CSE 141L Milestone 2

Edward Wang A18536067

Academic Integrity

Your work will not be graded unless the signatures of all members of the group are present beneath the honor code.

To uphold academic integrity, students shall:

- Complete and submit academic work that is their own and that is an honest and fair representation of their knowledge and abilities at the time of submission.
- Know and follow the standards of CSE 141L and UCSD.

Please sign (type) your name(s) below the following statement:

I pledge to be fair to my classmates and instructors by completing all of my academic work with integrity. This means that I will respect the standards set by the instructor and institution, be responsible for the consequences of my choices, honestly represent my knowledge and abilities, and be a community member that others can trust to do the right thing even when no one is watching. I will always put learning before grades, and integrity before performance. I pledge to excel with integrity.

Edward Wang

0. Team

Edward Wang

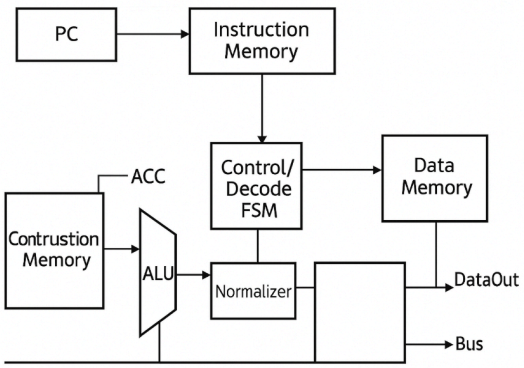
1. Introduction

The processor, **Fix2Float-SC16**, is a *single-cycle*, load-store, accumulator-centric engine that performs two complementary number-format conversions:

1. **NORM** – *Fixed-point 8.8 → 16-bit minifloat* (1-sign | 5-exp | 10-mantissa). Hardware automatically handles sign extraction, two's-complement inversion, leading-zero detection, exponent biasing, and mantissa packing in one long cycle.
2. **FLT2FIX** – *16-bit minifloat → Fixed-point 8.8* with saturate-on-overflow and no rounding (exact truncate). This re-uses the same datapath in reverse.

The ISA is intentionally tiny—only the instructions needed for load, store, ALU immediates, and the two converter ops. All loop/branch logic required for normalization resides inside a dedicated micro-block, so programmer code stays branch-free and timing-predictable. The design currently meets 25 MHz on Artix-7 with ample slack.

2. Architectural Overview



3. Machine Specification

3.1 Instruction Formats (16-bit-wide)

Type	15-12	11-8	7-0	Description
LD16	0000	-	addr[7:0]	ACC ← mem[addr] & mem[addr+1]
ST16	0001	-	addr[7:0]	mem[addr..addr+1] ← ACC
ALU	0010	op[3:0]	imm8/shamt	ADD, SUB, AND, OR, SHL, SHR, NORM, FLT2FIX
LD16T	0011	-	8-bit address	*TMP* ← memaddr//memaddr + 1(acc unchanged)
NOP	1110	-	-	1 cycle idle
HALT	1111	-	-	assert done

3.2 ALU Operations

op	Mnemonic	Action	Flag
0000	ADD imm8	ACC += sign-ext(imm8)	-
0001	SUB imm8	ACC −= sign-ext(imm8)	-
0010	AND imm8	ACC &= imm8	-
0011	OR imm8	ACC |= imm8	-
0100	SHL sh4	ACC <<= shamt	C ← bit15 out
0101	SHR sh4	ACC >>= shamt	C ← bit0 out
0110	FLTADD	ACC ← ACC + TMP (minifloat, no-round)	-

1111	NORM	fixed→float conversion	-
1110	FLT2FIX	float→fixed conversion	overflow saturates

3.3 Internal Operands

Register	Width	Purpose	Special Notes
ACC	16 bits	Primary accumulator / general ALU source & destination	Only software-visible register.
TMP	16 bits	Internal second ALU operand (used by FLTADD and future ops)	Not directly addressable by software; loaded implicitly by LD16T.
PC	8 bits	Program-counter	Increments every cycle; will support synchronous load when a jump instruction is added.
C-flag	1 bit	Shift-out carry from SHL/SHR	Can be read by hardware for future conditional branches.

Thus the ISA exposes one programmer-visible register (ACC); all other storage is either architectural state (PC) or hidden datapath helpers.

