO.

Program Counter

See *PC*.

Prologue/Epilogue

The setup and teardown code sequences at the start and end of a function. A typical SAMPO prologue pushes `RA` and callee-saved registers onto the stack and allocates local variable space. The epilogue reverses these steps and returns via **`JR RA`**.

PUSH/POP

Stack instructions that write to and read from the stack. **`PUSH Rs`** decrements `SP` by 2 and writes the register value to `mem[SP]`. **`POP Rd`** reads `mem[SP]` into the register and increments `SP` by 2. The register file has a dedicated second write port for `SP` to allow simultaneous register and stack-pointer updates during **`POP`**.

RAM

Random Access Memory. Volatile read/write memory used for data storage and the stack. In the

SAMPO SoC, RAM is implemented as a synchronous memory module (`ram.v`) with a one-cycle read latency, mapped to BRAM resources on the FPGA.

RDRF
Receive Data Register Full. Bit 0 of the UART status register at port `0x80`. When set, it indicates that a byte has been received and is available for reading from the data register at port `0x81`. Reading the data register clears this bit.

Register
A fast storage location within the CPU, typically implemented as a group of flip-flops. SAMPO has 16 general-purpose 16-bit registers (`R0`–`R15`), plus 8 alternate registers (`R4'`–`R11'`), a program counter, and a FLAGS register.

Register File
The collection of CPU registers, implemented in SAMPO as a dual-read, dual-write memory structure in the `regfile.v` module. It provides two read ports and two write ports (one general-purpose, one dedicated to `SP` for **`PUSH`**/**`POP`**), and handles the **`EXX`** swap between primary and alternate register sets.

RETI
Return from Interrupt. A system instruction (opcode `0xE`, func `0x4`) that returns execution to the point where an interrupt occurred. RETI restores the saved program counter and re-enables interrupts by setting the I flag.

Return Address (RA)
Register `R1` in the SAMPO architecture, used by convention to store the address to which a called function should return. The **`JALR`** and **`JALX`** (call) instructions write the return address into `RA` (or a specified register), and functions return via **`JR RA`**.

RISC
Reduced Instruction Set Computer. An architectural philosophy emphasizing a small number of simple, fixed-length instructions that execute in few cycles, with a load/store memory model and a large register file. SAMPO is primarily a RISC design, with select CISC-inspired extensions for Z80 workload compatibility.

ROM
Read-Only Memory. Non-volatile memory used to store program code and constant data. In the SAMPO memory map, the region `0x0100`–`0x7FFF` is designated for program code. On the FPGA, ROM and RAM are both implemented in BRAM initialized from hex files.

Rotate
A shift-like operation in which bits shifted out of one end of a register are shifted back in at the other end, preserving all bit values. SAMPO supports rotate left (**`ROL`**) and rotate right (**`ROR`**), as well as rotate through carry (**`RCL`**, **`RCR`**), in 1-bit, 4-bit, and 8-bit amounts.

Sequential Logic
Digital logic whose output depends on both the current inputs and the internally stored state (typically held in flip-flops). In SAMPO, the FSM, register file, cycle counter, and all latching registers are examples of sequential logic. Contrast with *combinational logic*.

Shadow Register
An alternate copy of a register that is not directly accessible but can be swapped with the primary register. In SAMPO, registers `R4'–R11'` are shadow registers that can be exchanged with `R4–R11` using the **`EXX`** instruction. See also *alternate register set*.

Shift
An operation that moves the bits of a register left or right by a specified number of positions. SAMPO supports logical shifts (**`SLL`**, **`SRL`**), arithmetic right shift (**`SRA`**), and rotates, in fixed amounts (1, 4, 8) and variable amounts (via extended instructions **`SLLX`**, **`SRLX`**, **`SRAX`**).

Sign Extension
The process of extending a narrow value to a wider bit width by replicating its most significant (sign) bit into all the new upper bits. For example, the 8-bit signed immediate in **`ADDI`** is sign-extended to 16 bits before being added to the register value. The **`LB`** instruction sign-extends an 8-bit byte to 16 bits.

Simulation
Running a software model of a hardware design to verify its functional behavior before physical implementation. The SAMPO project uses Icarus Verilog (iverilog) for simulation, with testbench modules that provide stimulus (clock, reset, test programs) and check outputs against expected results.

SoC
System-on-Chip. A complete computing system integrated into a single module, combining a CPU core with memory and peripherals. The SAMPO SoC (`soc.v`) integrates the CPU, synchronous RAM, and an MC6850-compatible UART, along with LED status outputs.

SP
Stack Pointer. Register `R2` in the SAMPO architecture, used to point to the current top of the stack. The stack grows downward (toward lower addresses). **`PUSH`** decrements `SP` by 2, and **`POP`** increments it by 2.

Stack
A last-in, first-out (LIFO) data structure in memory, managed by the stack pointer `SP`. The SAMPO stack is used for saving return addresses, preserving register values across function calls, passing arguments beyond the first four, and allocating local variables.

Stack Frame
The region of the stack allocated for a single function invocation, containing the saved return

address, saved callee-saved registers, and local variables. The function prologue creates the stack frame, and the epilogue tears it down before returning.

Store
An instruction that writes data from a CPU register to memory. SAMPO store instructions include **`SW`** (store word), **`SB`** (store byte), and the extended **`SWX`** (store word with 16-bit offset). Store instructions do not modify flags.

Sub-opcode
A secondary opcode field that further specifies the operation within a primary opcode category. In SAMPO, the `func` field (bits [3:0]) serves as a sub-opcode for load, store, shift, multiply/divide, and miscellaneous instructions. Extended instructions (opcode `0xF`) use the `sub` field (bits [3:0]) to select among 16 extended operations.

