

VLSI System Design

Yu-Chi Chu

1. Introduction

1. Instruction Set Format

For this project, I have chosen the RISC-V instruction set architecture (ISA) as the foundation. We have implemented a five-stage pipelined CPU based on this ISA. To ensure correct execution and performance, mechanisms such as forwarding (or bypassing) and hazard detection and resolution (including data hazards, control hazards, and structural hazards, if applicable) are incorporated into the design to facilitate the operation of the complete CPU.

2. Instruction Set Format Field Names, Lengths, and Descriptions

The RISC-V instruction set architecture (ISA) uses 32-bit binary instructions. These instructions are classified into six fundamental formats: R-type, I-type, S-type, SB-type (or B-type), U-type, and UJ-type (or J-type). A detailed explanation of each format is provided as follows:

R-type

7	5	5	3	5	7
funct7	rs2	rs1	funct3	rd	opcode

rs1: 1st register operand (register source) (5 bits)

rs2: 2nd register operand (5 bits)

rd: register destination (5 bits)

opcode: specifies operation(7 bits)

funct7+funct3: combined with opcode (10 bits)

R-type instructions consist of five fields: opcode, rs1, rs2, rd, and funct (which is further divided into funct3 and funct7), forming a 32-bit instruction code. The detailed composition is shown in the table above. The opcode for R-type instructions is 0110011 (binary). The specific operation to be performed is determined by the funct fields (funct3 and funct7). The operation is then performed on the values in registers rs1 and rs2, and the result is stored in register rd.

I-type

12		5	3	5	7
Imm[11:0]		rs1	funct3	rd	opcode
Imm[11:5]	shamt	rs1	funct3	rd	opcode

immediate: 12 bits number (12 bits)

shamt: shift amount (5 bits)

I-type instructions are composed of five main parts: opcode, rd, funct3, rs1, and imm (immediate), forming a 32-bit instruction code. While the term shamt (shift amount) is often associated with shift operations within I-type instructions, it's not a separate, dedicated field. Instead, shamt is encoded within the imm field and is only relevant for shift instructions (e.g., slli, srli, srai). Therefore, the core components of an I-type instruction are opcode, rd, funct3, rs1, and imm. The detailed composition is shown in the table below. The opcode for I-type instructions has two primary values: 0010011 (for arithmetic, logical, and shift operations) and 0000011 (for load instructions). The specific operation to be performed is determined by the funct3 field. For arithmetic/logical operations, the operation is performed on rs1 and imm, and the result is stored in rd. For load operations, the content of the memory location addressed by $rs1 + imm$ is loaded into rd.

S-type

7	5	5	3	5	7
Imm[11:5]	rs2	rs1	funct3	Imm[4:0]	opcode

S-type instructions consist of five main parts: opcode, rs1, rs2, imm (immediate), and funct3 (function code 3), forming a 32-bit instruction code. The opcode for S-type instructions is fixed at 0100011 (binary). The specific store operation to be performed is determined by the funct3 field. S-type instructions are used to store the value from register rs2 into memory at the address calculated by $rs1 + imm$. The notation $rs1[imm]$ means "the memory location pointed to by the value in rs1 as the base address, plus the offset imm."

U-type

20	5	7
Imm[31:12]	rd	opcode

U-type instructions consist of three parts: opcode, rd (destination register), and imm (immediate), forming a 32-bit instruction code. The U-type opcode has two main values: 0110111 (for the lui instruction) and 0010111 (for the auipc instruction).

- **lui (Load Upper Immediate):** Loads the 20-bit value of imm into the upper 20 bits (bits 31:12) of the rd register, filling the lower 12 bits (bits 11:0) with zeros.
- **auipc (Add Upper Immediate to PC):** Treats the 20-bit value of imm as a signed value, left-shifts it by 12 bits, adds it to the current Program Counter (PC) value, and stores the result in the rd

3. Branch, Jump:

SB-type

7	5	5	3	5	7
Imm[12 10:5]	rs2	rs1	funct3	Imm[4:1 11]	opcode

SB-type instructions consist of five parts: opcode, rs1, rs2, imm (immediate), and funct3 (function code 3), forming a 32-bit instruction code. The opcode for SB-type instructions is fixed at 1100011 (binary). The specific conditional branch instruction to execute is determined by the funct3 field. SB-type instructions compare the values in registers rs1 and rs2. If the comparison is true, the Program Counter (PC) is updated to PC + imm; if the comparison is false, the PC is updated to PC + 4 (i.e., the next sequential instruction is executed). The order of comparison is important: rs1 is compared against rs2.

UJ-type

20	5	7
Imm[20 10:1 11 19:12]	rd	1101111
12	5	3
Imm[11:0]	rs1	funct3
	rd	1100111

UJ-type instructions consist of *three* parts: opcode, rd (destination register), and imm (immediate), forming a 32-bit instruction code. rs1 and funct3 are *not* part of the UJ-type instruction encoding. They are used only during the

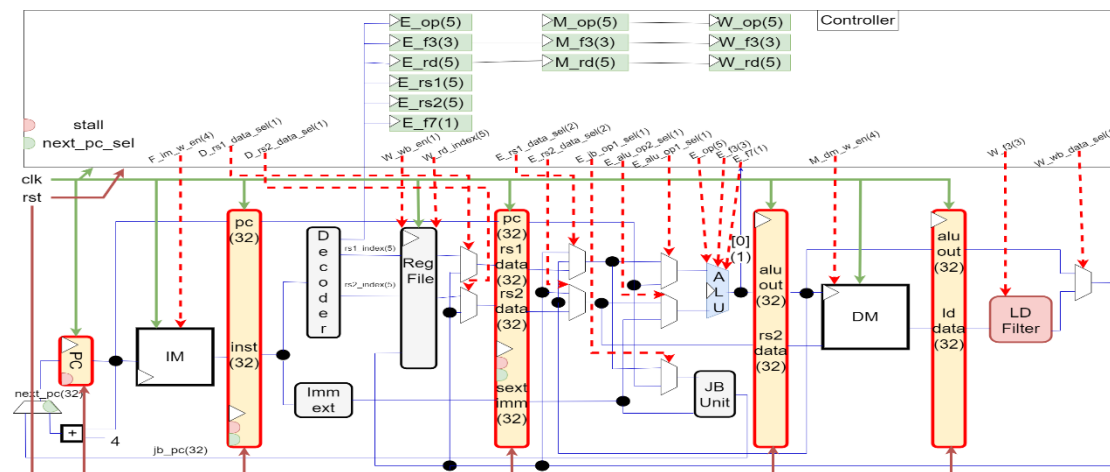
execution stage of the jalr instruction. In fact, the jalr instruction uses the I-type format, not the UJ-type format.

The UJ-type opcode has one main value: 1101111 (for the jal instruction). The opcode 1100111 is used for jalr instruction, but it is I-type format.

4. Architecture

i. Graph and Explain

The initial architectural prototype comes from the non-pipeline CPU, which consists of memory, decoder, register file (RegFile), ALU (Arithmetic Logic Unit), etc. Subsequently, based on the operation duration of each stage, five stages are distinguished and run synchronously. Some extra MUXes (multiplexers) and forwarding lines are added to solve hazards.



ii. Component

SRAM: The memory storage unit . By duplicating a memory unit, it handles fetch and load-related instructions separately, reducing hazard issues. This separation allows simultaneous instruction fetching and data loading without structural hazards. It's important to note this usually refers to separate instruction and data caches, not a full duplication of main memory.

Decoder: This is where RISC-V encoded instructions are disassembled into different parts. It generates meaningful segments that are used as references by subsequent execution units. The decoder identifies the opcode, registers (rs1, rs2, rd), immediate values, and function codes (funct3, funct7), which control the ALU and other units.

Imm_ext: Some instructions, such as jumps and branches, require immediate values. These immediate values are sent here for processing. The immediate extender performs sign extension or zero extension depending on the instruction type. Sign extension is used for signed immediates to maintain their correct value when used in arithmetic operations or address calculations. Zero extension is used for unsigned immediates.

Reg_file: This is the storage location for registers, acting as a buffer to temporarily store the values decoded from instructions. The register file typically has multiple read ports and write ports, allowing multiple instructions to read and write registers simultaneously (within the limits of the number of ports) during the pipeline stages.

ALU: This is where all arithmetic and logic instructions are executed. After the operation is complete, the result is passed to the next stage, MEM (memory access). The ALU performs operations like addition, subtraction, AND, OR, XOR, shifts, and comparisons.

JB Unit: When a jump or branch instruction is encountered, the execution path goes through this unit. It directly calculates the target PC address for the jump or branch. For branches, the JB Unit evaluates the branch condition (e.g., equality, less than) based on the comparison result from the ALU. If the condition is met, the PC is updated; otherwise, the PC proceeds to the next sequential instruction.