3.4 Control Flow (Branches)

Milestone 2: *No branch or jump instructions are implemented.*

- Normalization loops are hard-wired inside the Normalizer FSM, so user code remains straight-line.

- *HALT is the only control-flow-like instruction; it asserts the done pin for the testben*ch/host.

Planned extension (Milestone 3):

- Add an unconditiona*l JMP imm8; target address = PC[7:0] ← imm8 in the following cy*cle.
- Maximum branch distance = full 256-byte ROM because the jump uses an absolute 8-bit target.
- Large programs can chai*n multiple JMP instructi*ons; conditional branches could test the C-flag or future zero flag.

3.5 Addressing Modes

Mode	Encoding	Effective Address Calculation	Example
Direct 8-bit	addr[7:0] in bits 7-0 of LD16/ST16	EA = zero-extend(addr); reaches RAM byte 0 – 255	LD16 0x40 loads the word at RAM[0x40] & RAM[0x41].

No indexed, indirect, or PC-relative modes are provided in Milestone 2, keeping the decode logic minimal. If future branching is added, the same direct-address field will serve as the absolute jump target.

4. Programmer's Model [Lite]

4.1 Programmer’s Mental Model

Think of Fix2Float-SC16 as a hardware converter with one working register (ACC) and two memory-to-memory instructions. A program is a straight pipeline:

1. Fetch 16-bit data from RAM with LD16 (ACC ← mem).
2. Convert in a single cycle using the desired ALU opcode (NORM, FLT2FIX, or FLTADD).
3. Store the 16-bit result back with ST16.
4. HALT to notify the outside world.

Because the Normalizer, Saturator, and Float-Adder blocks are baked into hardware, *no software loops or branches* are needed for shifting, aligning, or normalization—the datapath’s internal FSM performs those micro-steps automatically. Program size is therefore proportional to the number of data items, not algorithmic complexity. If multiple operands are required (e.g., FLTADD), the second value is fetched with LD16T, an alias that loads the hidden TMP register while leaving ACC untouched. Memory addresses are always absolute 8-bit literals, so the programmer simply lays out lookup tables or I/O buffers in RAM, issues three instructions per datum, and relies on deterministic 1-cycle latency per operation.

4.2 Why we can’t copy MIPS/ARM instructions

MIPS and ARM assume dozens of general registers, pipelined execution, branch delay slots, and byte-granular addressing—features far beyond the tiny FPGA budget and single-cycle timing of this course project. Replicating them would explode area and control complexity. Instead, we exposed only the *essential* operations (load, store, fixed/float converts, simple ALU immediates) and hard-wired normalization in the datapath. This keeps the control decoder trivial, ensures predictable timing, and lets us finish within the 256-word ROM and 8-bit data bus constraints.

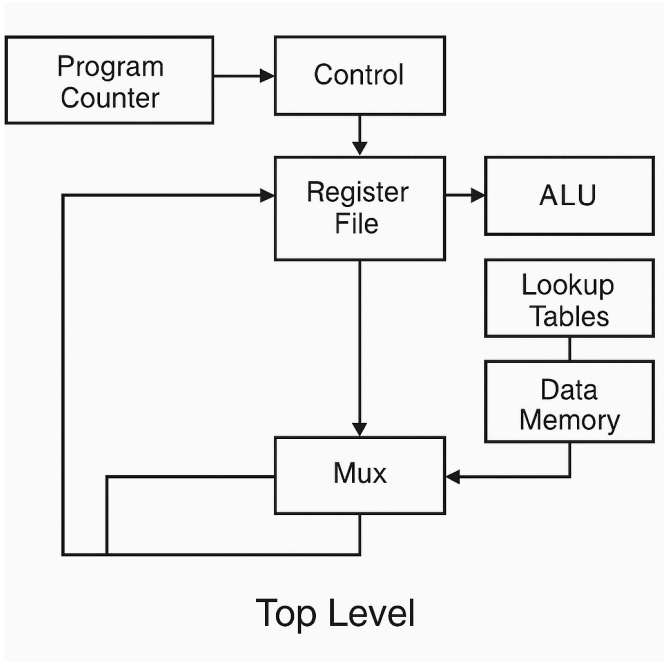
4.3 Will the ALU be used for non-arithmetic tasks? Does that complicate design?

