VLSI SYSTEM DESIGN

5 stage pipeline RV32I CPU

Table of Contents

I. System Introduction

A. Instruction Set Format

The format of instructions in the RV32I instruction set follows the RISC-V specification

Formats	32 Bits (RV32I)																												
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R type	func7							rs2					rs1					func3			rd					opcode						
I type	imme[11:0]												rs1					func3			rd					opcode						
S type	imme[11:5]							rs2					rs1					func3			imme[4:0]					opcode						
B type	{ imme[12], imme[10:5] }							rs2					rs1					func3			{ imme[4:1], imme[11] }					opcode						
U type	imme[31:12]																				rd					opcode						
J type	{ imme[20], imme[10:1], imme[11], imme[19:12] }																				rd					opcode						

B. Names, Widths, and Functions of the Fields in the Instruction Format

name	width	function
func7	7 bits	Instruction Type Classification
rs1	5 bits	source register1 address
rs2	5 bits	source register2 address
func3	3 bits	Instruction Type Classification
rd	5 bits	destination adress
opcode	7 bits	Instruction Type Classification
imme	Determine based on the type	Immediate

C. Branch & Jump Addressing

1. Branch Addressing

The target address for a branch instruction is calculated as:

$$PC+\{imm,1'b0\}$$

Since imm[0] must be 0, the immediate value is effectively shifted left by 1 bit. This ensures that the resulting address is a multiple of 4, i.e., word-aligned. The final result is the memory address to which the branch will jump.

2. JAL Addressing

   The jump address for the JAL instruction is calculated similarly:

   $$PC+\{imm20,1'b0\}$$

   Here, imm[0] is also implicitly 0, and the calculation is identical to that of a branch instruction.

3. JALR Addressing

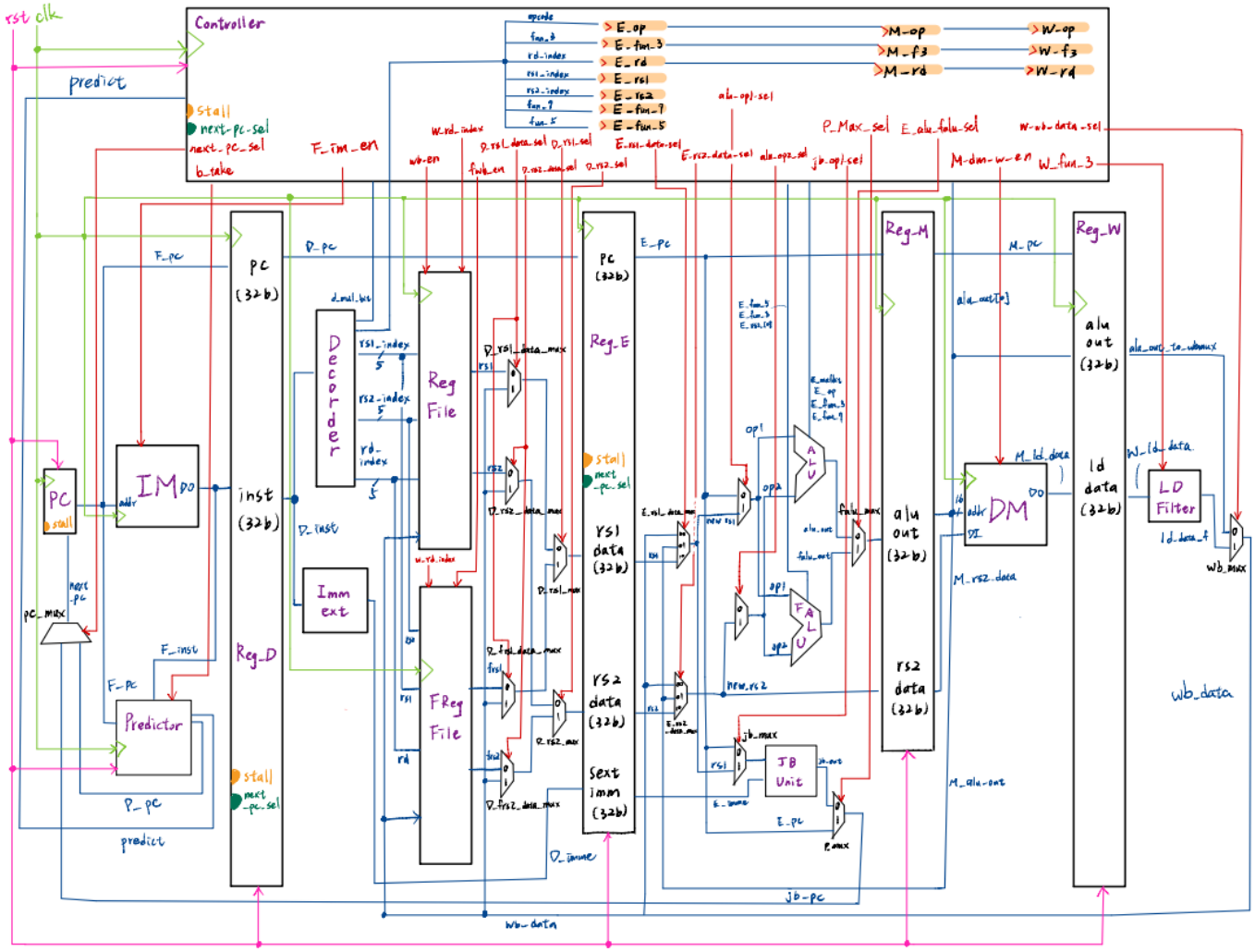
   The target address for JALR is computed as:

   $$(rs1+sext(imm12))\&\sim 32'd1$$

   The bitwise AND with ~32'd1 clears the least significant bit of the computed address, ensuring the result is word-aligned. This alignment is required by the RISC-V specification to prevent jumping to an invalid instruction boundary.

D. System Architecture

a. Architecture Diagram and Description



b. Module Descriptions

i. **IM (Instruction Memory)** – Stores the program instructions to be fetched by the CPU.

ii. **Decoder** – Breaks the 32-bit instruction into its constituent fields so that subsequent pipeline stages can interpret them.

iii. **ImmExt (Immediate Extender)** – Extends the immediate field to a full 32-bit value (sign- or zero-extended as required).

iv. **RegFile (Register File)** – A bank of 32 general-purpose registers used to hold operands, computation results, and addresses.

v. **ALU (Arithmetic Logic Unit)** – Executes arithmetic and logical operations on operand1 and operand2 according to the opcode.

vi. **JB_unit (Jump/Branch Unit)** – Calculates the target address

for branch and jump instructions.

vii. **DM (Data Memory)** – Stores and retrieves data operands for load-store instructions.

viii. **LD Filter (Load Filter)** – Selects the proper data width (word, half-word, or byte) for load instructions and performs sign or zero extension.

ix. **Controller** – Generates all multiplexor select signals and other control lines required by the pipeline stages.

II. Supported Instructions

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

imm[11:0]		rs1	010	rd	0000111	FLW
imm[11:5]	rs2	rs1	010	imm[4:0]	0100111	FSW
0000000	rs2	rs1	rm	rd	1010011	FADD.S
0000100	rs2	rs1	rm	rd	1010011	FSUB.S
0001000	rs2	rs1	rm	rd	1010011	FMUL.S
0010100	rs2	rs1	000	rd	1010011	FMIN.S
0010100	rs2	rs1	001	rd	1010011	FMAX.S
1100000	00000	rs1	rm	rd	1010011	FCVT.W.S
1100000	00001	rs1	rm	rd	1010011	FCVT.WU.S
1110000	00000	rs1	000	rd	1010011	FMV.X.W
1010000	rs2	rs1	010	rd	1010011	FEQ.S
1010000	rs2	rs1	001	rd	1010011	FLT.S
1010000	rs2	rs1	000	rd	1010011	FLE.S
1101000	00000	rs1	rm	rd	1010011	FCVT.S.W
1101000	00001	rs1	rm	rd	1010011	FCVT.S.WU
1111000	00000	rs1	000	rd	1010011	FMV.W.X

0000001	rs2	rs1	000	rd	0110011	MUL

III. Verification Methodology and Results Analysis

A. Verification Methodology

1. Basic-Instruction Test: prog0 is a test program that exercises the core RV32I instruction set (e.g., add, sub, addi). After execution, each instruction writes its result to data memory. The testbench then compares these values against a golden data set; if all comparisons match, the console prints "simulation pass."

```
Done

DM['h9000] = fffffff0, pass
DM['h9004] = fffffff8, pass
DM['h9008] = 00000008, pass
DM['h900c] = 00000001, pass
DM['h9010] = 00000001, pass
DM['h9014] = 78787878, pass
DM['h9018] = 000091a2, pass
DM['h901c] = 00000003, pass
DM['h9020] = fefcfefd, pass
DM['h9024] = 10305070, pass
DM['h9028] = cccccccc, pass
DM['h902c] = ffffffcc, pass
DM['h9030] = ffffcccc, pass
DM['h9034] = 000000cc, pass
DM['h9038] = 0000cccc, pass
DM['h903c] = 00000d9d, pass
DM['h9040] = 00000004, pass
DM['h9044] = 00000003, pass
DM['h9048] = 000001a6, pass
DM['h904c] = 00000ec6, pass
DM['h9050] = 2468b7a8, pass
DM['h9054] = 5dbf9f00, pass
DM['h9058] = 00012b38, pass
DM['h905c] = fa2817b7, pass
DM['h9060] = ff000000, pass
DM['h9064] = 12345678, pass
DM['h9064] = 12345678, pass
DM['h9068] = 0000f000, pass
DM['h906c] = 00000f00, pass
DM['h9070] = 000000f0, pass
DM['h9074] = 0000000f, pass
DM['h9078] = 56780000, pass
DM['h907c] = 78000000, pass
DM['h9080] = 00005678, pass
DM['h9084] = 00000078, pass
DM['h9088] = 12345678, pass
DM['h908c] = ce780000, pass
DM['h9090] = fffff000, pass
DM['h9094] = fffff000, pass
DM['h9098] = fffff000, pass
DM['h909c] = fffff000, pass
DM['h90a0] = fffff000, pass
DM['h90a4] = fffff000, pass
DM['h90a8] = 13579d7c, pass
DM['h90ac] = 13578000, pass
DM['h90b0] = fffff004, pass
    ****************************
    **                        **
    **  Waku Waku !!          **
    **                        **
    **  Simulation PASS !!    **
    **                        **
    ****************************
```

2. **Multiply-Instruction Test:** prog4 exercises the RV32M multiply (mul) instruction. After execution, each result is written to data memory, and the testbench compares these values against a golden reference. If every comparison matches, the console prints “simulation pass.”

```
DM['h9000] = 00000002, pass
DM['h9004] = 00000004, pass
DM['h9008] = fffffff8, pass
DM['h900c] = 00000010, pass
DM['h9010] = ffffffe0, pass
DM['h9014] = 00000040, pass
DM['h9018] = ffffff80, pass
DM['h901c] = ffffff00, pass
DM['h9020] = fffffe00, pass
DM['h9024] = 00000000, pass
    ****************************
    **                        **
    **  Waku Waku !!          **
    **                        **
    **  Simulation PASS !!    **
    **                        **
    ****************************
```

3. **Floating-Point Instruction Test:** prog3 targets the RV32F extension, exercising instructions such as FLW and FSW. Once the program finishes, each result is written to data memory; the testbench then compares these values with a golden data set. If every comparison matches, the console outputs **“simulation pass.”**

```
DM['h9000] = c0100000, pass
DM['h9004] = 40100000, pass
DM['h9008] = c0400000, pass
DM['h900c] = fffffffd, pass
DM['h9010] = 00000002, pass
DM['h9014] = ffffffff, pass
DM['h9018] = fffffffe, pass
DM['h901c] = 00000003, pass
DM['h9020] = 00000002, pass
DM['h9024] = 00000001, pass
DM['h9028] = 00000002, pass
DM['h902c] = c0400000, pass
DM['h9030] = 3fc00000, pass
DM['h9034] = bf000000, pass
DM['h9038] = bfc00000, pass
DM['h903c] = 40800000, pass
DM['h9040] = 40200000, pass
DM['h9044] = bf800000, pass
DM['h9048] = c0200000, pass
DM['h904c] = 40400000, pass
DM['h9050] = 3f800000, pass
DM['h9054] = bf800000, pass
DM['h9058] = c0400000, pass
DM['h905c] = 40400000, pass
DM['h9060] = 3f800000, pass
DM['h9064] = 3f800000, pass
DM['h9068] = 40400000, pass
DM['h906c] = 3e000000, pass
DM['h9070] = 3fa00000, pass
DM['h9074] = 3fa00000, pass
DM['h9078] = 3e000000, pass
DM['h907c] = 00000000, pass
DM['h9080] = 00000001, pass
DM['h9084] = 00000000, pass
DM['h9088] = 00000000, pass
DM['h908c] = 00000000, pass
DM['h9090] = 00000001, pass
    ****************************
    **                        **
    **  Waku Waku !!          **
    **                        **
    **  Simulation PASS !!    **
    **                        **
    ****************************
```

IV. Results Analysis (nWave Screenshots)

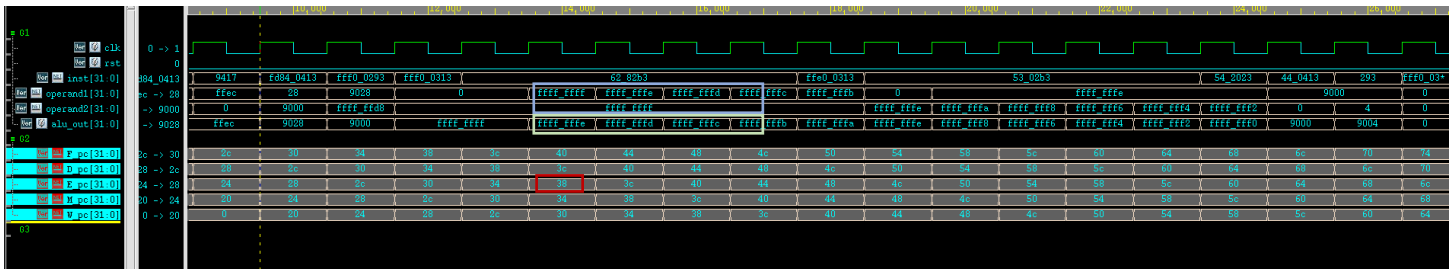
A. Single-Instruction Correctness

1. **add rd,rs1,rs2**:

a. The value in rs1 is read into operand1, and the value in rs2 is read into operand2 (highlighted in blue on the waveform). The ALU then produces the resulting value (highlighted in green).

b. At program counter PC = 0x38 (marked in red), the fetched instruction is ADD, so the ALU output equals operand1 + operand2.

c. In the left-most boxed area of the nWave screenshot, the example shows (−1) + (−1) = −2.

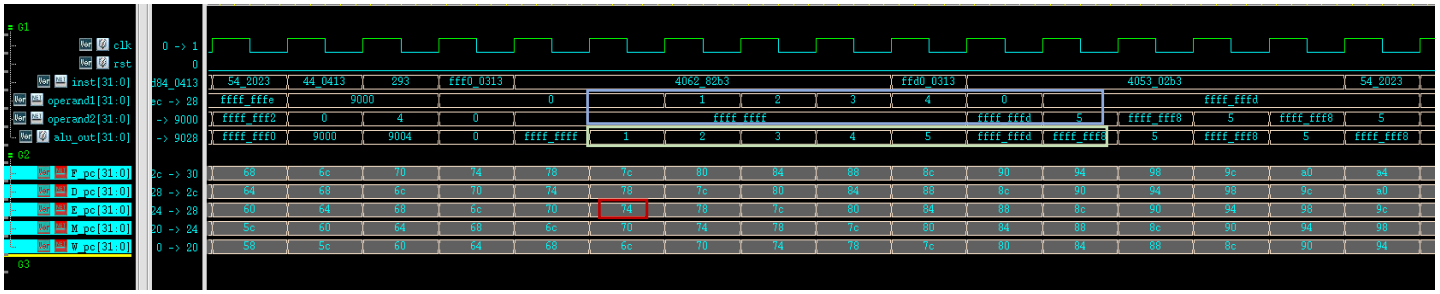


2. **sub rd,rs1,rs2**

   a. The value in rs1 is read into operand1, while the value in rs2 is read into operand2 (highlighted in **blue** on the waveform). The ALU then produces the result (highlighted in **green**).

   b. At program counter PC = 0x74 (marked in **red**), the fetched instruction is SUB, so the ALU output equals operand1 − operand2.

   c. In the left-most boxed region of the nWave screenshot, the example shows **(0) − (−1) = 1**.

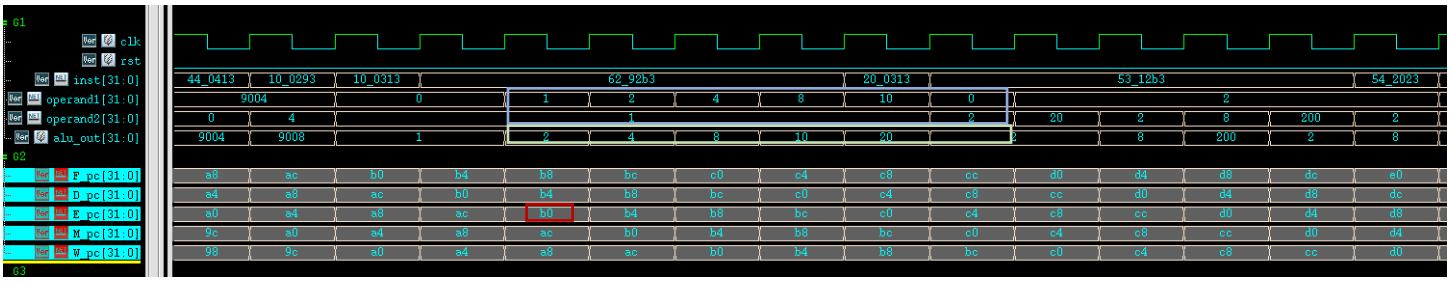


3. **sll rd,rs1,rs2**

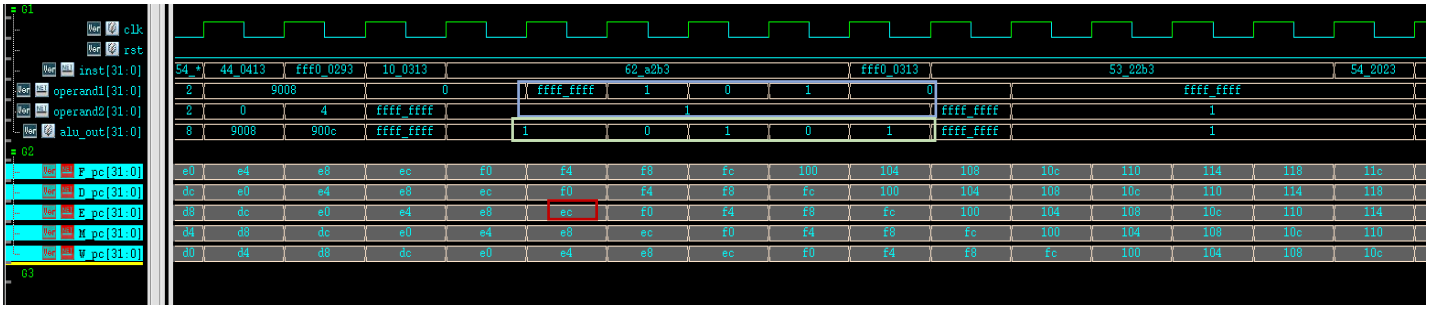
   a. The value in rs1 is loaded into operand1, and the value in rs2 is loaded into operand2 (highlighted in blue on the waveform). The ALU then generates the result (highlighted in green).

   b. At program counter PC = 0xB0 (marked in red), the fetched instruction is SLL, so the ALU output equals operand1 << operand2.

   c. In the left-most boxed area of the nWave screenshot, the example shows 1 << 1 = 2.