Synthesis
The process of transforming an HDL description into a gate-level or LUT-level netlist. For SAMPO, Yosys reads the Verilog source files, performs logic optimization, and outputs a netlist targeting the Lattice ECP5 FPGA architecture.

Synchronous
A design methodology in which all state changes occur on the active edge of a global clock signal, ensuring predictable timing behavior. The entire SAMPO CPU is synchronous, with all flip-flops clocked by the same `clk` signal and reset by the active-low `rst_n`.

TDRE
Transmit Data Register Empty. Bit 1 of the UART status register at port `0x80`. When set, it indicates that the UART transmitter is ready to accept a new byte for transmission. Software should poll this bit before writing to the data register at port `0x81`.

Testbench
A Verilog module that provides stimulus (clock generation, reset sequencing, test data) to a design under test (DUT) and optionally checks outputs against expected values. The SAMPO testbench (`sampo_tb.v`) instantiates the SoC, loads a hex program into RAM, runs the simulation, and monitors the CPU's outputs.

Timing Closure
The process of ensuring that all signal paths in a synthesized design meet the setup and hold time requirements of flip-flops at the target clock frequency. During FPGA place-and-route, nextpnr reports whether the SAMPO design achieves timing closure for the specified clock frequency on the ECP5.

Two's Complement
A signed integer representation in which a negative number is obtained by inverting all bits of the corresponding positive number and adding one. In SAMPO's 16-bit word, two's complement

represents values from −32768 (`0x8000`) to +32767 (`0x7FFF`). The N and V flags are defined in terms of two's complement arithmetic.

UART
: Universal Asynchronous Receiver/Transmitter. A serial communication peripheral that converts parallel data to a serial bitstream and vice versa. The SAMPO SoC includes an MC6850-compatible UART (`uart.v`) accessible at I/O ports `0x80` and `0x81`, providing the primary means of text I/O.

Unaligned Access
: Accessing a multi-byte value at a memory address that is not a natural multiple of the value's size. SAMPO supports unaligned 16-bit word accesses (words at odd addresses), though aligned accesses are preferred for performance and simplicity.

VCD
: Value Change Dump. A standard waveform file format (IEEE 1364) produced by Verilog simulators to record signal transitions over time. The SAMPO testbench uses `$dumpfile` and `$dumpvars` to generate VCD files, which can be viewed in GTKWave for debugging.

Verilog
: An IEEE-standard (IEEE 1364) hardware description language widely used for digital design and verification. The primary SAMPO RTL implementation is written in Verilog, comprising modules for the CPU, ALU, register file, shifter, decoder, UART, RAM, and SoC top level.

VHDL
: VHSIC (Very High Speed Integrated Circuit) Hardware Description Language. An IEEE-standard (IEEE 1076) HDL and alternative to Verilog. While the SAMPO project uses Verilog and Amaranth rather than VHDL, VHDL is discussed in the handbook for comparison.

Von Neumann Architecture
: A computer architecture in which instructions and data share a single memory space and bus. The SAMPO CPU uses a von Neumann architecture: instructions and data reside in the same 64 KB address space and are accessed through the same memory interface.

Waveform
: A graphical representation of digital signal values (high, low, or multi-bit bus values) plotted against time. Waveforms are the primary tool for debugging SAMPO hardware designs in simulation, viewed in GTKWave from VCD files generated by Icarus Verilog.

Word
: The native data unit of the SAMPO architecture: 16 bits (2 bytes). Registers, ALU operations, and the memory data bus all operate on 16-bit words. Instructions are either one word (16 bits) or two words (32 bits for extended format).

Writeback

The final CPU pipeline or FSM stage in which computation results are written to the register file. In the SAMPO FSM, the WRITEBACK state writes the execution result or memory load data to the destination register, and updates the stack pointer for **PUSH**/**POP** instructions.

Yosys

An open-source logic synthesis framework used in the SAMPO FPGA toolchain. Yosys reads the Verilog RTL source files, performs elaboration and logic optimization, and produces a technology-mapped netlist targeting the Lattice ECP5 FPGA. It is the synthesis front-end paired with nextpnr for place-and-route.

Zero Extension

The process of extending a narrow value to a wider bit width by padding the upper bits with zeros. In SAMPO, the **LBU** (load byte unsigned) instruction zero-extends an 8-bit byte to fill a 16-bit register, preserving the unsigned interpretation of the value.

Zero Flag (Z)

Bit 6 of the SAMPO FLAGS register. Set when the result of an operation is exactly zero (all 16 bits are 0). The zero flag is tested by the **BEQ** (branch if equal / branch if zero) and **BNE** (branch if not equal / branch if not zero) instructions.

Zero Register

Register `R0` in the SAMPO architecture, which is hardwired to always read as zero. Writes to `R0` are silently discarded. This convention (borrowed from RISC-V and MIPS) simplifies instruction encoding by providing a constant zero source—for example, `ADD Rd, Rs, R0` acts as a move, and `ADD Rd, R0, R0` clears a register.

