VLSI System Design
Project Report

One. Introduction to the system

1. Instruction set format

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						

2. The name, length, and description of the fields in the instruction set format

Instruction set format name	Length	Illustrate
func7	7 bits	Further determine the type of instruction
rs1	5 bits	source register1 address
rs2	5 bits	source register2 address
func3	3 bits	Further determine the type of instruction
rd	5 bits	destination adress
opcode	7 bits	Determine the type of instruction
imme	Depending on the type	constant

3. Branch directives and Jump directives

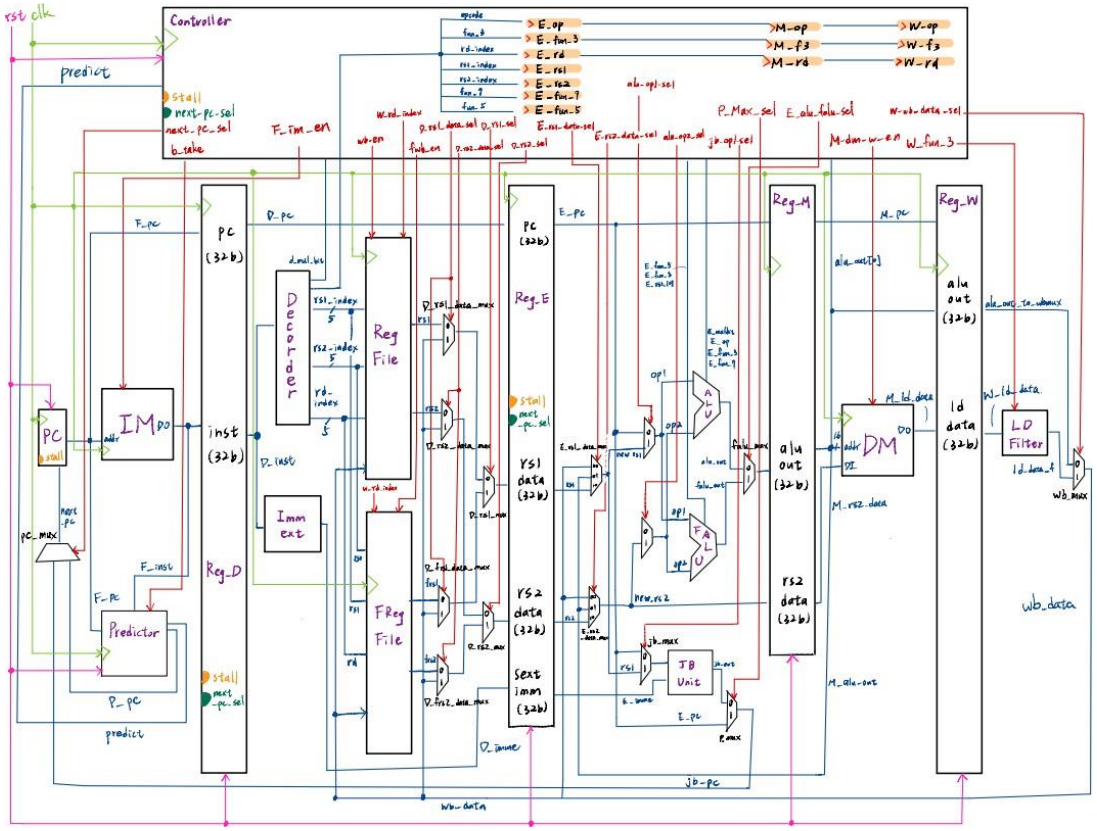
   Branch address is pc+{imme,1'b0}. The imm[0] here must be 0, and the calculation should be shifted to the left by 1 bit to give a multiple of 4. The result is the memory address that should be jumped.

   JAL's address is pc+{imm20,1'b0}. The imm[0] here must be 0 as well, and the same branch address is calculated

   Jalr's address is rs1+sext(imm12)&(~32'd1). &(~32'd1) is used to change the last digit of the address to 0.

4. Schema description

   i. Architecture diagram and description



ii. Description of individual components

1. IM: the memory that stores the instruction.
2. Decoder: Disassembles the instruction command into various parts for easy use at the next level.
3. immext: used to extend the constant to 32bits.
4. regfile: 32 registers for calculation results, addresses, etc.
5. ALU: The unit of operation that calculates operand1 and operand2 results based on opcode. 6.JB_unit: Calculate the branch address.
7. DM: the memory in which the data is stored.
8. LD filter: load command is divided into word, half word, and byte, and the LD filter determines the size to be loaded.
9. controller: provides various sel signals.

Two. Instructions that can be executed by the system

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

Three. System validation method and result analysis

1. Verification method

i. Test Basic Instructions: prog0 is a program that tests the basic instruction set (RV32I), including add, sub, addi, and other commands. After the test is completed, the answers to each instruction are written into the data memory, and then the answers are matched with the golden data. If the test is successful, the simulation pass message will pop up.

```
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 !!    **
    **                        **
    ****************************
```

ii. Test multiplication: prog4 is the program for testing extended instructions (RV32M-MUL), mul commands. After the test is completed, the answers to each instruction are written into the data memory, and then the answers are matched with the golden data. If the test is successful, the simulation pass message will pop up.

```
.text

.global main
```

```
main:
    addi sp, sp, -4
    sw s0, 0(sp)
    la s0, _answer          #addr


mul:

    li t0, 1
    li t1, 2
    li t2, -2
    mul t0, t0, t1          # t0 = 2
    sw t0, 0(s0)
    addi s0, s0, 4
    mul t0, t0, t1          # t0 = 4
    sw t0, 0(s0)
    addi s0, s0, 4
    mul t0, t0, t2          # t0 = -8
    sw t0, 0(s0)
    addi s0, s0, 4
    mul t0, t0, t2          # t0 = 16
    sw t0, 0(s0)
    addi s0, s0, 4
    mul t0, t0, t2          # t0 = -32
    sw t0, 0(s0)
    addi s0, s0, 4
    mul t0, t0, t2          # t0 = 64
    sw t0, 0(s0)
    addi s0, s0, 4
    mul t0, t0, t2          # t0 = -128
    sw t0, 0(s0)
    addi s0, s0, 4
    mul t0, t0, t1          # t0 = -256
    sw t0, 0(s0)
    addi s0, s0, 4
    mul t0, t0, t1          # t0 = -512
    sw t0, 0(s0)
    addi s0, s0, 4
    mul t0, t0, x0          # t0 = 0
    sw t0, 0(s0)
```

```
main_exit:
    lw s0, 0(sp)
    addi sp, sp, 4
    ret
```

```
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 !!   **
    **                       **
    ***************************
```

iii. Test floating-point number: prog3 is a program for testing extended instructions (RV32F), including FLW, FSW and other commands. After the test is completed, the answers to each instruction are written into the data memory, and then the answers are matched with the golden data. If the test is successful, the simulation pass message will pop up.

```
.text
.global main


main:
    addi sp, sp, -4
    sw s0, 0(sp)
    la s0, _answer          #addr

############float test###################
fadd:
    li t0, 0x3FA00000
    sw t0, 0(s0)
    flw f0, 0(s0)               # f0 = 1.25
    li t1, 0x3E000000
    sw t1, 0(s0)
    flw f1, 0(s0)               # f1 = 0.125
```

```
        fadd.s f0, f0, f1          # f0 = 1.375
        fadd.s f0, f0, f1          # f0 = 1.5

        li t2, 0xBF800000
        sw t2, 0(s0)
        flw f2, 0(s0)              # f2 = -1.0
        fadd.s f0, f0,             # f0 = 0.5
        f2 fadd.s f0,              # f0 = -0.5
        f0, f2 fadd.s              # f0 = -1.5
        f0, f0, f2                 # f0 = -2.5
        fadd.s f0, f0,             # f0 = -2.375
        f2 fadd.s f0,              # f0 = -2.25

        fsw f0, 0(s0)
        addi s0, s0, 4

fsub:
        li t0, 0x3FA00000
        sw t0, 0(s0)
        flw f0, 0(s0)              # f0 = 1.25
        li t1, 0x3F400000
        sw t1, 0(s0)
        flw f1, 0(s0)              # f1 = 0.75

        fsub.s f0, f0, f1          # f0 = 0.5
        fsub.s f0, f0, f1          # f0 = -0.25
        fsub.s f0, f0, f1          # f0 = -1.0
        fsub.s f0, f0, f1          # f0 = -1.75

        li t2, 0xBF800000
        sw t2, 0(s0)
        flw f2, 0(s0)              # f2 = -1.0

        fsub.s f0, f0, f2          # f0 = -0.75
        fsub.s f0, f0, f2          # f0 = 0.25
        fsub.s f0, f0, f2          # f0 = 1.25
        fsub.s f0, f0, f2          # f0 = 2.25
```

```
        fsw f0, 0(s0)
        addi s0, s0, 4

fmul:

        li t0, 0x3FC00000
        sw t0, 0(s0)
        flw f0, 0(s0)
                                # f0 = 1.5
        li t1, 0x40000000
        sw t1, 0(s0)
        flw f1, 0(s0)
                                # f1 = 2.0


        fmul.s f0, f0, f1       # f0 = 3.0
        fmul.s f0, f0, f1       # f0 = 6.0


        li t2, 0xBF000000
        sw t2, 0(s0)
        flw f2, 0(s0)
                                # f2 = -0.5
        fmul.s f0, f0, f2       # f0 = -3.0
        fmul.s f0, f0, f2       # f0 = 1.5
        fmul.s f0, f0, f2       # f0 = -0.75
        fmul.s f0, f0, f1       # f0 = -1.5
        fmul.s f0, f0, f1       # f0 = -3.0


        fsw f0, 0(s0)
        addi s0, s0, 4


fcvt_w_s:                       # float to int
        fcvt.w.s t0, f0         # 3.0 -> 3
        sw t0, 0(s0)
        addi s0, s0, 4



        li t1, 0x3FC00000
        sw t1, 0(s0)
        flw f1, 0(s0)
                                # f1 = 1.5
        fcvt.w.s t1, f1         # 1.5 -> 2  (round-to-even)
        sw t1, 0(s0)
        addi s0, s0, 4
```

```
        fcvt.w.s t2, f2             # -0.5 -> 0 (round-to-even)
        sw t2, 0(s0)
        addi s0, s0, 4

        li t3, 0xBFC00000
        sw t3, 0(s0)
        flw f3, 0(s0)               # f3 = -1.5
        fcvt.w.s t3, f3             # -1.5 -> -2 (round-to-even)
        sw t3, 0(s0)
        addi s0, s0, 4

fcvt_wu_s:                          # float to unsigned int
        fcvt.wu.s t0, f0            # 3.0 -> 3
        sw t0, 0(s0)
        addi s0, s0, 4

        fcvt.wu.s t1, f1            # 1.5 -> 2
        sw t1, 0(s0)
        addi s0, s0, 4

        fcvt.wu.s t2, f2            # -0.5 -> 0
        sw t2, 0(s0)
        addi s0, s0, 4

        fcvt.wu.s t3, f3            # -1.5 -> 0
        sw t3, 0(s0)
        addi s0, s0, 4

fmv_x_w:                            #float transfer to int (IEEE-754)
        fmv.x.w t0, f0              # 3.0 -> 0x40400000
        sw t0, 0(s0)
        addi s0, s0, 4

        fmv.x.w t1, f1              # 1.5 -> 0x3FC00000
        sw t1, 0(s0)
        addi s0, s0, 4
```

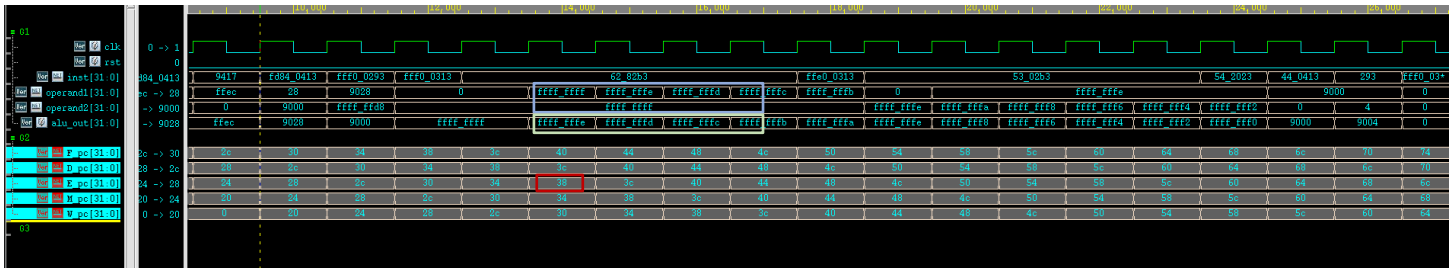
```
    fmv.x.w t2, f2          # -0.5 -> 0xBF000000
    sw t2, 0(s0)
    addi s0, s0, 4

    fmv.x.w t3, f3          # -1.5 -> 0xBFC00000
    sw t3, 0(s0)
    addi s0, s0, 4

fmv_w_x:                    # int transfer to float (IEEE-754)
    fmv.w.x f0, t0          # 0x40400000 -> 3.0
    fsw f0, 0(s0)
    addi s0, s0, 4

    fmv.w.x f1, t1          # 0x3FC00000 -> 1.5
    fsw f1, 0(s0)
    addi s0, s0, 4

    fmv.w.x f2, t2          # 0xBF000000 -> -0.5
    fsw f2, 0(s0)
    addi s0, s0, 4

    fmv.w.x f3, t3          # 0xBFC00000 -> -1.5
    fsw f3, 0(s0)
    addi s0, s0, 4

fcvt_s_w:                   # int to float
    li t0, 3
    fcvt.s.w f0, t0         # 3 -> 3.0
    fsw f0, 0(s0)
    addi s0, s0, 4

    li t1, 1
    fcvt.s.w f1, t1         # 1 -> 1.0
    fsw f1, 0(s0)
    addi s0, s0, 4

    li t2, -1
    fcvt.s.w f2, t2         # -1 -> -1.0
```

```
        fsw f2, 0(s0)
        addi s0, s0, 4

        li t3, -3
        fcvt.s.w f3, t3             # -3 -> -3.0
        fsw f3, 0(s0)
        addi s0, s0, 4


fcvt.s.wu:                          # unsigned int to float
        fcvt.s.wu f0, t0            # 3 -> 3.0
        fsw f0, 0(s0)
        addi s0, s0, 4

        fcvt.s.wu f1, t1            # 1 -> 1.0
        fsw f1, 0(s0)
        addi s0, s0, 4

        fcvt.s.wu f2, t2            # -1 -> 0x7F800000(+inf)
        fsw f2, 0(s0)
        addi s0, s0, 4

        fcvt.s.wu f3, t3            # -3 -> 0x7F800000(+inf)
        fsw f3, 0(s0)
        addi s0, s0, 4

fmin.s:
    li t0, 0x3FA00000
    sw t0, 0(s0)
    flw f0, 0(s0)                   # f0 = 1.25
    li t1, 0x3E000000
    sw t1, 0(s0)
    flw f1, 0(s0)                   # f1 = 0.125
    fmin.s f3,f0,f1       # f3 = 0.125
    fmin.s f4,f0,f0       # f4 = 1.25
    fsw f3, 0(s0)                 # mem[answer] = 0.125
    addi s0, s0, 4
    fsw f4, 0(s0)                 # mem[answer+4] = 1.25
```

```
    addi s0, s0, 4

fmax.s:
    fmax.s f3,f0,f1      # f3 = 1.25
    fmax.s f4,f1,f1      # f4 = 0.125
    fsw f3, 0(s0)            # mem[answer+8] = 1.25
    addi s0, s0, 4
    fsw f4, 0(s0)            # mem[answer+12] = 0.125
    addi s0, s0, 4

feq.s:
    feq.s t0,f0,f1       # t0 = 0
    sw t0, 0(s0)             # mem[answer+16]=0
    addi s0, s0, 4
    feq.s t0,f0,f0       # t0 = 1
    sw t0, 0(s0)             # mem[answer+20]=1
    addi s0, s0, 4

flt.s:
    flt.s t0,f0,f1       # t0 = 0
    sw t0, 0(s0)
    addi s0, s0, 4           # mem[answer+24]=0
    flt.s t0,f0,f0       # t0 = 0
    sw t0, 0(s0)
    addi s0, s0, 4       # mem[answer+28]=0

fle.s:
    fle.s t0,f0,f1       # t0 = 0
    sw t0, 0(s0)
    addi s0, s0, 4           # mem[answer+32]=0
    fle.s t0,f0,f0       # t0 = 1
    sw t0, 0(s0)



main_exit:
      lw s0, 0(sp)
      addi sp, sp, 4
      ret
```