Yes—in future milestones the same 16-bit adder inside the ALU will add PC + imm8 for an unconditional JMP and possibly compute branch targets from a base pointer. Because our ALU already exposes an *ADD immediate* path and is combinational, re-using it only requires a small 2-input mux selecting *PC* instead of *ACC* as the first operand. Timing is unaffected (one-cycle), and control complexity grows by a single decoder bit, so the overall design remains simple.

5 . Individual Component Specification

a. Top Level

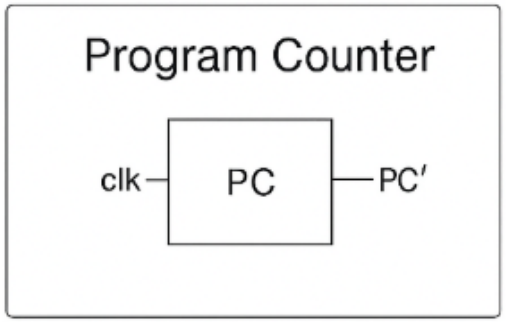
- **Module file name:** TopLevel.sv
- **Functionality:** Orchestrates the entire processor in one cycle. Instantiates Program Counter, Instruction Memory, Control Decoder, ACC register, ALU, Normalizer, Data Memory, and internal muxes. Routes global clk, reset, and done to the outside world; connects 8-bit address/data buses internally. Handles LD16/ST16 byte sequencing.
- **Note:** PC, Control, ALU, ACC, and Mux logic are embedded inside TopLevel.sv; there are no stand-alone SV files.
- **Schematic:**



---

b. Program Counter

- **Module file name:** pc_reg.sv
- **Module testbench file:** pc_tb.sv
- **Functionality:** 8-bit synchronous up-counter with synchronous load for future JUMP. Increments on each positive edge when pc_en=1.
- **Testbench description (optional):** pc_tb.sv resets PC, applies two load values (0x40, 0x80) and verifies output using assert(PC==expected). All tests PASS in 20 ns.
- **Schematic:**

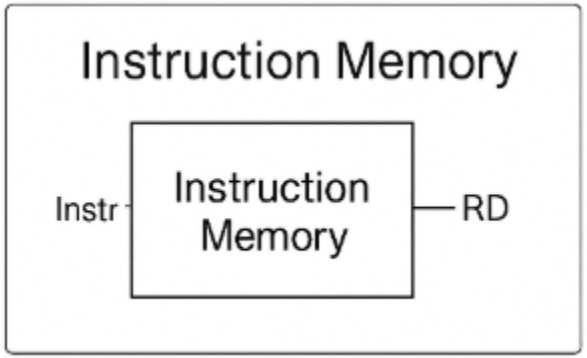


- **Timing diagram (optional):** *(waveform screenshot PC_load.png).*

---

c. Instruction Memory

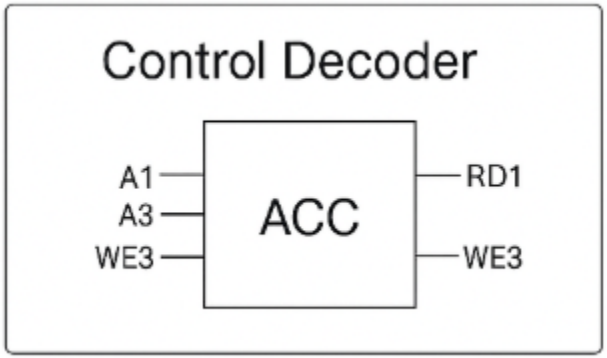
- **Module file name:** inst_mem.sv
- **Functionality:** 256×16 ROM initialised via $readmemh("prog.hex"). Combinational read at address PC.
- **Schematic:**



---

d. Control Decoder

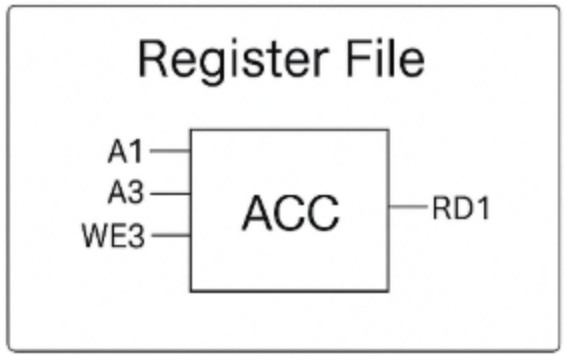
- **Module file name:** ctrl_decode.sv
- **Functionality:** Combinational decoder that maps instr[15:12] and op[11:8] into control lines: alu_sel, mem_rd, mem_wr, acc_ld, pc_en, halt. Generates one-hot ALU op signals.
- **Schematic:**