Index

.ascii directive, 788
.asciz directive, 788
.db directive, 788
.dw directive, 788
.equ directive, 787
.include directive
    absent, 795
.org directive, 787
$readmemh, 439
    address directive, 204
`\$readmemh`, 321
32-bit instructions, *see* extended instructions

ABI
    register names, 153
abstract syntax tree, 164
ACIA, 61, 136–139, 737, 799
ADC, 69, 725
ADD, 69, 725, 782
    bit-level encoding, 744
    encoding, 741
ADD instruction, 355
adder, 300–303
ADDI, 61, 75, 726, 782
    bit-level encoding, 747
    encoding, 742
    register constraint, 704
    two-operand, 25, 781
    two-operand form, 232
ADDI instruction
    encoding, 181
    two-operand, 182
ADDIX, 28, 730, 787
    encoding, 743
    three-operand form, 751
address bus, 799
address decoding, 321, 438
address space, 10, 23, 736
address-shift bug, 211
addressing modes, 31
    immediate, 32
    offset, 90
    PC-relative, 32
    register direct, 31
    register indirect, 32
alternate register set, 731, 799
alternate registers, 21
ALU, 4, 69–86, 351–367, 799
    addition, 354
    Amaranth implementation, 651
    bitwise operations, 361
    design overview, 351
    division, 356
    emulator, 229
    flag generation, 358
    multiplication, 356
    NEG operation, 363
    NOT operation, 363
    operation codes, 353
    operation encoding, 769
    operation selection, 352
    overflow detection, 359

pass-through operations, 361
shift operations, 364
subtraction, 354
testing, 365
variable shifts, 382
Verilog module, 768
`alu` module, 768–770
alu.v, 8
`alu_tb.v`, 451
always block, 800
Amaranth HDL, 4, 623–643, 800
ALU, 651
clock domains, 627
CPU module, 645
Elaboratable, 624
elaboration, 627
enumerations, 630
FPGA synthesis, 669
FSM, 631
Module, 624
overview, 623
platform, 639
Sampo implementation, 645–673
Signal, 624
simulator, 632
Verilog generation, 637
vs. Verilog, 639
choosing, 663
side-by-side, 659
AND, 72, 726, 782
bit-level encoding, 744
encoding, 741
AND gate, 292
AND instruction, 292, 362
ANDIX, 731, 787
encoding, 743
ANSI escape sequences, 267
App struct, 252
architecture comparison, 11, 33
area vs. speed, 695–697
arithmetic instructions, 69–86, 781
summary, 725
arithmetic logic unit, *see* ALU
arithmetic logic unit (ALU), 303–307
arithmetic shift
rounding, 386
ARM, 3
Thumb-2, 12
ARM-style carry, 230, 733, 800
`Array`
in Amaranth, 655
ASCII, 800
assembler, 800
architecture, 143
code generation, 175
example, 796
expression syntax, 789
limitations, 792
passes, 175, 791
reference, 781
role in compiler flow, 720
assembler directive, 804
assembler directives, 786
assembly language, 6, 53–67, 800
program structure, 53
AST, 164

backward trace, 499
bank switching, 374
barrel shifter, *see* shifter, *see* shifter, *see* `shifter` module, 800
baud rate, 444
BCC, 27, 728, 784
encoding, 750
BCD, 30, 800
BCS, 27, 728, 784
encoding, 750
BEQ, 27, 60, 728, 784
encoding, 750
BGE, 27, 728, 784

encoding, 750
BGEU, 27, 728, 784
carry convention, 735
encoding, 750
BGT, 27, 107, 728, 784
encoding, 750
BHI, 27, 107, 728, 784
encoding, 750
BHT, 688
big-endian, 801
bin2hex.py, 46, 203, 596
`bin2hex.py`, 489
binary format, 201
binary literals, 151
binary toolchain, 201
bitstream, 801
BLE, 27, 107, 728, 784
encoding, 750
block operations, 130–131, 801
register conventions, 731
summary, 729
block RAM, 518–521
block transfer
emulator, 235
blocking assignment, 493
BLS, 27, 107, 728, 784
encoding, 750
BLT, 27, 728, 784
encoding, 750
BLTU, 27, 728, 784
carry convention, 735
encoding, 750
BLTU instruction, 420
BMI, 27, 107, 728, 784
encoding, 750
BNE, 27, 728, 784
encoding, 750
Boolean algebra, 294–297, 801
BPL, 27, 107, 728, 784
encoding, 750
BRAM, 801
branch, 801
emulator, 236
offset calculation, 187
offset encoding, 735
range, 784
range checking, 795
branch condition, 687
encoding, 188
branch condition encoding, 394
branch conditions
evaluation, 419
symmetry, 750
truth table, 733
branch delay slot, 688
branch encoding, 108
branch history table, 688
branch instructions, 26, 103–119, 783
encoding, 749
summary, 728
branch offset calculation, 757
branch penalty, 686
definition, 687
branch prediction, 801
dynamic, 688
static, 687
branch target calculation, 742
bubble (pipeline), 683
bug
address-shift, 211
build pipeline, 717–720
building from source, 43
bus, 801
busy-wait loop, 60, 137
BVC, 27, 728, 784
encoding, 750
BVS, 27, 728, 784
encoding, 750
bypass network, 684
byte, 801

byte addressing, 212
byte enable, 321, 802
byte enables, 438
byte load
    sign extension, 423
byte order
    instruction storage, 759
byte swap, 388