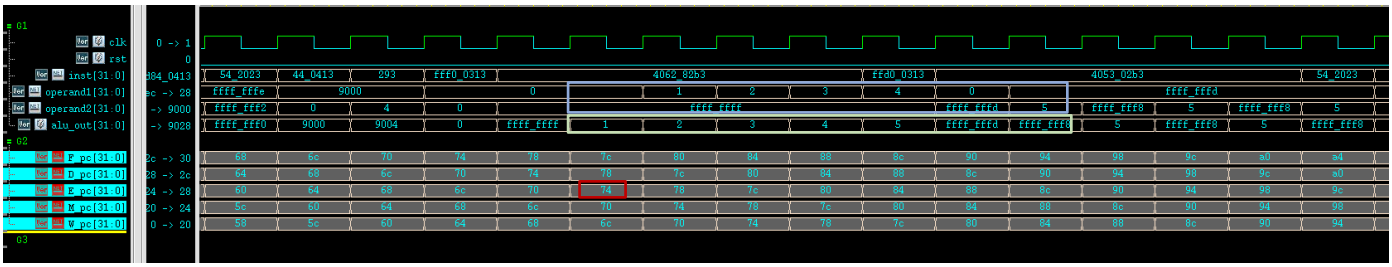
```
DM['h9000] = c0100000, pass
DM['h9004] = 40100000, pass
DM['h9008] = c0400000, pass
DM['h900c] = fffffffd, pass
DM['h9010] = 00000002, pass
DM['h9014] = ffffffff, pass
DM['h9018] = fffffffe, pass
DM['h901c] = 00000003, pass
DM['h9020] = 00000002, pass
DM['h9024] = 00000001, pass
DM['h9028] = 00000002, 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 !!    **
    **                        **
    ****************************
```

2. Analysis of the results (with a screenshot of nWave attached).
   i. "Single" instruction correctness
      - **add rd,rs1,rs2**



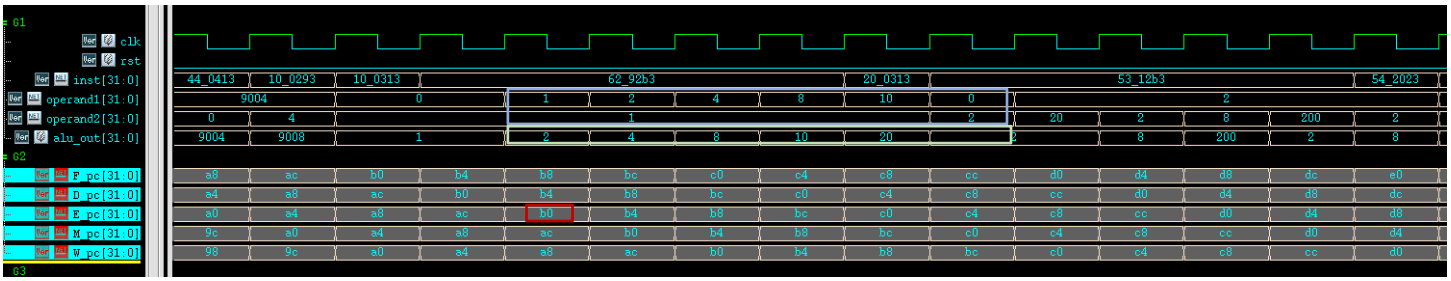
The value of rs1 is read to operand1, the value of rs2 is read to operand2 (blue), and finally the answer is calculated by alu (green).
In the code, pc=38 (red) is the add command, so the ALU output is operand1+operand2. The leftmost part of the box shows (-1)+(-1)=-2.

- **sub rd,rs1,rs2**



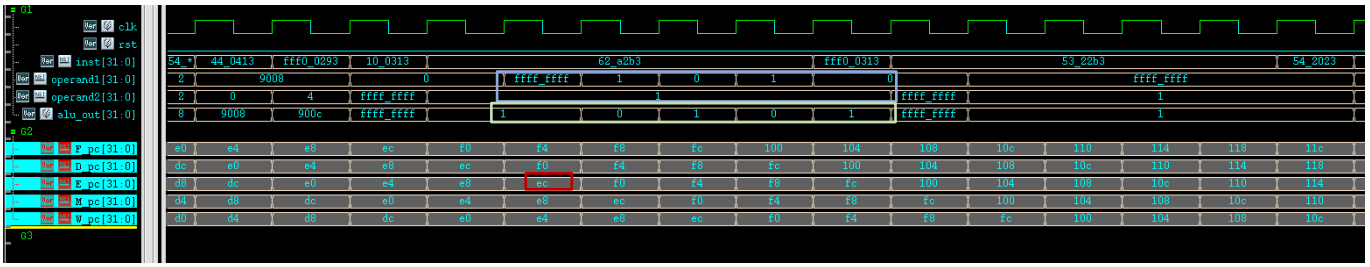
The value of rs1 is read to operand1, the value of rs2 is read to operand2 (blue), and finally the answer is calculated by alu (green).
In the code, pc=74 (red) is the sub instruction, so the alu output is operand1-operand2. The leftmost part of the box shows (0)-(-1)=1.

- **sll rd,rs1,rs2**



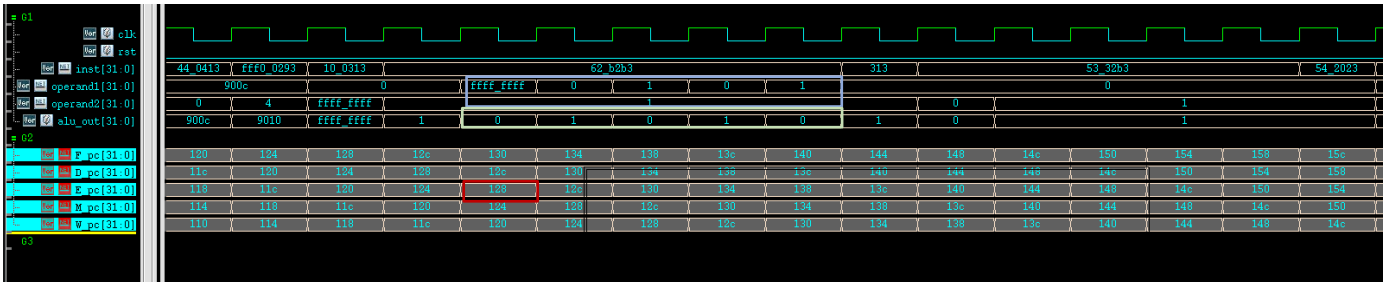
The value of rs1 is read to operand1, the value of rs2 is read to operand2 (blue), and finally the answer is calculated by alu (green).
In the code, pc=b0 (red) is the sll command, so the alu output is operand1<<operand2. The leftmost side of the box shows 1<<1=1*2=2

- **slt rd,rs1,rs2**



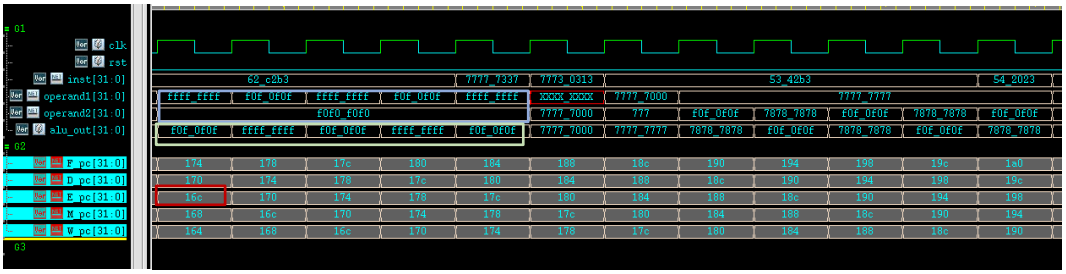
The value of rs1 is read to operand1, the value of rs2 is read to operand2 (blue), and finally the answer is calculated by alu (green).
In the code, PC=EC (red) is the SLT command, so the ALU output is the BOOL value of operand1<operand2.
The leftmost side of the box shows ((-1)<1)=true=1.

- **sltu rd,rs1,rs2**



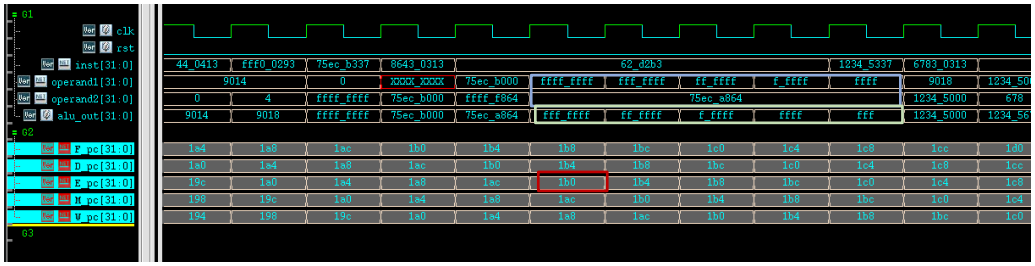
The value of rs1 is read to operand1, the value of rs2 is read to operand2 (blue), and finally the answer is calculated by alu (green).
In the code, PC=128 (red) is the SLTU instruction, so the ALU output is the BOOL value of operand1<operand2 (unsigned).
The leftmost side of the box shows ((ffff_ffff)<1)=false=0.

- **xor rd,rs1,rs2**



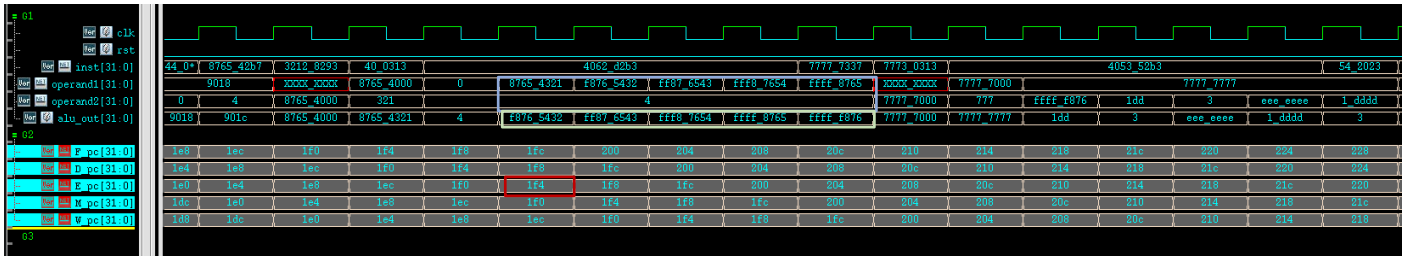
The value of rs1 is read to operand1, the value of rs2 is read to operand2 (blue), and finally the answer is calculated by alu (green).
In the code, PC=16C (red) is the XOR instruction, so the ALU output is the value of operand1^operand2. The leftmost side of the box shows ((ffff_ffff)^(f0f0_f0f0))=0f0f_0f0f

- **srl rd,rs1,rs2**



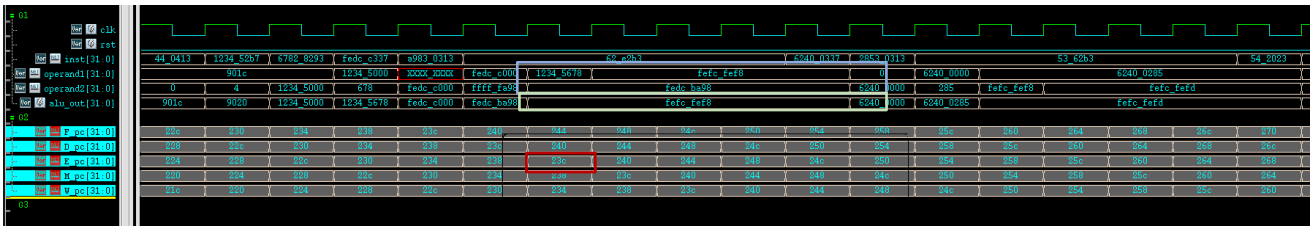
The value of rs1 is read to operand1, the value of rs2 is read to operand2 (blue), and finally the answer is calculated by alu (green).
In the code, PC=1B0 (red) is the SRL instruction, so the ALU output is the value of operand1>>operand2[4:0].
The leftmost side of the box shows (ffff_ffff)>>$(00100)_{(2)}$=fff_ffff.

- **sra rd,rs1,rs2**



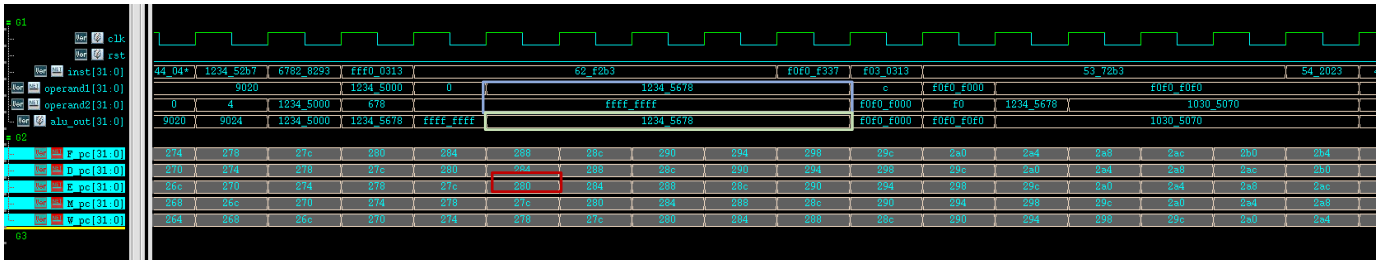
The value of rs1 is read to operand1, the value of rs2 is read to operand2 (blue), and finally the answer is calculated by alu (green).
In the code, PC=1F4 (red) is the SRA instruction, so the ALU output is the value of operand1>>operand2[4:0] (signed extension).
The leftmost side of the box shows ((8765_4321)>>(00100)$_{(2)}$)=f876_5432

- **or rd,rs1,rs2**



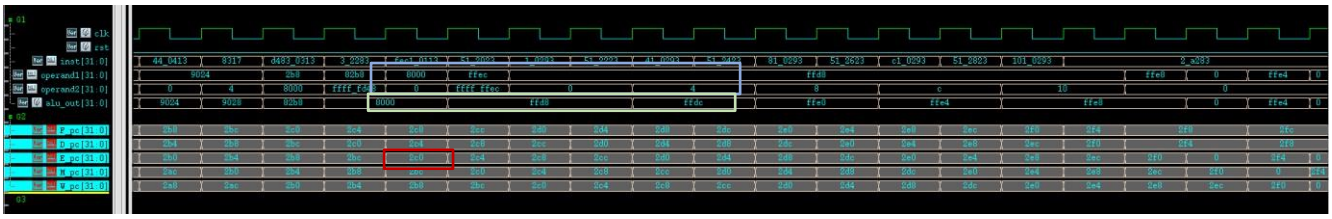
The value of rs1 is read to operand1, the value of rs2 is read to operand2 (blue), and finally the answer is calculated by alu (green).
In the code, PC=23C (red) is the OR instruction, so the ALU output is the value of operand1|operand2. On the far left of the box is ((1234_5678)|(fedc_ba98))=f876_5432