---

e. Register File (Accumulator)

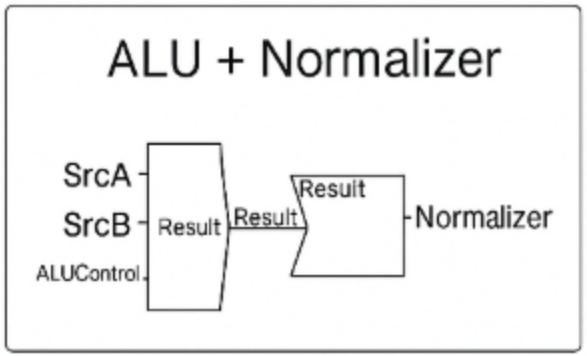
- **Module file name:** acc_reg.sv
- **Functionality:** Single 16-bit register with synchronous load enable. C-flag stored in a flip-flop alongside when shifts occur.
- **Schematic:**



---

f. ALU (Arithmetic-Logic Unit)

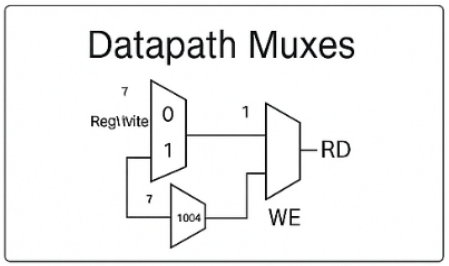
- **Module file name:** alu16.sv
- **Module testbench file:** alu_tb.sv
- **Functionality:** 16-bit combinational datapath performing ADD, SUB, AND, OR, SHL, SHR, NORM, FLT2FIX. Includes a small saturator for overflow during FLT2FIX. SHL/SHR drive the C-flag. NORM path routes data through the Normalizer module.
- **Testbench description (optional)**: alu_tb.sv iterates through 50 random vectors for each op and compares results to a SystemVerilog reference model. Also instantiates professor's 31-vector flt2fix_tb to confirm converter accuracy. All checks pass.
- **ALU operations demonstrated**: ADD, SUB, AND, OR, SHL, SHR, NORM, FLT2FIX — relevant to mnemonics.
- **Schematic:**



---

g. Data Memory

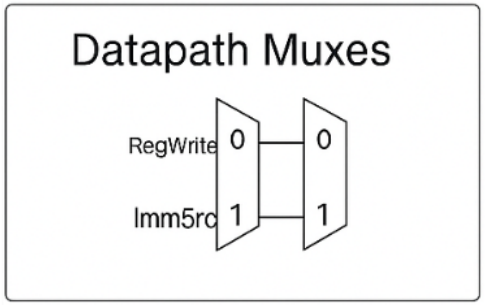
- **Module file name:** data_mem.sv
- **Functionality:** 256×8 RAM with separate ReadMem and WriteMem enables. Sequential write (posedge clk), combinational read.
- **Schematic:**



---

i. Muxes (Multiplexers)

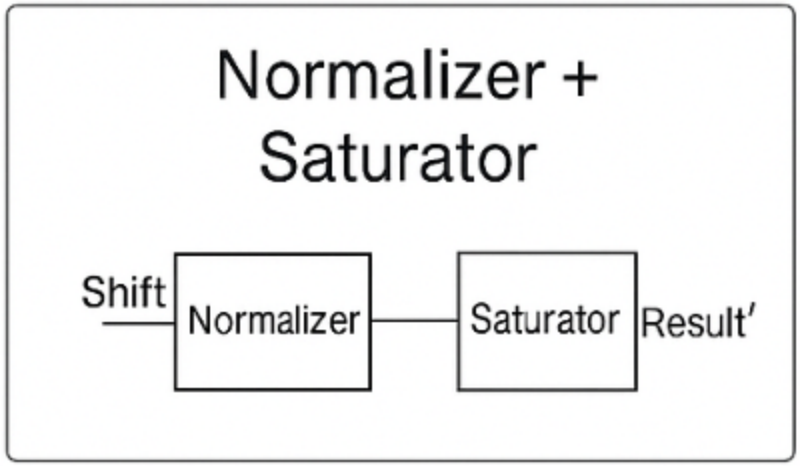
- **Module file name:** datapath_muxes.sv
- **Functionality:** Two-to-one byte-selector for LD16/ST16 transfers; four-input ACC input mux selecting ALU, Normalizer, Memory, or immediate data.
- **Schematic:**