4. **slt rd,rs1,rs2**

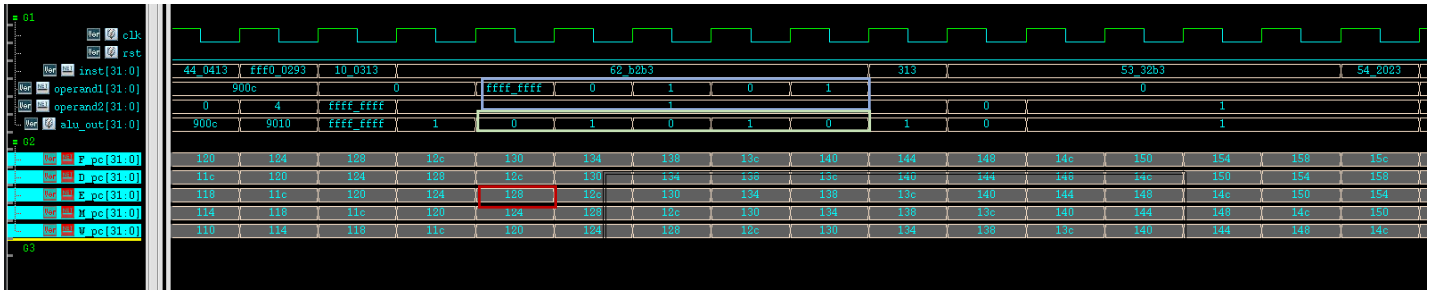


a. The value in rs1 is loaded into operand1, and the value in rs2 is loaded into operand2 (highlighted in blue on the waveform). The ALU then produces the result (highlighted in green).

b. At program counter PC = 0xEC (marked in red), the fetched instruction is SLT, so the ALU output is the Boolean result of operand1 < operand2.

c. In the left-most boxed area of the nWave screenshot, the example shows (−1) < 1 → true → 1.

5. **sltu rd,rs1,rs2**

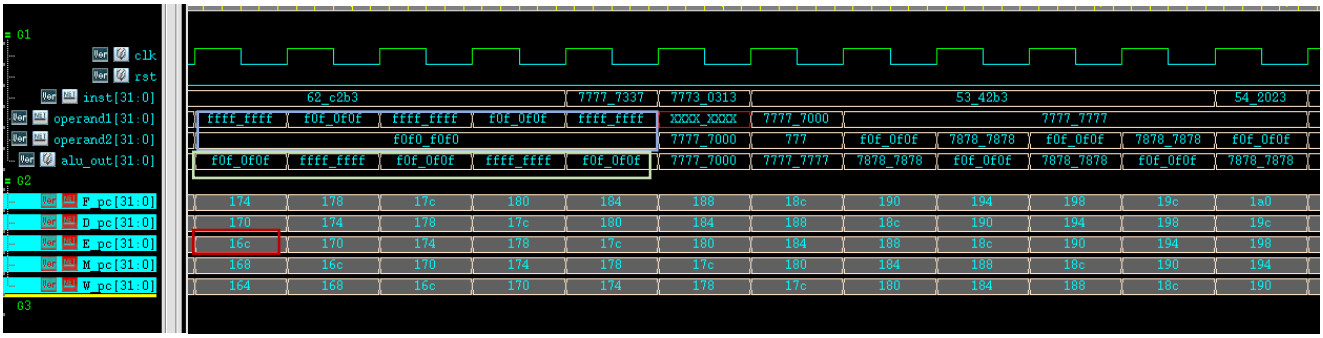


a. The value in rs1 is loaded into operand1, and the value in rs2 is loaded into operand2 (highlighted in blue on the waveform). The ALU then produces the result (highlighted in green).

b. At program counter PC = 0x128 (marked in red), the fetched instruction is SLTU, so the ALU output is the Boolean result of the unsigned comparison operand1 < operand2.

c. In the left-most boxed area of the nWave screenshot, the example shows 0xFFFF_FFFF < 1 → false → 0.

6. **xor rd,rs1,rs2**

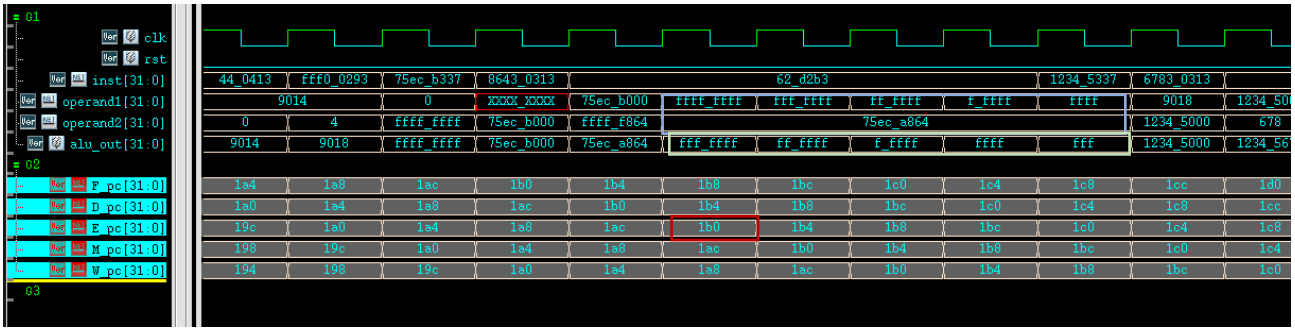


a. The value in rs1 is loaded into operand1, and the value in rs2 is loaded into operand2 (highlighted in blue on the waveform). The ALU then computes the result (highlighted in green).

b. At program counter PC = 0x16C (marked in red), the fetched instruction is XOR, so the ALU output equals operand1 ^ operand2.

c. In the left-most boxed area of the nWave screenshot, the example shows 0xFFFF_FFFF ^ 0xF0F0_F0F0 = 0x0F0F_0F0F.

7. **srl rd,rs1,rs2**

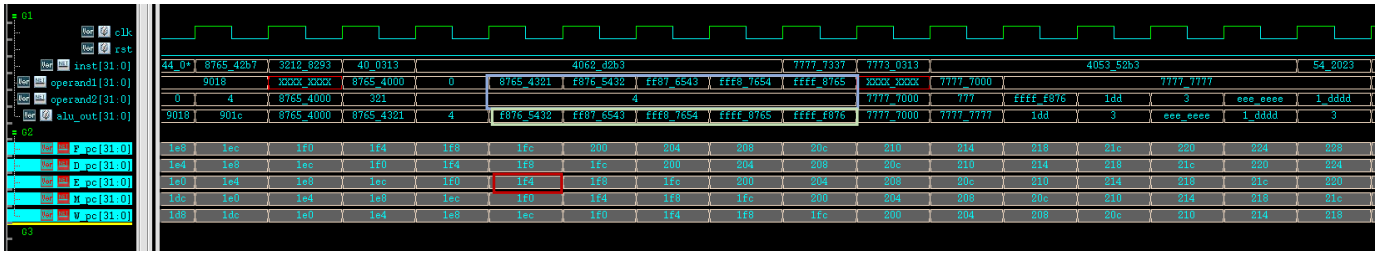


a. The value in rs1 is loaded into operand1, and the value in rs2 is loaded into operand2 (highlighted in blue on the waveform). The ALU then computes the result (highlighted in green).

b. At program counter PC = 0x1B0 (marked in red), the fetched instruction is SRL, so the ALU output equals operand1 >> operand2[4:0].

c. In the left-most boxed region of the nWave screenshot, the example shows 0xFFFF_FFFF >> 2 = 0x3FFF_FFFF.

8. **sra rd,rs1,rs2**

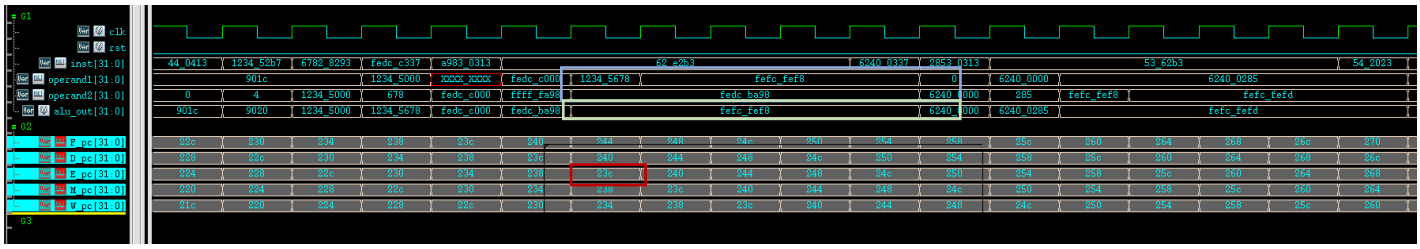


a. The value in rs1 is loaded into operand1, and the value in rs2 is loaded into operand2 (highlighted in blue on the waveform). The ALU then produces the result (highlighted in green).

b. At program counter PC = 0x1F4 (marked in red), the fetched instruction is SRA (arithmetic right shift). Consequently, the ALU output equals operand1 >> operand2[4:0], with sign-extension.

c. In the left-most boxed area of the nWave screenshot, the example shows 0x8765_4321 >> 2 = 0xF876_5432.

9. **or rd,rs1,rs2**

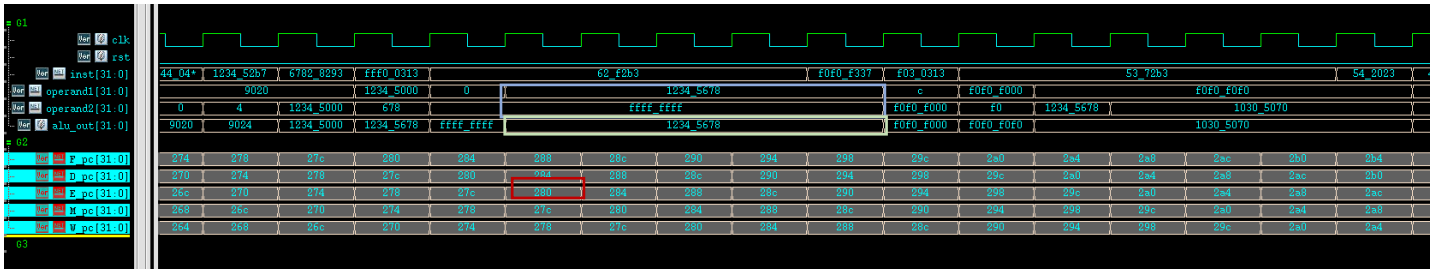


a. The value in rs1 is loaded into operand1, and the value in rs2 is loaded into operand2 (highlighted in blue on the waveform). The ALU then computes the result (highlighted in green).

b. At program counter PC = 0x23C (marked in red), the fetched instruction is OR, so the ALU output equals operand1 | operand2.

c. In the left-most boxed area of the nWave screenshot, the example shows 0x1234_5678 | 0xFEDC_BA98 = 0xF876_5432.

10. **and rd,rs1,rs2**

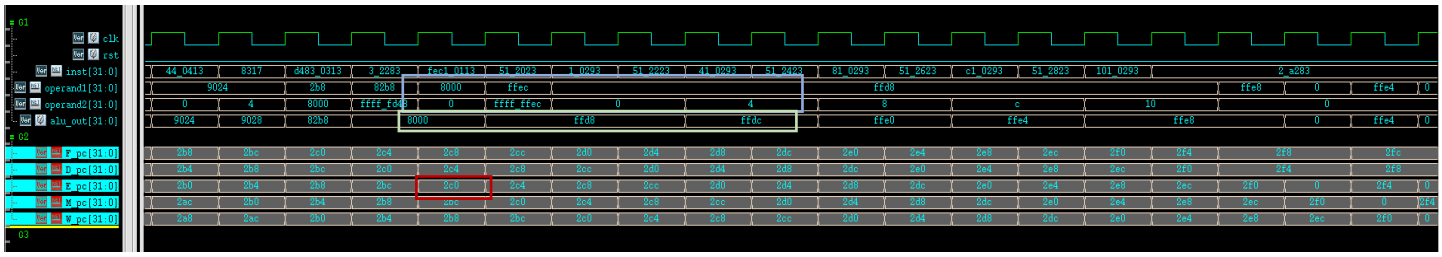


a. The value in rs1 is loaded into operand1, and the value in rs2 is loaded into operand2 (highlighted in blue on the waveform). The ALU then produces the result (highlighted in green).

b. At program counter PC = 0x280 (marked in red), the fetched instruction is AND, so the ALU output equals operand1 & operand2.

c. In the left-most boxed region of the nWave screenshot, the example shows 0x1234 5678 & 0xFFFF FFFF = 0x1234 5678.

11. **lw rd,imm(rs1)**

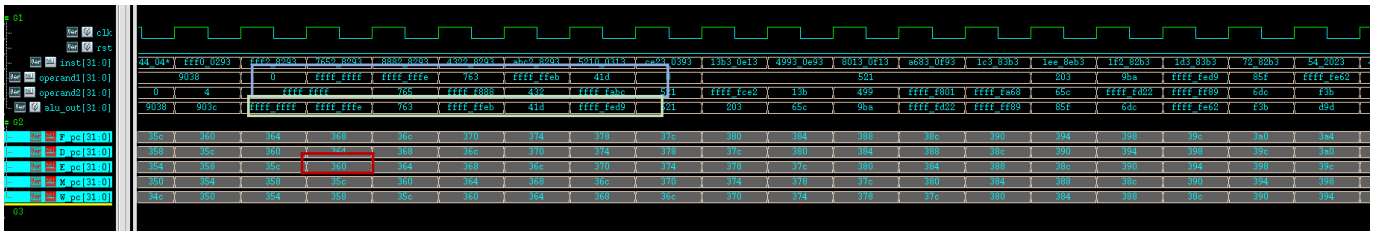


a. The value in rs1 is loaded into operand1, and the immediate value is loaded into operand2 (highlighted in blue on the waveform). The ALU then computes the effective load address (highlighted in green).

b. At program counter PC = 0x2C0 (marked in red), the fetched instruction is LW, so the ALU output equals rs1 + imm, i.e., the address from which the data will be loaded.

c. In the left-most boxed region of the nWave screenshot, the example shows 0x8000 + 0 = 0x8000.

12. **addi rd,rs1,imm**

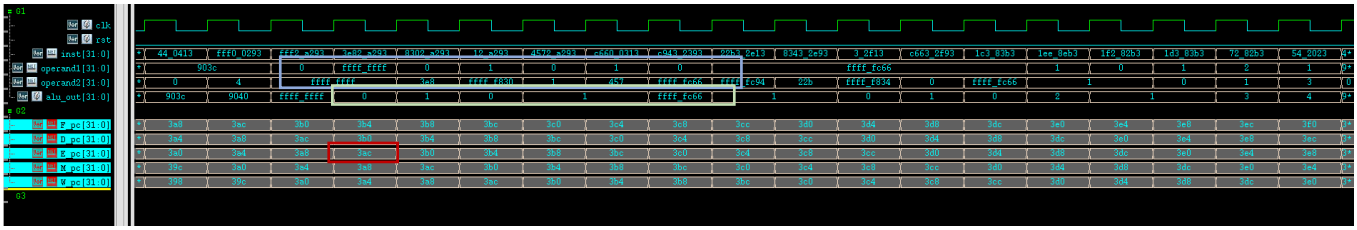


a. The value in rs1 is loaded into operand1, and the immediate value is loaded into operand2 (highlighted in blue on the waveform). The ALU then produces the result (highlighted in green).

b. At program counter PC = 0x360 (marked in red), the fetched instruction is ADDI, so the ALU output equals operand1 + operand2.

c. In the left-most boxed region of the nWave screenshot, the example shows (−1) + (−1) = −2.

13. **slti rd,rs1,imm**

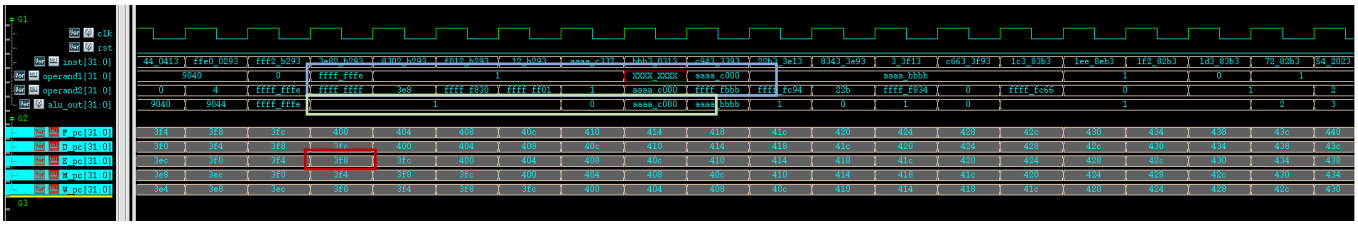


a. The value in rs1 is loaded into operand1, while the immediate value is loaded into operand2 (highlighted in blue on the waveform). The ALU then produces the result (highlighted in green).

b. At program counter PC = 0x360 (marked in red), the fetched instruction is SLTI, so the ALU output is the Boolean result of operand1 < operand2.

c. In the left-most boxed area of the nWave screenshot, the example shows (−1) < (−1) → false → 0.

14. **sltiu rd,rs1,imm**

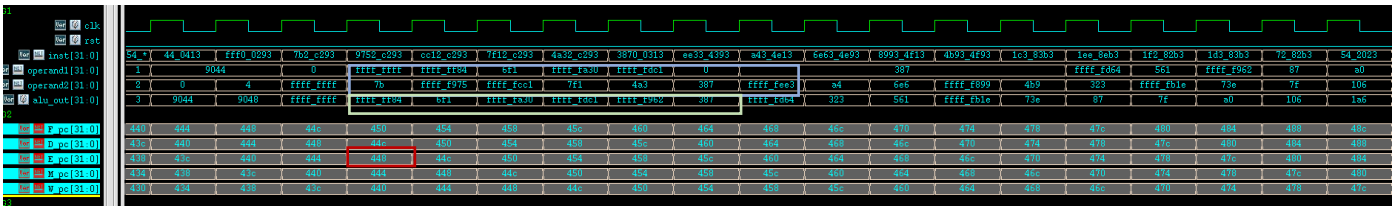


a. The value in rs1 is loaded into operand1, and the immediate value is loaded into operand2 (highlighted in blue on the waveform). The ALU then produces the result (highlighted in green).

b. At program counter PC = 0x3F8 (marked in red), the fetched instruction is SLTIU, so the ALU output is the Boolean result of the unsigned comparison operand1 < operand2.

c. In the left-most boxed region of the nWave screenshot, the example shows (−2) < (−1) when interpreted as unsigned values, which evaluates to true → 1.

15. **xori rd,rs1,imm**

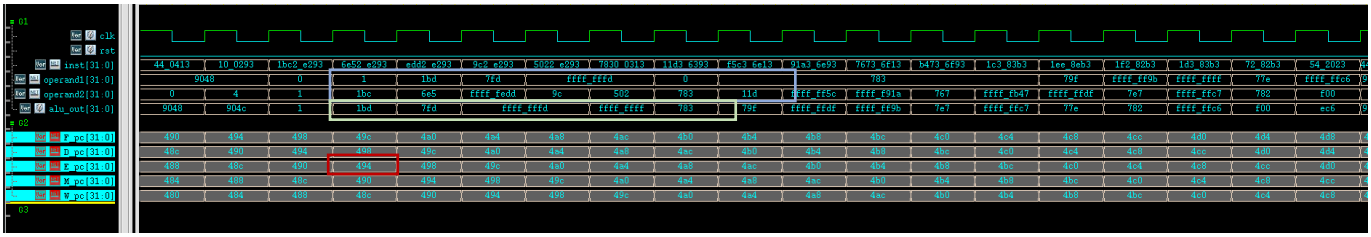


a. The value in rs1 is loaded into operand1, and the immediate value is loaded into operand2 (highlighted in blue on the waveform). The ALU then produces the result (highlighted in green).

b. At program counter PC = 0x448 (marked in red), the fetched instruction is XORI, so the ALU output equals operand1 ^ operand2.

c. In the left-most boxed region of the nWave screenshot, the example shows 0xFFFF_FFFF ^ 0x0000_007B = 0xFFFF_FF84.