C flag, 30
cache, 802
callee-saved registers, 802
caller-saved registers, 802
calling convention, 20, 97, 113, 704–705, 802
    register usage, 731
Cargo, 37
carry chain, 517–518, 696
carry flag, 229, 354, 802
    ARM style, 707
    ARM-style, 79–86, 484, 733, 800
    ARM-style subtraction, 355
    ARM-style vs. x86-style, 356
    from shifts, 389
    subtraction, 303
    subtraction convention, 30
case equality, 458
case sensitivity, 791
CCF, 30, 134, 730, 786
    encoding, 753
character literal, 156
check_eq, 476
CI/CD
    FPGA testing, 617
    lab runner, 617
    script example, 618
    what it catches, 619
CISC, 11, 802
clock, 316–317, 802
    divider selection, 570
    ULX3S, 533
clock domain, 316–317
    crossing, 317
clock domain crossing, 508
clock generation, 453
clock management, 521
clock period, 453
clock skew, 319
CMP, 29, 60, 81, 726, 782
    encoding, 747
CMP instruction
    decoding, 405
    encoding, 184
CMPIX, 731, 787
    encoding, 743
code generation, 175, 792
combinational logic, 291–307, 802
    in Amaranth, 627
comments, 58, 791
    assembly, 150
compilation pipeline, 700
compiler backend, 699–723
condition code, 803
condition codes, 733
conditional branches
    encoding, 749
continuous integration
    hardware, 617
control flow, 103–119
control hazard, 686–689
control signals, 396
control unit, 803
conventions, 14
counter, 314–316, 803
CPI, 677–681, 803
CPIR, 130, 729, 785
    encoding, 747
CPU design
    educational value, 3
    motivation, 3
cpu module, 765–768

FSM states, 766
state transitions, 767
CPU state machine, 411–432
design philosophy, 411
execute, 417
fetch, 415
memory, 422
pipeline hazard, 425
states, 412
transitions, 413
writeback, 422
CPU struct, 222
cpu.v, 8
Cranelift, 721
critical path, 318, 552, 680
cross-assembler, 803
crossterm, 39, 249
cycle count, 803
cycle counter, 679
cycle counting, 240

D flip-flop, 310
DAA, 30, 129, 726, 782, 803
encoding, 746
data bus, 803
data forwarding, 684
data hazard, 683–686
data layout string, 707
De Morgan's theorem, 295
debug trace, 463
debug tracing, 346
debugging, 493–510
`decode` module, 773–775
computed values, 773
control flags, 773
extracted fields, 773
memory control, 773
decode.v, 8
`Decoder`
Amaranth, 656
decoder, 297–300, *see* instruction decoder, 804
Amaranth implementation, 656
delay slot, 688
design under test, 452
development environment, 37
DI, 30, 134, 730, 786
encoding, 753
digital logic, 7
directive, 804
DirectiveArg enum, 166
directives
lexing, 156
disassemble function, 260
disassembler, 212, 258
opcode conflict, 214
distributed RAM, 518
DIV, 126, 726, 782
encoding, 746
DIV instruction, 356
divide, 126
emulator, 241
divide instructions, 782
encoding, 746
summary, 726
division by zero, 358
DIVU, 126, 726, 782
encoding, 746
dual write, *see* register file, SP write port
`$dumpfile`, 463
`$dumpvars`, 463
DUT, *see* design under test

ECP5, 4, 42, 522–526, 804
resource usage, 695
ecppack, 8, 41, 545, 551
EI, 30, 134, 730, 786
encoding, 753
Either enum, 192
elaboration, 624, 804
emulator, 62–64, 219–247

endianness, 804
big-endian, 801
little-endian, 736, 808
`Enum`
in Amaranth, 630
epilogue, 811
error handling
assembler, 167
escape sequences, 154, 791
exclusive OR, *see* XOR gate
EXECUTE state, 417
exit code, 462
expression evaluation
absent, 793
extended instructions, 28, 179, 399, 786
assembler support, 786
complete reference, 751
emulator, 227
encoding, 190, 743
in pipeline, 692
not implemented, 794
summary, 730
EXX, 10, 21, 135, 730, 731, 785, 804
encoding, 747
pipeline considerations, 693
register bank, 21
testbench, 460
`EXX`
Amaranth implementation, 655
EXX instruction, 374
encoding, 184

factorial, 115
fetch, 220
FETCH state, 415
fetch-decode-execute, 219
FETCH_EXT state, 415
FILL, 130, 235, 716, 729, 785
encoding, 747
finite state machine, 805
CPU states, 766
in Amaranth, 631, 648
five-stage pipeline, 682
fixup, 193, 804
Absolute16, 792
Relative12, 792
Relative8, 792
fixup application, 792
flag, 804
flags, 305–306, 358
C (carry), 77, 359
subtraction semantics, 79
carry, 733, 802
computation, 229
effect summary, 733
effects per instruction, 731
H (half-carry), 77
half-carry, 733, 806
I (interrupt enable), 77
interrupt enable, 733, 807
N (negative), 77, 358
negative, 733, 809
overflow, 733, 810
testing, 483
update rules, 420
V (overflow), 77, 80, 359
Z (zero), 77, 358
zero, 733, 816
FLAGS register, 10, 20, 29, 77–86, 731, 805
layout, 731
update rules, 29
flags register
emulator, 223
flip-flop, 309–311, 805
D, 310
JK, 311
Fmax, 680
fork/join, 457
forward reference, 176
resolution, 193

forwarding, 684
FP alias, 790
FPGA, 4, 513–527, 805
  architecture, 513
  flip-flops, 311
  implementation, 8
  loop patch, 206
  programming, 48, 588
  running on hardware, 561–575
  toolchain, 8
  toolchain installation, 41
FPGA testing
  automated execution, 584
  capture strategies, 585
  double-read pattern, 599
  full suite, 601
  hardware differences, 608
  individual tests, 603
  infrastructure, 579
  interpreting results, 603
  per-test builds, 592
  pipeline overview, 579
  running all tests, 601
  timing, 580
`fpga-%` target, 589
`fpga-tests` target, 589, 601
fpga_capture.py, 572–574, 581
fpga_test.py, 584
  exit codes, 588
FSM, *see* CPU state machine, 805
FTDI
  buffer management, 597
  stale data, 597
  USB re-enumeration, 585
FTDI FT231X, 534–538
full adder, 301, 805
function epilogue, 98, 114, 811
function prologue, 97, 114, 811
functional completeness, 294

gate, 805
GETF, 31, 78, 483, 730, 785
  encoding, 747
global pointer, 805
GP, 805
GP (global pointer), 19
GPIO, 12
GPIO peripheral, 283
GTKWave, 40, 345–347, 463, 498, 805
  installation, 40
  save file, 463
  waveform viewing, 47