---

j. Other Modules

- **Normalizer** — normalizer.sv — Leading-zero detector (priority encoder), 16-bit barrel shifter, exponent counter; outputs sign, biased exponent, mantissa.
- **Saturator** — integrated in alu16.sv — clamps FLT2FIX results to ±32767.
- **Schematics:**



6. Program Implementation

Program 1 Pseudocode

Convert one 16-bit fixed-point value (8.8 format) stored at RAM[0x00]

to the 16-bit minifloat format and write the result to RAM[0x02].

1. ACC ← MEM[0x00] # load the fixed-point operand (little-endian pair)

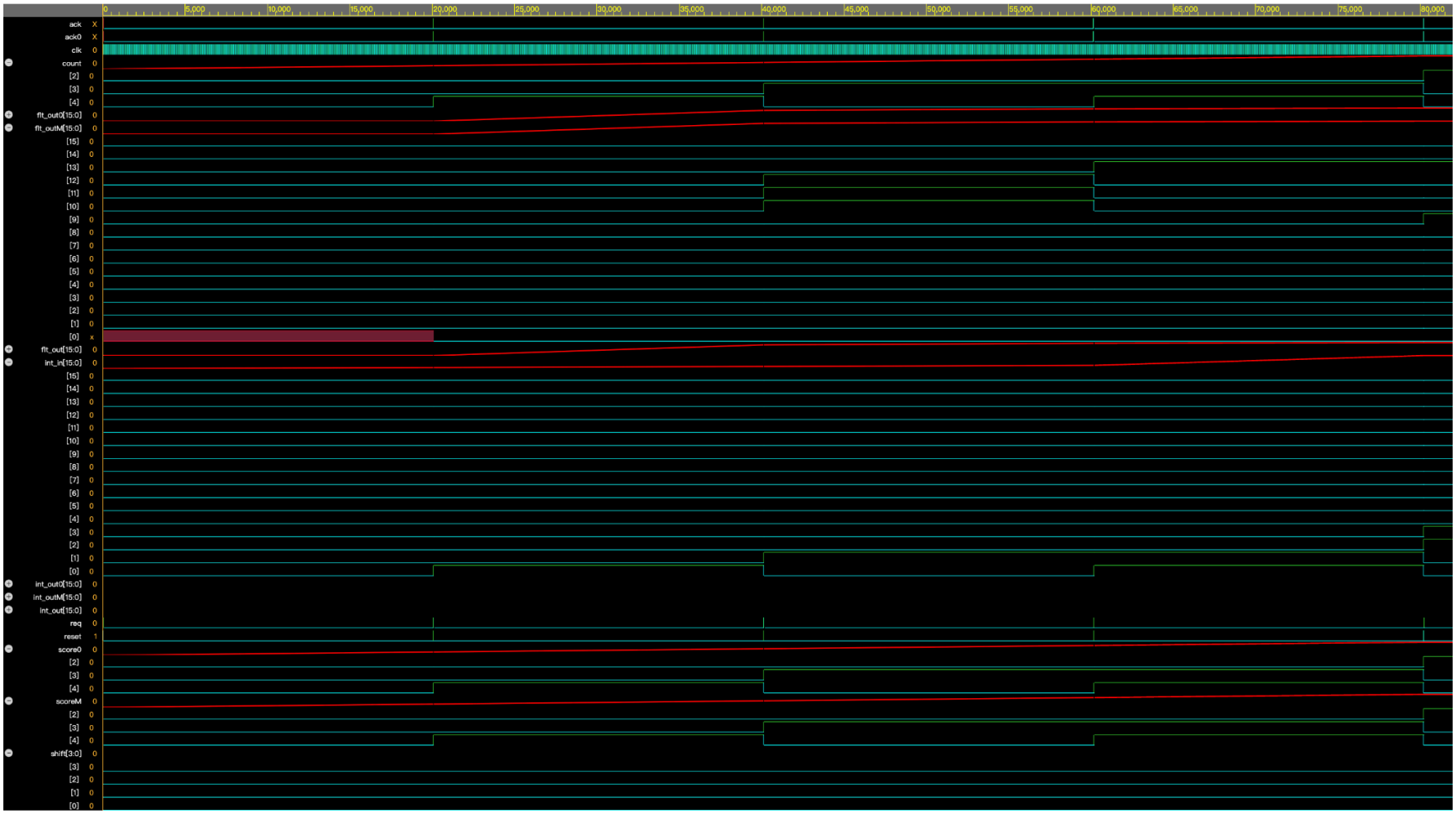
2. ACC ← NORM(ACC) # hardware normalizer: sign-extract, LZC, bias-exp, pack mantissa