16. **ori rd,rs1,imm**

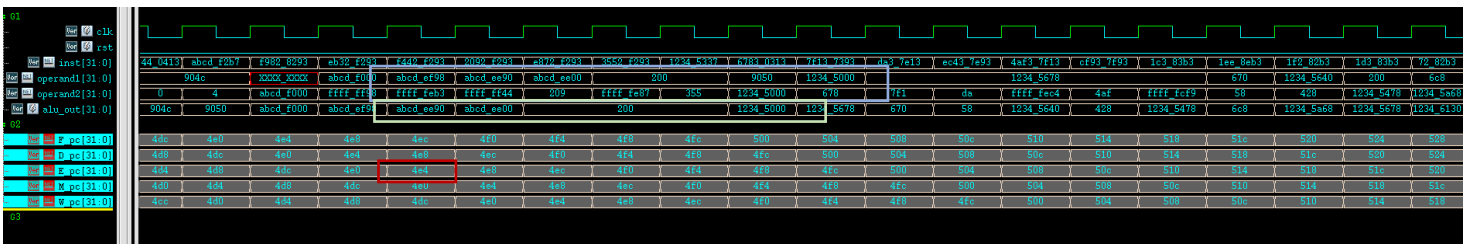


a. The value in rs1 is loaded into operand1, and the immediate value is loaded into operand2 (highlighted in blue on the waveform). The ALU then computes the result (highlighted in green).

b. At program counter PC = 0x494 (marked in red), the fetched instruction is ORI, so the ALU output equals operand1 | operand2.

c. In the left-most boxed area of the nWave screenshot, the example shows 0x1 | 0x1BC = 0x1BD.

17. **andi rd,rs1,imm**

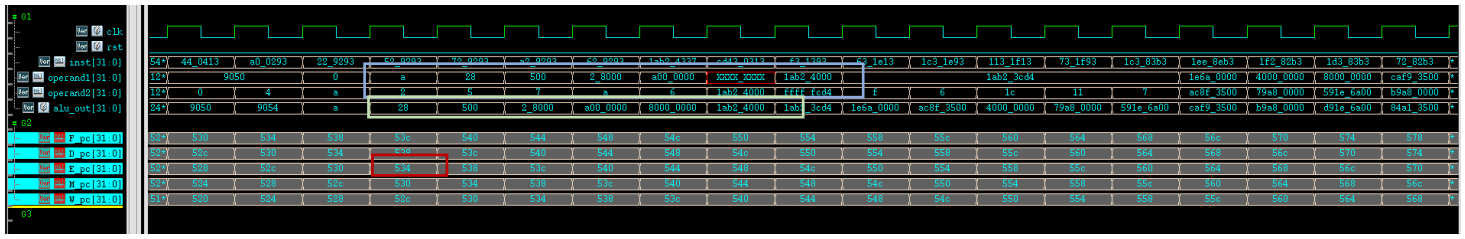


a. The value in rs1 is loaded into operand1, and the immediate value is loaded into operand2 (highlighted in blue on the waveform). The ALU then produces the result (highlighted in green).

b. At program counter PC = 0x4E4 (marked in red), the fetched instruction is ANDI, so the ALU output equals operand1 & operand2.

c. In the left-most boxed region of the nWave screenshot, the example shows 0xABCD_EF98 & 0xFFFF_FEB3 = 0xABCD_EE90.

18. **slli rd,rs1,imm**

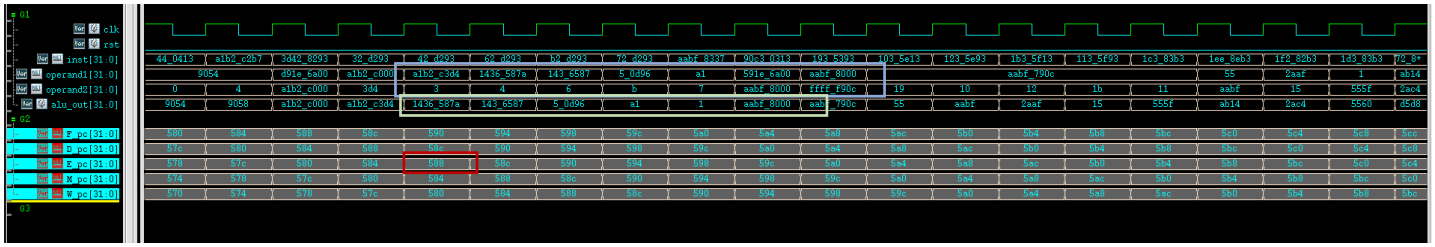


a. The value in rs1 is loaded into operand1, and the immediate value is loaded into operand2 (highlighted in blue on the waveform). The ALU then produces the result (highlighted in green).

b. At program counter PC = 0x534 (marked in red), the fetched instruction is SLLI, so the ALU output equals operand1 << operand2.

c. In the left-most boxed region of the nWave screenshot, the example shows 0xA << 2 = 0x28 (binary 1010 shifted left by 2, which is 40 in decimal).

19. **srli rd,rs1,imm**

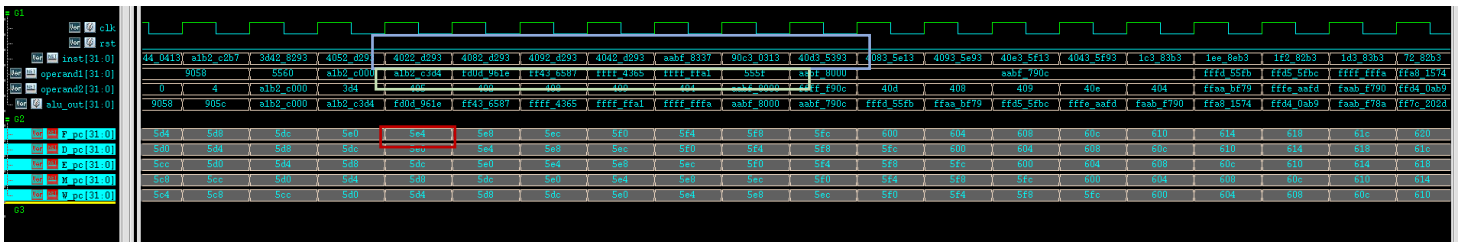


a. The value in rs1 is loaded into operand1, and the immediate value is loaded into operand2 (highlighted in blue on the waveform). The ALU then produces the result (highlighted in green).

b. At program counter PC = 0x588 (marked in red), the fetched instruction is SRLI, so the ALU output equals operand1 >> operand2.

c. In the left-most boxed region of the nWave screenshot, the example shows 0xA1B2_C3D4 >> 3 = 0x1436_587A.

20. **srai rd,rs1,imm**

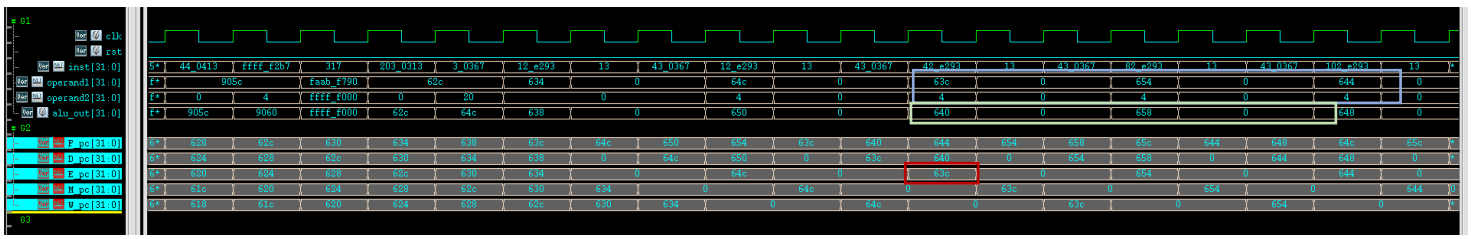


a. The value in rs1 is loaded into operand1, and the immediate value is loaded into operand2 (highlighted in blue on the waveform). The ALU then produces the result (highlighted in green).

b. At program counter PC = 0x5DC (marked in red), the fetched instruction is SRAI (arithmetic right-shift immediate). Accordingly, the ALU output equals operand1 >> operand2[4:0], with sign extension.

c. In the left-most boxed area of the nWave screenshot, the example shows 0xA1B2_C3D4 >> 5 = 0xFD0D_961E.

21. **jalr rd,rs1,imm**

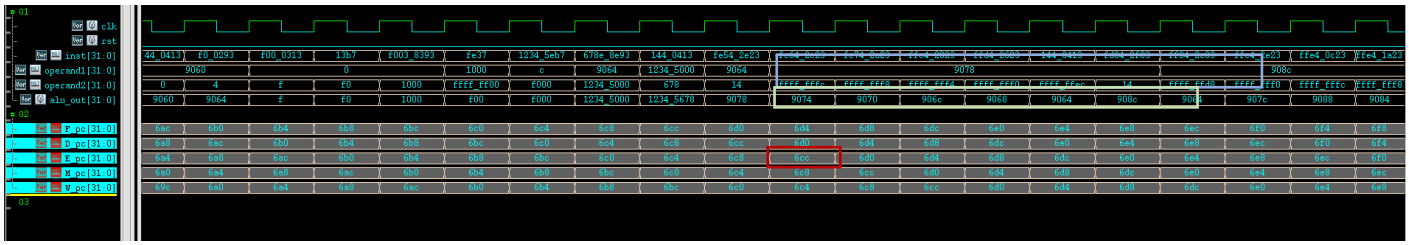


a. The value in rs1 is loaded into operand1, and the immediate value is loaded into operand2 (highlighted in blue on the waveform). The ALU then produces the result (highlighted in green).

b. At program counter PC = 0x5DC (marked in red), the fetched instruction is SRAI (arithmetic right-shift immediate). Accordingly, the ALU output equals operand1 >> operand2[4:0], with sign extension.

c. In the left-most boxed area of the nWave screenshot, the example shows 0xA1B2_C3D4 >> 5 = 0xFD0D_961E.

22. **sw rd,imm(rs1)**

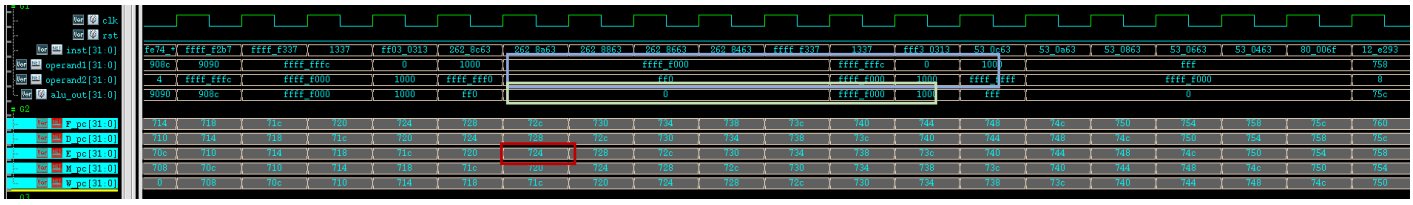


a. The value in rs1 is loaded into operand1, and the immediate value is loaded into operand2 (highlighted in blue on the waveform). The ALU then computes the store (write) address, shown in green.

b. At program counter PC = 0x6CC (marked in red), the fetched instruction is SW, so the ALU output gives the effective address where the data will be written (rs1 + imm).

c. In the left-most boxed region of the nWave screenshot, the example shows 0x9078 + (−4) = 0x9074.

23. **beq rd,rs1,imm**

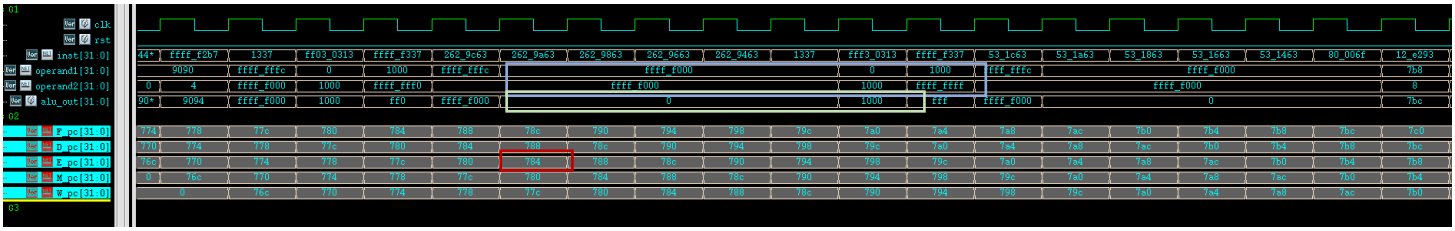


a. rs1 的值讀到 operand1，imm 的值讀到 operand2 (藍)，最後由 alu 計算得到 operand1!=operand2　的 bool 值(綠)。

b. 程式碼中 pc=784 (紅)是 bne 指令，所以 alu 輸出為 (operand1!=operand2)。

c. 方框最左邊展示的是(ffff_f000 != ffff_f0000)=0。

24. **bne rd,rs1,imm**



a. The value in rs1 is loaded into operand1, and the immediate value is loaded into operand2 (highlighted in blue on the waveform). The ALU then evaluates the inequality, producing the Boolean result shown in green.

b. At program counter PC = 0x784 (marked in red), the fetched instruction is BNE (branch-if-not-equal); therefore, the ALU output reflects operand1 != operand2.

c. In the left-most boxed region of the nWave screenshot, the example shows 0xFFFF_F000 ≠ 0xFFFF_F0000 → false → 0.

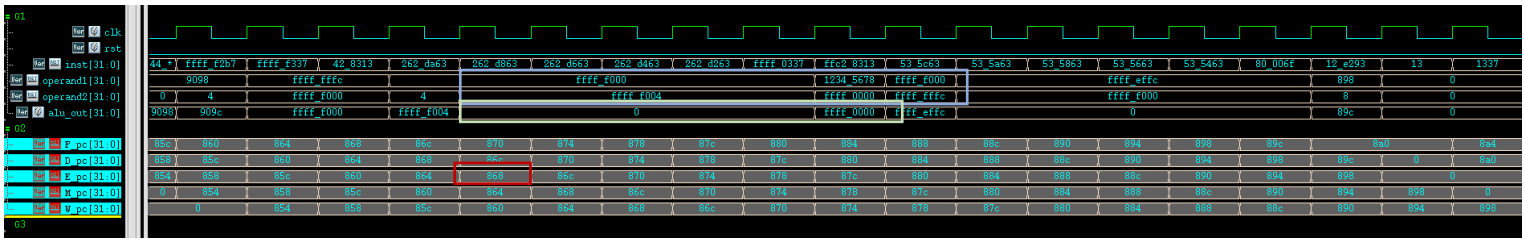
25. **blt rd,rs1,imm**

a. The value in rs1 is loaded into operand1, and the immediate value is loaded into operand2 (highlighted in blue on the waveform). The ALU then performs a signed comparison, producing the Boolean result shown in green.

b. At program counter PC = 0x7E4 (marked in red), the fetched instruction is BLT (branch-if-less-than), so the ALU output reflects whether operand1 < operand2 (signed comparison).

c. In the left-most boxed area of the nWave screenshot, the example comparison 0xFFFF_F000 < 0xFFFF_EFFC evaluates to false → 0.

26. **bge rd,rs1,imm**

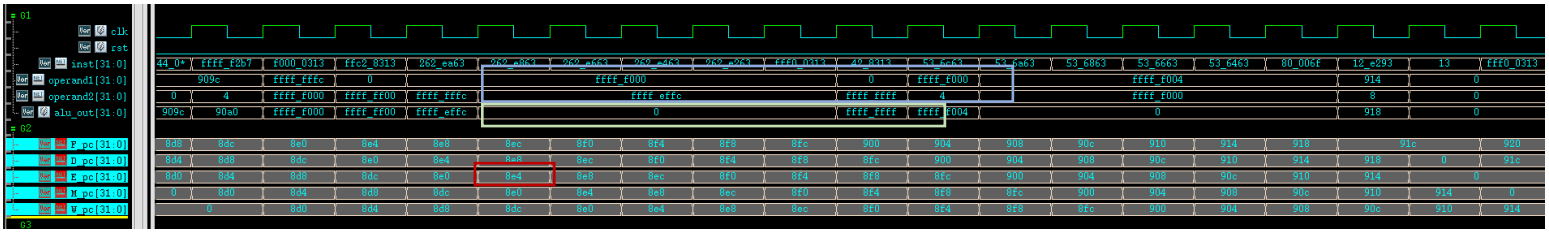


a. The value in rs1 is loaded into operand1, and the immediate value is loaded into operand2 (highlighted in blue on the waveform). The ALU then performs a signed comparison, producing the Boolean result shown in green.

b. At program counter PC = 0x868 (marked in red), the fetched instruction is BGE (branch-if-greater-than-or-equal), so the ALU output reflects whether operand1 ≥ operand2 (signed comparison).

c. In the left-most boxed area of the nWave screenshot, the example comparison 0xFFFF_F000 ≥ 0xFFFF_F004 evaluates to false → 0.

27. **bltu rd,rs1,imm**

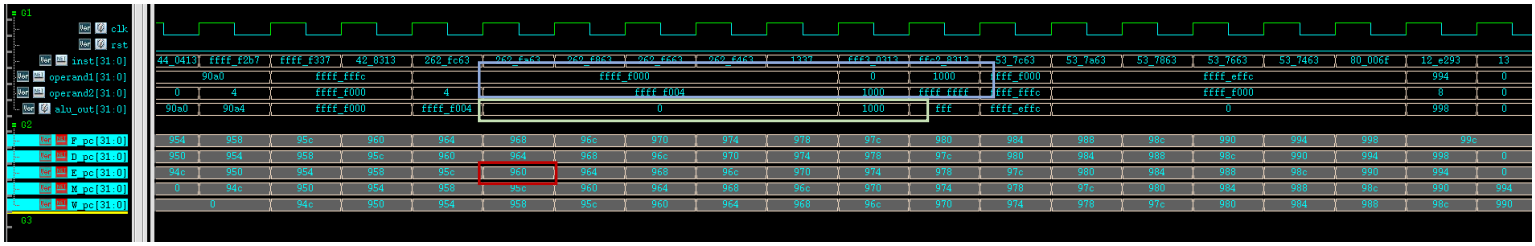


a. The value in rs1 is loaded into operand1, and the immediate value is loaded into operand2 (highlighted in blue on the waveform). The ALU then performs an unsigned comparison, producing the Boolean result shown in green.

b. At program counter PC = 0x8E4 (marked in red), the fetched instruction is BLTU (branch-if-less-than unsigned), so the ALU output reflects whether operand1 < operand2 in unsigned arithmetic.

c. In the left-most boxed region of the nWave screenshot, the example comparison 0xFFFF_F000 < 0xFFFF_EFFC evaluates to false → 0.

28. **bgeu rd,rs1,imm**

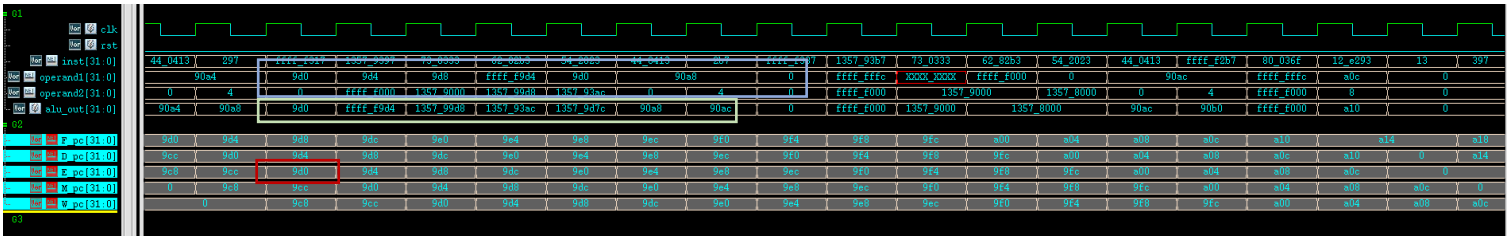


a. The value in rs1 is loaded into operand1, and the immediate value is loaded into operand2 (highlighted in blue on the waveform). The ALU then performs an unsigned comparison, producing the Boolean result shown in green.

b. At program counter PC = 0x960 (marked in red), the fetched instruction is BGEU (branch-if-greater-than-or-equal unsigned), so the ALU output reflects whether operand1 ≥ operand2 in unsigned arithmetic.

c. In the left-most boxed area of the nWave screenshot, the example comparison 0xFFFF_F000 ≥ 0xFFFF_F004 evaluates to false → 0.