LD Filter: To handle hazards, especially load-use data hazards, this unit specifically handles the write-back of load instructions to the RegFile. This forwarding mechanism allows subsequent instructions that depend on the loaded data to receive it directly from the MEM/WB pipeline register, avoiding stalls in many cases. This is a form of forwarding specifically designed for load instructions. It is sometimes also referred to as a "load interlock" or "load delay slot" in simpler designs without full forwarding.

2. Instructions

R-type

R-type	funct7	rs2	rs1	funct3	rd	opcode
ADD	0000000	rs2	rs1	000	rd	0110011
SUB	0100000	rs2	rs1	000	rd	0110011
SLL	0000000	rs2	rs1	001	rd	0110011
SLT	0000000	rs2	rs1	010	rd	0110011
SLTU	0000000	rs2	rs1	011	rd	0110011
XOR	0000000	rs2	rs1	100	rd	0110011
SRL	0000000	rs2	rs1	101	rd	0110011
SRA	0100000	rs2	rs1	101	rd	0110011
OR	0000000	rs2	rs1	110	rd	0110011
AND	0000000	rs2	rs1	111	rd	0110011

I-type

I-type	Imm[11:0]		rs1	funct3	rd	opcode
ADDI	Imm[11:0]		rs1	000	rd	0010011
SLTI	Imm[11:0]		rs1	010	rd	0010011
SLTIU	Imm[11:0]		rs1	011	rd	0010011
XORI	Imm[11:0]		rs1	100	rd	0010011
ORI	Imm[11:0]		rs1	110	rd	0010011
ANDI	Imm[11:0]		rs1	111	rd	0010011
SLLI	0000000	shamt	rs1	001	rd	0010011
SRLI	0000000	shamt	rs1	101	rd	0010011
SRAI	0100000	shamt	rs1	101	rd	0010011
LB	imm[11:0]		rs1	000	rd	0000011
LH	imm[11:0]		rs1	001	rd	0000011
LW	imm[11:0]		rs1	010	rd	0000011
LBU	imm[11:0]		rs1	100	rd	0000011
LHU	imm[11:0]		rs1	101	rd	0000011

S-type

S-type	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[11:5]	rs2	rs1	000	imm[4:0]	0100011
SH	imm[11:5]	rs2	rs1	001	imm[4:0]	0100011
SW	imm[11:5]	rs2	rs1	010	imm[4:0]	0100011

SB-type

SB-type	imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
BEQ	imm[12]	imm[10:5]	rs2	rs1	000	imm[4:1]	imm[11]	1100011
BNE	imm[12]	imm[10:5]	rs2	rs1	001	imm[4:1]	imm[11]	1100011
BLT	imm[12]	imm[10:5]	rs2	rs1	100	imm[4:1]	imm[11]	1100011
BGE	imm[12]	imm[10:5]	rs2	rs1	101	imm[4:1]	imm[11]	1100011
BLTU	imm[12]	imm[10:5]	rs2	rs1	110	imm[4:1]	imm[11]	1100011
BGEU	imm[12]	imm[10:5]	rs2	rs1	111	imm[4:1]	imm[11]	1100011

U-type

U-type	Imm[31:12]	rd	opcode
LUI	Imm[31:12]	rd	0110111
AUIPC	Imm[31:12]	rd	0010111

UJ-type

UJ-type	Imm[20]	Imm[10:1]	Imm[11]	Imm[19:12]	rd	opcode
JAL	Imm[20]	Imm[10:1]	Imm[11]	Imm[19:12]	rd	1101111
JALR	Imm[11:0]		Rs1	000	rd	1100111

3. Verification

1. Methods

To verify the correct operation of the ALU in a CPU, we first perform basic functionality tests. These include testing with single instructions and testing with combinations of different instruction types. Then, we check whether the forwarding functionality and hazard elimination mechanisms are working correctly. After these tests are completed, we proceed with meaningful overall program verification. We use design events to detect when these situations occur and check if the Program Counter (PC) stops updating, and clear any incorrectly executed instructions, allowing the previously executing instructions to complete.

After verifying the correct operation of single instructions, forwarding, and hazard elimination, we begin the remaining program verification. The programs used for verification are two types: sorting and Fibonacci sequence generation. For sorting, we use bubble sort for verification. This means the program will compare the magnitudes of numbers stored in

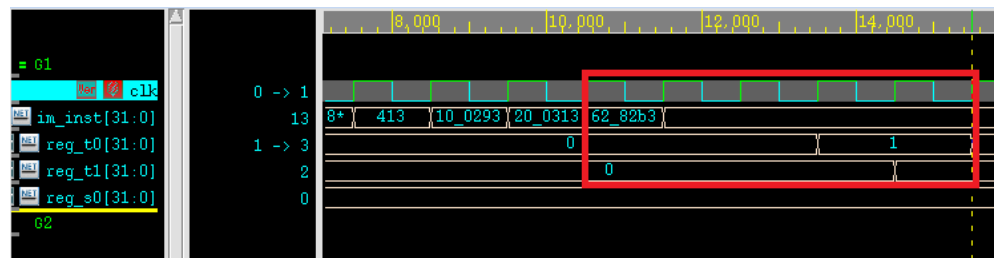
adjacent memory locations and sort them, progressing from the first number to the last. After this process, the sorted result is the output of the bubble sort. The second verification uses the Fibonacci sequence, where we expect to observe that each output value is the sum of the two preceding values. The entire output sequence is then the Fibonacci sequence.

2. Analysis

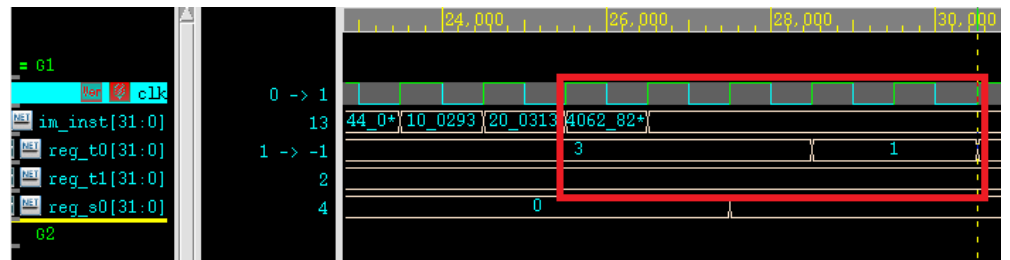
i. Single Instructions

A. R-type

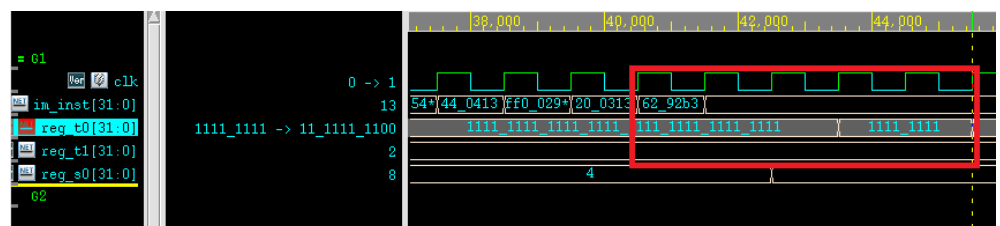
add t0,t0,t1 (t0 = 1 + 2)



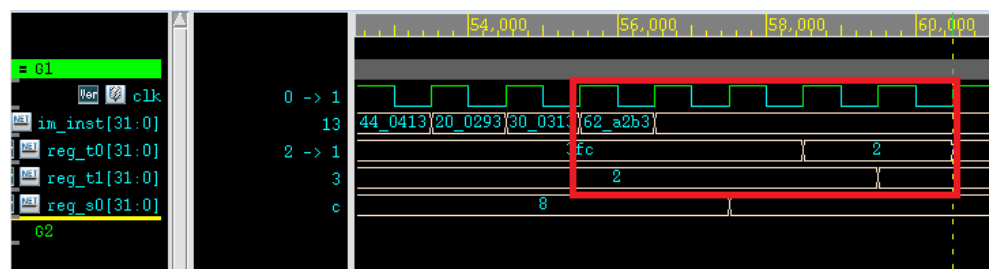
sub t0,t0,t1 (t0 = 1 - 2)