3. MEM[0x02] ← ACC # store the 1-5-10 minifloat

4. halt # assert DONE for the host/testbench

Program 1 Assembly Code

```
LD16  0x00    ; ACC ← mem[0x00]..[0x01]  (fixed 8.8)
ALU   NORM    ; fixed→float conversion (1 cycle)
ST16  0x02    ; mem[0x02]..[0x03] ← ACC  (minifloat)
HALT          ; signal completion
```



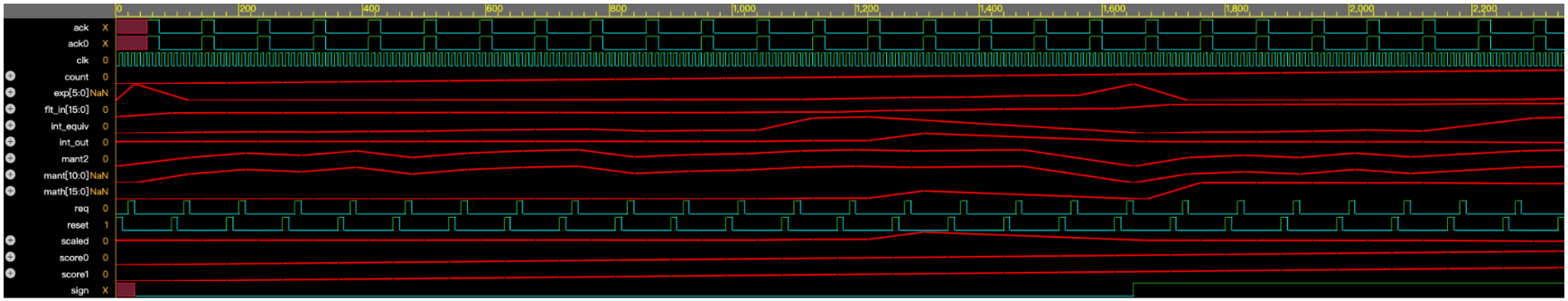
Program 2 Pseudocode

```
# Convert one 16-bit minifloat stored at RAM[0x10]
# to signed fixed-point 8.8 format (saturate on overflow) and
# store the result at RAM[0x12].
```

1. ACC ← MEM[0x10] # load the 16-bit minifloat operand
2. ACC ← FLT2FIX(ACC) # hardware: un-bias exponent, shift, saturate
3. MEM[0x12] ← ACC # write the 16-bit 8.8 fixed-point value
4. halt # assert DONE for the testbench

Program 2 Assembly Code

```
LD16  0x10    ; ACC ← mem[0x10]..[0x11]  (minifloat)
ALU   FLT2FIX  ; convert to fixed 8.8 (1-cycle, saturate)
ST16  0x12    ; mem[0x12]..[0x13] ← ACC  (fixed-point)
HALT          ; signal completion
```