- **and rd,rs1,rs2**



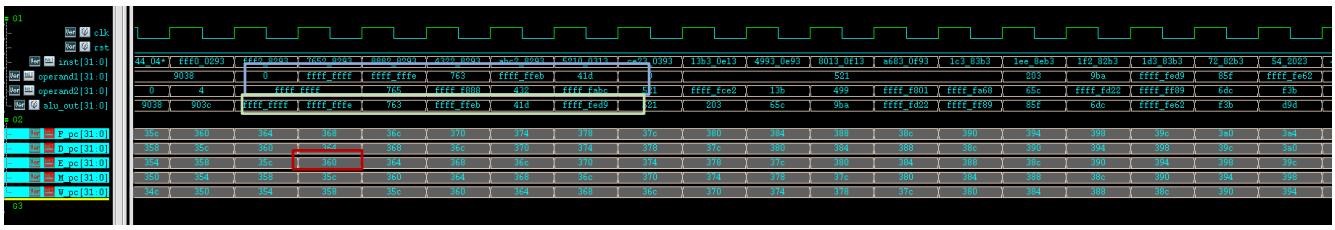
The value of rs1 is read to operand1, the value of rs2 is read to operand2 (blue), and finally the answer is calculated by alu (green).
In the code, pc=280 (red) is the AND command, so the alu output is the value of operand1&operand2.
The leftmost box shows ((1234_5678)&(ffff_ffff))=1234_5678

- **lw rd,imm(rs1)**



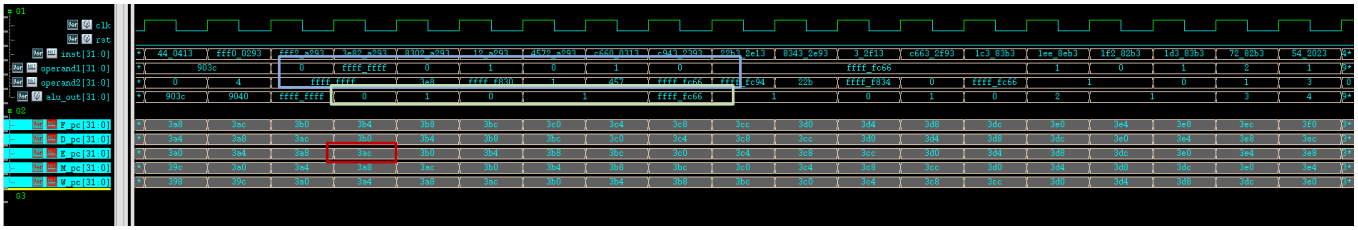
The value of rs1 is read to operand1, the value of imm is read to operand2 (blue), and finally the value address (green) is calculated by alu.
In the code, PC=2C0 (red) is the LW instruction, so the ALU output is the value address (the value of RS1
+imm)。
The leftmost part of the box shows ('h8000+0)='h8000.

- **addi rd,rs1,imm**



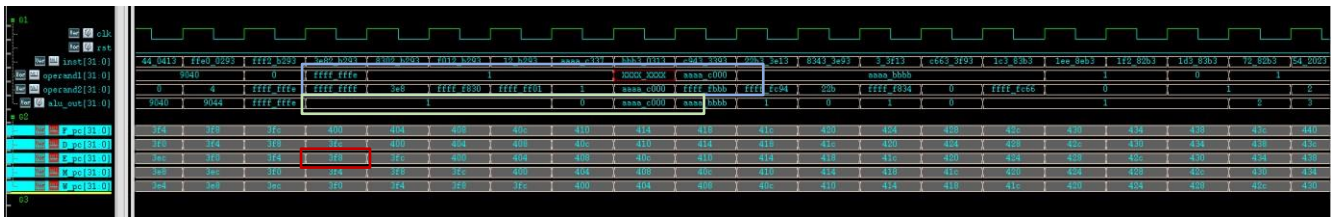
The value of RS1 is read to operand1, the value of IMM is read to operand2 (blue), and finally the result is calculated by ALU (green).
In the code, PC=360 (red) is the ADDI instruction, so the ALU output is operand1+operand2. The leftmost part of the box shows (-1)+(-1)=-2.

- **slti rd,rs1,imm**



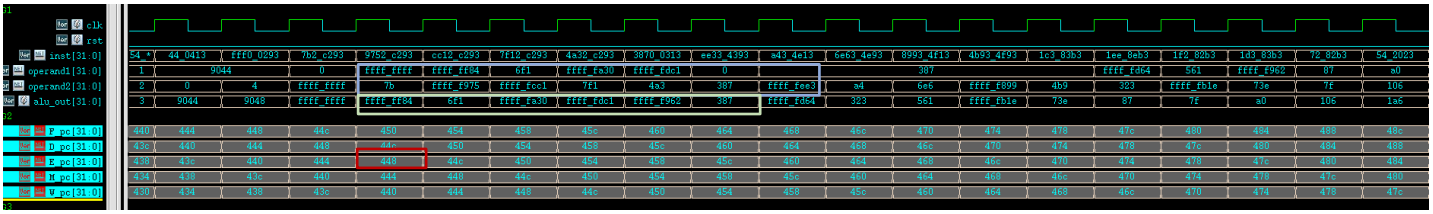
The value of RS1 is read to operand1, the value of IMM is read to operand2 (blue), and finally the result is calculated by ALU (green).
In the code, PC=360 (red) is the SLTI command, so the ALU output is the BOOL value of (operand1<operand2).
The leftmost side of the box shows ((-1)<(-1))=false=0.

- **sltiu rd,rs1,imm**



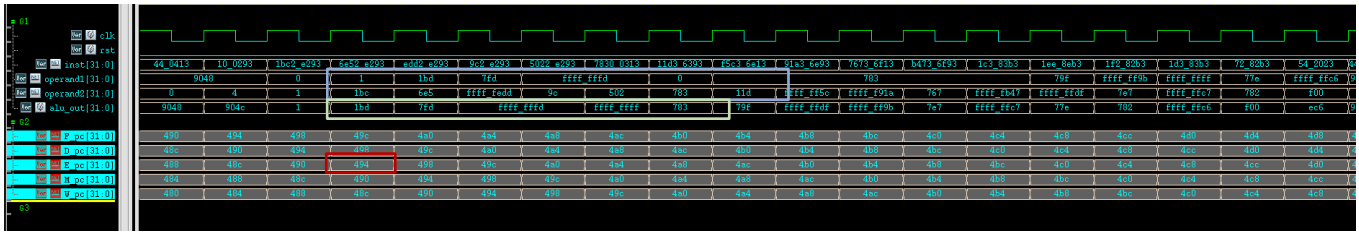
The value of RS1 is read to operand1, the value of IMM is read to operand2 (blue), and finally the result is calculated by ALU (green).
In the code, PC=3F8 (red) is the SLTIU instruction, so the ALU output is the BOOL value of (operand1<operand2) (unsigned).
The leftmost side of the box shows ((-2)<(-1))=true=1.

- **xori rd,rs1,imm**



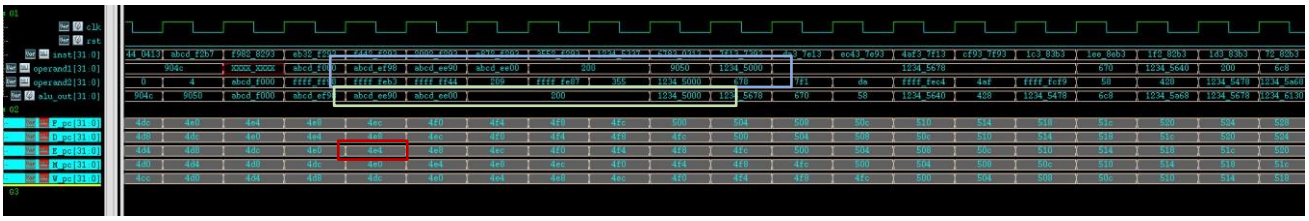
The value of RS1 is read to operand1, the value of IMM is read to operand2 (blue), and finally the result is calculated by ALU (green).
In the code, pc=448 (red) is the xori command, so alu outputs as the value of (operand1^operand2).
The leftmost side of the box shows ((ffff_ffff)^(0000_007b))=ffff_ff84.

- **ori rd,rs1,imm**



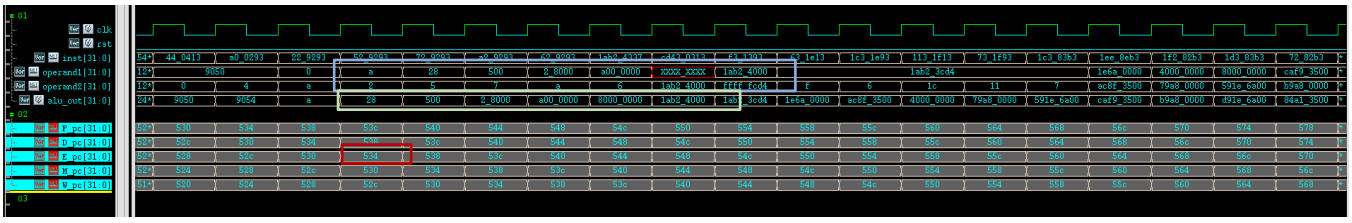
The value of RS1 is read to operand1, the value of IMM is read to operand2 (blue), and finally the result is calculated by ALU (green).
In the code, PC=494 (red) is the ori command, so the ALU output is the value of (operand1| operand2). On the far left of the box is ((1)|(1bc))=1bd。

- **andi rd,rs1,imm**



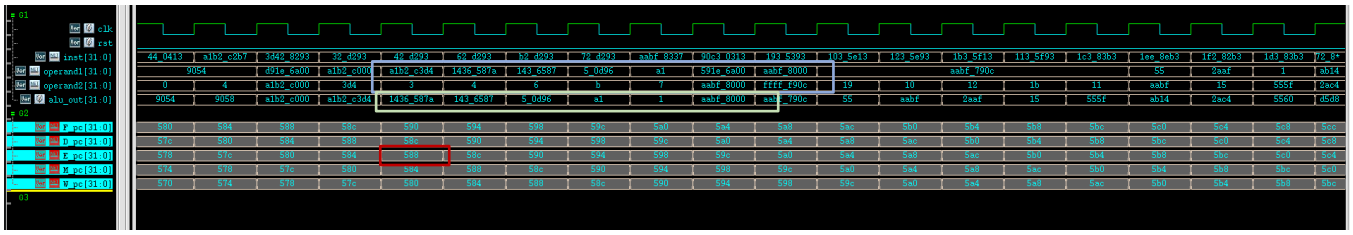
The value of RS1 is read to operand1, the value of IMM is read to operand2 (blue), and finally the result is calculated by ALU (green).
In the code, PC=4E4 (red) is the andi instruction, so ALU outputs as the value of (operand1&operand2).
On the far left of the box is ((abcd_ef98)|(ffff_feb3))=abcd_ee90。

- **slli rd,rs1,imm**



The value of RS1 is read to operand1, the value of IMM is read to operand2 (blue), and finally the result is calculated by ALU (green).
In the code, PC=534 (red) is the SLLI instruction, so the ALU output is the value of (operand1<<operand2).
On the far left of the box is ((a)<<(2))=1010<<2=28 (16).

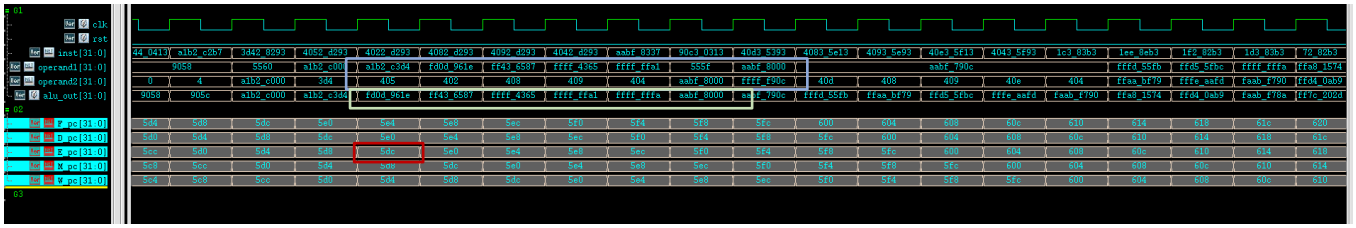
- **srli rd,rs1,imm**



The value of RS1 is read to operand1, the value of IMM is read to operand2 (blue), and finally the result is calculated by ALU (green).
In the code, PC=588 (red) is the SRLI instruction, so the ALU output is the value of (operand1>>operand2).

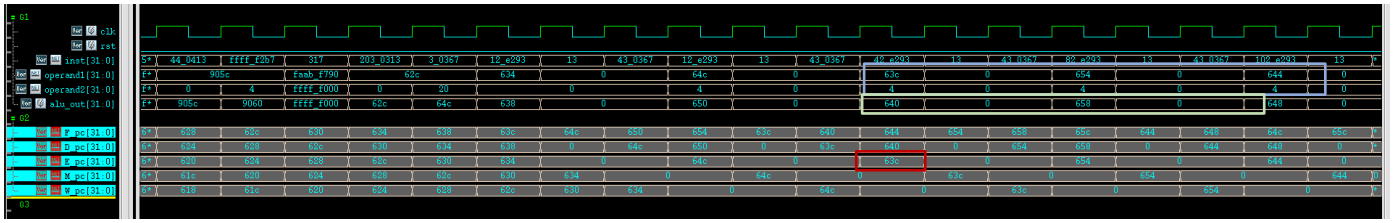
The leftmost side of the box shows (a1b2_c3d4)>>3)=1436_587a.

- **srai rd,rs1,imm**



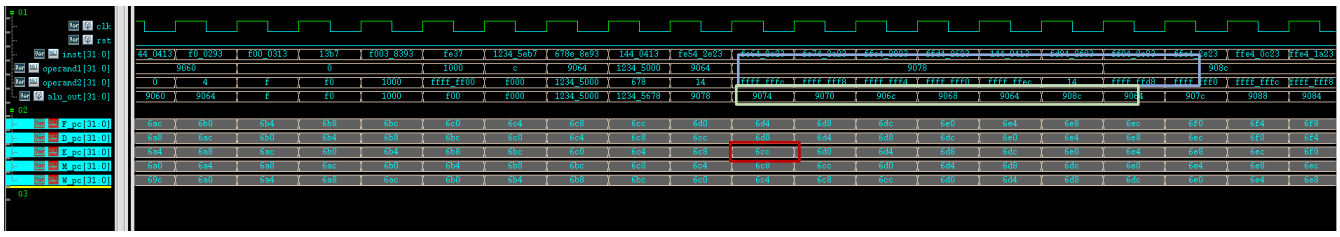
The value of RS1 is read to operand1, the value of IMM is read to operand2 (blue), and finally the result is calculated by ALU (green).
In the code, PC=5DC (red) is the SRAI instruction, so the ALU output is the value of (operand1>>operand2[4:0]).
The leftmost side of the box shows ((a1b2_c3d4)>>5)=fd0d_961e.

- **jalr rd,rs1,imm**



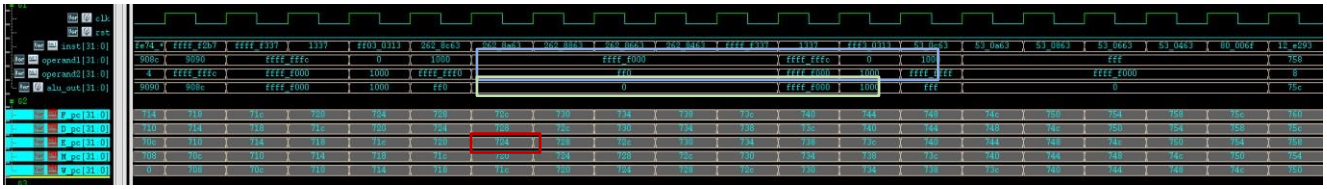
The value of rs1 is read to operand1, the value of imm is read to operand2 (blue), and finally the hop address (green) is calculated by alu. Move PC+4 to RD again.
In the code, PC=63C (red) is the JALR command, so the ALU output is the jump address. The leftmost part of the box shows 63c+4=640.