H flag, 30
half adder, 300
half-carry flag, 806
HALT, 5, 62, 134, 730, 786, 806
  emulator, 242
  encoding, 753
  loop patching, 594
HALT instruction
  encoding, 185
hand-assembly, 754
handshake, 806
hardware debugging, 493–510
hardware description language, 806
Harvard architecture, 690, 806
hazard
  control, 686–689
  data, 683–686
HDL, 806
Hello, Sampo, 564
  program, 59–62
hello.s
  end-to-end, 44
  trace, 243
hex dump, 257
  reading instructions, 760
hex file, 806
hex format

Verilog, 596
hex_loop_patch.py, 206, 565–567, 594
hexadecimal, 151
hold time, 318–319, 507
host I/O, 284

I flag, 30
I-type instructions
encoding, 747
I/O
data width, 753
emulator, 238
port-based, 5, 10
I/O instructions, 132–133, 785
decoding, 403
encoding, 741, 752
summary, 730
I/O model, 737
I/O port, 806
address space, 736
I/O ports, 273–288
UART, 776
Icarus Verilog, 40, 339–341, 807
installation, 40
immediate, 806
encoding, 186
IN, 132, 730, 785
encoding, 752
index register, 806
indirect addressing
parsing, 161
inferred latch, 495
INI, 5, 61, 132, 730, 785
encoding, 752
INI instruction, 179
encoding, 192
limitation, 793
workaround, 793
initial values, 506
inline assembly, 716
instruction
representation, 165
instruction count, 4
instruction decoder, 4, 393–409, 804
branch/jump, 401
control signals, 395
extended instructions, 399
field extraction, 393
I/O control, 403
immediate extraction, 398
J/JR/JALR conflict, 406
load/store, 401
misc operations, 404
register selection, 397
system instructions, 408
Verilog module, 773
instruction decoding
emulator, 226
procedure, 760
instruction encoding, 10, 180, 741
emulator, 227
field extraction, 393
two-tier, 11
worked examples, 754
instruction execution
walkthrough, 429
instruction formats, 25
B-type, 26, 742
encoding, 741
extended, 28, 743
I-type, 25, 742
J-type, 27, 743
R-type, 25, 741
S-type, 26, 742
summary, 739
instruction register, 807
instruction selection, 703
challenges, 711
instruction set
complete survey, 121–139

quick reference, 725
summary, 725
instruction set architecture, 807
instruction timing, 415
`InstType`, 656
interrupt, 807
interrupt enable flag, 807
interrupt handling, 22
interrupt vector, 23, 807
interrupt vector register, 731
interrupts
emulator, 242
inverter, *see* NOT gate
INX, 132, 276, 731, 785
encoding, 743
port range, 751
ISA, 807
quick reference, 725
ISA test suite, 473–491
FPGA results, 601
iteration detection, 581
Iverilog, 807
iverilog
parameter override, 466
`iverilog`, 339–341
IVR (interrupt vector register), 20

J, 729, 784
bit-level encoding, 750
encoding, 743
J (jump), 61, 109
J instruction
encoding conflict, 406
J/JR/JALR encoding conflict, 567–570
JAL, 19, 784
uses JALX encoding, 785, 795
JALR, 19, 27, 109, 729, 784
bit-level encoding, 751
encoding, 743
JALR instruction
encoding, 190
encoding conflict, 406
JALX, 28, 33, 111, 731, 784
encoding, 743
encoding detail, 751
JK flip-flop, 311
JR, 19, 27, 109, 729, 784
bit-level encoding, 751
encoding, 743
JR instruction
as return, 407
encoding, 190
encoding conflict, 406
JTAG, 541, 561
jump, 807
emulator, 236
jump instruction
decode conflict, 208
encoding, 189
jump instructions, 27, 784
encoding, 750
summary, 728
jump offset
lost range, 407
jump target calculation, 743
JX, 28, 33, 111, 731, 784
encoding, 743
encoding detail, 751

Kalevala, 8
Karnaugh map, 296
keyboard controls, 262
keyboard input forwarding, 264

label, 807
label collection, 177, 791
labels, 57, 789
duplicate, 796
latch, 808
inference, 495
inferred, 496

SR, 309
unintentional, 334
Lattice Semiconductor, 522
LB, 24, 88, 727, 783
encoding, 748
LBU, 24, 60, 88, 727, 783
encoding, 748
LDD, 130, 729, 785
encoding, 747
LDDR, 130, 729, 785
encoding, 747
LDI, 130, 729, 785
encoding, 747
LDIR, 130, 235, 716, 729, 785
encoding, 747
learning objectives, 6
LED, 538, 808
LED output, 444
LED status output, 764
lexer, 143
implementation, 146
limitations, 720
linker script, 714
little-endian, 10, 23, 196, 300, 438, 741, 808
binary output, 202
definition, 23
instruction encoding, 759
memory access, 234
LIX, 5, 28, 59, 93, 731, 787
encoding, 743
encoding detail, 751
LIX instruction
encoding, 191
LLVM, 699–723
backend, 700
overview, 699
small targets, 700
TableGen, 700
load instruction, 808
encoding, 182
offsets, 183
load instructions, 783
encoding, 741, 748
offset variants, 90
summary, 727
load-use hazard, 685
load/store architecture, 87–101, 808
local variables, 98
logic gate, 805
logic gates, 291–294
logic instructions, 782
summary, 726
logical shift, 808
Look-Up Table, *see* LUT
look-up table, 808
lookup table (LUT), 296
looping problem, 564–565
LPF file, 530–532
LUI, 727, 783
encoding, 748
LUI instruction, 365
LUT, 515–517, 808
LUT count, 695
LW, 24, 87, 727, 783
encoding, 748
with offset, 727
LWX, 26, 92, 731, 787
encoding, 743
offset range, 751