29. **auipc rd,imm**

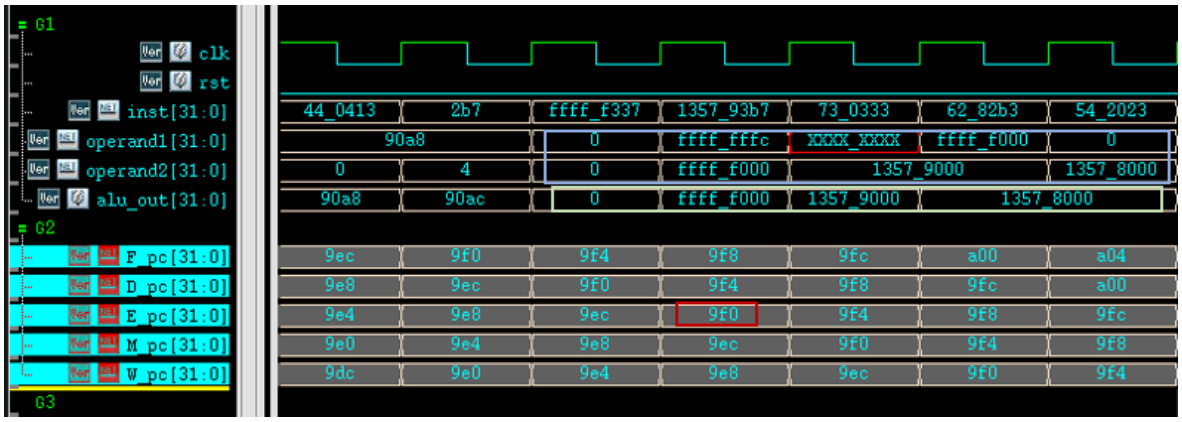


a. The current PC value is loaded into operand1, and the 20-bit immediate (shifted left by 12) is loaded into operand2 (highlighted in blue on the waveform). The ALU then adds the two, producing the result shown in green.

b. At program counter PC = 0x9D0 (marked in red), the fetched instruction is AUIPC (Add Upper Immediate to PC), so the ALU output is operand1 + operand2.

c. In the left-most boxed region of the nWave screenshot, the example shows 0x9D0 + 0 = 0x9D0.

30. **lui rd,imm**

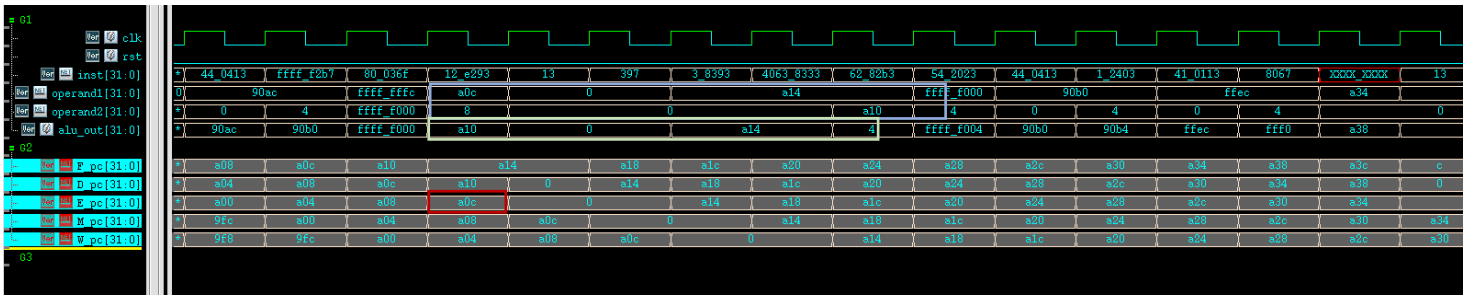


a. The 20-bit immediate value is loaded into operand2 (highlighted in blue on the waveform). The ALU then concatenates the immediate with twelve trailing zeros—{imm, 12'b0}—producing the result shown in green.

b. At program counter PC = 0x9F0 (marked in red), the fetched instruction is LUI (Load Upper Immediate), so the ALU output is exactly {imm, 12'b0}.

c. In the left-most boxed region of the nWave screenshot, the example shows 0xFFFFF → 0xFFFF_F000.

31. **jal rd,imm**



a. The ALU calculates PC + 4 and writes it to rd.

b. At program counter PC = 0xA0C (marked in red), the fetched instruction is JAL, so the ALU output equals PC + 4.

c. In the left-most boxed region of the waveform screenshot, the example shows 0xA0C + 4 = 0xA10.

V. Program Correctness

A. Sorting (Assembly Language)

1. The code shown above is an assembly implementation of **merge sort**.The test data consist of three sets:

test1: 3, 41, 18, 8, 40, 6, 45, 1, 18, 10, 24, 46, 37, 23, 43, 12, 3, 37, 0, 15, 11, 49, 47, 27, 23, 30, 16, 10, 45, 39, 1, 23, 40, 38

test2: −3, −23, −22, −6, −21, −19, −1, 0, −2, −47, −17, −46, −6, −30, −50, −13, −47, −9, −50

test3: −46, 0, −29, −2, 23, −46, 46, 9, −18, −23, 35, −37, 3, −24, −18, 22, 0, 15, −43, −16, −17, −42, −49, −29, 19, −44, 0, −18, 23

2. Below are the merge-sort execution results (displayed in hexadecimal).

Test1	Test2	Test3
		DM['h90d4] = fffffcf, pass DM['h90d8] = ffffffd2, pass DM['h90dc] = ffffffd2, pass DM['h90e0] = ffffffd4, pass DM['h90e4] = ffffffd5, pass DM['h90e8] = ffffffd6, pass DM['h90ec] = ffffffdb, pass DM['h90f0] = ffffffe3, pass DM['h90f4] = ffffffe3, pass DM['h90f8] = ffffffe8, pass DM['h90fc] = ffffffe9, pass DM['h9100] = ffffffee, pass DM['h9104] = ffffffee, pass

```
DM['h9000] = 00000000, pass
DM['h9004] = 00000001, pass
DM['h9008] = 00000001, pass
DM['h900c] = 00000003, pass
DM['h9010] = 00000003, pass
DM['h9014] = 00000006, pass
DM['h9018] = 00000008, pass
DM['h901c] = 0000000a, pass
DM['h9020] = 0000000a, pass
DM['h9024] = 0000000b, pass
DM['h9028] = 0000000c, pass
DM['h902c] = 0000000f, pass
DM['h9030] = 00000010, pass
DM['h9034] = 00000012, pass
DM['h9038] = 00000012, pass
DM['h903c] = 00000017, pass
DM['h9040] = 00000017, pass
DM['h9044] = 00000017, pass
DM['h9048] = 00000018, pass
DM['h904c] = 0000001b, pass
DM['h9050] = 0000001e, pass
DM['h9054] = 00000025, pass
DM['h9058] = 00000025, pass
DM['h905c] = 00000026, pass
DM['h9060] = 00000027, pass
DM['h9064] = 00000028, pass
DM['h9068] = 00000028, pass
DM['h906c] = 00000029, pass
DM['h9070] = 0000002b, pass
DM['h9074] = 0000002d, pass
DM['h9078] = 0000002d, pass
DM['h907c] = 0000002e, pass
DM['h9080] = 0000002f, pass
DM['h9084] = 00000031, pass
```

```
DM['h9088] = fffffffce, pass
DM['h908c] = fffffffce, pass
DM['h9090] = fffffffd1, pass
DM['h9094] = fffffffd1, pass
DM['h9098] = fffffffd2, pass
DM['h909c] = fffffffe2, pass
DM['h90a0] = fffffffe9, pass
DM['h90a4] = fffffffea, pass
DM['h90a8] = fffffffeb, pass
DM['h90ac] = fffffffed, pass
DM['h90b0] = fffffffef, pass
DM['h90b4] = ffffffff3, pass
DM['h90b8] = ffffffff7, pass
DM['h90bc] = fffffffa, pass
DM['h90c0] = fffffffa, pass
DM['h90c4] = fffffffd, pass
DM['h90c8] = fffffffe, pass
DM['h90cc] = ffffffff, pass
DM['h90d0] = 00000000, pass
```

```
DM['h9108] = fffffffee, pass
DM['h910c] = fffffffef, pass
DM['h9110] = ffffffff0, pass
DM['h9114] = ffffffffe, pass
DM['h9118] = 00000000, pass
DM['h911c] = 00000000, pass
DM['h9120] = 00000000, pass
DM['h9124] = 00000003, pass
DM['h9128] = 00000009, pass
DM['h912c] = 0000000f, pass
DM['h9130] = 00000013, pass
DM['h9134] = 00000016, pass
DM['h9138] = 00000017, pass
DM['h913c] = 00000017, pass
DM['h9140] = 00000023, pass
DM['h9144] = 0000002e, pass
    **************************
    **                      **
    **  Waku Waku !!        **
    **                      **
    **  Simulation PASS !!  **
    **                      **
    **************************
```

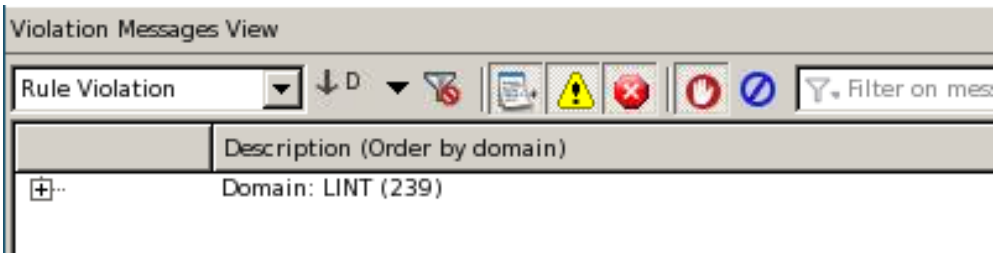
B. Fibonacci Sequence (Assembly Language)

1. The following are the Fibonacci-sequence results (displayed in hexadecimal).

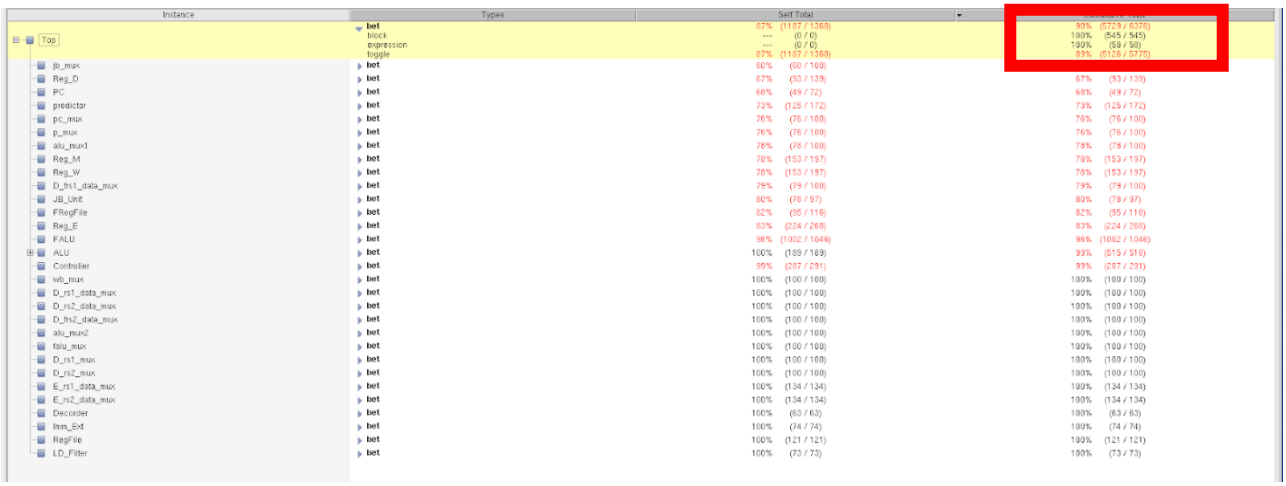
```
DM['h9000] = 00000000, pass
DM['h9004] = 00000001, pass
DM['h9008] = 00000001, pass
DM['h900c] = 00000002, pass
DM['h9010] = 00000003, pass
DM['h9014] = 00000005, pass
DM['h9018] = 00000008, pass
DM['h901c] = 0000000d, pass
DM['h9020] = 00000015, pass
DM['h9024] = 00000022, pass
DM['h9028] = 00000037, pass
DM['h902c] = 00000059, pass
DM['h9030] = 00000090, pass
DM['h9034] = 000000e9, pass
DM['h9038] = 00000179, pass
DM['h903c] = 00000262, pass
DM['h9040] = 000003db, pass
DM['h9044] = 0000063d, pass
DM['h9048] = 00000a18, pass
DM['h904c] = 00001055, pass
DM['h9050] = 00001a6d, pass
    **************************
    **                      **
    **  Waku Waku !!        **
    **                      **
    **  Simulation PASS !!  **
    **                      **
    **************************
```

VI. SuperLint and ICC Verification Results

A. **SuperLint Results** (include a SuperLint screenshot)



B. **ICC Verification Results** (include an ICC screenshot)



1. block: 100%
2. expression:100%
3. toggle:89%

VII. Synthesis Results

A. **Speed** (include a screenshot; both setup-time slack and hold-time slack are > 0). The clock period is set to 14 ns per cycle, giving a frequency of 7.2 × 10^7 Hz.

1. Setup time

```
 # A fanout number of 1000 was used for high fanout net computations.

Operating Conditions: fast   Library: fast
Wire Load Model Mode: top

  Startpoint: M_ld_data[15]
              (input port clocked by clk)
  Endpoint: Reg_W/ld_data_out_reg[15]
            (rising edge-triggered flip-flop clocked by clk)
  Path Group: clk
  Path Type: min

  Des/Clust/Port     Wire Load Model       Library
  ------------------------------------------------
  Top                tsmc18_wl10           slow

  Point                                                   Incr       Path
  ---------------------------------------------------------------------------
  clock clk (rise edge)                                   7.00       7.00
  clock network delay (ideal)                             0.00       7.00
  input external delay                                    0.20       7.20 f
  M_ld_data[15] (in)                                      0.04       7.24 f
  Reg_W/ld_data[15] (Reg_W)                               0.00       7.24 f
  Reg_W/ld_data_out_reg[15]/D (DFFRHQX2)                  0.00       7.24 f
  data arrival time                                                  7.24

  clock clk (rise edge)                                   7.00       7.00
  clock network delay (ideal)                             0.00       7.00
  Reg_W/ld_data_out_reg[15]/CK (DFFRHQX2)                 0.00       7.00 r
  library hold time                                      -0.03       6.97
  data required time                                                 6.97
  ---------------------------------------------------------------------------
  data required time                                                 6.97
  data arrival time                                                 -7.24
  ---------------------------------------------------------------------------
  slack (MET)                                                        0.27


***** End Of Report *****
```

2. Hold time

```
Operating Conditions: slow   Library: slow
Wire Load Model Mode: top

  Startpoint: Controller/M_op_reg[4]
              (rising edge-triggered flip-flop clocked by clk)
  Endpoint: PC/current_pc_reg[13]
            (rising edge-triggered flip-flop clocked by clk)
  Path Group: clk
  Path Type: max

  Des/Clust/Port     Wire Load Model       Library
  ------------------------------------------------
  Top                tsmc18_wl10           slow

  Point                                                    Incr       Path
  ----------------------------------------------------------------------------
  clock clk (rise edge)                                    7.00       7.00
  clock network delay (ideal)                              0.00       7.00
  Controller/M_op_reg[4]/CK (DFFRX4)                       0.00 #     7.00 r
  Controller/M_op_reg[4]/QN (DFFRX4)                       0.49       7.49 f
  Controller/U132/Y (AOI2BB1X4)                            0.25       7.74 f
  Controller/U140/Y (AOI31X2)                              0.21       7.95 r
  Controller/U104/Y (AND3X4)                               0.26       8.21 r
  Controller/E_rs2_data_sel[0] (Controller)                0.00       8.21 r
  E_rs2_data_mux/sel[0] (MuxThree_1)                       0.00       8.21 r
  E_rs2_data_mux/U101/Y (INVX8)                            0.08       8.30 f
  E_rs2_data_mux/U31/Y (INVX20)                            0.29       8.58 r
  E_rs2_data_mux/U109/Y (NOR2X4)                           0.16       8.74 f
  E_rs2_data_mux/U91/Y (CLKINVX8)                          0.08       8.82 r
  E_rs2_data_mux/U85/Y (INVX8)                             0.05       8.87 f
  E_rs2_data_mux/U32/Y (AOI222X4)                          0.49       9.36 r
  E_rs2_data_mux/U107/Y (INVX4)                            0.06       9.42 f
  E_rs2_data_mux/out_tri[15]/Y (TBUFX16)                   0.19       9.61 f
  E_rs2_data_mux/out[15] (MuxThree_1)                      0.00       9.61 f
  alu_mux2/in1[15] (MuxTwo_5)                              0.00       9.61 f
  alu_mux2/U68/Y (AOI2BB2X4)                               0.19       9.80 r
  alu_mux2/U82/Y (INVX8)                                   0.12       9.91 f
  alu_mux2/out[15] (MuxTwo_5)                              0.00       9.91 f
  FALU/operand2[15] (FALU)                                 0.00       9.91 f
```

```
  FALU/sub_177/U37/Y (OA21X4)                             0.10      18.47 f
  FALU/sub_177/U133/Y (XNOR2X4)                           0.23      18.70 f
  FALU/sub_177/DIFF[8] (FALU_DW01_sub_9)                  0.00      18.70 f
  FALU/U1811/Y (NOR2BX4)                                  0.17      18.86 f
  FALU/U2825/Y (OR4X4)                                    0.38      19.24 f
  FALU/out[0] (FALU)                                      0.00      19.24 f
  falu_mux/in1[0] (MuxTwo_3)                              0.00      19.24 f
  falu_mux/U23/Y (AOI2BB2X4)                              0.20      19.44 r
  falu_mux/U37/Y (INVX8)                                  0.09      19.53 f
  falu_mux/out[0] (MuxTwo_3)                              0.00      19.53 f
  Controller/alu_out (Controller)                         0.00      19.53 f
  Controller/U105/Y (NAND2X2)                             0.14      19.67 r
  Controller/U230/Y (NAND3X4)                             0.16      19.83 f
  Controller/next_pc_sel (Controller)                     0.00      19.83 f
  U6/Y (BUFX20)                                           0.18      20.01 f
  pc_mux/sel (MuxTwo_7)                                   0.00      20.01 f
  pc_mux/U28/Y (CLKINVX8)                                 0.13      20.14 r
  pc_mux/U42/Y (CLKINVX8)                                 0.15      20.29 f
  pc_mux/U49/Y (MX2X2)                                    0.31      20.60 r
  pc_mux/out[13] (MuxTwo_7)                               0.00      20.60 r
  PC/next_pc[13] (Reg_PC)                                 0.00      20.60 r
  PC/U34/Y (NAND2X2)                                      0.08      20.68 f
  PC/U32/Y (NAND2X1)                                      0.19      20.88 r
  PC/current_pc_reg[13]/D (DFFRXL)                        0.00      20.88 r
  data arrival time                                                 20.88

  clock clk (rise edge)                                  21.00      21.00
  clock network delay (ideal)                             0.00      21.00
  PC/current_pc_reg[13]/CK (DFFRXL)                       0.00      21.00 r
  library setup time                                     -0.12      20.88
  data required time                                                20.88
  --------------------------------------------------------------------------
  data required time                                                20.88
  data arrival time                                                -20.88
  --------------------------------------------------------------------------
  slack (MET)                                                       0.00


***** End Of Report *****
```