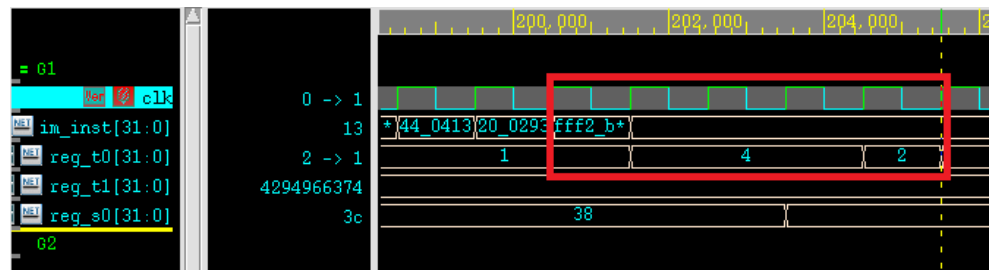
sll t0,t0,t1 (t0 = 1111_1111 << 2)



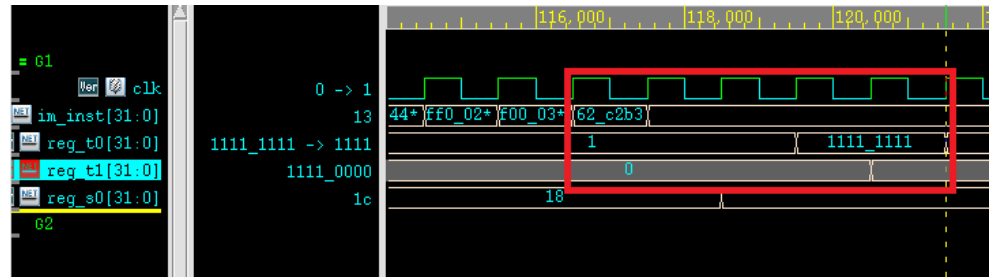
slt t0,t0,t1 (t0 = 2, t1 = 3) -> t0 = 1



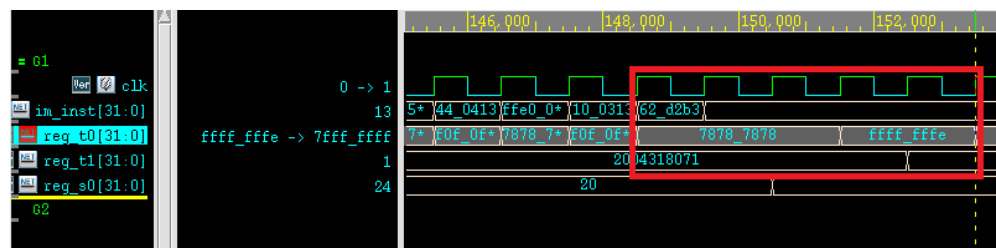
sltu t0,t0,t1 (t0 = 2, t1 = -1) -> t0 = 1



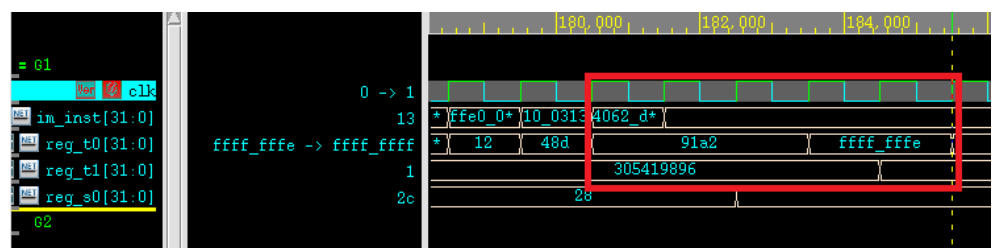
xor t0,t0,t1 (t0 = 1111_1111 ^ 1111_0000)



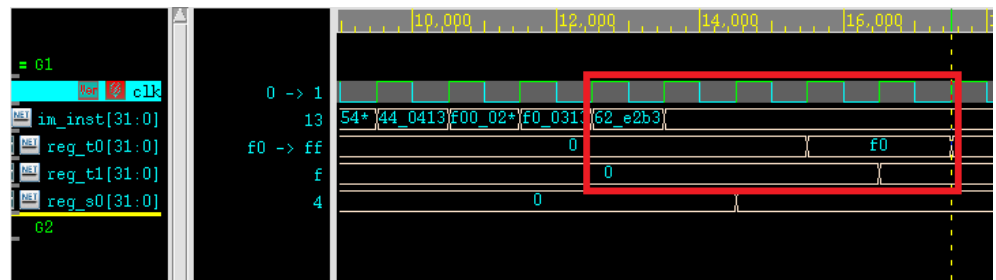
srl t0,t0,t1 (t0 = 0xffff_fffe >> 1)



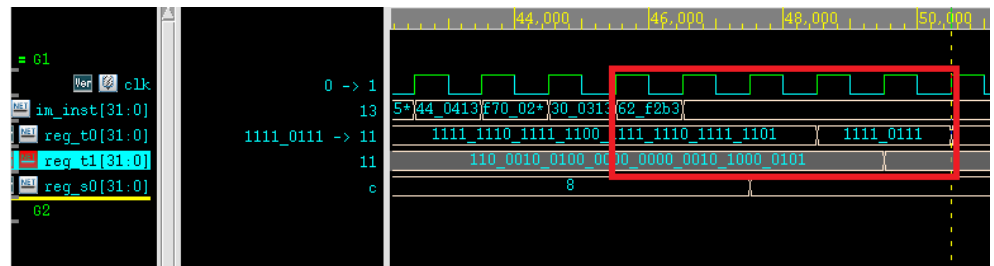
sra t0,t0,t1 (t0 = 0xffff_fffe >>> 1)



or t0,t0,t1 (t0 = f0 | f)

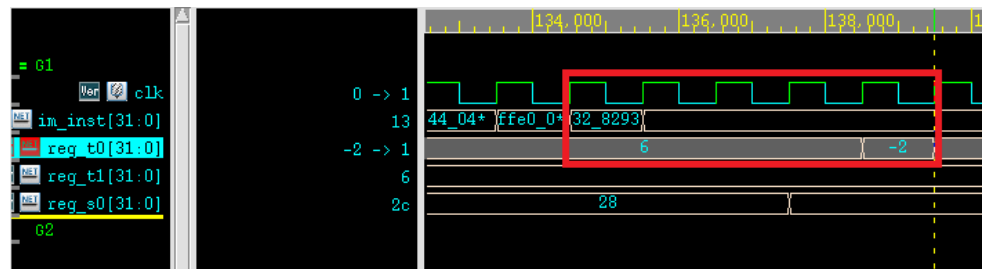


and t0,t0,t1 (t0 = 1111_0111 & 11)

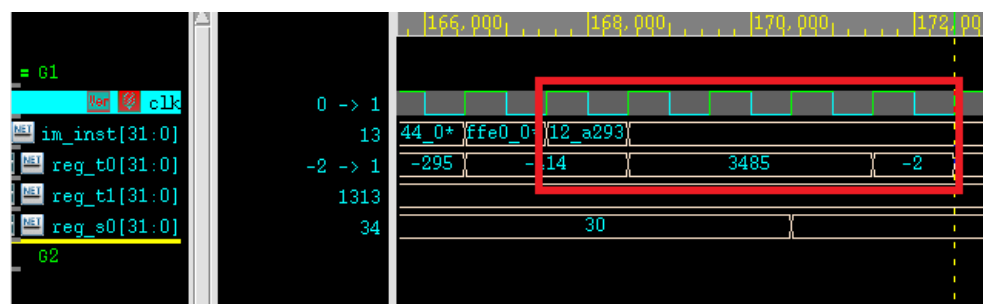


B. l-type

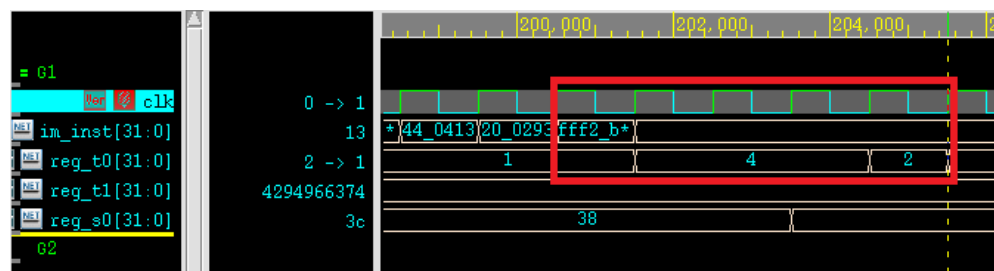
addi t0,t0,3 (t0 = -2 + 3)



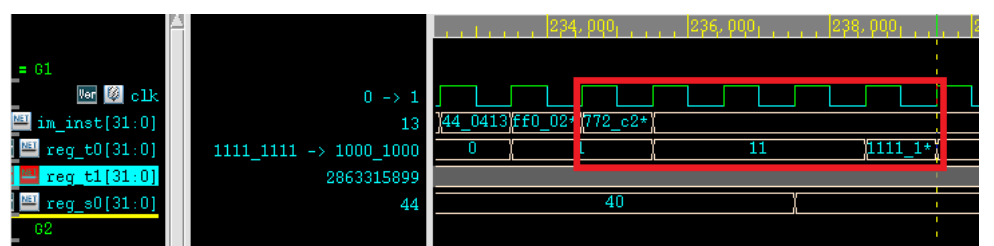
slti t0,t0,1 (t0 = -2, imm = 1) -> t0 = 1