- **sw rd,imm(rs1)**



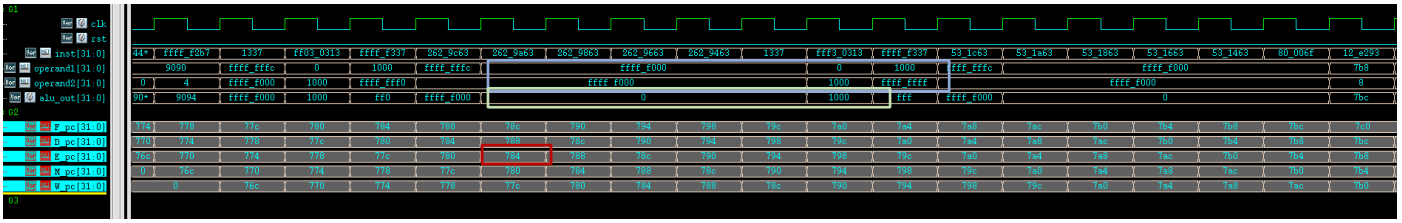
The value of rs1 is read to operand1, the value of imm is read to operand2 (blue), and finally the hop address (green) is calculated by alu.
In the code, PC=6cc (red) is the SW command, so the ALU output is the storage address. The leftmost side of the box shows 9078+(-4)=9074.

- **beq rd,rs1,imm**



The value of rs1 is read to operand1, the value of imm is read to operand2 (blue), and finally the operand1==operand2 (green) is calculated by alu. In the code, PC=724 (red) is the SW instruction, so the ALU output is (operand1==operand2). The leftmost side of the box shows (ffff_f000 == ff0)=0.

- **bne rd,rs1,imm**



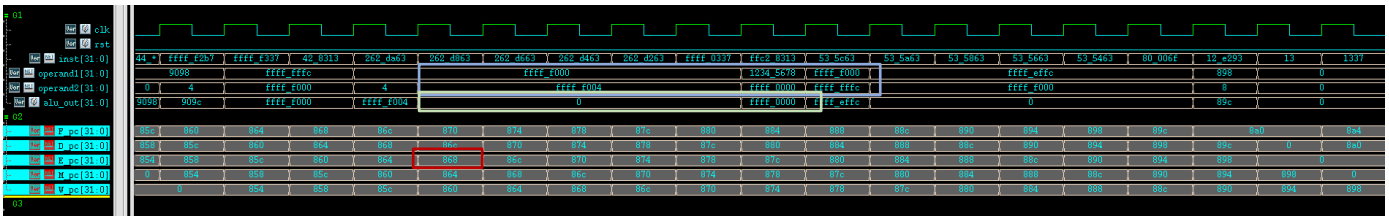
The value of rs1 is read to operand1， imm reads to operand2 (blue), which is finally calculated by ALU operand1!=operand2 bool value(green)。In the code, PC=784 (red) is the BNE instruction, so the ALU output is (operand1!=operand2). The leftmost side of the box shows (ffff_f000 != ffff_f0000)=0.

- **blt rd,rs1,imm**

The value of rs1 is read to operand1， imm reads to operand2 (blue), which is finally calculated by ALU operand1<operand2 bool value(green)。In the code, PC=7E4 (red) is the BLT instruction, so the ALU output is (operand1<operand2).

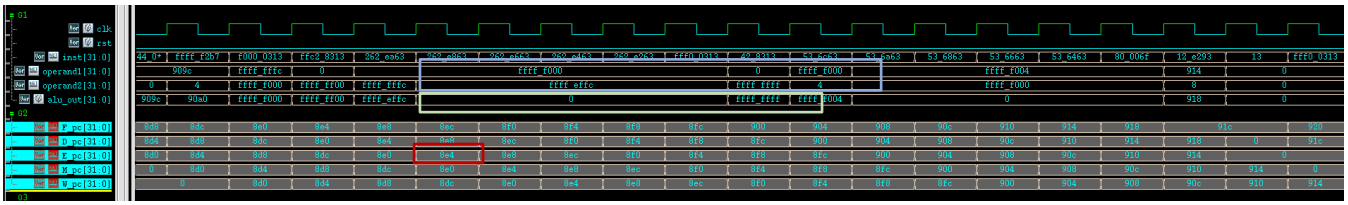
The leftmost side of the box shows (ffff_f000 < ffff_effc)=0.

- **bge rd,rs1,imm**



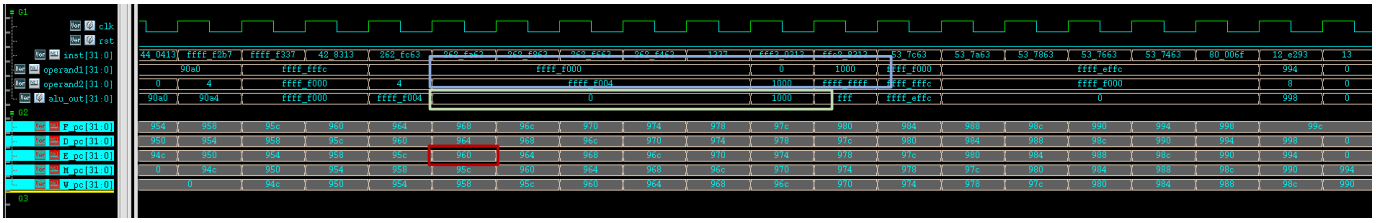
The value of rs1 is read to operand1， imm reads to operand2 (blue), which is finally calculated by ALU operand1>=operand2 bool value(green)。
In the code, PC=868 (red) is the BGE command, so the ALU output is (operand1>=operand2). The leftmost side of the box shows (ffff_f000 > = ffff_f004) = 0.

- **bltu rd,rs1,imm**



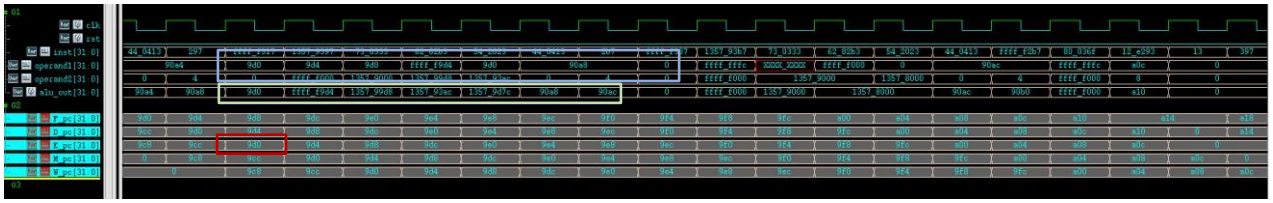
The value of rs1 is read to operand1， imm reads to operand2 (blue), which is finally calculated by ALU operand1<operand2 bool value(green)。
In the code, PC=8E4 (red) is the BLTU instruction,
so the ALU output is
(operand1<operand2)(unsunsigned).
The leftmost side of the box shows (ffff_f000 < ffff_effc)=0.

- **bgeu rd,rs1,imm**



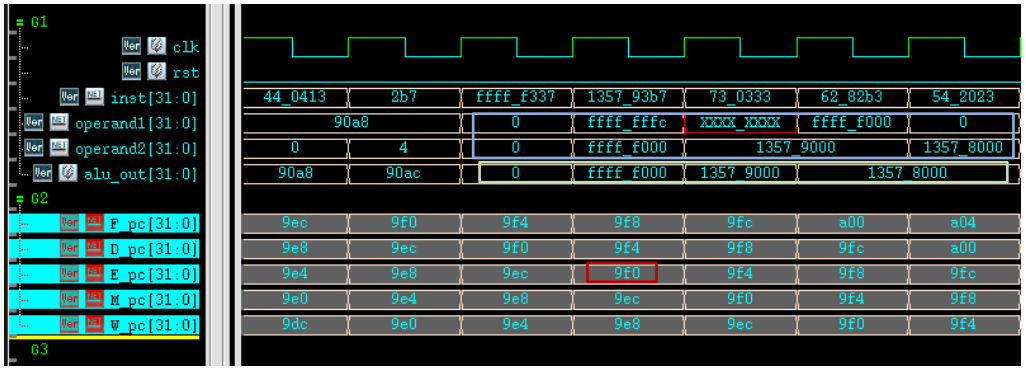
The value of rs1 is read to operand1， imm reads to operand2 (blue), and finally operand1>=operand2 is calculated by alu bool value(green)。
In the code, PC=960 (red) is the BGEU instruction,
so the ALU output is
(operand1>=operand2)(unsigned).
The leftmost side of the box shows (ffff_f000 > = ffff_f004)=0.

- **auipc rd,imm**



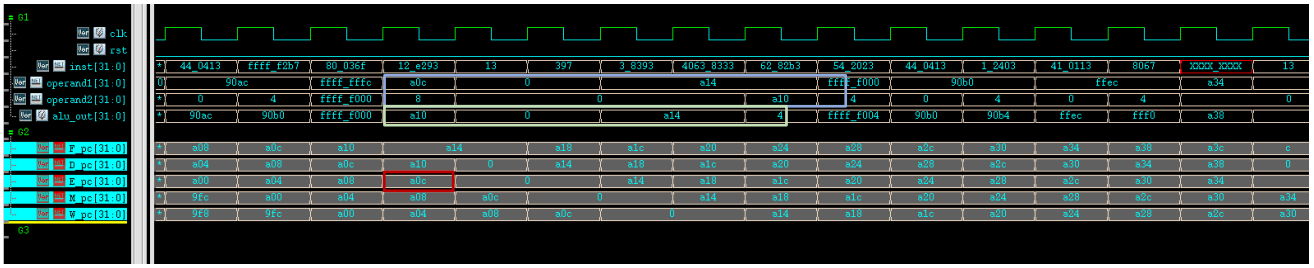
The value of the PC is read to operand1, the value of the IMM is read to operand2 (blue), and finally the operand1+operand2 is calculated by alu. In the code, PC=9D0 (red) is the AUIPC instruction, so the ALU output is operand1+operand2. The leftmost side of the box shows (9d0+0)=9d0.

- **lui rd,imm**



The value of the imm is read to operand2 (blue) and finally calculated by alu to get {imm,12{1'b0}}. In the code, PC=9F0 (red) is the LUI instruction, so the ALU output is {imm,12{1'b0}}. The leftmost part of the box shows 0xfffff->ffff_f000.

- **jal rd,imm**



ALU calculates PC+4 and deposits it into RD.
In the code, PC=A0C (red) is the JAL instruction, so the ALU output is PC+4. The leftmost part of the box shows a0c+4=a10.