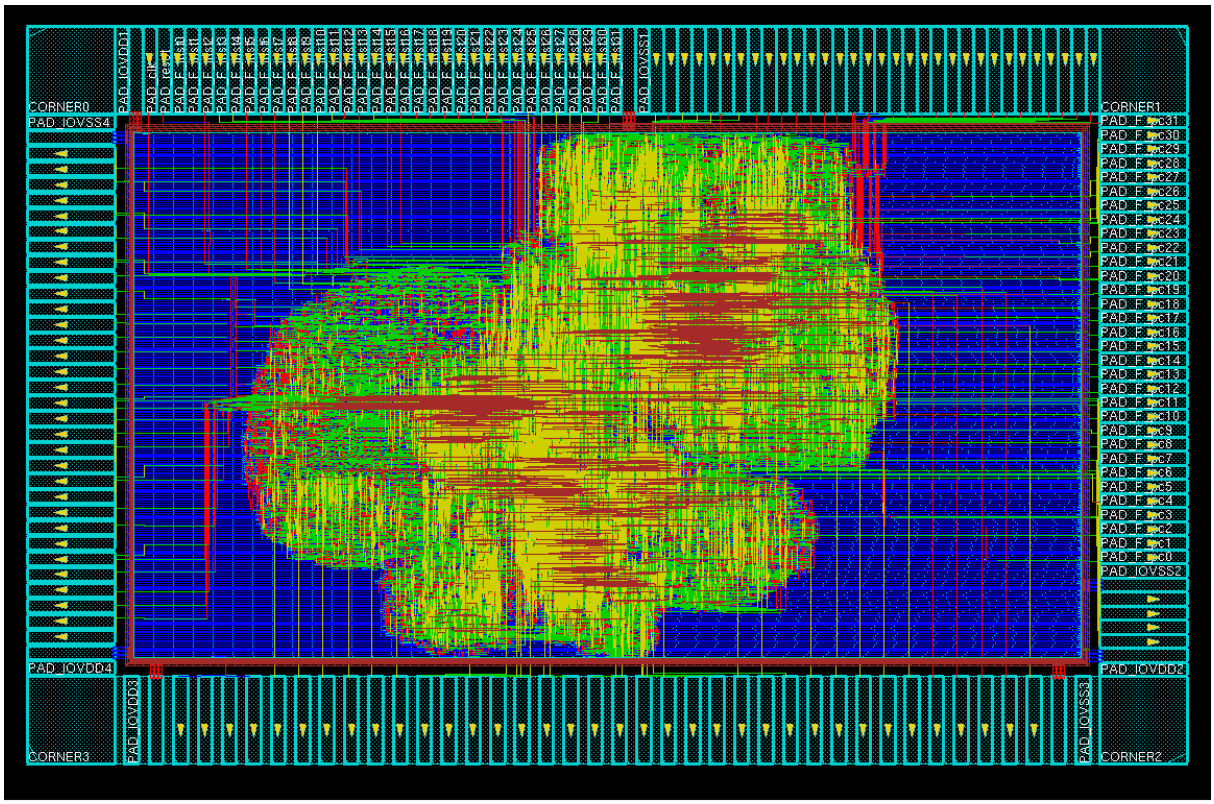
B. Area (attach a screenshot)

```
****************************************
Report : area
Design : Top
Version: Q-2019.12
Date   : Sat Jan  6 13:44:04 2024
****************************************

Library(s) Used:

    slow (File: /home/ncku_class/vsd2023/vsd202300/Desktop/vsd2023/synopsys/slow.db)

Number of ports:                         8565
Number of nets:                         30694
Number of cells:                        21801
Number of combinational cells:          19049
Number of sequential cells:              2591
Number of macros/black boxes:               0
Number of buf/inv:                       4721
Number of references:                      47

Combinational area:             383058.247397
Buf/Inv area:                    46659.413446
Noncombinational area:          193230.580820
Macro/Black Box area:                0.000000
Net Interconnect area:         2671633.442047

Total cell area:                576288.828217
Total area:                    3247922.270264

***** End Of Report *****
```

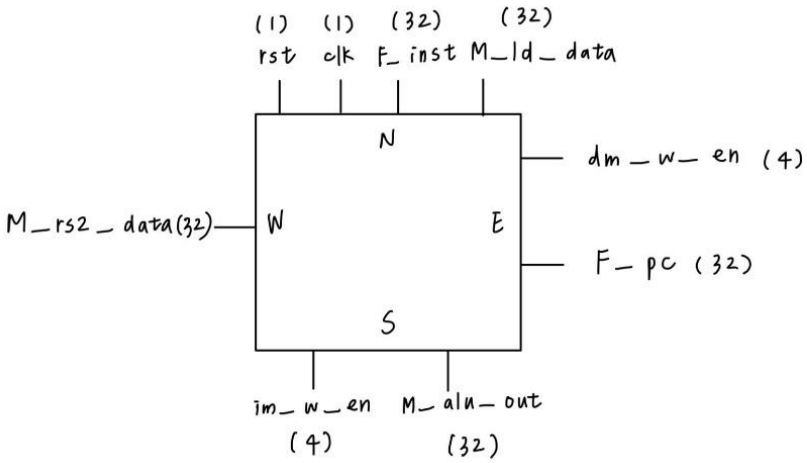
C. Power (attach a screenshot)

```
    slow (File: /home/ncku_class/vsd2023/vsd202300/Desktop/vsd2023/synopsys/slow.db)


Operating Conditions: slow   Library: slow
Wire Load Model Mode: top

Design        Wire Load Model            Library
------------------------------------------------
Top                   tsmc18_wl10       slow


Global Operating Voltage = 1.62
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000pf
    Time Units = 1ns
    Dynamic Power Units = 1mW    (derived from V,C,T units)
    Leakage Power Units = 1pW


  Cell Internal Power  =  12.7239 mW   (82%)
  Net Switching Power  =   2.7432 mW   (18%)
                         ---------
Total Dynamic Power    =  15.4672 mW  (100%)

Cell Leakage Power     =  20.5333 uW


                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )  Attrs
--------------------------------------------------------------------------------------------------
io_pad             0.0000           0.0000              0.0000             0.0000  (   0.00%)
memory             0.0000           0.0000              0.0000             0.0000  (   0.00%)
black_box          0.0000           0.0000              0.0000             0.0000  (   0.00%)
clock_network      0.0000           0.0000              0.0000             0.0000  (   0.00%)
register          11.2185           0.1021          4.7251e+06            11.3254  (  73.13%)
sequential     1.0077e-02        1.8859e-03          2.6181e+04         1.1990e-02  (   0.08%)
combinational      1.4953           2.6392          1.5782e+07             4.1503  (  26.80%)
--------------------------------------------------------------------------------------------------
Total             12.7239 mW        2.7432 mW       2.0533e+07 pW         15.4877 mW

***** End Of Report *****
```

VIII. Layout Results



A. Pin Diagram (attach pin-assignment screenshot)



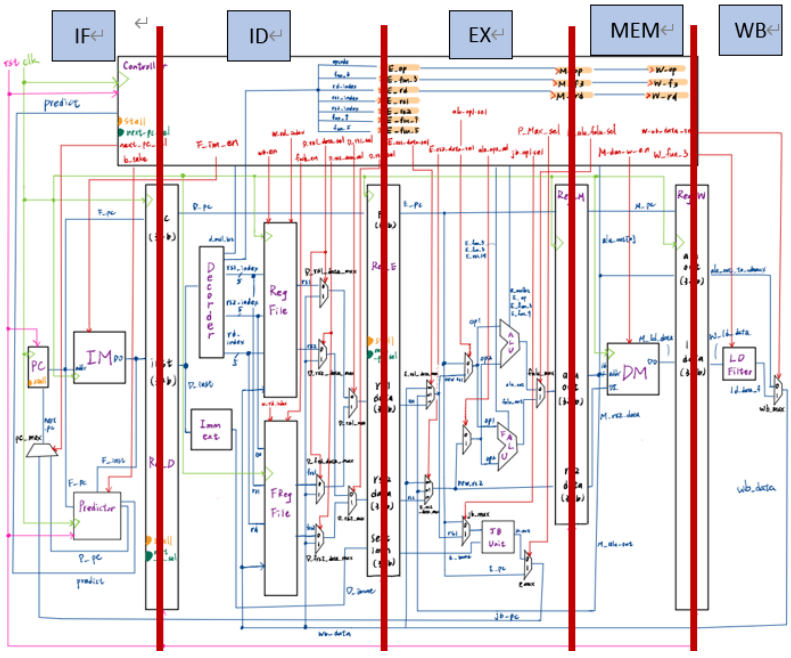
B. Area (report core/total area here)

```
   ----------------------------
   Standard Cells in Netlist
   ----------------------------
             Cell Type    Instance Count  Area (um^2)
                 FILL8            10953     291472.4736
                FILL64             5119    1089781.8624
"summaryReport.rpt" 6574L, 258686C                          1,1           Top
```

IX. Pipelining

A. Design Overview

1. The CPU is implemented as a classic five-stage pipeline—IF, ID, EX, MEM and WB—to overlap instruction execution and improve throughput.



B. Stage Partitioning

Stage	Name	Function
IF	*Instruction Fetch*	Uses the current PC to fetch the instruction from instruction memory and passes it to the next stage.
ID	*Instruction Decode*	Decodes the fetched instruction, reads operands from the register file, sign/zero-extends the immediate to 32 bits, and forwards all operands to EX.
EX	*Execute*	Performs the required ALU, FPU, or address-generation operation according to the decoded control signals, then forwards the result and control flags to MEM.
MEM	*Memory Access*	Writes data to, or reads data from, data memory based on opcode and dm_w_en, then forwards the result to WB.
WB	*Write Back*	Writes the result from MEM back to the destination register in the register file.

C. Pipeline Hazards and Mitigations

Hazard Type	Cause	Mitigation
Structure Hazard	Concurrent instruction fetch and data-memory access could contend for a single memory port.	**Split memory** into separate **IM** (instruction memory) and **DM** (data memory) so fetch and load/store never share the same resource.
Control Hazard	Until a branch or jump target is resolved, the pipeline does not know which instruction to fetch next.	Employ a **branch predictor**; if a mis-prediction is detected, the pipeline **flushes** the incorrect instructions.
Data Hazard	An instruction needs a result that has not yet been written back.	**Forward (bypass)** results from the MEM and WB stages directly to the EX stage, eliminating most read-after-write stalls.

X. **Special Design Features(Branch Predictor, Booth Multiplier**, **Floating-Point Unit)**

A. Branch predictor

1. **Design Rationale:**

a. A **one-level bimodal branch predictor** is employed. It uses a 2-bit saturating counter for each entry, offering four states:

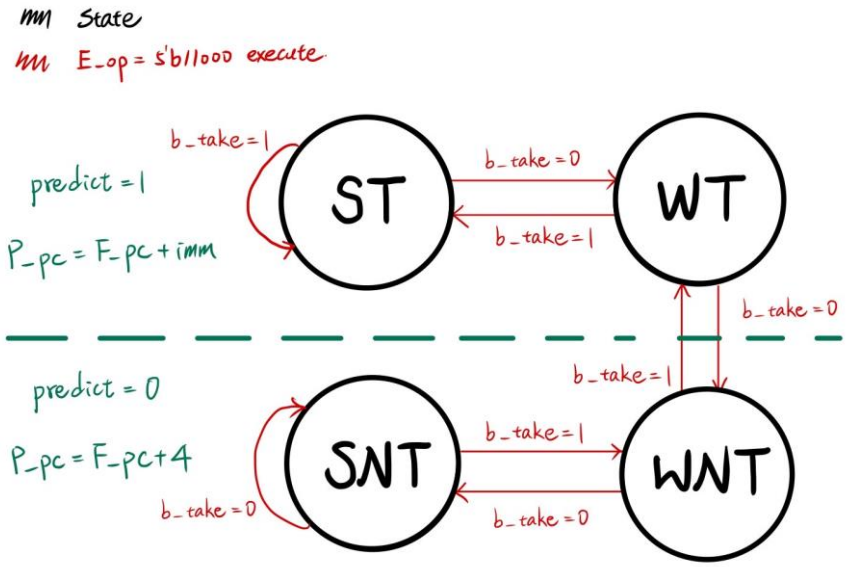
State	Abbreviation	Meaning

Strongly Not Taken	**SNT**	Predictor is confident the branch will *not* be taken.
Weakly Not Taken	**WNT**	Predictor leans toward not-taken but is uncertain.
Weakly Taken	**WT**	Predictor leans toward taken but is uncertain.
Strongly Taken	**ST**	Predictor is confident the branch *will* be taken.

b. **Taken states (WT, ST):** The predictor assumes the branch *will* be taken and fetches the instruction at the computed branch target.

c. **Not-taken states (WNT, SNT):** The predictor assumes the branch *will not* be taken and fetches the sequential instruction at **PC + 4**.

d. The 2-bit counter is updated after the actual branch outcome is known, gradually moving toward the correct state and providing hysteresis to avoid quick oscillation on noisy branches.



2. **State-Update Procedure**

a. The predictor's **current state** is initialised to **2 = WNT (Weakly Not Taken)**.

b. **If the branch is *not* taken**, the state is incremented by 1; thus WNT (2) becomes **SNT (3)**.

c. **If the branch *is* taken**, the state is decremented by 1; thus WNT (2) becomes **WT (1)**.

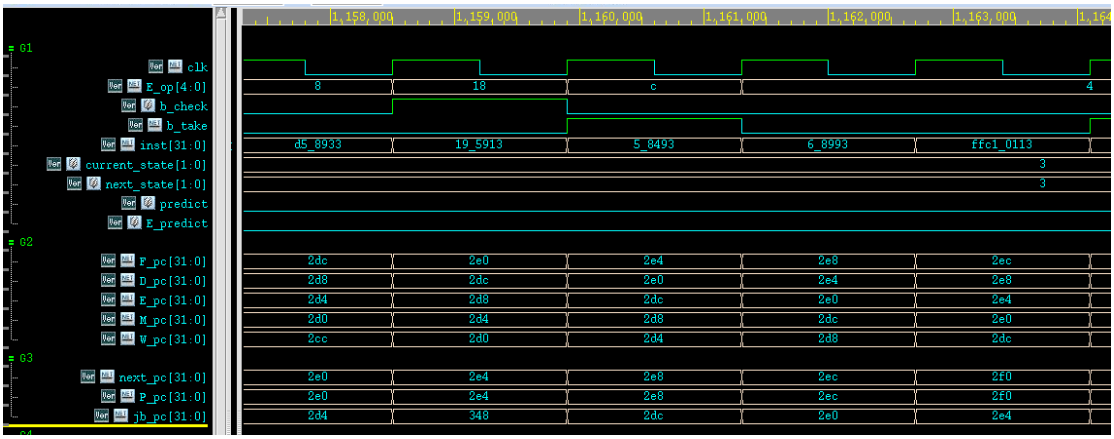
d. The 2-bit counter is **saturating**:

i. At **0 = ST (Strongly Taken)** the value cannot be decremented further.

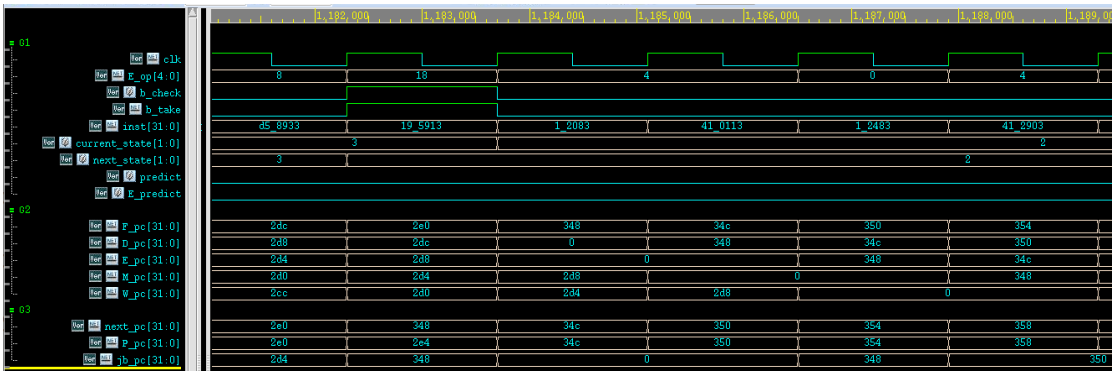
ii. At **3 = SNT (Strongly Not Taken)** the value cannot be incremented further.

3. Simulation Result Analysis

a. **SNT -> SNT:** At PC = 0x2D8, the fetched instruction is a branch, so b_check = 1. The branch condition evaluates to not taken (b_taken = 0), so no jump occurs. Because the predictor correctly predicted “not taken,” it remains in the SNT (Strongly Not Taken) state and the predict output is unchanged.

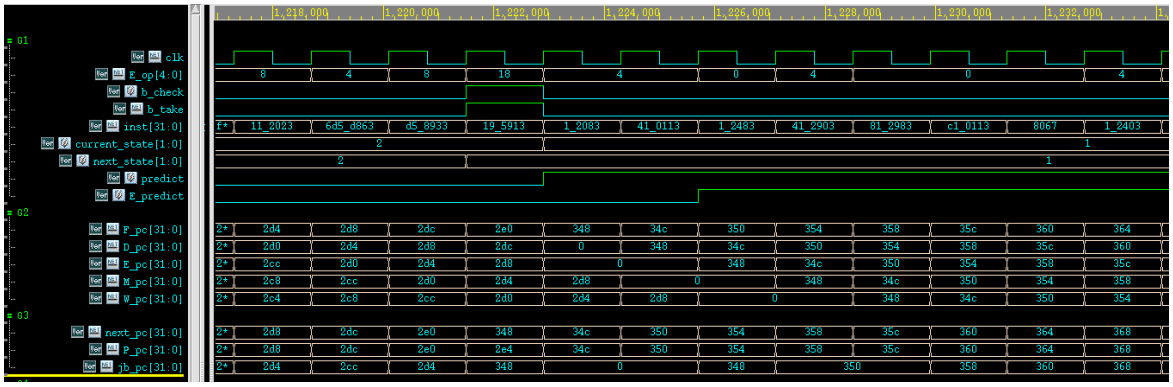


b. **SNT -> WNT:** At PC = 0x2D8 the fetched instruction is a branch, so b_check = 1. This time the branch condition evaluates to taken (b_taken = 1), so a jump must occur. The predictor was in SNT (Strongly Not Taken) and therefore predicted not taken—a misprediction. The saturating counter is updated from SNT (3) to WNT (2). Because of the misprediction, the pipeline flushes the instructions that were speculatively fetched, while the predict output itself remains unchanged for this cycle.

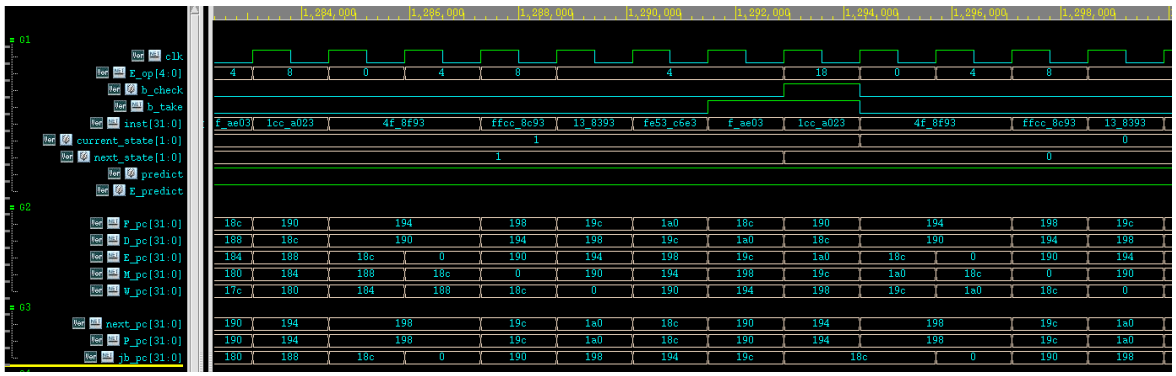


c. **WNT -> WT:** At PC = 0x2D8 the instruction is a branch, so b_check = 1. The branch condition evaluates to taken (b_taken = 1), requiring a jump. Because the predictor had been in WNT (Weakly Not Taken), it mispredicted; the state is therefore updated to WT (Weakly Taken). A pipeline flush is issued for the speculatively fetched instructions, and the

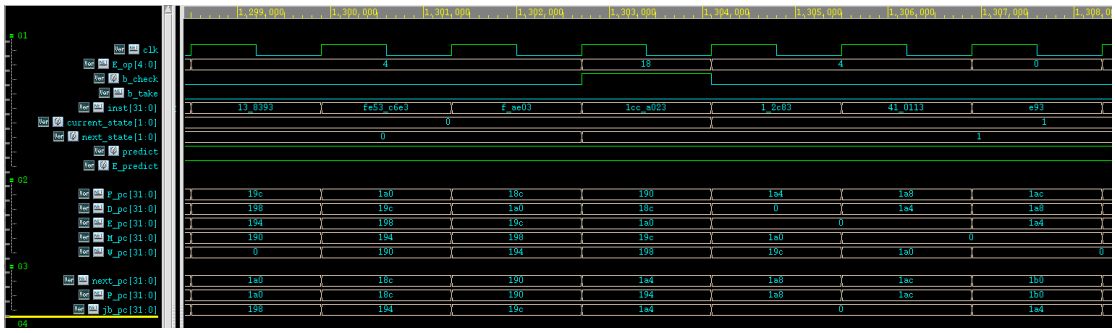
predict signal flips from 0 (not taken) to 1 (taken).