machine code, 741, 808
macro
not supported, 794
`make isa-<name>`, 489
Makefile, 44, 339
.PRECIOUS, 490, 592
FPGA targets, 555–559, 589
ISA tests, 488
loop patch, 210
pattern rules, 589

test automation, 466
MC6850, 537, *see* UART, 809
MC6850 ACIA, 60, 136–139, 276
mem_ready bug, 425
`Memory`
in Amaranth, 671
memory, 87–101
emulator, 224
properties, 736
memory bus, 435
memory hazard, 610
memory initialization, 506
memory interface, 23
memory map, 23, 736, 809
memory model, 23
memory operations
emulator, 232
memory ready signal, 436, 765
MEMORY state, 422
FETCH hazard, 610
MEMORY to WRITEBACK fix, 501–506
discovery, 487
memory viewer, 257
memory-mapped I/O, 434, 809
metastability, 317, 507, 536
minterm, 295
MIPS, 11
miscellaneous instructions, 785
encoding, 741, 746
summary, 729
mnemonic, 809
complete list, 781
module, 809
MOV, 729, 785
encoding, 747
MUL, 126, 726, 782
encoding, 746
MUL instruction, 356
MULH, 126, 726, 782
encoding, 746
MULHU, 726, 782
encoding, 746
multi-cycle CPU, 411
multi-cycle FSM, 677
multiplexer, 297–300, 809
4-to-1, 297
multiply, 126
emulator, 241
multiply instruction
encoding, 183
multiply instructions, 782
encoding, 741, 746
summary, 726
multiply/divide unit, 809
mux, 809

N flag, 30
NAND gate, 294
NEG, 72, 725, 782
pseudo-instruction, 781
NEG instruction, 363
encoding, 185
limitation, 793
negative flag, 809
netlist, 810
nextpnr, 4, 8, 41, 545, 549–810
nibble packing, 384
nMigen, *see* Amaranth HDL
no_std, 712–717
non-blocking assignment, 375, 493, 810
NOP, 133, 730, 786, 810
encoding, 753
NOP instruction
encoding, 185
NOR gate, 294
NOT, 72, 726, 782
pseudo-instruction, 782
NOT gate, 291
NOT instruction, 363
encoding, 185

limitation, 793
number literals, 151
numeric literals, 58, 789

offset, 810
opcode, 810
map, 737
shared encoding, 209
opcode dispatch, 226
opcode map, 29
complete, 754
openFPGALoader, 8, 41, 541–542, 561, 588
operand, 810
Operand enum, 165
operand formats, 790
OR, 72, 726, 782
bit-level encoding, 744
encoding, 741
OR gate, 293
OR instruction, 362
origin
padding, 202
origin padding, 180
ORIX, 731, 787
encoding, 743
OSS CAD Suite, 41
OUT, 132, 730, 785
encoding, 752
OUTI, 5, 61, 132, 730, 785
encoding, 752
OUTI instruction, 179
encoding, 192
limitation, 793
output buffer, 268
OUTX, 132, 276, 731, 785
encoding, 743
port range, 751
overflow detection, 359
addition, 360
and branches, 361
subtraction, 360
overflow flag, 229, 810

parallelism
combinational, 306
parameter
Verilog, 322
parameter override, 466
parser, 143
implementation, 157
navigation methods, 158
operand parsing, 161
PC, 810
PC (program counter), 20
performance analysis, 677–698
performance metrics, 271
peripheral
adding new, 280
peripheral simulation, 273–288
pin constraints, 530
pipeline, 811
assembler, 145
pipeline design, 689–694
pipeline flush, 692
pipeline hazard, 425
discovery, 484
MEMORY to WRITEBACK, 501
pipeline registers, 689
pipeline stages, 681
pipeline stall, 683, 692
load-use, 685
pipelining, 677–698
place and route, 545–559, 811
PLL, 521–522
ECP5, 525
POP, 19, 95, 729, 785, 811
encoding, 747
field layout, 747
pipeline considerations, 693
POP instruction

decoding, 405
encoding, 184
execution, 427
port, 811
port I/O
emulator, 238
port-based I/O, 736, 806, 811
port-mapped I/O, 273, 434
power-on defaults, 736
predict-not-taken, 687
prerequisites, 14
print_str, 475
program counter, 315, 731, 810, *see* PC
update logic, 418
Program struct, 166
Project Trellis, 526, 545
prologue, 811
propagation delay, 302
pseudo-instruction, 185
PUSH, 19, 95, 729, 785, 811
encoding, 747
field layout, 747
pipeline considerations, 693
stack pointer update, 505
PUSH instruction
decoding, 405
encoding, 184
execution, 427
Python HDL, *see* Amaranth HDL