ii. Procedural correctness

A. Sorting

```
.text


_start:


init_stack:
    # set stack pointer
    la sp, _stack


SystemInit:
    # jump to main
    jal main


SystemExit:
    # End simulation
    # Write -1 at _sim_end(0xfffc)
    la t0, _sim_end
    li t1, -1
    sw t1, 0(t0)


dead_loop:
    # infinite loop
    j dead_loop
.data
num_test: .word 3
TEST1_SIZE: .word 34
TEST2_SIZE: .word 19
TEST3_SIZE: .word 29
test1: .word 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: .word -3,-23,-22,-6,-21,-19,-1,0,-2,-47,-17,-46,-6,-30,-50,-13,-47,-9,-50
```

```
test3: .word -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



.text

.global main
setup:
	li	ra, -1

	li	sp, 0x7ffffff0
main:
############################################################

	# <  Variable >

	#	s0 : test_size

	#	t0 : address of test

################ callee save ##############################

	addi sp,sp,-8	#  Given storage space

				# sp=sp-8

	sw	ra,0(sp)	# ra->mem[@sp-4]
	sw	s0,4(sp)
################ merge test1 ##############################

	la	t0,test1		# t0 = address of test1
	lw	s0,TEST1_SIZE # s0 = t1_size
	addi s0,s0,-1		# t1_size-1



	mv	a0,t0			# a0= address of test1
	mv	a1,x0			 # a1= 0

	mv	a3,s0			# a3= size-1



	# call mergesort and do caller saving
	addi sp,sp,-4
	sw	ra,0(sp)

	jal  ra,mergesort
```

```
    lw      ra,0(sp)
    addi sp,sp,4



################ merge test2 ############################

    la      t0,test2

    lw      s0,TEST2_SIZE

    addi s0,s0,-1



    mv      a0,t0

    mv      a1,x0

    mv      a3,s0



    # call mergesort and do caller saving
    addi sp,sp,-4
    sw      ra,0(sp)

    jal ra,mergesort
    lw      ra,0(sp)
    addi sp,sp,4


################ merge test3 ############################

    la      t0,test3

    lw      s0,TEST3_SIZE

    addi s0,s0,-1



    mv      a0,t0

    mv      a1,x0

    mv      a3,s0



    # call mergesort and do caller saving
```

```
        addi sp,sp,-4
        sw      ra,0(sp)
        jal ra,mergesort
        lw      ra,0(sp)
        addi sp,sp,4
############## resave data#################################

        lw          ra,0(sp)

        lw          s0,4(sp)

        addi        sp,sp,8



# move data to assigned address #

        la          t1,test1
        lw              t2,TEST1_SIZE

        lw              t3,TEST2_SIZE

        lw              t4,TEST3_SIZE

        add         t2,t2,t3

        add         t2,t2,t4                    # t2 = size1 + size2 +size3

        li          t5,0                        # t5 = i

        li          t3,0x01000000               # t3 = answer target

datamove:

        lw          t4,0(t1)

        sw          t4,0(t3)                    #  MOVE data to t3

        addi        t5,t5,1

        addi        t1,t1,4

        addi        t3,t3,4

        blt         t5,t2,datamove          # if i < total size  do loop



        ret
```

```
merge:
#########################################################
    # < Function >
    #       merge
    # < Parameters >

    #       a0 : array address

    #       a1 : start

    #       a2 : mid

    #       a3 : end
#########################################################
    # <  Variable >

    #       t0 : temp_size

    #       t1 : space for temp_size
    array #  t2 : i
    #       t3 : Dump Array Data Section

    #       t4 : left_index

    #       t5 : right_index

    #       t6 : add[i+start]



    #       s0 : left_max

    #       s1 : right_max

    #       s2 : arr_index

    #       s3 : 4*left_index+ temp address
    #       s4 : 4*right_index+ temp
    address #             s5 : 4*s2+address of
    arr
    #       s6 : *(s3)

    #       s7 : *(s4)

    #       s8 : 0x01000000

    #       s9 : addtress of temp

################ callee save ############################
```

```
    addi    sp, sp, -44                 # Allocate stack space
                                            # sp = @sp - 44


    sw      ra, 0(sp)                   # @ra -> MEM[@sp - 4]
    sw      s0, 4(sp)
    sw      s1, 8(sp)

    sw      s2, 12(sp)

    sw      s3, 16(sp)

    sw      s4, 20(sp)

    sw      s5, 24(sp)

    sw      s6, 28(sp)

    sw      s7, 32(sp)

    sw      s8, 36(sp)

    sw      s9, 40(sp)
############################################################
    sub     t0,a3,a1                    # t0=temp_size=end-
    start+1 addi                        t0,t0,1


    slli    t1,t0,2                     # give temp array a space

    addi    s9,sp,-4                    # now s9 =address of
    temp sub                            sp,sp,t1


    li      t2,0                        # i=0

    slli    t6,a1,2                     # t6=4*start

    add     t6,t6,a0                    # t6=address of arr[start]



    addi    sp, sp, -4

    sw      s9, 0(sp)
temp_duplicated_for:
    lw      t3,0(t6)                        # t3=arr[i+start]
```

```
sw          t3,0(s9)                        # temp[i]=t3


addi        t6,t6,4                         # arr[i+START+1]

addi        s9,s9,-4                        # temp[i+1]


addi        t2,t2,1                         # i++

blt         t2,t0,temp_duplicated_for       # if i<temp_size loop


lw          s9, 0(sp)

addi        sp, sp, 4


#       load some variable #

li          t4,0            # left_index=0


sub         t5,a2,a1        # right_index=mid-start+1
addi        t5,t5,1


sub         s0,a2,a1        # left_max = mid-start
sub         s1,a3,a1        # right_max = end-
start mv    s2,a1             # arr_index = start


slli        s3,t4,2

sub         s3,s9,s3        # s3= temp[left_index] address




slli        s4,t5,2

sub         s4,s9,s4        # s4= temp[right_index] address




slli        s5,s2,2         # s5= address of arr[array index]
```

```
        add         s5,s5,a0



        blt         s0,t4,while_loop_A_end  # if left_max < index

        blt         s1,t5,while_loop_A_end # or right_max < index endloop
while_loop_sort_A:
        lw          s6,0(s3)              # s6=*s3
        lw          s7,0(s4)              # s7=*s4

        bge         s7,s6,if_a            # temp[left_index] <= temp[right_index] go if

        jal         x0,else_a




if_a:
        sw          s6,0(s5)              # arr[arr_index] = temp[left_index]

        addi        s2,s2,1               # arr_index++

        addi        t4,t4,1               # left_index++

        addi        s3,s3,-4

        addi        s5,s5,4

        lw          s6,0(s3)



        blt   s0,t4,while_loop_A_end  blt
        s1,t5,while_loop_A_end          jal
        x0,while_loop_sort_A
else_a:

        sw          s7,0(s5)              # arr[arr_index] = temp[right_index]
        addi        s2,s2,1               # arr_index++
        addi        t5,t5,1               # right_index++
        addi        s4,s4,-4
        addi   s5,s5,4

        lw          s7,0(s4)



        blt         s0,t4,while_loop_A_end
```

```
        blt         s1,t5,while_loop_A_end
                    x0,while_loop_sort_A


while_loop_A_end:

        bge          s0,t4,while_loop_B    # left_max >= left_index do loop B
        bge          s1,t5,while_loop_C    # right_max>= right_index do loop C

        jal         x0,merge_end



while_loop_B:

        sw           s6,0(s5)                 # arr[arr_index] = temp[left_index]
        addi        s2,s2,1
        addi        t4,t4,1

        addi        s3,s3,-4

        addi        s5,s5,4

        lw           s6,0(s3)



        bge          s0,t4,while_loop_B

        jal         x0,while_loop_A_end



while_loop_C:

        sw           s7,0(s5)                 # arr[arr_index] = temp[right_index]
        addi        s2,s2,1
        addi        t5,t5,1

        addi        s4,s4,-4

        addi        s5,s5,4

        lw           s7,0(s4)



        bge          s1,t5,while_loop_C

        jal         x0,while_loop_A_end

######## ################ end merge ##################### ############
```

```
######## ################ resave Parameters ##################### ############

merge_end:

        add         sp,sp,t1



        lw          ra, 0(sp)
        lw          s0, 4(sp)
        lw          s1, 8(sp)
        lw          s2, 12(sp)
        lw          s3, 16(sp)
        lw          s4, 20(sp)
        lw          s5, 24(sp)
        lw          s6, 28(sp)
        lw          s7, 32(sp)
        lw          s8, 36(sp)
        lw          s9, 40(sp)
        addi        sp, sp, 44
        ret

mergesort:
#############################################################
        # < Function >
        #       merge
        # < Parameters >

        #       a0 : array address

        #       a1 : start

        #       a3 : end



        #       s1 : start

        #       s2 : mid

        #       s3 : end

################ callee save ############################
```

```
        addi        sp,sp,-12

        sw          s1,0(sp)

        sw          s2,4(sp)

        sw          s3,8(sp)

        addi        sp, sp, -4                  # Allocate stack space

                                                    # sp = @sp - 4



        sw          ra, 0(sp)                    # @ra -> MEM[@sp - 4]
        bge         a1,a3,end
if_b:

        add         s2,a1,a3

        srli        s2,s2,1                     #
        mid=(end+start)/2 mv                        s1,a1
        mv          s3,a3

        ################ caller save ############################

        addi        sp,sp,-4      #  Given storage space

                                  # sp=sp-4

        sw          ra,0(sp)      # ra->mem[@sp-4]
        mv          a3,s2
        jal         ra,mergesort

        lw          ra,0(sp)

        addi        sp,sp,4

        ################ caller save ############################

        addi        s2,s2,1             # mid+1
        mv          a1,s2                 # start(argument)=mid+1(para)
        mv          a3,s3

        addi        sp,sp,-4      #  Given storage space
                                  # sp=sp-4
        sw          ra,0(sp)      # ra->mem[@sp-4]
        jal         ra,mergesort
```

```
    lw          ra,0(sp)
    addi        sp,sp,4
    ################ caller save ############################
    addi        s2,s2,-1            # mid+1-1=mid
    mv          a1,s1
    mv          a2,s2
    mv          a3,s3
    addi        sp,sp,-4    #  Given storage
                            space
                            # sp=sp-4
    sw          ra,0(sp)    # ra->mem[@sp-4]
    jal         ra,merge
    lw          ra,0(sp)
    addi        sp,sp,4


end:
    lw          ra,0(sp)
    addi        sp,sp,4


    lw          s1,0(sp)
    lw          s2,4(sp)
    lw          s3,8(sp)
    addi        sp,sp,12
    ret
```

The above is the assembly code of merge sort. There are three groups of capital measurements:

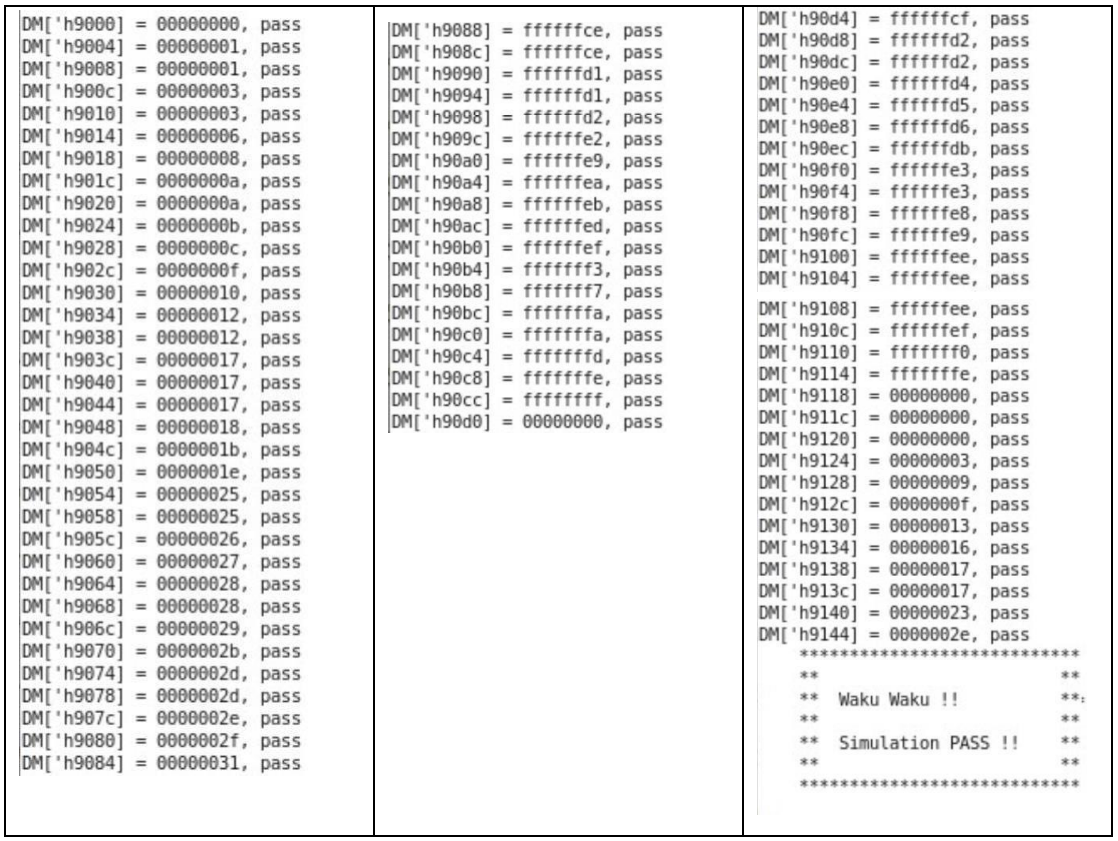
test1: 3,41,18,8,40,6,45,1,18,10,24,46,37,23,43,12,3,37,0,15,11,49,47,27,23,30,16,10,45,39,1,23,40,38

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

Here is the result of the Merge Sort: (denoted in 16 digits).

Test1:	Test2:	Test3:



B. The Fibonacci Sequence

```
.text

_start:

init_stack:

    # set stack pointer

    la sp, _stack

SystemInit:

    # jump to main

    jal main

SystemExit:

    # End simulation

    # Write -1 at _sim_end(0xfffc)

    la t0, _sim_end

    li t1, -1

    sw t1, 0(t0)
```

```
dead_loop:

    # infinite loop
    j dead_loop
.text

.global main



main:

        # TODO

        # remember to exit the program by setting $a0 to 0 and using ecall



        addi      sp, sp, -16                                  # allocate stack space       sp = @sp - 4
        sw              ra, 12(sp)                                    # ra -> @sp - 4
        sw              s0, 8(sp)

        sw              s1, 4(sp)

        li           a5, 2                                       # used by for loop int i = 2
        sw              a5, 0(sp)

        addi      s0, sp, 0

        la           s1, _answer

        sw              zero, 0(s1)                                   # '0' -> @M[0x00000100]

        li           a3, 1                                       # a3 is function argument (a3 = 1)
        addi      s1, s1, 4
        sw              a3, 0(s1)                                     # 1 -> @M[0x00000100+4]

        j loop



loop:

        addi      s1, s1, 4
        lw              a2, -8(s1)                                    # load f(n-2) to a2

        lw              a3, -4(s1)                                    # load f(n-1) to a3

        add             a4, a2, a3                                    # a4 = a2 + a3          => result = a + b

        sw              a4, 0(s1)                                     # store to mem
```

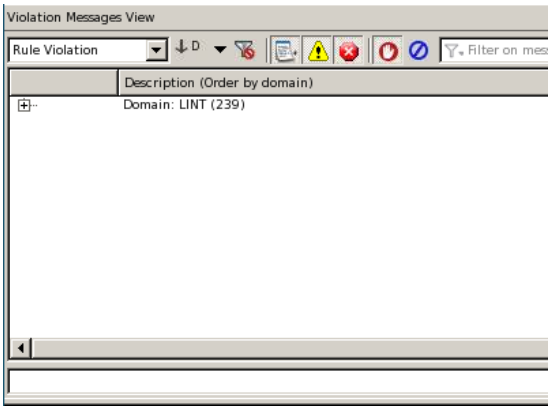
```
    lw          a5, 0(s0)
    addi    a5, a5, 1                       # i = i + 1
    sw          a5, 0(s0)
    li      a6, 20                          # a6 = 21
    bge         a6, a5, loop                # check if a5(i) <= a6(20), if true branch to loop
main_exit:
    lw      ra, 12(sp)
    lw          s0, 8(sp)
    sw          s1, 4(sp)
    lw          a5, 0(sp)
    addi    sp,sp,16
    ret
```

Here are the results of the Fayser series :(expressed inhexadecimal).

```
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 !!    **
    **                        **
    ****************************
```

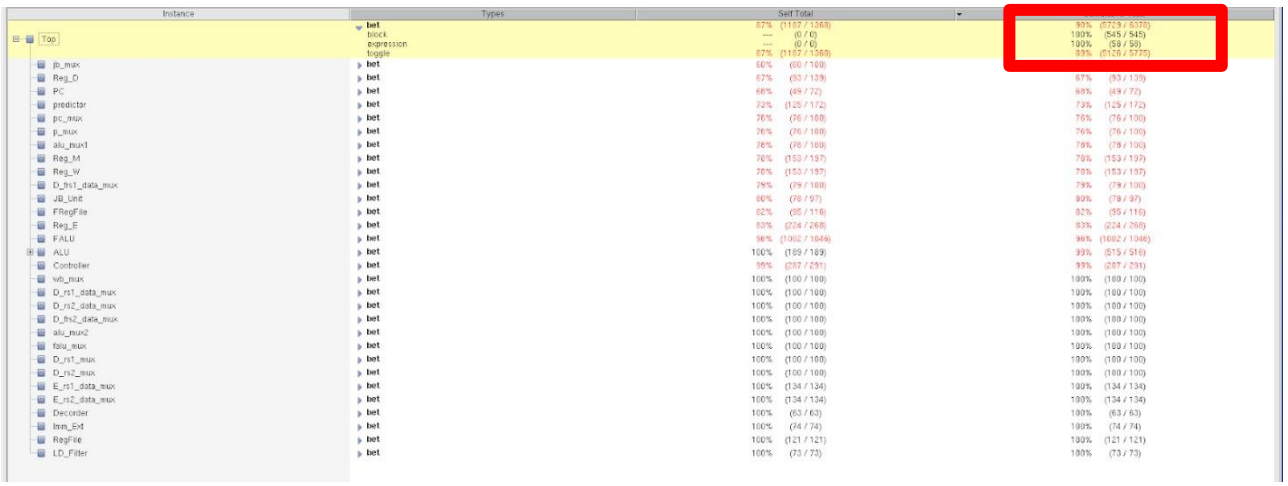
Four. SuperLint and ICC test results

1. SuperLint check result (a screenshot of SuperLint must be attached).




```
INFO (ISL018): Finished building reset tree
INFO (ISL018): Violation Count: Errors = 0 Warnings = 321 Info = 0
321
```

2. ICC test results (ICC screenshots must be attached).



block: 100%

expression:100%

toggle:89%

Five. Synthetic results

1. Speed (both Setup time and Hold time slack are >0)

The CLK of each cylcle is set to 14ns, so the frequency is 7.2*107 Hz

- 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 *****
```

- 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 *****
```

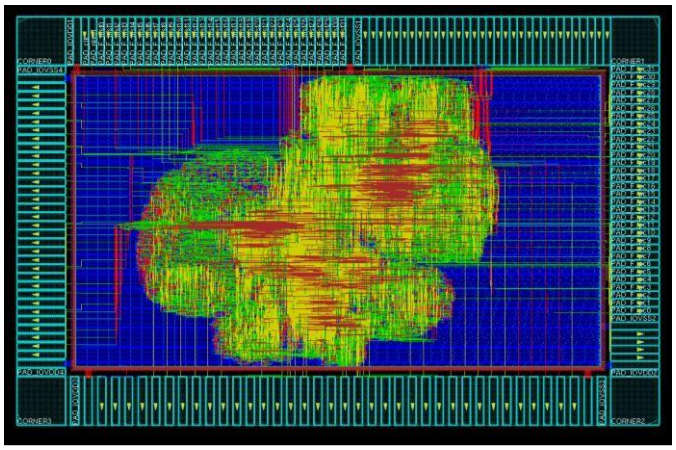
2. Area

```
****************************************
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 *****
```