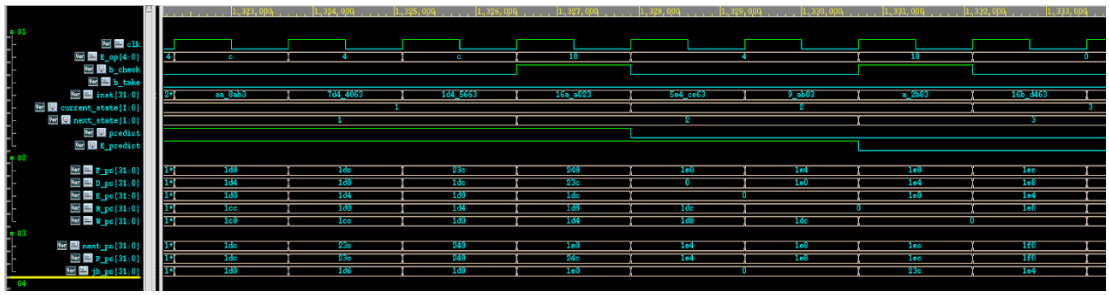
d. **WT -> ST:** At PC = 0x1A0 the fetched instruction is a branch, so b_check = 1. Because the branch condition is satisfied (b_taken = 1), a jump must occur. The predictor was in WT (Weakly Taken) and, having predicted taken correctly, it is updated to ST (Strongly Taken). Since the prediction was correct, no pipeline flush is required, and the predict signal remains unchanged.



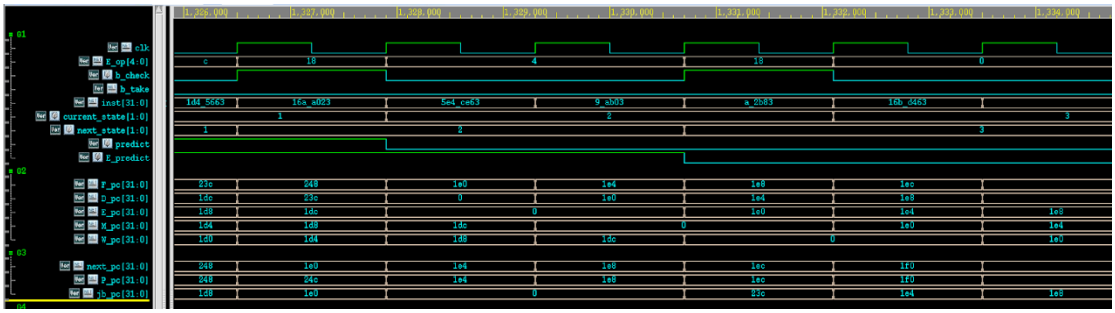
e. **ST -> WT:**At PC = 0x1A0 the fetched instruction is a branch, so b_check = 1. Because the branch condition evaluates to not taken (b_taken = 0), no jump is performed. The predictor was in ST (Strongly Taken) and therefore mispredicted; its state is updated to WT (Weakly Taken). A pipeline flush is triggered for the speculatively fetched instructions, while the predict output itself remains unchanged.



f. **WT -> WNT:** At PC = 0x1DC the fetched instruction is a branch, so b_check = 1. Because the branch condition evaluates to not taken (b_taken = 0), no jump is performed. The predictor was in WT (Weakly Taken) and therefore mispredicted; its state is updated to WNT (Weakly Not Taken). A pipeline flush is issued for the speculatively fetched instructions, and the predict output flips from 1 (taken) to 0 (not taken).



g. **WNT -> SNT:** At PC = 0x1E0 the fetched instruction is a branch, so b_check = 1. Because the branch condition evaluates to not taken (b_taken = 0), no jump is performed. The predictor was in WNT (Weakly Not Taken) and predicted correctly, so its state moves to SNT (Strongly Not Taken). Since the prediction was correct, no pipeline flush is required and the predict signal remains unchanged.



B. Floating point number Arithmetic Logic Unit

1. Supported Instructions

imm[11:0]		rs1	010	rd	0000111	FLW
imm[11:5]	rs2	rs1	010	imm[4:0]	0100111	FSW
0000000	rs2	rs1	rm	rd	1010011	FADD.S
0000100	rs2	rs1	rm	rd	1010011	FSUB.S
0001000	rs2	rs1	rm	rd	1010011	FMUL.S
0010100	rs2	rs1	000	rd	1010011	FMIN.S
0010100	rs2	rs1	001	rd	1010011	FMAX.S
1100000	00000	rs1	rm	rd	1010011	FCVT.W.S
1100000	00001	rs1	rm	rd	1010011	FCVT.WU.S
1110000	00000	rs1	000	rd	1010011	FMV.X.W
1010000	rs2	rs1	010	rd	1010011	FEQ.S
1010000	rs2	rs1	001	rd	1010011	FLT.S
1010000	rs2	rs1	000	rd	1010011	FLE.S
1101000	00000	rs1	rm	rd	1010011	FCVT.S.W
1101000	00001	rs1	rm	rd	1010011	FCVT.S.WU
1111000	00000	rs1	000	rd	1010011	FMV.W.X

Instruction	**Description** (Unless otherwise noted, rs1, rs2, and rd refer to registers in FREG.)
FLW	Loads the word located at **(REG rs1 + imm[11:0])** from memory and writes it to **FREG rd**.
FSW	Stores the value in **FREG rs1** to memory at the address **(REG rs1 + imm[11:0])**.
FADD.S	Adds **FREG rs1** and **FREG rs2**; writes the single-precision result to **rd**.
FSUB.S	Subtracts **FREG rs2** from **FREG rs1**; writes the result to **rd**.
FMUL.S	Multiplies **FREG rs1** by **FREG rs2**; writes the result to **rd**.
FMIN.S	Compares **FREG rs1** and **FREG rs2**; writes the *smaller* value to **rd**.
FMAX.S	Compares **FREG rs1** and **FREG rs2**; writes the larger value to rd.
FCVT.W.S	Rounds the value in **FREG rs1** to the nearest 32-bit *signed* integer and stores it in **REG rd**.

FCVT.WU.S	Rounds the *absolute value* of **FREG rs1** to the nearest 32-bit *unsigned* integer and stores it in **REG rd**.
FMV.X.W	Moves the raw bit pattern from **FREG rs1** directly to **REG rd** (no conversion).
FEQ.S	Sets **rd = 1** if **FREG rs1 == FREG rs2**, else **rd = 0**.
FLT.S	Sets **rd = 1** if **FREG rs1 < FREG rs2**; otherwise **rd = 0**.
FLE.S	Sets **rd = 1** if **FREG rs1 ≤ FREG rs2**; otherwise **rd = 0**.
FCVT.S.W	Converts **REG rs1** (32-bit *signed* integer) to single-precision floating-point and stores it in **FREG rd**.
FCVT.S.WU	Converts the *absolute value* of **REG rs1** (32-bit *unsigned* integer) to single-precision floating-point and stores it in **FREG rd**.
FMV.W.X	Moves the raw bit pattern from **REG rs1** directly to **FREG rd** (no conversion).

2. Floating-Point Set:

   The implementation follows the **IEEE 754** standard for single-precision floating-point representation.

   a. The 32-bit operand is divided as follows:

      i. operand[31]: sign bit

      ii. operand[30:23]: 8-bit exponent

      iii. operand[22:0]: mantissa (fraction)

   b. The **exponent** is stored using a bias of **127**, which allows the exponent to represent both positive and negative powers of two.

      i. An actual exponent of 0 is encoded as 127,

      ii. Values greater than 127 represent positive exponents,

      iii. Values less than 127 represent negative exponents.

      iv. Thus, the range of representable exponents is from −126 to +127.

   c. The **mantissa** is normalized such that the first leading 1 is implicitly placed before the binary point (i.e., **1.xxx...**). Only the fractional part is stored.

   d. **Rounding Rule:**

The design uses **"round to nearest, ties away from zero"**, which rounds intermediate results to the nearest representable value, and if exactly halfway, it rounds away from zero.

e. In accordance with the RISC-V instruction set architecture, the processor includes **32 floating-point registers (f0 to f31)** specifically for storing and operating on floating-point values.

3. Implementation Principles:

a. FADD.S、FSUB.S:

i. **Absolute Value Extraction:**

Extract bits [30:0] from operand1 and operand2 as their absolute values: ABS_1 and ABS_2.

ii. **Exponent Alignment:**

Compare ABS_1 and ABS_2 to determine which is larger.
Compute the exponent difference ($\Delta exp = |exp1 - exp2|$), and record which operand is larger using a flag lar (lar = 0 if ABS_1 is larger, lar = 1 if ABS_2 is larger).

iii. **True Sign Determination:**

Define the true sign logic:

If signs differ and the operation is addition, or if signs match and the operation is subtraction, then true_sign = 1 (indicating subtraction). Restore the implicit leading 1 in the mantissa (hidden bit), and append zeros to form the full extended mantissa (add_man).

iv. **Mantissa Shifting:**

Shift the smaller operand's mantissa (add_man) **right** by the exponent difference so that both mantissas are aligned to the same base.

v. **Mantissa Arithmetic:**

Based on true_sign, perform **addition** or **subtraction** on the aligned mantissas.
Determine the result's **sign bit** depending on operand signs and magnitudes.

vi. **Normalization:**

Use a leading-one detector to identify the position of the first 1 in the result, which determines how many bits to shift to normalize the mantissa. This also helps adjust the exponent accordingly.

vii. **Final Shift & Exponent Adjustment:**

Shift the result mantissa as needed, and subtract the shift amount from the exponent to form the final **normalized mantissa** and **exponent**.
These are then re-packed into IEEE 754 format for write-back.

b. FMUL.S

i. Exponent Calculation: Compute the resulting exponent by adding the exponents of the two operands and then subtracting 127 (the IEEE 754 bias). exp_result = exp1 + exp2 – 127.This accounts for the bias used to represent both positive and negative powers of two.

ii. **Mantissa Restoration and Multiplication:** Restore the hidden leading 1 in both operands' mantissas (i.e., convert from **1.fraction** form), then perform a **full 24-bit × 24-bit** multiplication. The resulting product will be 48 bits wide.

iii. Normalization and Final Adjustment: Because both mantissas were normalized, the most significant bit of the product (bit 47 or 46) will always be 1. Use a conditional check to determine whether to shift the product and adjust the exponent accordingly: If bit 47 is 1, the mantissa is already normalized. If bit 46 is 1, shift left by 1 and decrement the exponent by 1. The final normalized mantissa and adjusted exponent are then packed into IEEE 754 format as the result of the multiplication.

c. FMIN.S、FMAX.S

i. Based on the sign bits of the two operands, if both numbers have the same sign, compare their absolute values (ABS). Then, depending on the value of func3 (func3 = 1 for FMAX.S, func3 = 0 for FMIN.S), output the correct minimum or maximum value accordingly.

d. FEQ.S、FLT.S、FLE.S

i. Compare the sign bits and the absolute values (ABS) of the two operands to determine whether the condition of equality (==), less than (<), or less than or equal to (<=) is satisfied.

e. FCVT.W.S、FCVT.WU.S

i. Determine whether the exponent is positive or negative and store the result in r1 (r1 = 0 for positive, r1 = 1 for negative). Then subtract 127 from the exponent to obtain its actual (unbiased) absolute value.

ii. iRestore the hidden leading 1 in the mantissa, then right-shift it by the absolute exponent value to compute the correct numerical value.

iii. Check whether the result is an integer.

- If it is, output the original value.
- If it is a fraction, truncate the fractional part and add 1 to perform rounding away from zero (i.e., forced upward rounding).

iv. Check func1 to determine the output type:

- If func1 = 1, the result is treated as unsigned and directly output.
- If func1 = 0, the result is signed; if the number is negative, apply two's complement to produce the correct signed integer output.

f. FCVT.S.W、FCVT.S.WU

i. Use XOR and the sign bit to determine the operand's sign:

- If operand1 is positive, use the original value.
- If operand1 is negative, apply two's complement to obtain its absolute value.

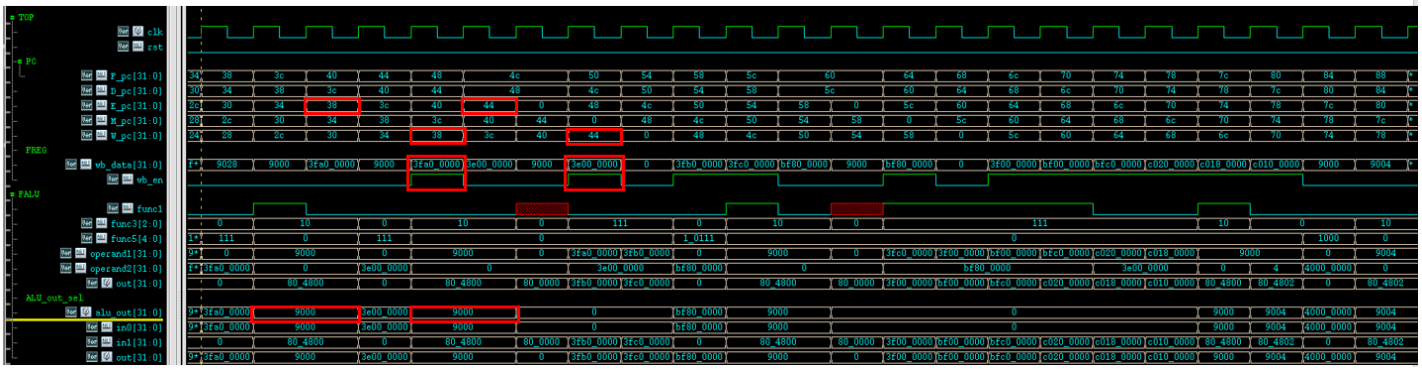
ii. Normalization: Identify the position of the most significant 1 in the absolute value to determine how far it must be shifted. This position reflects the difference between the actual exponent and the bias (127).

iii. Shift the original value according to the normalization result to form the correct mantissa. Then, calculate the adjusted exponent by adding the shift-derived difference to 127, forming the correct biased exponent.

iv. Based on func1 (func1 = 0 for signed, func1 = 1 for unsigned), determine the proper sign bit. Use this along with the computed exponent and mantissa to assemble the final IEEE 754 floating-point representation.

4. Simulation Result

a. **flw f1, 0(s0):**



At PC = 0x38 and PC = 0x44, the fetched instructions are both FLW. Thus, the ALU calculates the memory address from which the floating-point data

will be loaded. When W_pc = 0x38, the wb_data (write-back data) is 0x3FA0_0000, which corresponds to 1.25 in IEEE 754 format. When W_pc = 0x44, the wb_data is 0x3E00_0000, which represents 0.125.

b. **fadd.s f1, f1, f2:**

At PC = 0x38 and PC = 0x44, the fetched instructions are both FLW. Thus, the ALU calculates the memory address from which the floating-point data will be loaded. When W_pc = 0x38, the wb_data (write-back data) is 0x3FA0_0000, which corresponds to 1.25 in IEEE 754 format. When W_pc = 0x44, the wb_data is 0x3E00_0000, which represents 0.125.

c. **fadd.s f1, f1, f3:**

At PC = 0x5C ~ 0x70, there are six consecutive FADD.S instructions. The FALU performs a chain of additions, where the result from each FADD.S is forwarded directly to the operand1 of the next instruction.

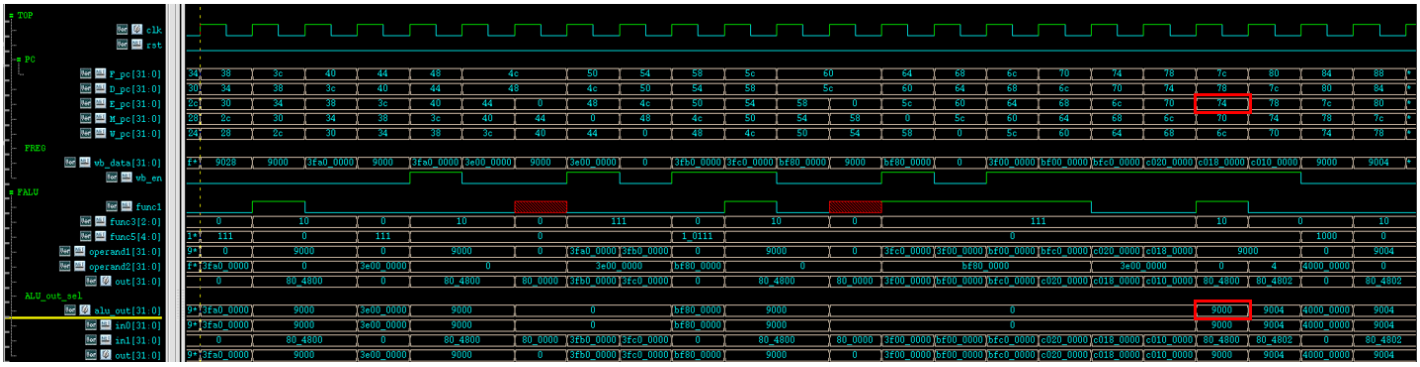
i. The first addition is: 0x3FE0_0000 (1.5) + 0xBF80_0000 (−1) → 0x3F00_0000 (0.5)

ii. The second: 0x3F00_0000 (0.5) + 0x3E00_0000 (0.125) → 0x3F20_0000 (0.625)

This process continues with further additions, reusing the

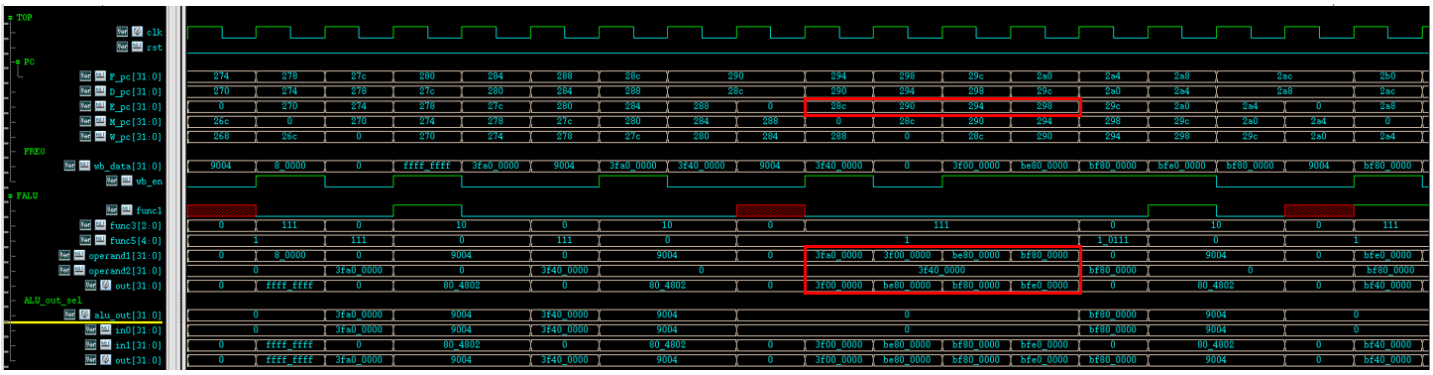
previous result each time. After all six additions, the final accumulated result becomes 0xC010_0000, which corresponds to −2.25 in IEEE 754 format. This shows that the forwarding mechanism is working correctly and that each result is immediately used as input for the next instruction without stalling.

d. **fsw f1, 0(s0)**:



At PC = 0x74, the fetched instruction is FSW (Store Word, Floating-Point). In this instruction, the ALU calculates the effective memory address by adding the base address in rs1 to the immediate offset.

e. **fsub.s f1, f1, f2:**



At PC = 0x28C ~ 0x298, there are four consecutive FSUB.S (floating-point subtraction) instructions. The FALU performs a chain of subtractions, where the result from each operation is forwarded directly to the operand1 of the next instruction.