quick reference card, 739

R-type instructions
encoding, 744
R0, 816
write discard, 483
zero register, 5, 10, 18
R0 hazard, 685
R0 register, 372
R1, 19
R2, 19
R3, 19
RA, 812
RA (return address register), 19
RAM, 319–322, 811
Amaranth initialisation, 671
initialization, 321, 439, 609
module, 320
organization, 438
synchronous, 436
Verilog module, 775
`ram` module, 775–776
word addressing, 775
`RAM_INIT_FILE` parameter, 763
range checking, 195
ratatui, 39, 249–272
RAW hazard, 683
RCL, 727
encoding, 745
RCL instruction, 387
RCR, 727
encoding, 745
RCR instruction, 387
RDRF, 812
read-after-write, 377
read-after-write hazard, 683
reading paths, 13
full stack, 14
hardware designer, 13
programmer, 13
recursion, 115
`regfile` module, 770–771
regfile.v, 8
register, 312–314, 812
aliases, 153
register aliases, 789
register allocation
order, 703
register conventions, 370
register display, 255
register file, 4, 18, 312–314, 369–379, 812

alternate registers, 370
Amaranth implementation, 654
array design, 369
bank switching, 374
combinational read, 371
dual write ports, 771
emulator, 223
EXX swap, 771
FPGA synthesis, 378
implementation, 312
internal storage, 770
LLVM description, 701
R0 hardwired zero, 372, 771
read ports, 371
reset, 377
SP write port, 372
timing, 377
Verilog module, 770
write ports, 371
`RegisterFile`
Amaranth, 654
registers, 18
aliases, 59
alternate set, 135
callee-saved, 802
caller-saved, 802
conventions, 731
names, 789
special, 731
usage example, 55
regression testing, 615
tracking results, 616
REM, 126, 726, 782
encoding, 746
REMU, 126, 726, 782
encoding, 746
reset
emulator, 225
sequence, 454
simulation vs. hardware, 609
reset state, 736
reset synchronization, 508
reset synchronizer, 539
`RESET_VECTOR` parameter, 763
resource utilization, 553
RETI, 134, 730, 786, 812
encoding, 753
return address, 812
ripple-carry adder, 301
RISC, 9, 11, 812
RISC-V, 3, 11
comparison, 6
register conventions, 10
ROL, 25, 121, 727, 783
encoding, 745
ROL instruction, 386
ROL4, 727
encoding, 745
ROL8, 727
encoding, 745
ROM, 812
ROR, 25, 121, 727, 783
encoding, 745
ROR instruction, 386
rotate, 812
rotate instructions, 121–126, 783
encoding, 745
summary, 726
rotate through carry, 388
RTLIL, 639
run loop, 221
Rust, 6
installation, 37
no_std, 712
on Sampo, 708–712
target specification, 717
rustup, 38

SAMPO, 3
architecture, 17

architecture summary, 10
design goals, 9
etymology, 8
project overview, 8
project structure, 9
sampo_pkg.vh, 337
sampo_tb.v, 453
SampoCPU, 645
sasm, 4, 37, 53–67, 143, 781
building, 38
directives, 55
.ascii, 57
.asciz, 57
.db, 56
.dw, 56
.equ, 56
.org, 56
labels, 57
pipeline, 143
SB, 24, 89, 728, 783
encoding, 742, 749
SBC, 69, 725
SCF, 30, 134, 730, 786
encoding, 753
SelectionDAG, 703
semu, 4, 37, 62–64, 219–247
building, 39
interactive mode, 63
trace mode, 63
sensitivity list, 495
sequential logic, 309–322, 813
serial output
TUI, 265
serial port, 136, 563
auto-detection, 587
buffer flushing, 597
capture, 581
timeout, 584
SETF, 30, 31, 78, 730, 785
encoding, 747
setup time, 318–319, 507
shadow register, 813
shift, 813
logical, 808
shift instruction
encoding, 183
shift instructions, 121–126, 783
encoding, 741, 745
fixed variants, 783
limitation, 794
multi-bit, 124
summary, 726
shift operations
emulator, 240
shift register, 314–316
shifter, 4, 381–392
Amaranth implementation, 653
byte swap, 388
carry output, 389
CPU integration, 391
design, 381
function codes, 383
rotate, 386
rotate right, 387
rotate through carry, 387
shift left logical, 384
shift right arithmetic, 385
shift right logical, 385
shifter module, 771–773
shift functions, 772
shifter.v, 8
sign extension, 186, 398, 742, 813
Signal, *see* Amaranth HDL, Signal
signal tracing, 500
simulation, 451–472, 813
Amaranth, 632
vs. hardware, 608
vs. synthesis, 506
SLL, 25, 121, 727, 783
encoding, 745

SLL instruction, 383
SLL4, 124, 727
    encoding, 745
SLL8, 124, 727
    encoding, 745
SLLX, 126, 731, 787
    encoding, 743
    shift amount, 752
SoC, *see* System on Chip, 813
soc module, 763–765
soc.v, 8
SP, 813
SP (stack pointer), 19
special registers, 20
SPI, 12
SR latch, 309
SRA, 25, 121, 727, 783
    encoding, 745
SRA instruction, 383
SRA4, 124, 727
    encoding, 745
SRA8, 124, 727
    encoding, 745
SRAM, 319–322
    cell, 319
SRAX, 126, 731, 787
    encoding, 743
    shift amount, 752
SRL, 25, 121, 727, 783
    encoding, 745
SRL instruction, 383
SRL4, 124, 727
    encoding, 745
SRL8, 124, 727
    encoding, 745
SRLX, 126, 731, 787
    encoding, 743
    shift amount, 752
stack, 94–101, 813
stack frame, 97–100, 813
stack instructions, 785
    summary, 729
stack pointer, 94, 813
    dual write, 372
    initialization, 378, 482
    update, 505
stack viewer, 261
stale mem_ready, 503
state machine, 4, *see* finite state machine
    CPU, 8
Statement enum, 165
status bar, 270
store instruction, 814
    encoding, 182, 398
store instructions, 783
    encoding, 748
    summary, 727
string literals, 154
SUB, 69, 725, 782
    bit-level encoding, 744
    carry convention, 27
    encoding, 741
SUB instruction, 355
sub-opcode, 814
SUBIX, 731, 787
    encoding, 743
subroutines, 113–119
subtraction, 302
    carry convention, 230
SW, 24, 89, 728, 783
    encoding, 742, 749
SWAP, 25, 121, 727, 783
    encoding, 745
SWAP instruction, 388
SWI, 134, 730, 786
    encoding, 753
SWX, 26, 92, 731, 787
    encoding, 743
    offset range, 751
symbol table, 177, 193