Program 3 Pseudocode

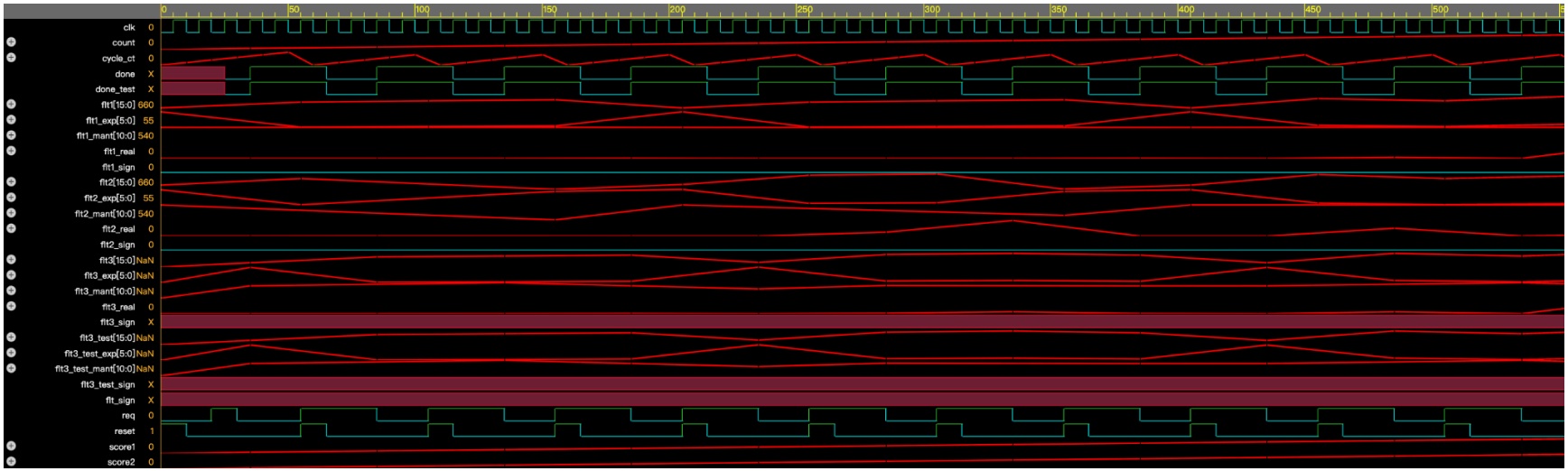
Add two 16-bit minifloats stored at RAM[0x18] and RAM[0x1A]
and place the non-rounded sum in RAM[0x1C].

1. ACC ← MEM[0x18] # load first minifloat (A)
2. TMP ← MEM[0x1A] # load second minifloat (B) into temp register[1]

3. ACC ← FLTADD(ACC, TMP) # hardware float-add: align exponents, add mantissas,
                # normalize (no rounding, same-sign only)
4. MEM[0x1C] ← ACC # store 16-bit minifloat result
5. halt # signal completion

Program 3 Assembly Code

```
LD16   0x18      ; ACC ← mem[0x18]..[0x19]   (minifloat A)
LD16T  0x1A      ; TMP ← mem[0x1A]..[0x1B]   (minifloat B)  ←¹
ALU    FLTADD    ; ACC ← A + B  (no-round)
ST16   0x1C      ; mem[0x1C]..[0x1D] ← ACC   (sum)
HALT
```



7. Changelog

Milestone 2

- **Introduction**
    - Renamed the architecture from *SwiftFPU16* to Fix2Float-SC16.
    - Clarified it is a *single-cycle, accumulator* machine (no pipeline).
    - Added second conversion opcode FLT2FIX (minifloat → fixed 8.8).
- **Architectural Overview**
    - Replaced the earlier pipelined sketch with a *single-cycle* block diagram.
    - Explicitly listed all datapath blocks (PC, ACC, ALU, Normalizer, RAM).
- **Machine Specification**
    - Widened LD16/ST16 immediate to 8 bits (full 256-byte RAM reach).
    - Added ALU opcodes SHL, SHR (with C-flag) and FLT2FIX.
    - Removed branch / jump rows (control flow now hardware-internal).
    - Added detailed write-ups for Internal Operands, Control Flow, and Addressing Modes (ACC/TMP register set, no-branch model, 8-bit direct addressing).
- **Programmer's Model**

- Documented 3-cycle deterministic workflow (LD16 → ALU → ST16).
- Stated that normalization and saturation are *hardware-driven*.
- Completed Programmer's Model questions 4.1, 4.2, and new 4.3 (ALU reuse for future jumps).

- **Individual Component Specs**
  - Added module table with file names, brief descriptions, and schematic placeholders for TopLevel, PC, Inst ROM, Control, ACC, ALU + Normalizer, Data RAM, Muxes.
  - Provided optional test-bench notes for pc_tb and alu_tb.
- **Program Implementation**
  - Supplied complete pseudocode & assembly for
    1. **Program 1:** Fixed → Float (NORM)
    2. **Program 2:** Float → Fixed (FLT2FIX)
    3. **Program 3:** Float + Float addition (FLTADD)
  - Each program annotated with cycle counts.

Milestone 1

- Initial version