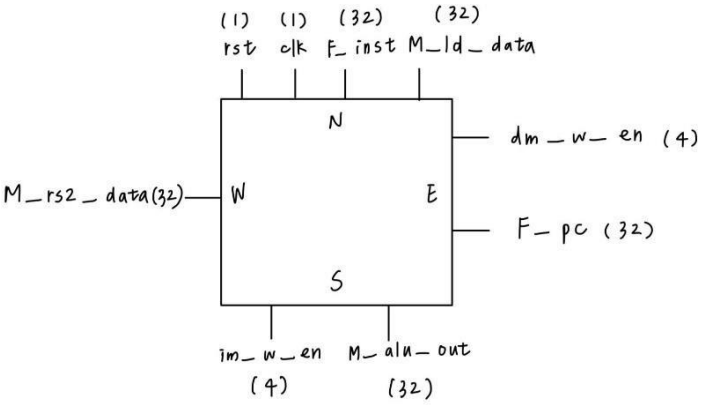
3. Power consumption

```
    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 *****
```

Six. Layout result



Pin Foot Diagram:



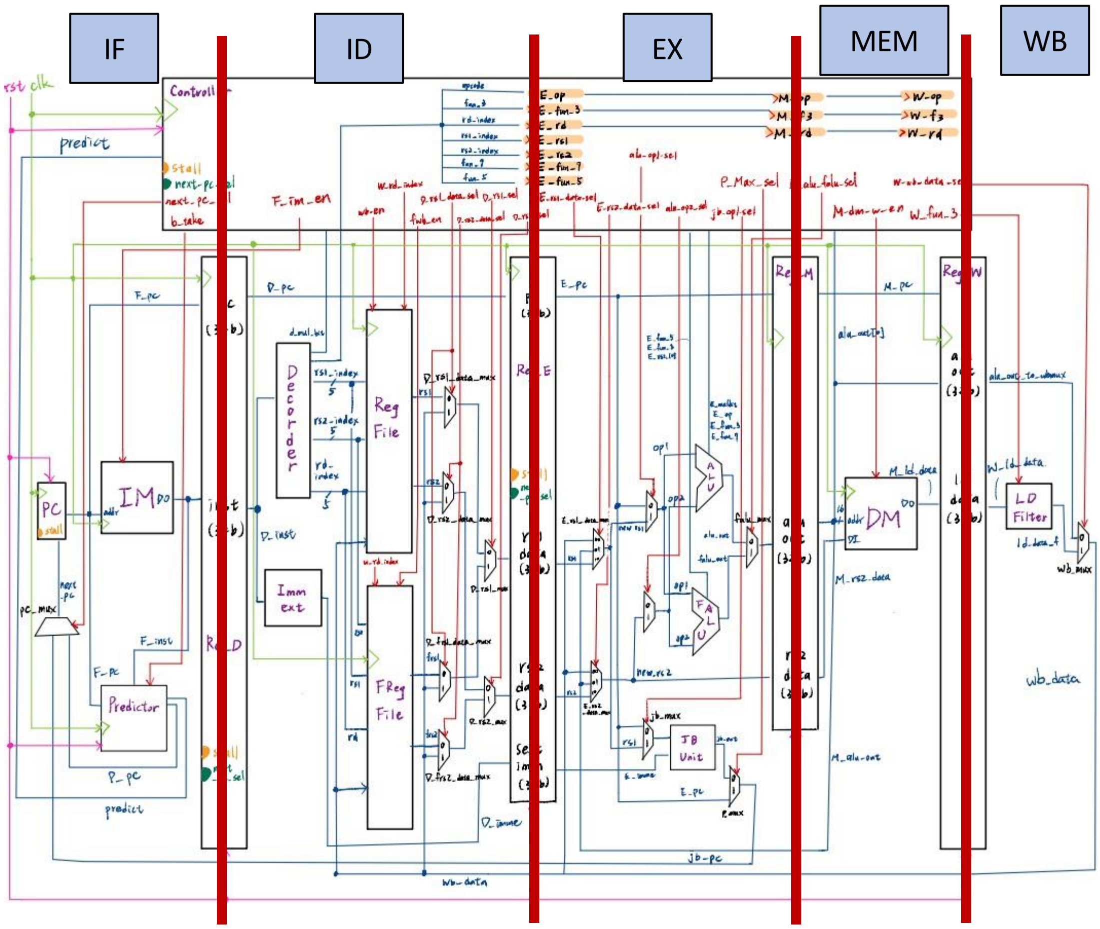
Area:

```
    ------------------------------
    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
```

```
##################################################
##                                              ##
##          C A L I B R E    S Y S T E M        ##
##                                              ##
##              L V S   R E P O R T             ##
##                                              ##
##################################################


REPORT FILE NAME:         lvs.rep
LAYOUT NAME:              CHIP.spi ('chip')
SOURCE NAME:              CHIP.spi ('chip')
RULE FILE:                Calibre-lvs-cur
HCELL FILE:               (-automatch)
CREATION TIME:            Mon Jan  8 10:19:57 2024
CURRENT DIRECTORY:        /project/dr562/pj12/pj1210/Calibre/lvs
USER NAME:                pj1210
CALIBRE VERSION:          v2020.2_14.12    Thu Apr 2 15:39:27 PDT 2020


                          OVERALL COMPARISON RESULTS



                 #         ###################      _   _
                #          #                 #       *   *
           #   #           #     CORRECT     #         |
            # #            #                 #       \___/
             #             ###################
```

Seven. Pipeline

1. Design Description



2. Divisions of phases

IF:instruction fetch。 The PC gets the instructions from the instruction memory and passes them to the next level.

ID: instruction decode. Disassemble the instruction from the previous level and transfer it to the regfile, the data in the scratchpad and the IMM that is extended to 32 bits are transmitted to the next level.

EX: execute. Receive instructions and information from the upper level. The calculation mode is determined according to the instructions transmitted by the upper level, and the data is calculated according to this mode and then transmitted downward.

MEM: memory access. Determine whether to write data to MEM based on the control signal, opcode, and dm_w_en. Or read data from the MEM and send it to the next level.

WB: write back. Upload the data obtained in the previous level back to the regfile.

3. Pipeline hazards & Solutions
   i. Structure hazard: There may be a conflict between reading instructions and writing data, and it is not possible to obtain the address and store the data at the same time
      Solution: Divide the memory into IM and DM, and store instructions and data respectively.
   ii. Control hazard: jump command is not known to be able to jump successfully until it has been calculated. Solution: Use the branch predictor, if jump is needed, command will be flushed.
   iii. Data hazard: Data is needed before the calculated result is stored
      How to resolve: forward Data back EX stage from MEM and WB stage.

Eight. Special design

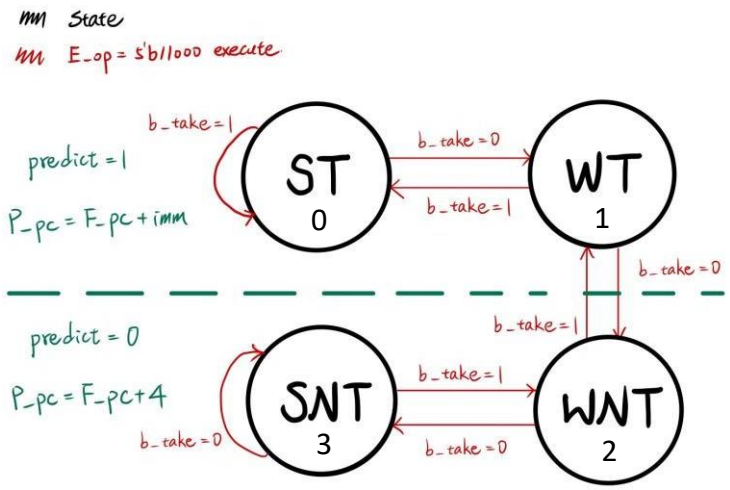
A. Branch predictor

- Design Concept :

The one level bimodal predictor has four states: Strongly not taken (SNT), Weakly not taken (WNT), Weakly taken (WT) and Strongly taken (ST). When the status is taken, the predictor will predict the jump of the order, and the next command will execute the command of the jump address; On the other hand, if the status is not taken, it is predicted that the current PC+4 command will be executed without jumping.

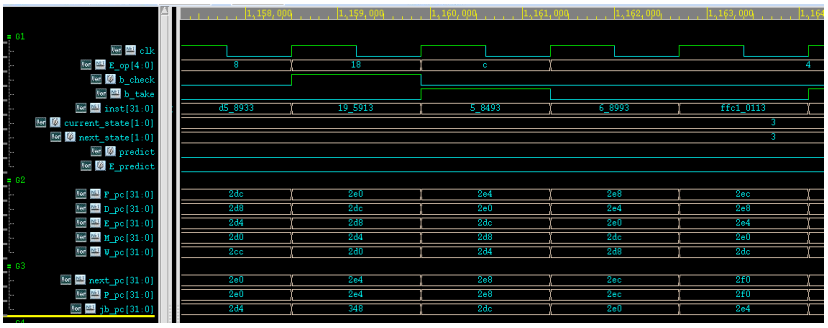
- Status Execution:

The current state in the program is 2(WNT), when the branch command does not need to be redirected, the next state will be +1 passed to the current state, so the state is changed to 3(SNT). Conversely, when the branch command needs to be redirected, the next state will be passed to the current state, so the state will be changed to 1(WT). When the state is 0(ST), do not continue -1; Do not continue with +1 when the status is 3(SNT).



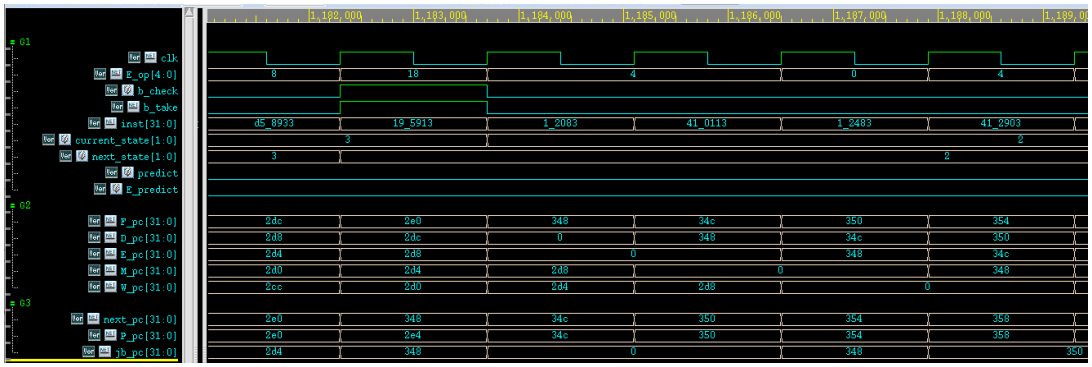
- Analysis of simulation results:
  - **SNT -> SNT:**

    When Pc = 2d8 is a branch instruction, b_check = 1, the jump condition is not met b_taken=0 does not need to be redirected, the predictor prediction is correct, the SNT state remains unchanged, and the predict does not change.



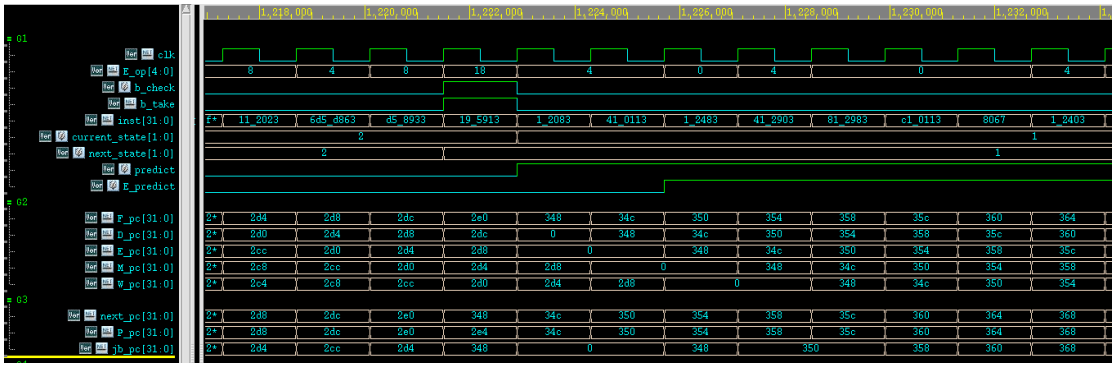
  - **SNT -> WNT:**

    When Pc = 2d8 is a branch instruction so b_check = 1, the jump condition is met b_taken=1 needs to be redirected, and the status changes from SNT to WNT and predictor prediction error, so Flush is performed on subsequent instructions, and predict remains unchanged.



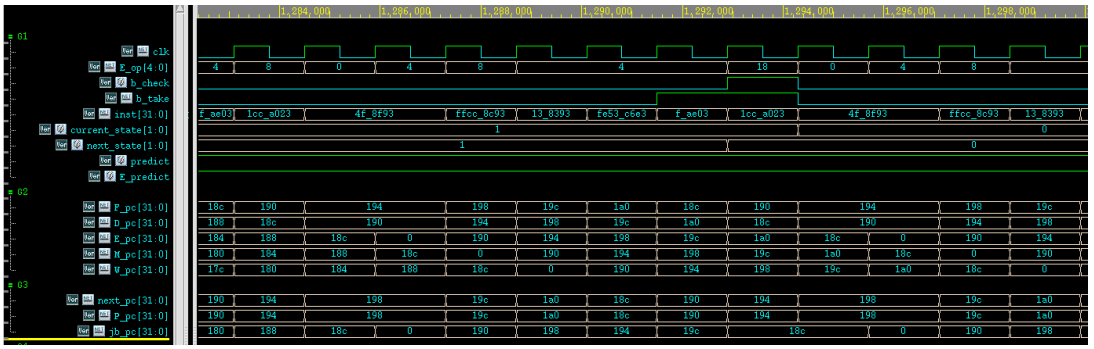
■ **WNT -> WT:**

When Pc = 2d8 is a branch instruction, b_check = 1, and the jump condition is met b_taken=1 needs to be redirected, and the state changes from WNT to WT and predictor is incorrect, so flush is performed on subsequent instructions, and predict changes from 0 to 1。



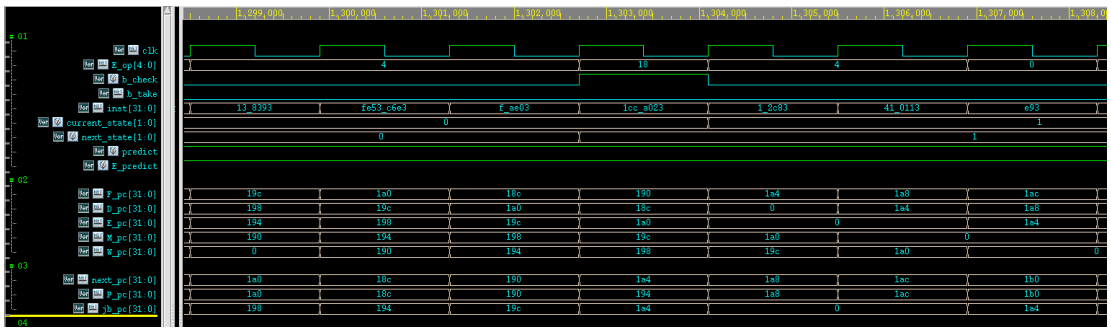
■ **WT -> ST:**

When Pc = 1a0 is a branch instruction so b_check = 1, the jump condition is met b_taken = 1 needs to be jumped, the status changes from WT to ST, and the predictor prediction is correct, so there is no need to flush, and the predict remains unchanged.