synchronous design, 316
synchronous logic, 814
    in Amaranth, 627
synchronous RAM, 321
synchronous write, 377
synthesis, 545–559, 814
    from Amaranth, 669
    mismatches, 506
    per-test, 592
system instructions, 133–136, 786
    encoding, 753
    summary, 730
System on Chip, 433–447
    architecture, 433
    integration, 445
    LED outputs, 444
    memory bus, 435
    RAM, 436
    UART, 440
system-on-chip, *see* soc module, 813

TableGen, 700
target description, 701–708
TDRE, 814
terminal emulator, 265
TEST, 30, 81, 726, 782
    encoding, 747
test coverage, 473, 614
    table, 479
test data area, 485
test framework, 475
test output
    format, 603
test pipeline, 488
test results
    full table, 610
test strategy, 473
test suite, 5, 473–491
    coverage, 605
test_addi, 605
test_alu, 605
`test_alu.s`, 481
test_branch, 607
test_extended, 607
test_jump, 607
test_loadstore, 606
`test_loadstore.s`, 484
test_misc, 607
test_muldiv, 606
test_shift, 605
test_stack, 607
testbench, 341–344, 451–472, 814
    Amaranth, 633
    anatomy, 451
    check task, 457
    parameterized, 465
    PASS/FAIL detection, 461
    Python, 664
    self-checking, 342, 457, 462
    termination, 456
testing
    132 tests, 601
    Amaranth, 664
    FPGA pipeline, 579
    regression, 615
    simulation-first workflow, 616
three-pass assembly, 175
throughput, 681
timeout, 457
timer peripheral, 281
‘`timescale`, 452
timing
    simulation vs. hardware, 608
timing analysis, 550
    reports, 552–553
timing closure, 696, 814
timing constraints, 318–319
timing violations, 507
Token enum, 147
tokenize method, 149

toolchain, 201
truth table, 295
TUI debugger, 64–66, 249–272
TUI layout, 253
two's complement, 187, 302, 814
    0x8000 edge case, 364
    negation, 363
two-operand instructions, 704
two-pass assembly, 791
two-stage pipeline, 697
tx_ready, 467
type legalization, 706

UART, 12, 440–447, 815
    baud rate, 444
    busy-wait, 476
    busy-wait output, 137
    emulator, 239
    hardware, 534–538
    port addresses, 404
    port assignments, 737
    receive, 278, 442
    serial capture, 581
    simulation, 276–280
    status register, 276, 440
    test reporting, 477
    testbench model, 467
    transmission delay, 469
    transmit, 441
    TX buffer overflow, 598
    Verilog module, 776
`uart` module, 776–777
    port map, 776
    RX handshake, 776
    TX handshake, 776
ULX3S, 4, 8, 42, 529–543
    baud rate, 444
    FTDI chip, 585
    hardware, 529
    setup, 42
unaligned access, 815
uninitialized signal, 497
universal gate, 294
USB
    re-enumeration, 597

V flag, 30
Value Change Dump, 815
VCD, 47, 462–815
    file size, 463
    from Amaranth, 636
VCD (Value Change Dump), 47
VCD file, 345–347
Verilator, 41
Verilog, 7, 325–347, 815
    $readmemh, 203
    always block, 328–331
    assign, 328–331
    blocking assignment, 331–333
    combinational block, 333–335
    common bugs, 493
    fork-join, 344
    generate, 335–338
    generation from Amaranth, 637
    implementation, 4
    `` `include ``, 337
    initial block, 331
    localparam, 336
    module, 325–328
    module hierarchy, 777
    module instantiation, 327
    module reference, 763–777
    non-blocking assignment, 331–333
    parameter, 335–338
    reg, 326
    reg vs register, 326
    sensitivity list, 330
    sequential block, 333–335
    simulation, 46
    task, 342

wire, 326
Verilog module, 809
VHDL, 815
von Neumann architecture, 815
von Neumann bottleneck, 690
VT220, 265

WAR hazard, 686
waveform, 815
Amaranth, 636
VCD, 462
waveform debugging, 498
waveform viewer, 345–347
WAW hazard, 686
width mismatch, 497
word, 815
word addressing, 212
word alignment, 24
write-after-read hazard, 686
write-after-write hazard, 686
writeback, 815
WRITEBACK state, 422
WSL, 38

x propagation, 497
x86, 3
XOR, 72, 726, 782
bit-level encoding, 745
encoding, 741
XOR gate, 293
XOR instruction, 362
XORIX, 731, 787
encoding, 743

Yosys, 4, 8, 41, 545–548, 816
RTLIL, 639

Z flag, 30
Z80, 11
carry convention, 31
comparison, 6
compatibility, 9
features retained, 12
limitations addressed, 12
programmer's RISC, 11
zero extension, 816
zero flag, 816
zero register, 18, 372, 816
rationale, 374