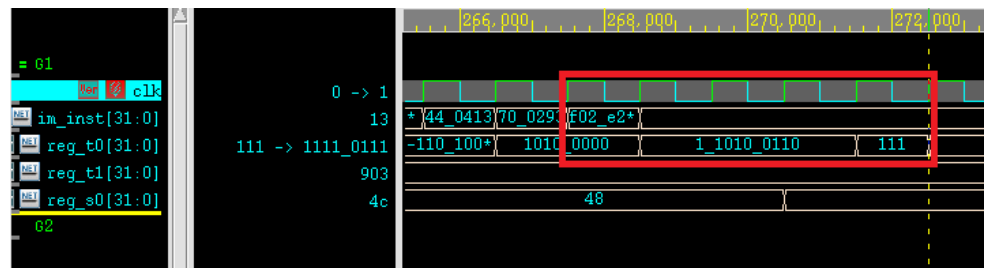
sltiu t0,t0,-1 (t0 = 2, imm = -1) -> t0 = 1



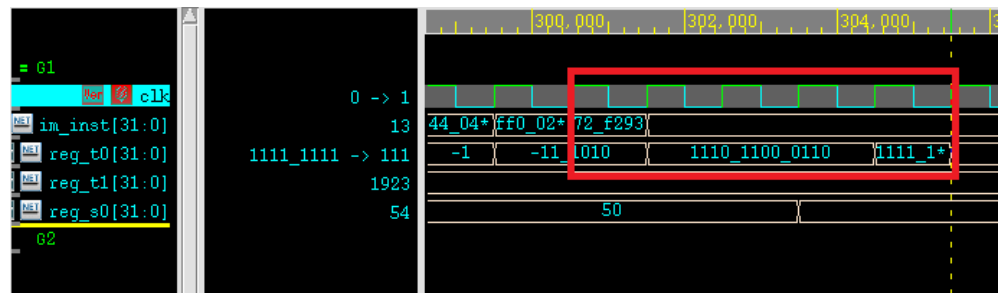
xori t0,t0,0x77 (t0 = 1111_1111, imm = 0x77) -> t0 = 1000_1000



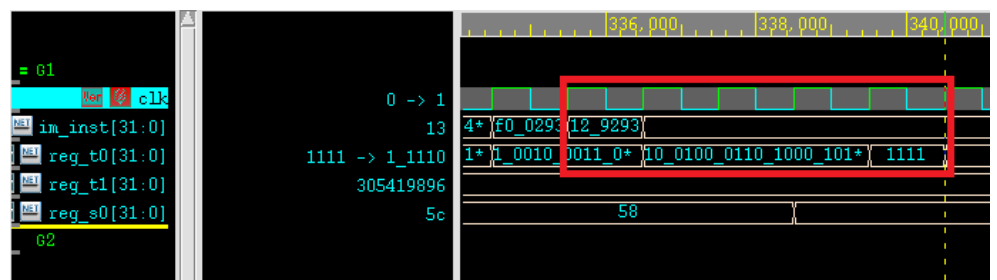
ori t0,t0,0xf0 (t0 = 111, imm = 0xf0) -> t0 = 1111_0111



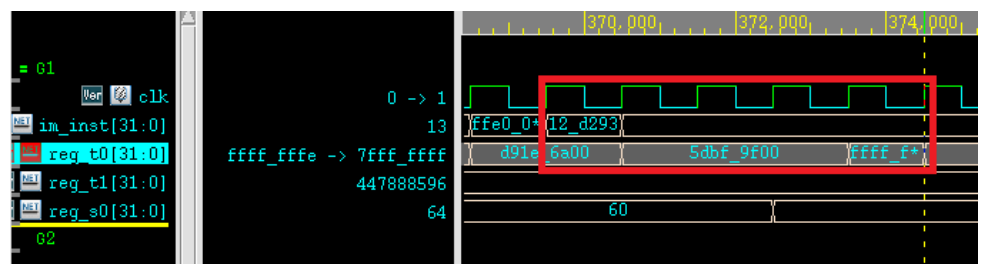
andi t0,t0,0x7 (t0 = 1111_1111, imm = 0x7) -> t0 = 111



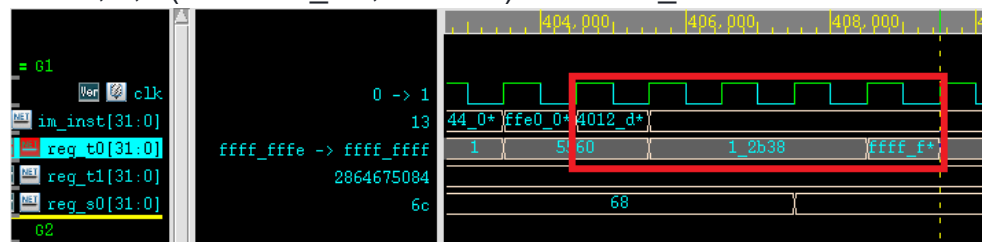
slli t0,t0,1 (t0 = 1111, imm = 1) -> t0 = 1_1110



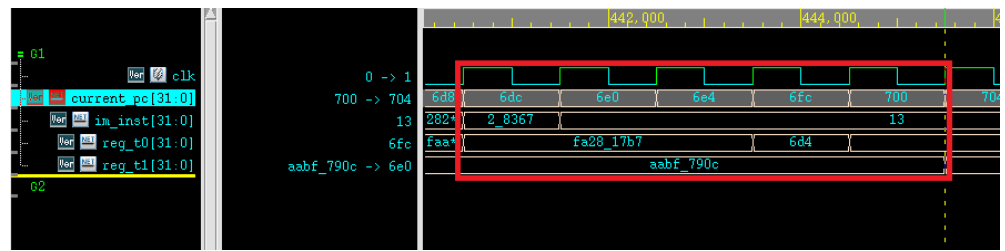
srlr t0,t0,1 (t0 = 0xffff_fffe, imm = 1) -> t0 = 7fff_ffff



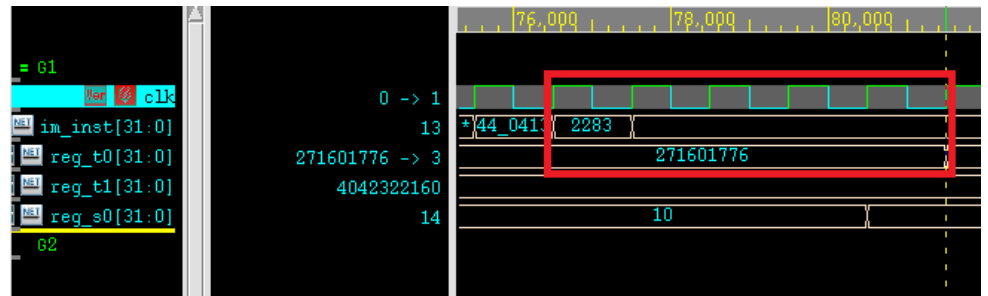
srai t0,t0,1 (t0 = 0xffff_fffe, imm = 1) -> t0 = ffff_ffff



jalr t1,t0,0 (store current PC in t1, PC set to t0)

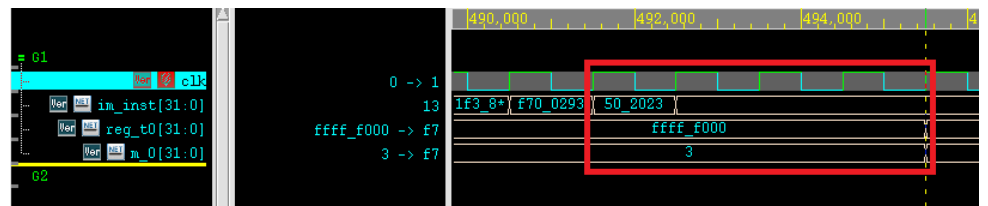


lw t0,0(x0) (load mem[0] into t0)