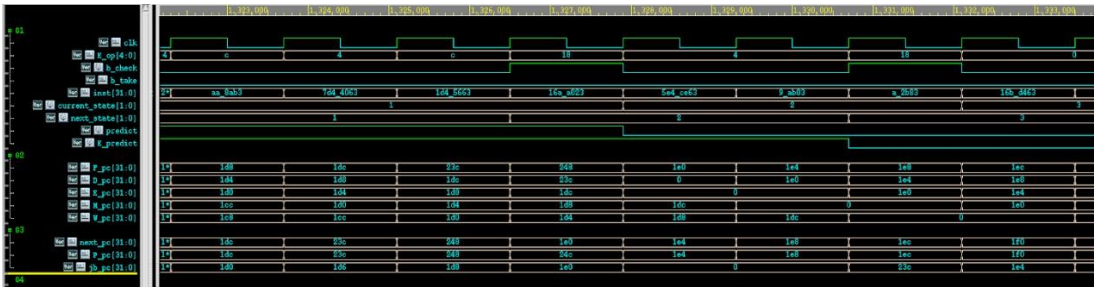
■ **ST -> WT:**

When Pc = 1a0, it is a branch instruction, so b_check = 1, and the jump condition is not met b_taken=0 does not need to be jumped, and the status changes from ST to WT, and the predictor prediction is wrong, so it is necessary to flush the subsequent instructions, and the predict remains unchanged.



- **WT -> WNT:**

When Pc = 1dc, it is a branch instruction, so b_check = 1, and the jump condition is not met b_taken=0 does not need to be jumped, and the state changes from WT to WNT, and the predictor prediction is wrong, so it is necessary to flush the subsequent instructions, and predict changes from 1 to 0。



- **WNT -> SNT:**

When Pc = 1e0, it is a branch instruction, so b_check = 1, the jump condition is not met b_taken=0 does not need to be jumped, the state changes from WNT to SNT, and the predictor prediction is correct, so there is no need to flush, and the predictor remains unchanged.

B. Floating point number Arithmetic Logic Unit

- Currently supported commands:

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

directives	Function description (rs1, rs2, and rd refer to the registers in FREG unless otherwise specified).
FLW	After the mem address + imm [11:0] stored in the rs1 of the REG, the value taken by the MEM is stored in the rd of the FREG.
FSW	After placing the mem address + imm [11:0] stored in the rs1 of the REG, according to the address The FREG data is stored in MEM.
FADD. S	Add the values in rs1 and rs2 of FREG and store the results in rd.
FSUB. S	Subtract the values in rs1 and rs2 of FREG and store the results in rd.
FMUL. S	Multiply the values in rs1 and rs2 of FREG and store the result in rd.
FMIN. S	Compare the rs1 of FREG with the value in rs2 and store the smaller value in rd.
FMAX. S	Compare the rs1 value of FREG with the value in rs2 and store the larger value in rd.
FCVT. W.S	The rs1 value of FREG is rounded to an integer and stored in the rd of REG.
FCVT. WU. S	The absolute value of rs1 of FREG is rounded to an integer and stored in the rd of REG.
FMV. X.W	The rs1 value of FREG is directly passed into the rd of the REG without any conversion or change.
FEQ. S	Determine whether the rs1 and rs2 values of FREG are equal, and enter the result into rd (equal: 1, not equal: 0).
FLT. S	Check whether the rs1 of FREG is less than the value of rs2, and if it is equal, enter the result into rd (less than: 1, greater than or equal to: 0).
FLE. S	Check whether the rs1 of FREG is less than or equal to rs2, and enter the result into rd (less than or equal to 1, greater than: 0).
FCVT. S.W	The rs1 value of the REG is converted to a floating-point number representation and stored in the rd of the FREG.
FCVT. S.WU	The absolute value of rs1 of the REG is converted to a floating-point number representation and stored in the rd of the FREG.
FMV. W.X	The rs1 value of the REG is directly passed into the rd of the FREG without any conversion or change.

- Floating point set:

Use the IEEE 754 specification:

1. operand[31] is the signed bit, operand[30:23] is exponential (8 bits), operand[22:0] is mantissa (base).。
2. The exponential part needs to be able to directly see the size of the exponent power, and there needs to be a negative part to represent the decimal number, so the value is offset by 127, with 127 as the power of 0, greater than 127 is a positive exponent, less than 127 is a negative exponent, indicating the range of -126~127.
3. Mantissa (base) specifies it as the first 1 shift to the first decimal place and the number after the decimal point.
4. Rounding rules: using round to nearest, ties away from zero, will round the decimal to the direction away from 0.
5. According to the RISC-V instruction set, 32 FREG(f0~f31) are added as the scratchpads for storing floating-point numbers

- How it works:
  - FADD. S、FSUB. S:
    i. take operand1[30:0] and operand2[30:0] as their absolute values ABS_1 ABS_2。
    ii. Compare the ABS and calculate the difference between the exponential of the two numbers in the form of a large decrease, and use lar to record which is greater ABS_1 (lar=0) and ABS_2 (lar=1).
    iii. Define true sign: 1 when different signs are added or picked up with the same sign, and the 1 that was previously omitted by mantissa is added back, and 0 is added after it to form the correct base (add_man).
    iv. Based on the lar results of the previous comparison, the correct base of the smaller number of ABS
    (add_man) Shift to the right and move to the same base as the other number.
    v. By judging the true sign, the 2 bases that have been moved to the same base are added and subtracted, and the sign bit (plus or minus) is determined.
    vi. Normalization: Using a determinant formula, record which bit of the addition and subtraction result is the first 1 (used to calculate the correct exponential) and how many bits the record needs to shift to produce the correct base.
    vii. Shift the addition and subtraction results, and subtract them from the previous normalization results to form the correct exponential and mantissa (bases).
  - FMUL. S
    i. Calculate the exponential after multiplication (add and subtract the exponential of two numbers
    127, the reason is that the exponential power greater than 127 is positive).

ii. After restoring the hidden 1 of the base, multiply it directly.

iii. Since there is a reduction of 1 to the base (23rd bit), there must be one of the 47th or 46th bits after multiplication as 1. Using the judgment formula, the multiplied exponential and mantissa are processed according to the result to form the correct exponential (exponential) and mantissa (base).

- FMIN. S、FMAX. S

i. According to the sign of two numbers bit(Plus or minus), if it is the same number, its absolute value is compared (ABS), and then according to func3(=1 is max,=0 for min), the output is correct min/max value.

- FEQ. S、FLT. S、FLE. S

i. Compare the sign of two numbers bit(Plus or minus), and absolute values(ABS), judging it to be equal to(=)Less than(<), less than or equal to(<=)Whether the situation is established.

- FCVT. W.S、FCVT. WU. S

i. To determine whether the index is positive or negative, subtract it from the R1 record (=0 is positive, =1 is negative).
127, the absolute value of the formation of the correct exponent.

ii. The base after the reduction of 1 is shifted to the absolute value of the exponent to form the correct value.

iii. Determine whether it is an integer, if it is an integer, the original value will be output, and the decimal will be unconditionally discarded and then added to 1 to form the result of unconditional base.

iv. Judgment func1 (=0 is signed, =1 is unsigned), unsigned is directly output, if signed, use the judgment formula to make it a negative number when the complement is used to form the correct output.

- FCVT. S.W、FCVT. S.WU

i. Using the calculation of xor and sign bit, if operand1 is positive, the original number is accessed, and operand2 is negative, the result of the two complements is accessed.

ii. Normalization: Calculate which bit its highest value of 1 appears in, so that you can see how much its index differs from 127.

iii. Shift the original number according to normalization to form the correct base, and add 127 to the difference between the exponent and 127 to form the correct indication representation.

iv. According to func1 (=0 is signed, =1 is unsigned), the correct sign bit (plus or minus signs) is calculated to form the correct floating point representation.

- Simulation Results:
  - **flw f1, 0(s0)**

In the code, PC=38 and PC44 are FLW instructions, so the result of ALU calculation is the address of memery, and when the W_pc is 38, the wb_data is 3fa0_0000 (1.25), and when the W_pc is 44, the wb_data is 3e00_0000 (0.125).。

- **fadd.s f1, f1, f2**

In the code, PC=48 and PC4C are two consecutive FADD instructions, so the result of FALU calculation is operand1(3fa0_0000:1.25) and operand2(3e00_0000:0.125) for addition. The result of the first result is directly forwarded to operand1 of the next fadd, becoming operand1(3fb0_0000:1.375) and operand2(3e00_0000:0.125) Do the addition and the result is 3fc0_0000:0.125.

- **fadd.s f1, f1, f3**

In the code, PC=5C~70 is six consecutive FADD instructions, and the results of FALU calculation are (3fe0_0000:1.5) and (bf80_0000:-1) and (3e00_0000:0.125), the result of the previous result is directly forwarded to operand1 of the next fadd, and the result is c010_0000:-2.25.

- **fsw f1, 0(s0)**

In the code, PC=74 is the FSW instruction, and the result of ALU calculation is the address of the MEM.

- **fsub.s f1, f1, f2**

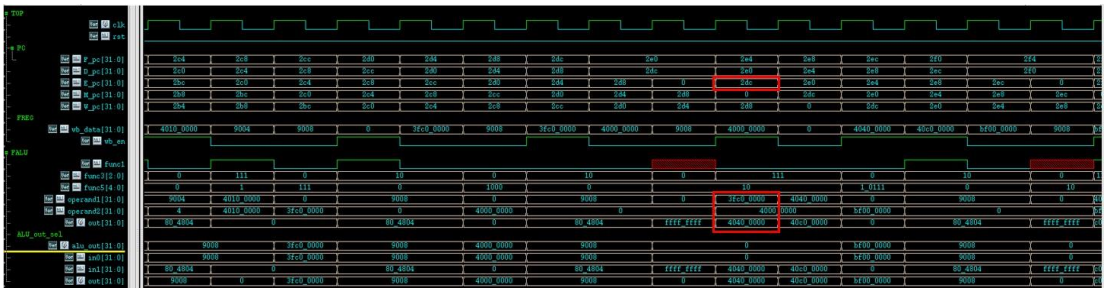
In the code, PC=28C~298 is four consecutive FSUB instructions, and the result of FALU calculation is (3fa0_0000:1.25) and (3f40_0000:0.75) to do subtraction. The results of the previous result are directly forwarded to operand1 of the next fsub, and the results are (3f00_0000: 0.5), (be80_0000: -0.25), (bf80_0000: -1.0), and respectively (bfe0_0000: -1.75)。

- **fsub.s f1, f1, f3**

In the code, PC=2A8~2B4 is four consecutive FSUB instructions, and the result of FALU calculation is (bfe0_0000:1.25) and (bf80_0000:-1) to do subtraction. The results of the previous result are directly forwarded to operand1 of the next fsub, and the results are (3f00_0000: -0.75), (be80_0000: 0.25), (bf80_0000: 1.25), and respectively

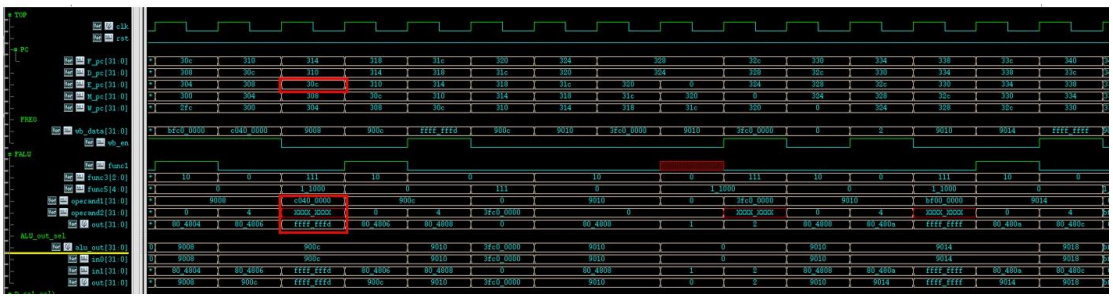
(bfe0_0000: 2.25)。

- **fmul.s f1, f1, f2**



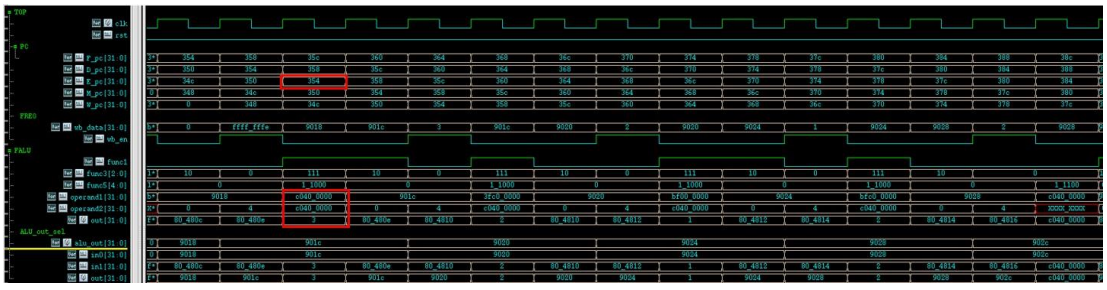
In the code, pc=2dc is the fmul instruction, so the falu output is operand1(3fc0_0000:1.5)*operand2(4000_0000:2)=(4040_0000:3)。

- **fcvt.w.s t0, f1**



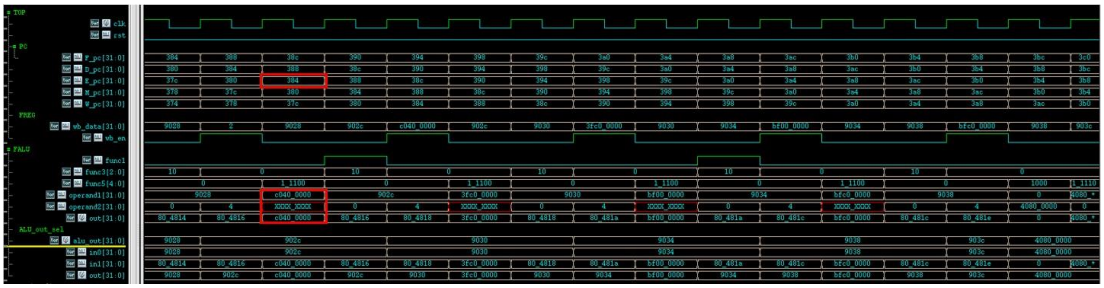
In the code, pc=30c is the fcvt.w.s instruction, so the falu output is will operand1(c040_0000:-3.0) is converted to an integer representation (ffff_fffd:-3).

- **fcvt.wu.s t0, f1**



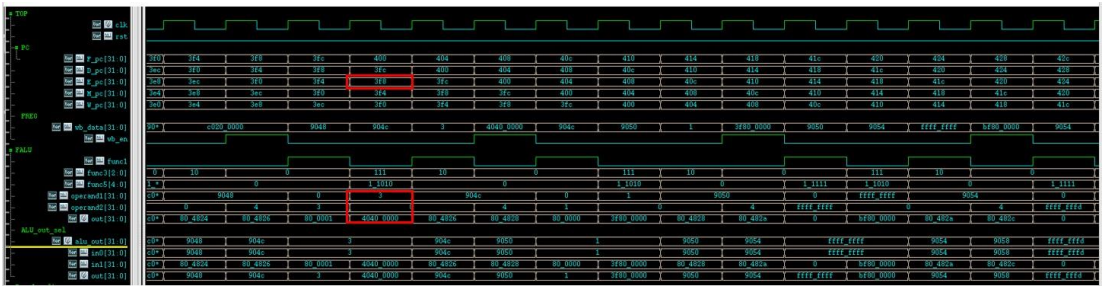
In the code, pc=354 is the fcvt.w.s instruction, so the FALU output is will operand1(c040_0000:-3.0) is converted to an integer representing 3.