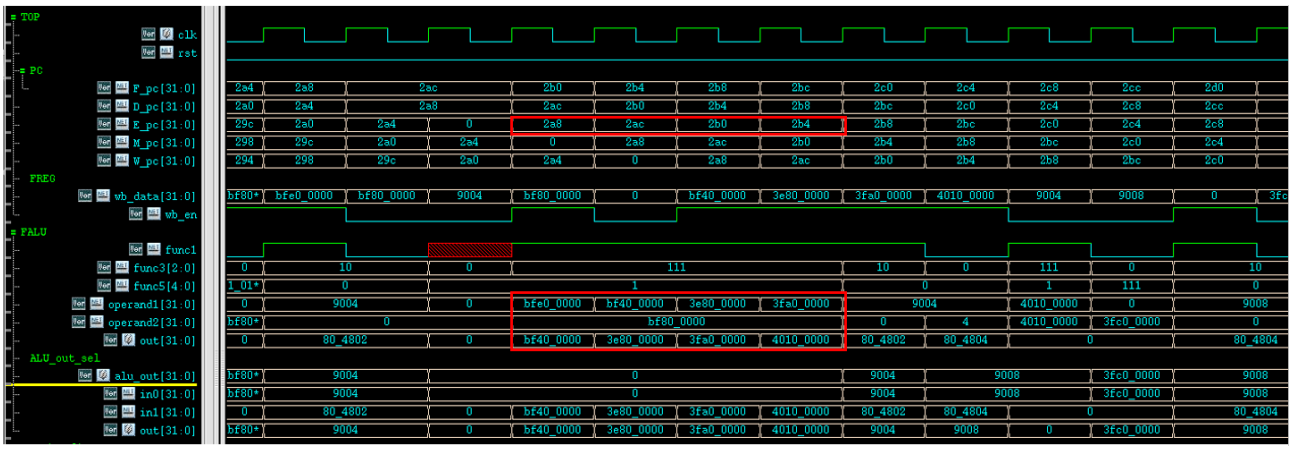
i. 1st FSUB (PC = 0x28C): 0x3FA0_0000 (1.25) − 0x3F40_0000 (0.75) → 0x3F00_0000 (0.5)

ii. 2nd FSUB (PC = 0x290): 0x3F00_0000 (0.5) − 0x3F40_0000 (0.75) → 0xBE80_0000 (−0.25)

iii. 3rd FSUB (PC = 0x294): 0xBE80_0000 (−0.25) − 0x3F40_0000 (0.75) → 0xBF80_0000 (−1.0)

iv. 4th FSUB (PC = 0x298): 0xBF80_0000 (−1.0) − 0x3F40_0000 (0.75) → 0xBFE0_0000 (−1.75)

f. **fsub.s f1, f1, f3:**



At PC = 0x2A8 ~ 0x2B4, four consecutive FSUB.S (floating-point subtraction) instructions are executed.The FALU performs repeated subtractions, where the result from each operation is forwarded to the next instruction's operand1.

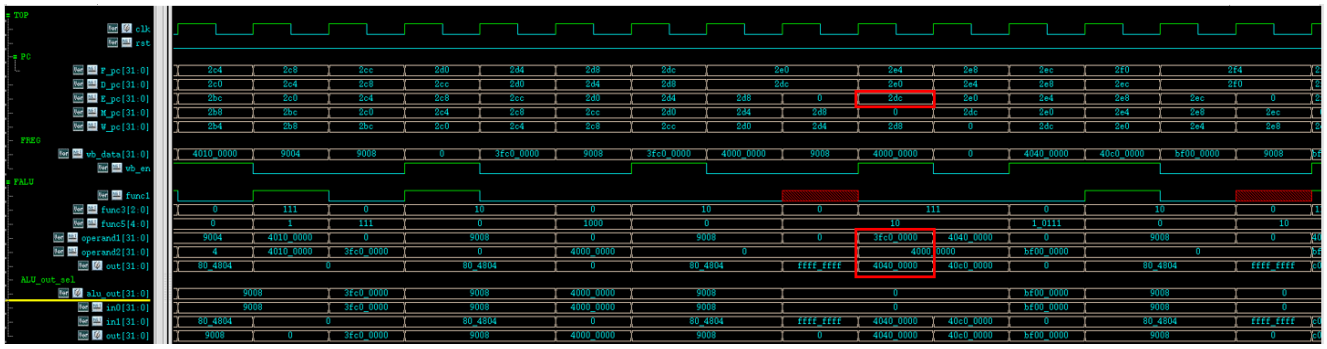
i. 1st FSUB (PC = 0x2A8): 0xBFE0_0000 (−1.25) − 0xBF80_0000 (−1.0) → 0x3F00_0000 (0.25)

ii. 2nd FSUB (PC = 0x2AC): 0x3F00_0000 (0.25) − 0xBF80_0000 (−1.0) → 0x3F80_0000 (1.25)

iii. 3rd FSUB (PC = 0x2B0): 0x3F80_0000 (1.25) − 0xBF80_0000 (−1.0) → 0x4000_0000 (2.25)

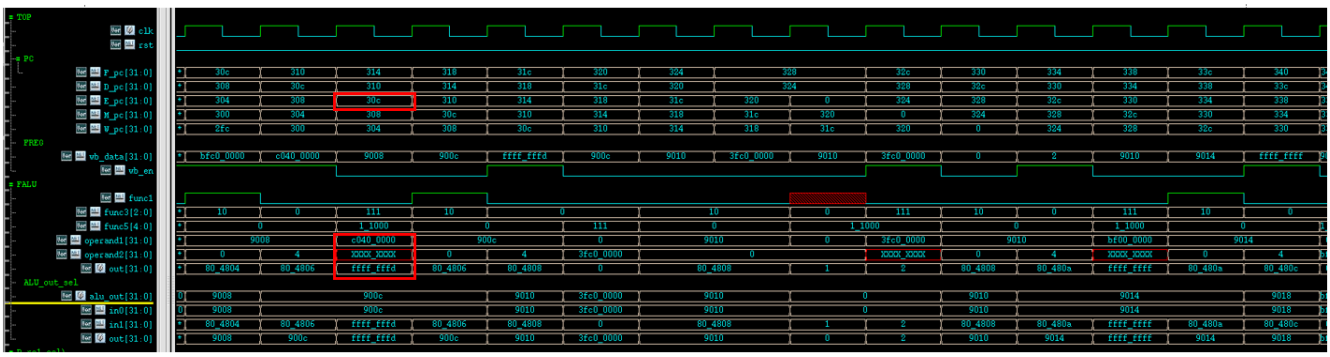
iv. 4th FSUB (PC = 0x2B4): 0x4000_0000 (2.25) − 0xBF80_0000 (−1.0) → 0x4040_0000 (3.25)

g. **fmul.s f1, f1, f2:**



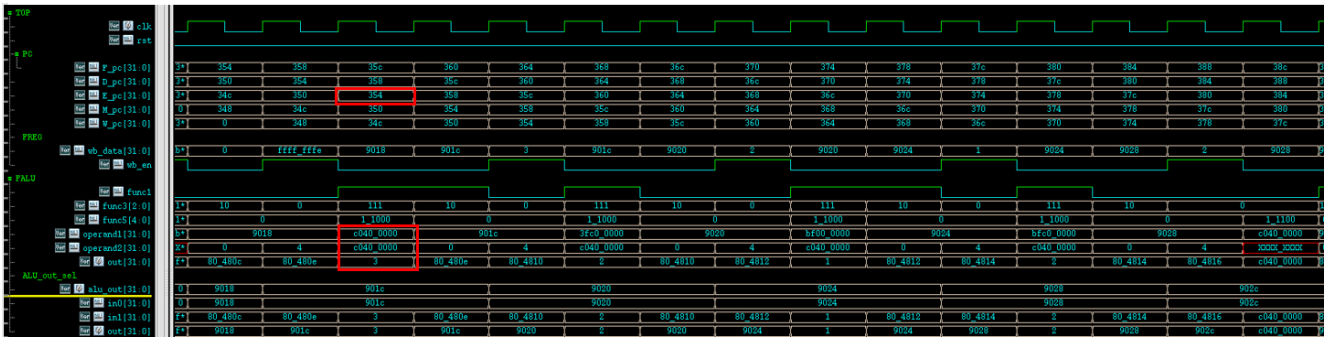
At PC = 0x2DC, the fetched instruction is FMUL.S (floating-point multiply). The FALU multiplies the following operands: Operand1 = 0x3FC0_0000, which equals 1.5. Operand2 = 0x4000_0000, which equals 2.0.The multiplication result is 1.5 × 2.0 = 3.0, which corresponds to 0x4040_0000 in IEEE 754 format. Thus, the FALU output is 0x4040_0000 (3.0).

h. **fcvt.w.s t0, f1:**



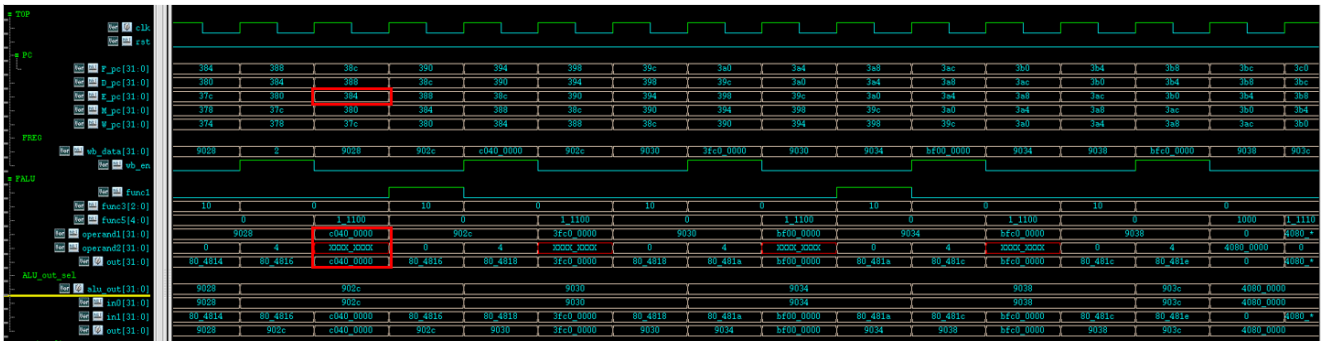
At PC = 0x30C, the fetched instruction is FCVT.W.S (floating-point to signed-integer conversion). The FALU converts the floating-point operand: operand1 = 0xC040_0000, which corresponds to −3.0 in IEEE 754 format. The conversion result is −3.0 rounded to an integer (using round to nearest, ties away from zero) = −3, Represented in 32-bit two's complement as 0xFFFF_FFFD. So, the FALU output is 0xFFFF_FFFD (−3).

i. **fcvt.wu.s t0, f1:**



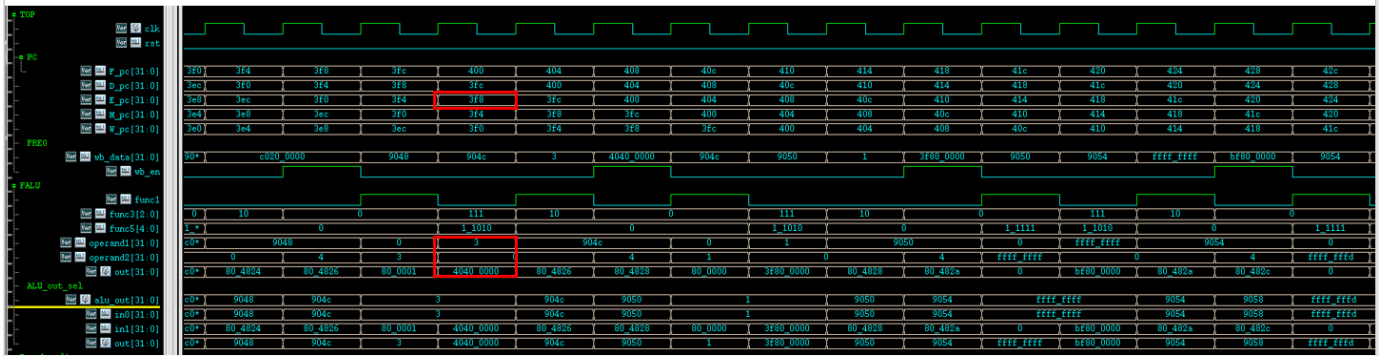
At PC = 0x354, the instruction is FCVT.WU.S, which converts a floating-point value to an unsigned integer. operand1 = 0xC040_0000, representing −3.0 in IEEE 754 format. Despite being negative, since this is an unsigned conversion, the implementation takes the absolute value and rounds it: |−3.0| → 3.0 → 0x0000_0003. So, the FALU output is 0x0000_0003 (3) — the unsigned integer result of converting −3.0.

j. **fmv.x.w t0, f1:**



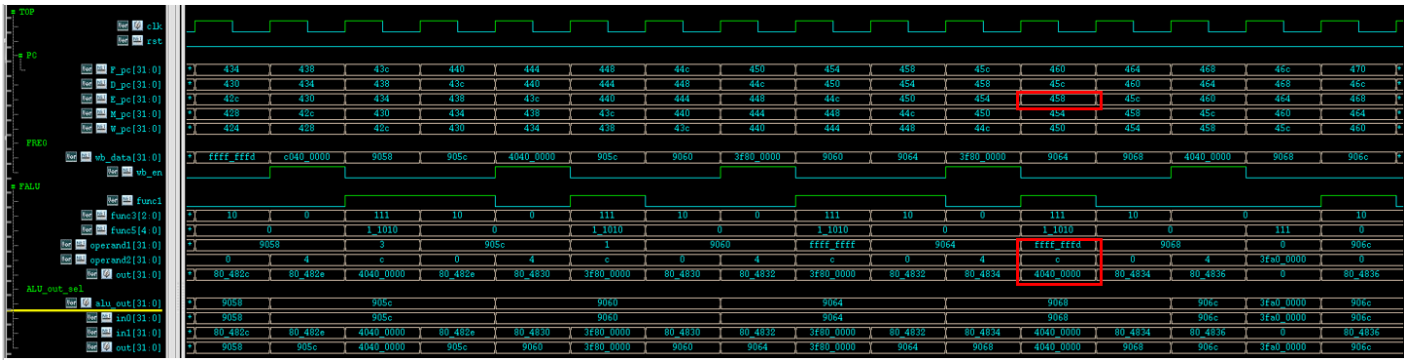
At PC = 0x384, the instruction is FMV.X.W. The FALU output is the bit pattern of operand1 = 0xC040_0000 (−3.0) passed directly and unmodified to t0.

k. **fcvt.s.w f1, t0:**



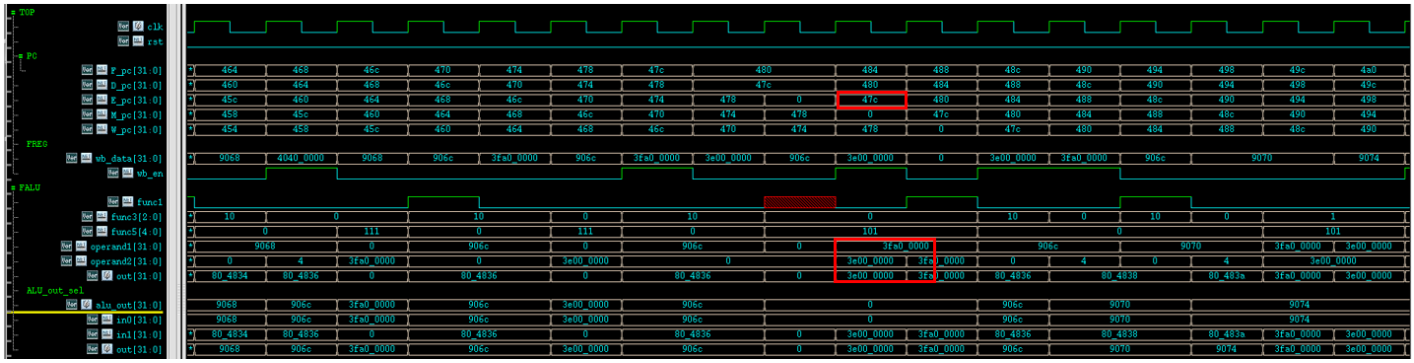
At PC = 0x3F8, the instruction is FCVT.S.W. The FALU converts operand1 = 3 into single-precision floating-point format, resulting in 0x4040_0000 (3.0).

l. **fcvt.s.wu f4, t3:**



At PC = 0x458, the instruction is FCVT.S.WU. The FALU takes the absolute value of operand1 = −3 and converts it into floating-point format, resulting in 0x4040_0000 (3.0).

m. **fmin.s f4, f1, f2:**

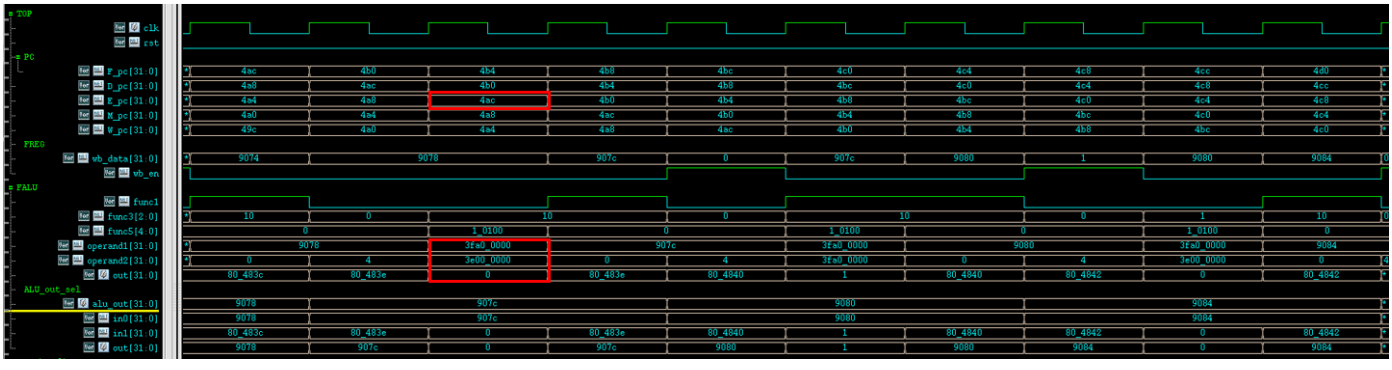


At PC = 0x47C, the instruction is FMIN.S. The FALU compares operand1 = 0x3FA0_0000 (1.25) and operand2 = 0x3E00_0000 (0.125), and outputs the smaller value, 0x3E00_0000 (0.125).

n. **fmax.s f4, f1, f2:**

At PC = 0x494, the instruction is FMAX.S. The FALU compares operand1 = 0x3FA0_0000 (1.25) and operand2 = 0x3E00_0000 (0.125), and outputs the larger value, 0x3FA0_0000 (1.25).