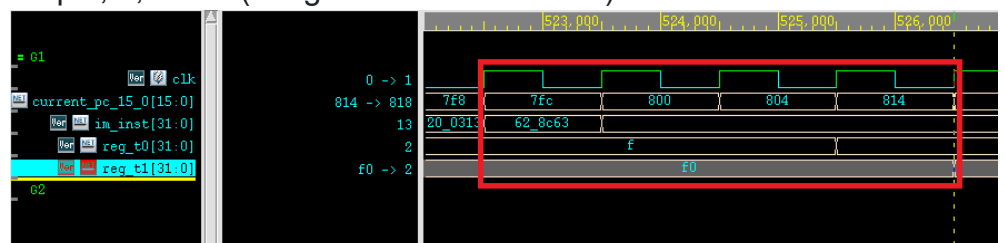
C. S-type

sw t0,0(x0) (store 0xf7 into mem[0])

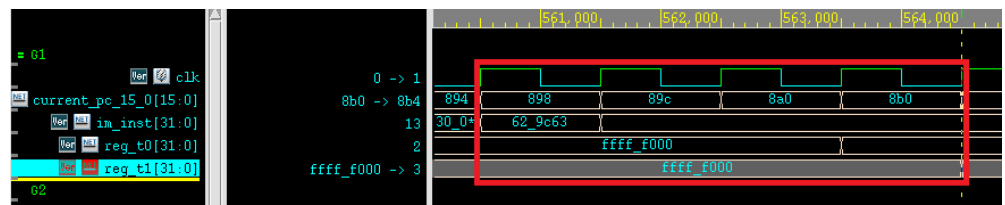


D. B-type (forwarding -> branch write-back)

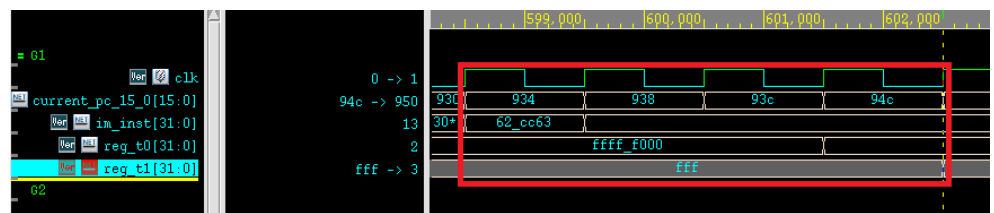
beq t0,t1,0x814 (PC goto 0x814 if t0 == t1)



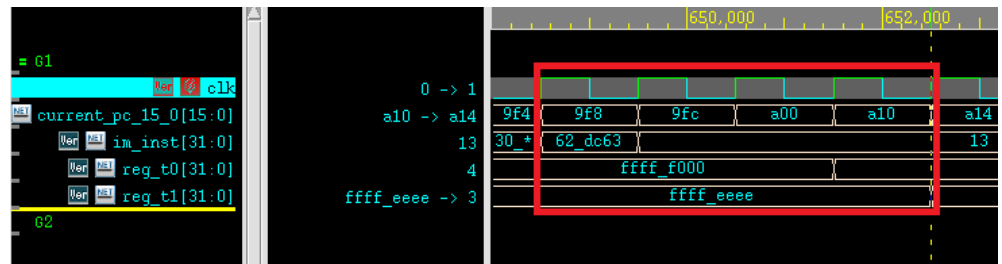
bne t0,t1,0xb0 (PC goto 0xb0 if t0 != t1)



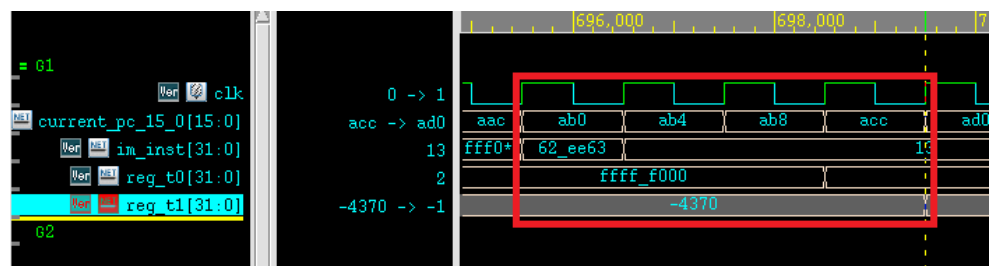
blt t0,t1,0x4c (PC goto 0x94c if t0 < t1)



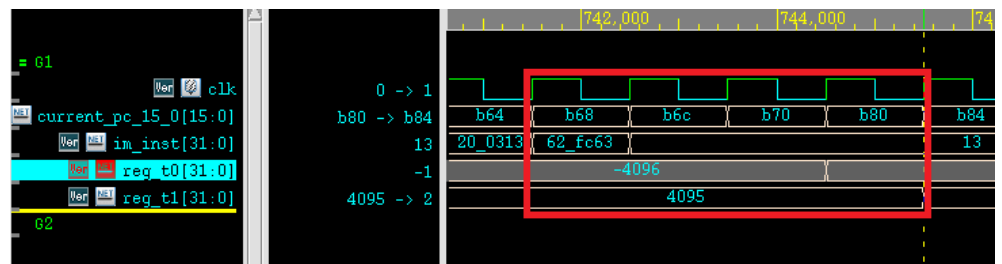
bge t0,t1,0xa10 (PC goto 0xa10 if t0 >= t1)



bltu t0,t1,0xacc

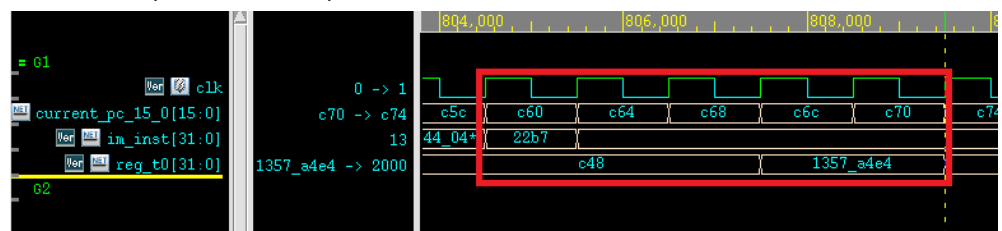


bgeu t0,t1,0xb80

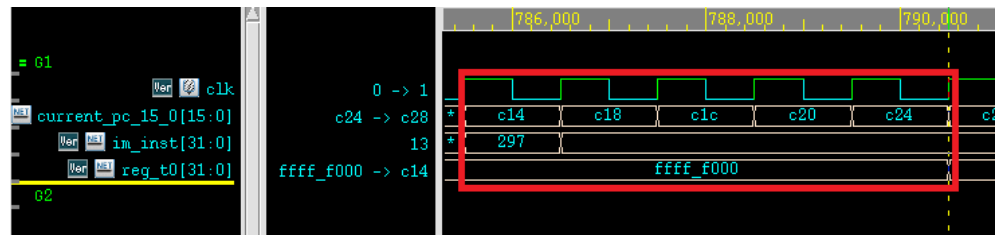


E. U-type

lui t0,0x2 (t0 = 0x2000)

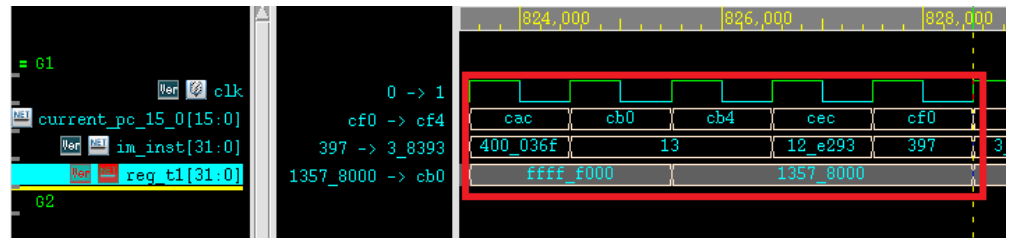


auipc t0,0 (t0 = PC)



F. J-type

jal t1,0xccec (store current PC in t1, PC goto 0xccec)



ii. Correctness

A. Sort

```
addi a0, x0, 3
sw a0, 0(x0)
addi a0, x0, 6
sw a0, 4(x0)
addi a0, x0, 1
sw a0, 8(x0)
addi a0, x0, 2
sw a0, 12(x0)
```