- **fmv.x.w t0, f1**



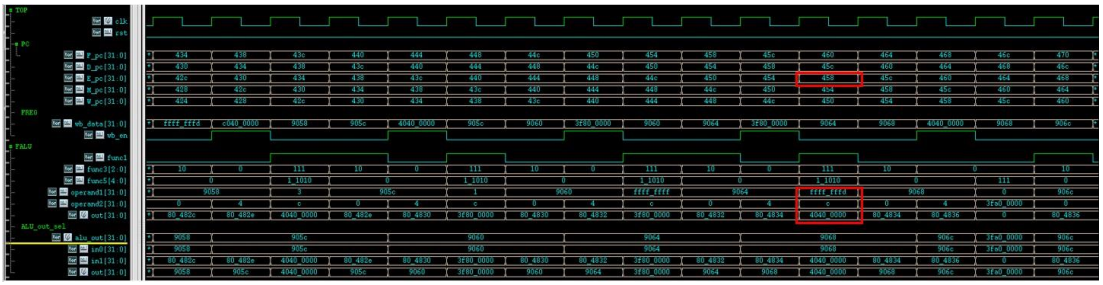
In the code, pc=384 is the fmv.x.w instruction, so the falu output is will operand1 (c040_0000: -3.0) incoming rs1 intact.

- **fcvt.s.w f1, t0**



In the code, pc=3f8 is the fcvt.s.w instruction, so the FALU output is to convert operand1=3 to a floating-point representation (4040_0000:3.0).

- **fcvt.s.wu f4,t3**

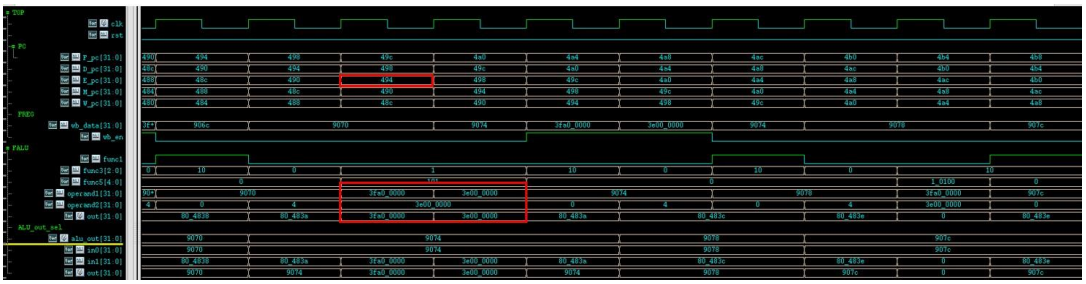


In the code, pc=458 is the fcvt.s.wu instruction, so the falu output is will operand1=-3 takes the absolute value and converts it into a floating-point number representation (4040_0000:3.0).

- **fmin.s f4, f1,f2**

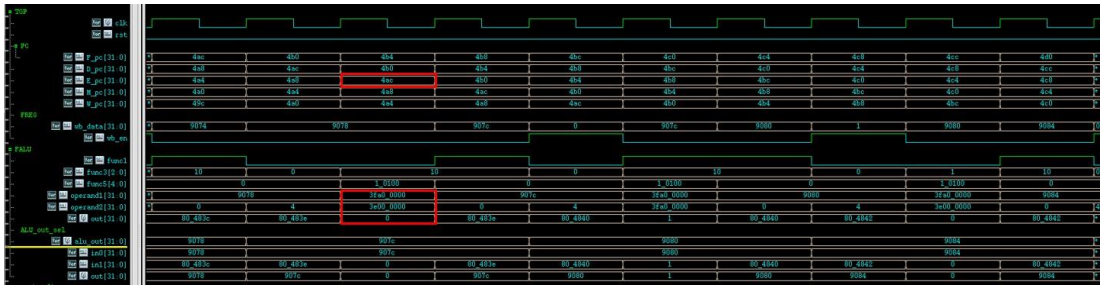
In the code, pc=47c is the fmin.s instruction, so the falu output is to compare operand1(3fa0_0000: 1.25) and operand2(3e00_0000: 0.125) to the smaller value.

- **fmax.s f4, f1, f2**



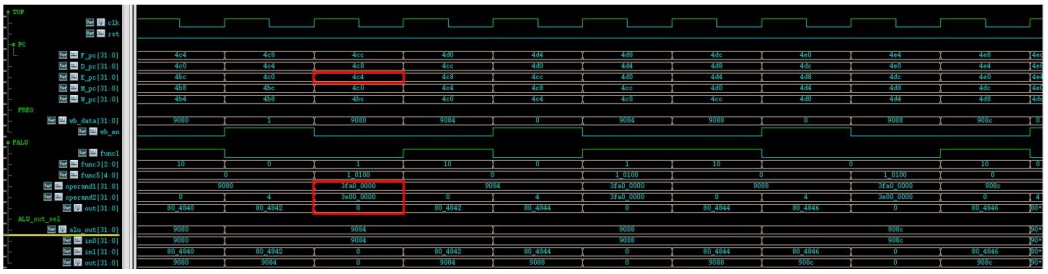
In the code, pc=494 is the fmax.s instruction, so the falu output is to compare operand1(3fa0_0000: 1.25) and operand2(3e00_0000: 0.125) to the larger value.

- **feq.s t0, f1, f2**



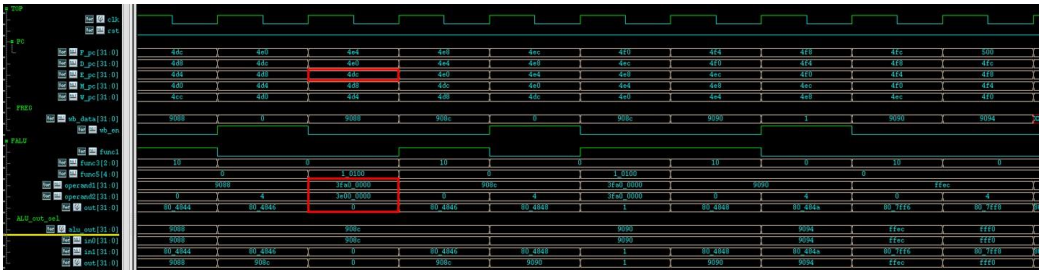
In the code, PC=4AC is the FEQ.S instruction, so the FALU output is to compare operand1(3fa0_0000: 1.25) and operand2(3e00_0000: 0.125) to output 0 (unequal).

- **flt.s t0, f1, f2**



In the code, PC=4AC is the FEQ.S instruction, so the FALU output is to compare operand1(3fa0_0000: 1.25) and operand2(3e00_0000: 0.125) to see if they are less than 0 (greater than or equal to).

- **fle.s t0, f1, f2**



In the code, PC=4AC is the FLE.S instruction, so the FALU output is to compare operand1(3fa0_0000: 1.25) and operand2(3e00_0000: 0.125) to output 0 (greater than).

A. Fast multiplier

- Booth algorithm
- Calculation principle

i. The core concept of the main operation is to find the two bits (01 and 10) in the multiplier that enter and leave consecutive 1,and treat 01 as +1,10 Considered -1, plus or minus the multiplier that has been displaced, if the adjacent bit is 00 and 11, it is in a continuous 1 or continuous 0, in this case, only the displacement is required, and the multiplier does not need to be added or subtracted. This process optimizes the traditional multiplication calculation, which requires the multiplier to be continuously shifted and added, so it is especially used when the multiplier has multiple consecutive 1s.

ii. The 0th bit of the multiplier have to be added to 0 before the calculation begins, to ensure that each multiplier has a right bit that can be compared to it

iii. Shifting one place to the left after each operation to determine the status of the multiplier is equivalent to excluding the processed bits from subsequent calculations. This means dividing the multiplier by 2 after each operation, gradually reducing the number of multipliers that need to be processed.

- Calculation steps

i. Add a 0 to the right of the 0th bit of the multiplier

ii. Make a 1 bit signed-extension with the highest digit of the multiplier, convert the result into a complement, and then save the original result and the result after the complement to the two registers (mulshift, mulshift_C) with two bits twice the number of the multipliedand declare that a portion of the product is used to store the multiplication answers

iii. Starting with the multiplier after 0, check each pair of adjacent digits, which may be 00, 01, 10, 11

iv. Take the following actions depending on the adjacent bits

(After all of the following operations, you must move Mulshift and mulshift_C one place to the left).

1. 00, 11: Do not perform addition actions
2. 01: Add mulshift to the partial product
3. 10: Add the partial product to mulshift_C

v. After checking each pair of adjacent digits, the final partial product is the result of multiplying the two numbers

- Example: Take 00010011 (8 bits multiplier) and 01001111 (8 bits multiplier) as examples

i. First, the multiplier is followed by 0 to form 010011110

ii. Then the multiplier is calculated by 2 captures (11101101), both of which are 1 bit signed extension, forming 000010011 and 111101101

iii. The multiplier after processing will be judged from the 0th bit to determine the value of two adjacent bits

The multiplier will be processed accordingly.

01001110 -> 10 -> -1

Partial volume: 111101101

000010011 -> 0000100110

111101101 -> 1111011010

01001111 -> 11

Partial product: No action

0000100110 -> 00001001100

1111011010 -> 11110110100

001001111 -> 11

Partial product: No action

00001001100 -> 000010011000

11110110100 -> 111101101000

00100 1111 -> 11

Partial product: No action

000010011000 -> 0000100110000

111101101000 -> 1111011010000

00101 1111 -> 01 ->+1

Partial product: 111101101 + 0000100110000 = 10000100011101

0000100110000 -> 00001001100000

1111011010000 -> 11110110100000

00100 1111 -> 00

Partial product: No action

00001001100000 -> 000010011000000

11110110100000 -> 111101101000000

00100 1111 -> 10 -> -1

Partial product: 10000100011101 + 111101101000000 = 1001110001011101

000010011000000 -> 0000100110000000

111101101000000 -> 1111011010000000

00 1001111 -> 01 -> +1

Partial product:

1001110001011101 + 0000100110000000 = 1010010111011101

0000100110000000 -> 00001001100000000

1111011010000000 -> 11110110100000000

001001111 -> 00

Partial product: 1010010111011101 (inactive).

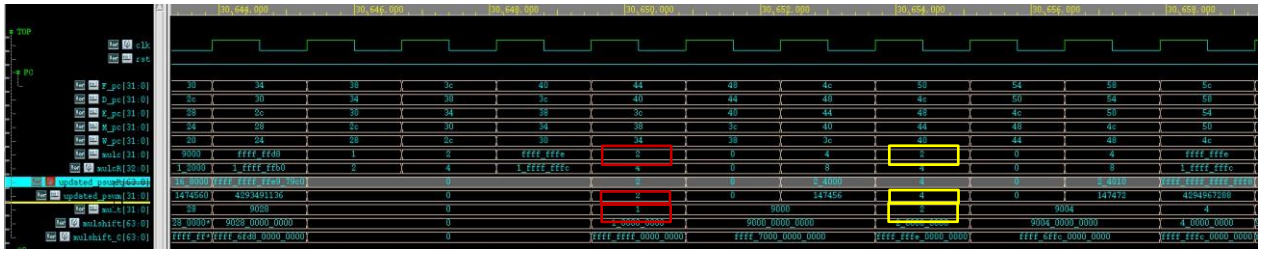
00001001100000000 -> 0000100110000000000

11110110100000000 -> 1111011010000000000

iv. You get 00010011*01001111=1010010111011101 (correct answer).

v. Take the first 8 bits of the final product and store it in the register representing the product. That is: 11011101

- The multiplier design uses RISC-V instructions: mul rd, rs1, rs2, and will eventually access the result before 32 bits.
- Waveform interpretation
  - **3c: mul t0, t0(1), t1(2)**
  - **48: mul t0, t0(2), t1(2)**

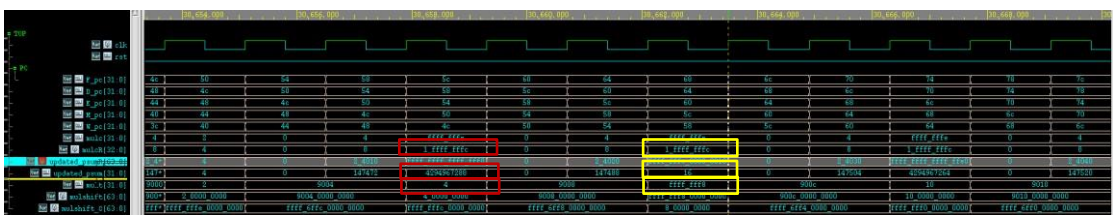


Set the multiplier (mult) and multiplier (mulc), and multiply the two by booth algorithm, the result will be stored in the updated_psumR, and the first 32 bits will be taken to the updated_psum. In the waveform diagram, mulcR represents the 0 bit of the multiplier at the beginning of the multiplier by 0 (which is used to judge and manipulate two adjacent bits). Mulshift and mulshift_c represent the result of multiplier shifting that would be performed in each of the adjacent bits.

The red box shows 1(mult) * 2(mulc) = 2(updated_psum).

The yellow box shows 2(mult) * 2(mulc) = 4(updated_psum).

  - **54: mul t0, t0(4), t1(-2)**
  - **60: mul t0, t0(-8), t1(-2)**



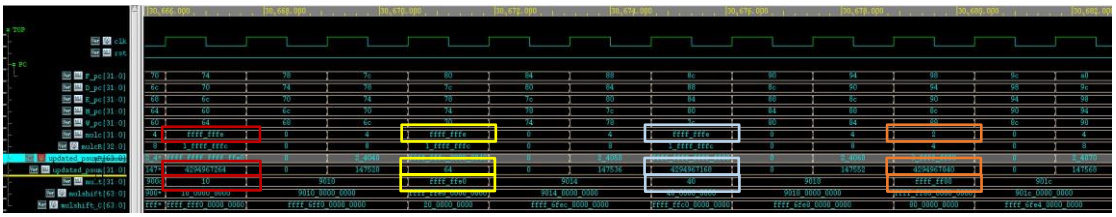
Set the multiplier (mult) and multiplier (mulc) and pass the two through the booth algorithm

The result will be stored in the updated_psumR and the first 32 bits will be taken to updated_psum. In the waveform diagram, mulcR represents the 0 bit of the multiplier at the beginning of the multiplier by 0 (which is used to judge and manipulate two adjacent bits). Mulshift and mulshift_c represent the result of multiplier shifting that would be performed in each of the two adjacent bits.

The red box shows 4 (mult) * (-2) (mulc) = (-8) (updated_psum).

The yellow box shows (-8)(mult) * (-2)(mulc) = 16(updated_psum).

- **6c: mul t0, t0(16), t1(-2)**
- **78: mul t0, t0(-32), t1(-2)**
- **84: mul t0, t0(64), t1(-2)**
- **90: mul t0, t0(-128), t1(2)**



Set the multiplier (mult) and multiplier (mulc), and multiply the two by booth algorithm, the result will be stored in the updated_psumR, and the first 32 bits will be taken to the updated_psum. In the waveform diagram, mulcR represents the 0 bit of the multiplier at the beginning of the multiplier by 0 (which is used to judge and manipulate two adjacent bits). Mulshift and mulshift_c represent the result of multiplier shifting that would be performed in each of the adjacent bits.

The red box shows 16(mult) * (-2)(mulc) = (-32)(updated_psum). The yellow box shows (-32)(mult) * (-2)(mulc) = 64(updated_psum). The light blue box shows 64 (mult) * (-2)(mulc) = (-128)(updated_psum)

The orange box shows (-128) (mult) * 2(mulc) = (-256)(updated_psum)。

- **9c: mul t0, t0(-256), t1(2)**
- **a8: mul t0, t0(-32), t1(-2)**

Set the multiplier (mult) and multiplier (mulc), and multiply the two by booth algorithm, the result will be stored in the updated_psumR, and the first 32 bits will be taken to the updated_psum. In the waveform diagram, mulcR represents the multiplier at the beginning of the 0 bit to make up 0 (use

to judge and operate two adjacent positions). Mulshift and mulshift_c represent the result of multiplier shifting that would be performed on each of the two adjacent bits.
The red box shows (-256)(mult) * 2(mulc) = (-512)(updated_psum).
The yellow box shows (-512)(mult) * 0(mulc) = 0(updated_psum)

- **Resources**

1. VLSI System Design Course Handouts
2. Computer organization course handouts
3. RISC-V instruction set manual