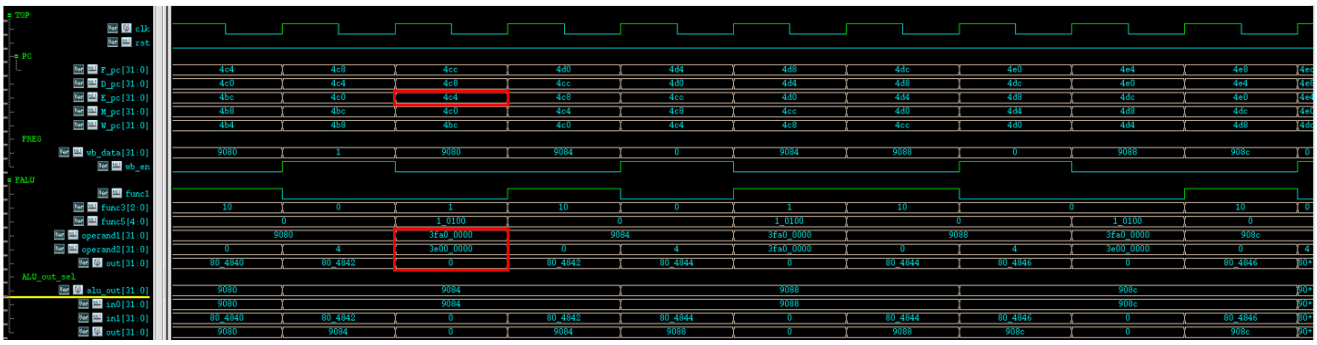
o. **feq.s t0, f1, f2:**



At PC = 0x4AC, the instruction is FEQ.S. The FALU compares operand1 = 0x3FA0_0000 (1.25) and operand2 = 0x3E00_0000 (0.125), and outputs 0

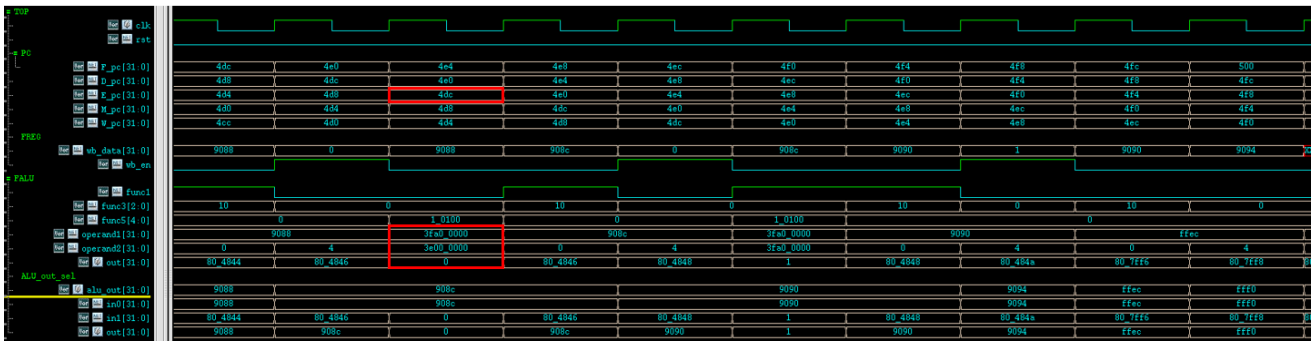
since the values are not equal.

p. **flt.s t0, f1, f2:**



At PC = 0x4AC, the instruction is FLT.S. The FALU compares whether operand1 = 0x3FA0_0000 (1.25) is less than operand2 = 0x3E00_0000 (0.125), and outputs 0 since 1.25 > 0.125.

q. **fle.s t0, f1, f2:**



At PC = 0x4AC, the instruction is FLE.S. The FALU compares whether operand1 = 0x3FA0_0000 (1.25) is less than or equal to operand2 = 0x3E00_0000 (0.125), and outputs 0 since 1.25 > 0.125.

C. Booth algorithm

The design uses the RISC-V instruction: mul rd, rs1, rs2. The final multiplication result is stored in the lower 32 bits of the full product.

a. Computational Principle

i. The core idea of Booth's multiplication is to identify transitions in the multiplier between sequences of 1s and 0s:

A transition from 0 → 1 (pattern 01) is treated as **+1**,

A transition from 1 → 0 (pattern 10) is treated as **−1**,

Sequences like 00 or 11 indicate a region of consecutive 0s or 1s and require **no addition/subtraction**, only

shifting.

This optimizes traditional multiplication, which would otherwise involve adding the multiplicand at every bit where the multiplier is 1. Booth encoding is particularly efficient when the multiplier contains multiple consecutive 1s.

ii. Before computation begins, a **0 is appended to the least significant bit (LSB)** of the multiplier to ensure that every bit has a "neighbor" to compare with.

iii. In each cycle, the algorithm checks a pair of adjacent bits (current + previous), performs the corresponding operation, and **left-shifts** the internal registers. This effectively excludes the processed bits from future operations and mimics dividing the multiplier by 2 each time.

b. Computation Steps

i. **Append a zero** to the right of the multiplier's 0th bit.

ii. Perform a **1-bit sign extension** on the multiplicand's MSB, and compute its **two's complement**.
Then store: the **original value** into mulshift, the **two's complement** into mulshift_C. Both stored in temporary registers **twice the width of the multiplicand**. Also initialize a **partial product register** to accumulate the result.

iii. Iterate over the multiplier bit pairs (including the appended 0), examining every two adjacent bits.
There are four possible patterns: 00, 01, 10, 11.

iv. Based on each pattern, perform the following operations:
(After each step, mulshift and mulshift_C are left-shifted by 1 bit)

v. Bit Pattern Operation

- 00 / 11 No addition; shift only
- 01 Add mulshift to the partial product
- 10 Add mulshift_C (i.e., subtract multiplicand) to the partial product.

vi. After all bit pairs have been examined, the final partial product represents the complete result of the multiplication.

c. Example: Multiply 00010011 (8-bit multiplicand) by 01001111 (8-bit multiplier)

i. Append a trailing 0 to the multiplier to form a 9-bit value: 010011110

ii. Two's complement of the multiplicand 00010011 is calculated as 11101101. Apply 1-bit sign extension to both the original and two's complement values:

Extended multiplicand: 000010011

Extended negative: 111101101

iii. Process the multiplier from bit 0 upward, examining each pair of adjacent bits and performing the corresponding operation:

- 10 → −1.

  Partial product: 111101101. Shift both multiplicand and negative versions left by 1:

  000010011 → 0000100110

  111101101 → 1111011010

- 11 → no operation

  Shift again:

  0000100110 → 00001001100

  1111011010 → 11110110100

- 11 → no operation

  Shift again:

  00001001100 → 000010011000

  11110110100 → 111101101000

- 11 → no operation

  Shift again:

  000010011000 → 0000100110000

  111101101000 → 1111011010000

- 01 → +1

  Add: 111101101 + 0000100110000 = 10000100011101

  Shift again:

  0000100110000 → 00001001100000

  1111011010000 → 11110110100000

- 00 → no operation

Shift again:

00001001100000 → 000010011000000

11110110100000 → 111101101000000

- 10 → −1

  Add: 10000100011101 + 111101101000000 = 1001110001011101

  Shift again:

  000010011000000 → 0000100110000000

  111101101000000 → 1111011010000000

- 01 → +1

  Add: 1001110001011101 + 0000100110000000 = 1010010111011101

  Shift again:

  0000100110000000 → 00001001100000000

  1111011010000000 → 11110110100000000

- 00 → no operation

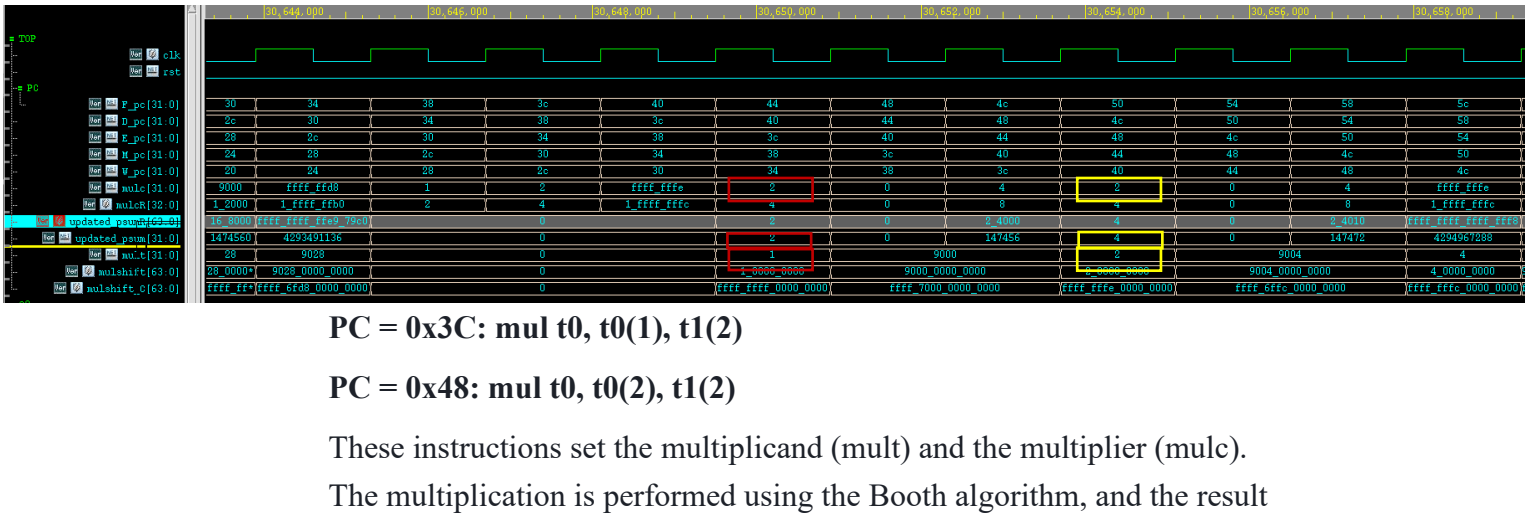
  Final partial product remains: 1010010111011101

iv. Therefore, the final result of 00010011 × 01001111 is: 1010010111011101

v. Taking the upper 8 bits of the final result gives the stored product value: 11011101

d. Waveform Explanation

i.



PC = 0x3C: mul t0, t0(1), t1(2)

PC = 0x48: mul t0, t0(2), t1(2)

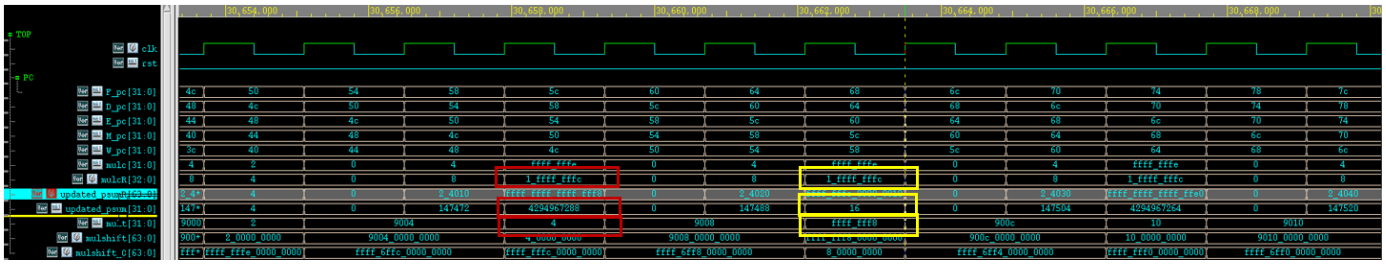
These instructions set the multiplicand (mult) and the multiplier (mulc). The multiplication is performed using the Booth algorithm, and the result

is stored in updated_psumR. The lower 32 bits of this result are extracted into updated_psum. In the waveform, mulcR represents the multiplier after appending a 0 to bit 0 (to enable pairwise bit comparison and operation). mulshift and mulshift_c show the shifted versions of the multiplicand, which are used for partial product accumulation based on the bit pair patterns.

Red box highlights the case 1 (mult) × 2 (mulc) = 2 (updated_psum)

Yellow box highlights the case 2 (mult) × 2 (mulc) = 4 (updated_psum)

ii.



PC = 0x54: mul t0, t0(4), t1(−2)

PC = 0x60: mul t0, t0(−8), t1(−2)

These instructions test multiplication with negative operands using Booth's signed handling, ensuring correct computation using two's complement and shift logic. The multiplicand (mult) and multiplier (mulc) are set for each multiplication instruction. The two operands are multiplied using the Booth algorithm, with the full result stored in updated_psumR. The lower 32 bits are extracted to updated_psum.

In the waveform:

- mulcR indicates the multiplier with a 0 appended to bit 0, used for detecting transitions in adjacent bits (i.e., 01, 10) to determine whether to add or subtract shifted versions of the multiplicand.
- mulshift and mulshift_c represent shifted forms of the multiplicand, used in each operation cycle based on the multiplier's bit pattern.

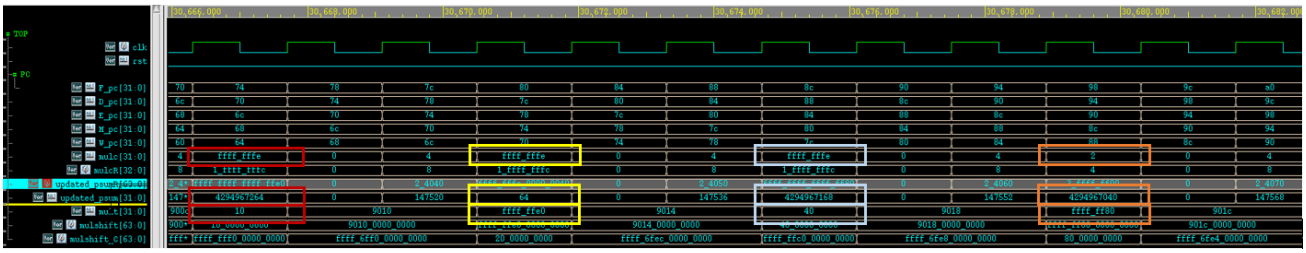
Red box highlights the multiplication:

4(mult) × (−2) (mulc) = −8 (updated_psum)

Yellow box highlights the multiplication:

−8 (mult) × (−2) (mulc) = 16 (updated_psum)

iii.



PC = 0x6C: mul t0, t0(16), t1(−2) → result: −32

PC = 0x78: mul t0, t0(−32), t1(−2) → result: 64

PC = 0x84: mul t0, t0(64), t1(−2) → result: −128

PC = 0x90: mul t0, t0(−128), t1(2) → result: −256

These waveform results verify that the Booth multiplier correctly handles signed values and shifting logic for both positive and negative operands. The multiplicand (mult) and multiplier (mulc) are set, and the Booth algorithm is used to compute their product. The full result is stored in updated_psumR, and the lower 32 bits are extracted into updated_psum.

In the waveform:

- mulcR: the multiplier with a 0 appended to the least significant bit, enabling pairwise bit comparison (01, 10, etc.).
- mulshift and mulshift_c: represent the shifted versions of the multiplicand, used in each operation cycle depending on the detected bit pattern.

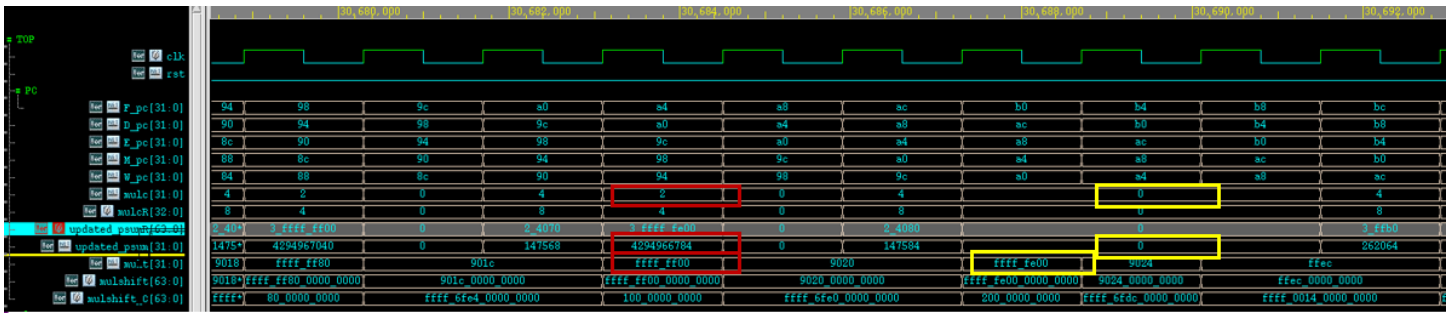
Red Box: 16 (mult) × (−2) (mulc) = −32

Yellow Box: −32 (mult) × (−2) (mulc) = 64

Light Blue Box: 64 (mult) × (−2) (mulc) = −128

Orange Box: −128 (mult) × 2 (mulc) = −256

iv.



PC = 0x9C: mul t0, t0(−256), t1(2) → Result: −512

PC = 0xA8: mul t0, t0(−32), t1(−2) → Result: 64

These results confirm that the Booth multiplier correctly handles positive and negative operands, dynamically applies signed extension and shifting logic, and efficiently computes the product via optimized partial sum accumulation. The multiplicand (mult) and multiplier (mulc) are processed using the Booth algorithm, with the full product stored in updated_psumR, and the lower 32 bits extracted into updated_psum.

In the waveform:

- mulcR represents the multiplier with an additional 0 appended to the least significant bit, used for detecting transition patterns (01, 10) during Booth encoding.
- mulshift and mulshift_c are the left-shifted versions of the multiplicand, updated every cycle based on the current two-bit Booth pattern.

Red Box: −256 (mult) × 2 (mulc) = −512 → updated_psum = −512

Yellow Box: −512 (mult) × 0 (mulc) = 0 → updated_psum = 0

v. These results validate that the Booth multiplier correctly performs:

- Sign-sensitive shifting
- Partial product accumulation
- And proper truncation to 32 bits

XI. Project Reflection

This project focused on the design and implementation of a pipelined CPU with support for floating-point operations and a high-speed multiplier. In addition to the standard five-stage pipeline (IF/ID/EX/MEM/WB), I integrated advanced components including a Floating Point Unit (FPU), a Branch Predictor, and a Booth Multiplier.

A. Hardware Design Workflow

I employed SuperLint for code linting and Design Compiler, ICC, and Layout tools to complete synthesis and physical design. For verification, I developed a comprehensive testbench and designed various test cases (integer, floating-point, logical, and control instructions) to achieve 100% coverage. Throughout the process, I gained experience across the entire hardware development flow—from RTL design, module integration, debugging, and verification, to physical implementation.

B. FPU Implementation & Challenges

The FPU simulates IEEE 754 single-precision format and supports

instructions such as FLW, FSW, FADD.S, FSUB.S, FMUL.S, FEQ.S, etc. I ensured correctness via waveform inspection and data forwarding mechanisms. Its implementation involves normalization, exponent adjustment, sign handling, and rounding. The logic is complex and debugging was time-consuming, especially for floating-point addition and subtraction, which require proper alignment of mantissas with different exponents and normalization of the results.

C. Booth Multiplier Design

The Booth multiplier uses a Radix-2 algorithm, which optimizes performance by reducing the number of addition operations in cases of consecutive 1s in the multiplier. By appending a 0 to the LSB of the multiplier, I can detect 01 and 10 patterns that correspond to +1 and -1, respectively. The operations are handled through shifted versions of the multiplicand (mulshift, mulshift_c) for both positive and negative terms. Although our custom multiplier is less efficient in terms of area and speed compared to the one synthesized by Design Compiler, this implementation helped us deeply understand how the algorithm maps to real hardware.

D. Debugging & Verification Experience

I encountered several challenges during the debugging phase, including: A one-cycle delay in the branch predictor signal, resulting in mis predicted jumps. Incomplete coverage, requiring edge cases (e.g., floating-point values with the leading 1 at different positions). Incorrect controller conditions causing FPU malfunctions. Post-synthesis issues such as port mismatches and clock continuity errors leading to layout violations. By using nWave to analyze waveforms and backtracking through the data flow, I gradually identified module-specific issues and resolved integration bugs.

E. Toolchain & Integration

This project also familiarized us with industrial tools like SuperLint, Design Vision, ICC, and Layout tools. I experienced the end-to-end ASIC flow from initial RTL to GDS, handled layout violations, and worked with pin and constraint definitions. The project emphasized the importance of clean coding style and modular design, as poor organization significantly increases debugging complexity during integration.

This project not only enhanced our skills in digital circuit design and

verification, but also deepened our understanding of system integration and debugging practices. It has been a valuable step toward preparing for real-world hardware development and academic research.

XII. References

A. Lecture Notes from the VLSI System Design Course

B. Lecture Notes from the Computer Organization Course

C. RISC-V Instruction Set Manual