Initialize memory

```
swap:
sw t1, 4(t3)
sw t2, 0(t3)
addi t0, t0, 1
j loop
```

Swap function

```

loop2:
beq t4, a0, loop2_done
addi t0, x0, 0

loop:
beq t0, a0, loop_done

add t3, t0, t0
add t3, t3, t3

lw t1, 0(t3)
lw t2, 4(t3)
bge t1, t2, swap

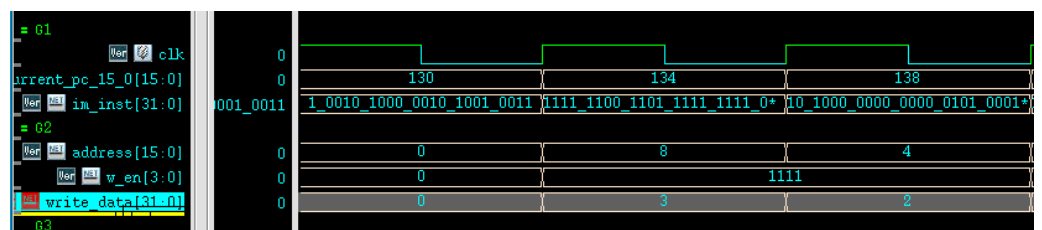
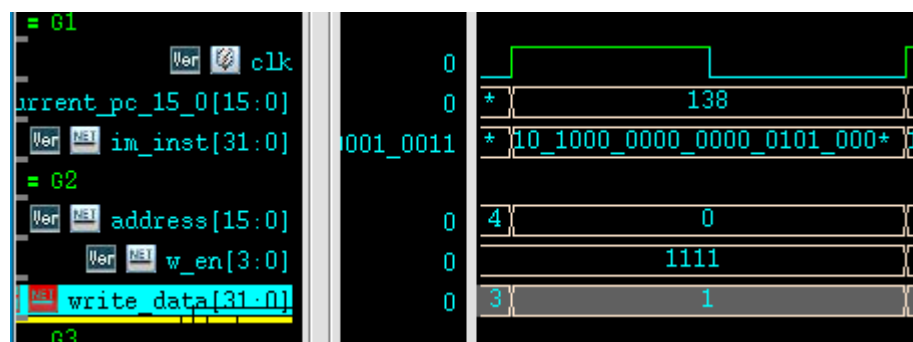
addi t0, t0, 1
j loop

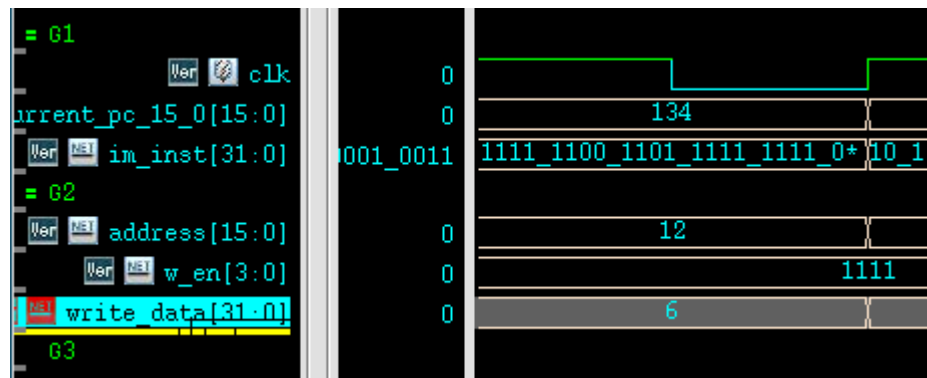
loop_done:
addi t4, t4, 1
j loop2

loop2_done:
addi a0, x0, 40
addi a1, x0, -1
sw a1, 0(a0)
hcf

```

For loop to detect the correctness of swap function





Mem[0]=1, Mem[4]=2, Mem[8]=3, Mem[12]=6

B. Fibonacci

```
addi    sp, x0, 256
addi    a0, x0, 9

# 開始進行 fibonacci 運算
addi    sp, sp, -4
sw      ra, 0(sp)
jal     fibonacci
lw      ra, 0(sp)
addi    sp, sp, 4

#pass ans to a0
addi    a0, a1, 0

j halt
```

Call Fibonacci function

```
fibonacci:
    addi    sp, sp, -12
    sw      a0, 0(sp)
    sw      a2, 4(sp)
    sw      a3, 8(sp)

    #t0=1
    addi    t0, x0, 1
    beq     a0, t0, ret_one

    #t0=0
    addi    t0, x0, 0
    beq     a0, t0, ret_zero
```

If t0=1, ret 1 ; If t0=0, ret 0

```
ret_zero:
    addi    a1, x0, 0
    lw      a3, 8(sp)
    lw      a2, 4(sp)
    lw      a0, 0(sp)
    addi    sp, sp, 12
    jr      ra
```

```
ret_one:
    addi    a1, x0, 1
    lw      a3, 8(sp)
    lw      a2, 4(sp)
    lw      a0, 0(sp)
    addi    sp, sp, 12
    jr      ra
```

Return 1/0 function

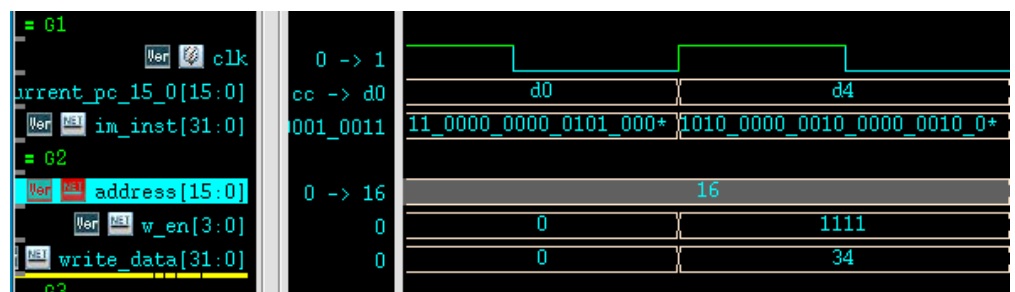
```
#進行fibonacci(n-1)
addi    a0, a0, -1
addi    sp, sp, -4
sw      ra, 0(sp)
jal     fibonacci
lw      ra, 0(sp)
addi    sp, sp, 4
#a2儲存a1值
addi    a2, a1, 0
```

```
#進行fibonacci(n-2)
addi    a0, a0, -1
addi    sp, sp, -4
sw      ra, 0(sp)
jal     fibonacci
lw      ra, 0(sp)
addi    sp, sp, 4
#a3儲存a1值
addi    a3, a1, 0
```

```
#get ans
add     a1, a2, a3

lw      a3, 8(sp)
lw      a2, 4(sp)
lw      a0, 0(sp)
addi    sp, sp, 12
jr      ra
```

Final Answer

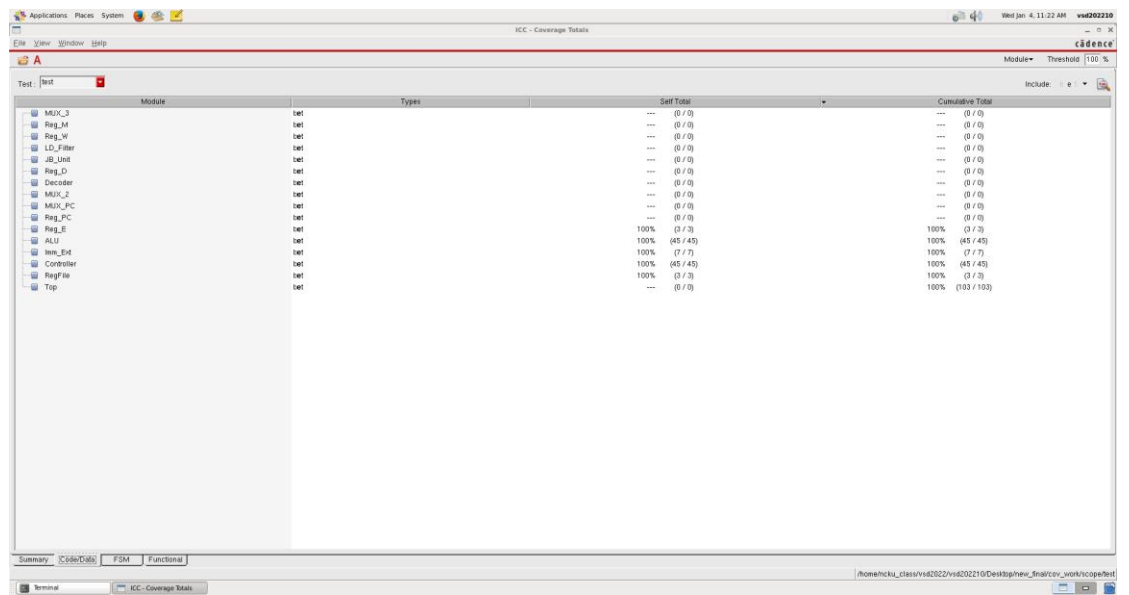


fibonacci(9)=34

$$\dim[16] = 34$$

Module	Type	Self Total	Cumulative Total
MDUX_3	test	100% (0/0)	100% (0/0)
MDUX_2	test	100% (0/0)	100% (0/0)
MDUX_PC	test	100% (0/0)	100% (0/0)
Reg_M	test	100% (0/0)	100% (0/0)
ALU	test	100% (0/0)	100% (0/0)
Reg_M	test	100% (0/0)	100% (0/0)
Reg_M	test	100% (0/0)	100% (0/0)
LQ_Filter	test	100% (0/0)	100% (0/0)
Mem_E4	test	100% (0/0)	100% (0/0)
JB_Unit	test	100% (0/0)	100% (0/0)
Controller	test	100% (0/0)	100% (0/0)
Reg_D	test	100% (0/0)	100% (0/0)
Decoder	test	100% (0/0)	100% (0/0)
RegFile	test	100% (0/0)	100% (0/0)
Trip	test	100% (0/0)	100% (0/0)
Reg_PC	test	100% (0/0)	100% (0/0)

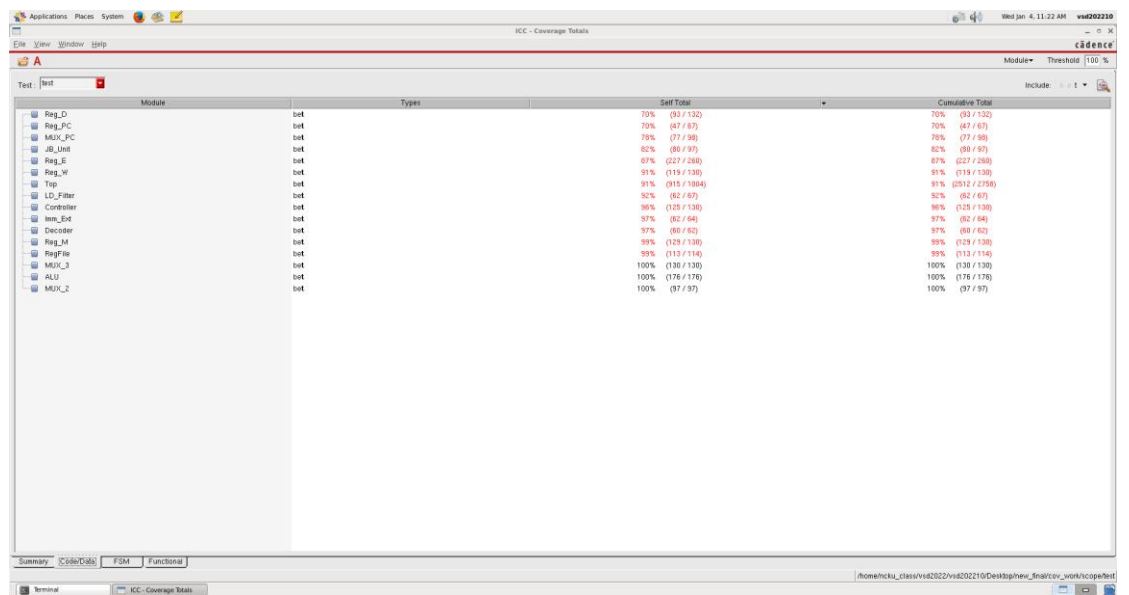
ii. Expression coverage: 100%



The screenshot shows the ICC - Coverage Totals window with the 'Test' dropdown set to 'test'. The table lists modules and their coverage statistics. All modules show 100% coverage.

Module	Type	Self Total	Cumulative Total
MUX_3	test	---	---
Reg_M	test	---	---
Reg_M	test	---	---
LD_Filter	test	---	---
JB_Unit	test	---	---
Reg_D	test	---	---
Decoder	test	---	---
MUX_2	test	---	---
MUX_PC	test	---	---
Reg_PC	test	---	---
Reg_E	test	100%	100%
ALU	test	100%	100%
Inv_Est	test	100%	100%
Controller	test	100%	100%
RegFile	test	100%	100%
Top	test	100%	100%

iii. Toggle coverage: 70% up



The screenshot shows the ICC - Coverage Totals window with the 'Test' dropdown set to 'test'. The table lists modules and their coverage statistics. The coverage for the first 15 modules is 70%, and for the last 5 modules, it is 100%.

Module	Type	Self Total	Cumulative Total
Reg_D	test	70% (93 / 132)	70% (93 / 132)
Reg_PC	test	70% (67 / 97)	70% (67 / 97)
MUX_PC	test	70% (77 / 98)	70% (77 / 98)
JB_Unit	test	82% (86 / 97)	82% (86 / 97)
Reg_E	test	87% (227 / 260)	87% (227 / 260)
Reg_M	test	91% (115 / 130)	91% (115 / 130)
Top	test	91% (915 / 1084)	91% (2512 / 2758)
LD_Filter	test	92% (82 / 87)	92% (82 / 87)
Controller	test	96% (125 / 130)	96% (125 / 130)
Inv_Est	test	97% (82 / 84)	97% (82 / 84)
Decoder	test	97% (86 / 87)	97% (86 / 87)
Reg_M	test	99% (129 / 130)	99% (129 / 130)
RegFile	test	99% (113 / 114)	99% (113 / 114)
MUX_3	test	100% (130 / 130)	100% (130 / 130)
ALU	test	100% (176 / 176)	100% (176 / 176)
MUX_2	test	100% (97 / 97)	100% (97 / 97)

iv. Overall coverage: 72% up

Module	Type	Self Total	Cumulative Total
Reg_D	bet	72% (100 / 138)	72% (100 / 138)
Reg_PC	bet	72% (52 / 72)	72% (52 / 72)
MUX_PC	bet	70% (77 / 90)	70% (77 / 90)
JB_Unit	bet	63% (81 / 98)	63% (81 / 98)
Reg_E	bet	88% (235 / 268)	88% (235 / 268)
Reg_W	bet	92% (122 / 133)	92% (122 / 133)
Top	bet	91% (215 / 234)	91% (215 / 234)
LD_Filter	bet	93% (73 / 78)	93% (73 / 78)
Imm_Ext	bet	97% (88 / 92)	97% (88 / 92)
Controller	bet	97% (282 / 287)	97% (282 / 287)
Decoder	bet	97% (83 / 85)	97% (83 / 85)
Reg_M	bet	99% (132 / 133)	99% (132 / 133)
RegFile	bet	99% (122 / 123)	99% (122 / 123)
MUX_2	bet	100% (130 / 130)	100% (130 / 130)
ALU	bet	100% (284 / 284)	100% (284 / 284)
MUX_2	bet	100% (97 / 97)	100% (97 / 97)

5. Simulation by Synopsys

1. Speed, Setup time/Hold time slack >0)

Report 5 - Timing

Path	Delay	Slack
jb_unit/add_8/U1_18/CO (ADDFX2)	0.40	44.17 f
jb_unit/add_8/U1_19/CO (ADDFX2)	0.40	44.17 f
jb_unit/add_8/U1_20/CO (ADDFX2)	0.40	44.56 f
jb_unit/add_8/U1_21/CO (ADDFX2)	0.40	44.96 f
jb_unit/add_8/U1_22/CO (ADDFX2)	0.40	45.35 f
jb_unit/add_8/U1_23/CO (ADDFX2)	0.40	45.75 f
jb_unit/add_8/U1_24/CO (ADDFX2)	0.40	46.14 f
jb_unit/add_8/U1_25/CO (ADDFX2)	0.40	46.54 f
jb_unit/add_8/U1_26/CO (ADDFX2)	0.40	46.93 f
jb_unit/add_8/U1_27/CO (ADDFX2)	0.40	47.33 f
jb_unit/add_8/U1_28/CO (ADDFX2)	0.40	47.72 f
jb_unit/add_8/U1_29/CO (ADDFX2)	0.40	48.12 f
jb_unit/add_8/U1_30/CO (ADDFX2)	0.40	48.52 f
jb_unit/add_8/U1_31/Y (XORX2)	0.27	48.79 f
jb_unit/add_8/SUM[31] (JB_Unit_DM01_add_0)	0.00	48.79 f
jb_unit/jb_out[31] (JB_Unit)	0.00	48.79 f
mux_next_pc/input[31] (MUX_PC)	0.00	48.79 f
mux_next_pc/U12/Y (A0122X1)	0.31	49.11 r
mux_next_pc/U11/Y (INVX1)	0.16	49.27 f
mux_next_pc/mux_out[31] (MUX_PC)	0.00	49.27 f
reg_pc/next_pc[31] (Reg_PC)	0.00	49.27 f
reg_pc/U9/Y (A0122X1)	0.32	49.59 r
reg_pc/U9/Y (INVX1)	0.14	49.72 f
reg_pc/current_pc_reg[31]/D (DFFRHQX1)	0.00	49.72 f
data arrival time		49.72
clock clk (rise edge)	75.00	75.00
clock network delay (ideal)	0.00	75.00
reg_pc/current_pc_reg[31]/CK (DFFRHQX1)	0.00	75.00 r
library setup time	-0.38	74.62
data required time		74.62
data arrival time		-49.72
slack (MET)		24.90

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

Report 6 - Timing

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

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

Startpoint: reg_m/Mout_rs2_data_reg[0]
(rising edge-triggered flip-flop clocked by clk)
Endpoint: reg_m_out_rs2_data[0]
(output port 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)	25.00	25.00
clock network delay (ideal)	0.00	25.00
reg_m/Mout_rs2_data_reg[0]/CK (DFFRHQX1)	0.00 #	25.00
reg_m/Mout_rs2_data_reg[0]/Q (DFFRHQX1)	0.32	25.32
reg_m/Mout_rs2_data[0] (Reg_M)	0.00	25.32
reg_m_out_rs2_data[0] (out)	0.00	25.32
data arrival time		25.32
clock clk (rise edge)	25.00	25.00
clock network delay (ideal)	0.00	25.00
output external delay	-0.10	24.90
data required time		24.90
data arrival time		-25.32
slack (MET)		0.42

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

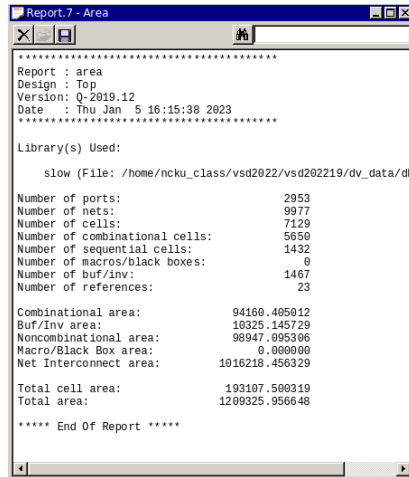
Synthesis result of time slack

Timing

Path type : max => slack = 24.9ns

Path type : min => slack = 0.42ns

2. Area



Synthesis result of area

Area

Total cell area = 193107.5 μm^2

Total area = 1209325.95 μm^2

3. Power

```

Report 8 - Power
*****
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 = 1.6943 mW (89%)
Net Switching Power = 211.4767 uW (11%)
-----
Total Dynamic Power = 1.9058 mW (100%)
Cell Leakage Power = 5.6774 uW

Power Group      Internal Power      Switching Power      Leakage Power      Total 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         1.6539              1.9210e-02          3.1079e+06           1.6762 ( 87.69%)
sequential       2.8727e-03          3.0203e-04          2.4641e+04           3.1994e-03 ( 0.17%)
combinational    3.7597e-02          0.1920              2.5449e+06           0.2321 ( 12.14%)
-----
Total            1.6943 mW           0.2115 mW           5.6774e+06 pW        1.9115 mW

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

```

Synthesis result of power

Power

Total dynamic power = 1.9058mW

Cell leakage power = 5.6774uW

6. Layout

```
11711 module chip ( PI_clk, PI_rst, PI_im_inst, PI_reg_w_in_ld_data, PO_current_pc_15_0,
11712 | PO_reg_m_out_alu_out_15_0, PO_F_im_w_en, PO_M_dm_w_en, PO_reg_m_out_rs2_data );
11713 input [31:0] PI_im_inst;
11714 input [31:0] PI_reg_w_in_ld_data;
11715 output [15:0] PO_current_pc_15_0;
11716 output [15:0] PO_reg_m_out_alu_out_15_0;
11717 output [3:0] PO_F_im_w_en;
11718 output [3:0] PO_M_dm_w_en;
11719 output [31:0] PO_reg_m_out_rs2_data;
11720 input PI_clk, PI_rst;
11721
11722 // Internal wires
11723 wire [31:0] WIRE_im_inst;
11724 wire [31:0] WIRE_reg_w_in_ld_data;
11725 wire [15:0] WIRE_current_pc_15_0;
11726 wire [15:0] WIRE_reg_m_out_alu_out_15_0;
11727 wire [3:0] WIRE_F_im_w_en;
11728 wire [3:0] WIRE_M_dm_w_en;
11729 wire [31:0] WIRE_reg_m_out_rs2_data;
11730 wire WIRE_clk, WIRE_rst;
11731
11732
11733 Top top(
11734 .clk(WIRE_clk),
11735 .rst(WIRE_rst),
11736 .im_inst(WIRE_im_inst),
11737 .current_pc_15_0(WIRE_current_pc_15_0),
11738 .reg_m_out_alu_out_15_0(WIRE_reg_m_out_alu_out_15_0),
11739 .F_im_w_en(WIRE_F_im_w_en),
11740 .M_dm_w_en(WIRE_M_dm_w_en),
11741 .reg_w_in_ld_data(WIRE_reg_w_in_ld_data),
11742 .reg_m_out_rs2_data(WIRE_reg_m_out_rs2_data)
11743 );
11744
11745 //input pads : PDIDG2
11746 PDIDG2 PAD_rst ( .PAD (PI_rst), .C (WIRE_rst));
11747 PDIDG2 PAD_clk ( .PAD (PI_clk), .C (WIRE_clk));
11748 PDIDG2 PAD_im_inst ( .PAD (PI_im_inst), .C (WIRE_im_inst));
11749 PDIDG2 PAD_reg_w_in_ld_data ( .PAD (PI_reg_w_in_ld_data), .C (WIRE_reg_w_in_ld_data));
11750 //Output pads : PDO02CDG
11751
11752 PDO02CDG PAD_WIRE_current_pc_15_0( .I (WIRE_current_pc_15_0), .PAD (PO_current_pc_15_0));
11753 PDO02CDG PAD_WIRE_reg_m_out_alu_out_15_0( .I (WIRE_reg_m_out_alu_out_15_0), .PAD (PO_reg_m_out_alu_out_15_0));
11754 PDO02CDG PAD_WIRE_F_im_w_en( .I (WIRE_F_im_w_en), .PAD (PO_F_im_w_en));
11755 PDO02CDG PAD_WIRE_M_dm_w_en( .I (WIRE_M_dm_w_en), .PAD (PO_M_dm_w_en));
11756 PDO02CDG PAD_WIRE_reg_m_out_rs2_data( .I (WIRE_reg_m_out_rs2_data), .PAD (PO_reg_m_out_rs2_data));
11757
11758 endmodule
```

original synthesized circuit and IO pads

```
1 #####
2
3 # Created by write_sdc on Thu Jan  5 16:17:00 2023
4
5 #####
6 set_sdc_version 1.5
7
8 set_operating_conditions -max slow -max_library slow\
9                           -min fast -min_library fast
10 set_wire_load_model -name tsmc18_wl10 -library slow
11 set_max_area 3000
12 set_driving_cell -lib_cell DFFX2 -library slow -dont_scale -no_design_rule \
13 [get_ports PI_rst]
14 set_load -pin_load 0.002323 [get_ports {PO_current_pc_15_0[15]}}
15 set_load -pin_load 0.002323 [get_ports {PO_current_pc_15_0[14]}}
16 set_load -pin_load 0.002323 [get_ports {PO_current_pc_15_0[13]}}
17 set_load -pin_load 0.002323 [get_ports {PO_current_pc_15_0[12]}}
18 set_load -pin_load 0.002323 [get_ports {PO_current_pc_15_0[11]}}
19 set_load -pin_load 0.002323 [get_ports {PO_current_pc_15_0[10]}}
20 set_load -pin_load 0.002323 [get_ports {PO_current_pc_15_0[9]}}
21 set_load -pin_load 0.002323 [get_ports {PO_current_pc_15_0[8]}}
22 set_load -pin_load 0.002323 [get_ports {PO_current_pc_15_0[7]}}
23 set_load -pin_load 0.002323 [get_ports {PO_current_pc_15_0[6]}}
24 set_load -pin_load 0.002323 [get_ports {PO_current_pc_15_0[5]}}
25 set_load -pin_load 0.002323 [get_ports {PO_current_pc_15_0[4]}}
26 set_load -pin_load 0.002323 [get_ports {PO_current_pc_15_0[3]}}
27 set_load -pin_load 0.002323 [get_ports {PO_current_pc_15_0[2]}}
28 set_load -pin_load 0.002323 [get_ports {PO_current_pc_15_0[1]}}
29 set_load -pin_load 0.002323 [get_ports {PO_current_pc_15_0[0]}}
30 set_load -pin_load 0.002323 [get_ports {PO_reg_m_out_alu_out_15_0[15]}}
31 set_load -pin_load 0.002323 [get_ports {PO_reg_m_out_alu_out_15_0[14]}}
32 set_load -pin_load 0.002323 [get_ports {PO_reg_m_out_alu_out_15_0[13]}}
33 set_load -pin_load 0.002323 [get_ports {PO_reg_m_out_alu_out_15_0[12]}}
34 set_load -pin_load 0.002323 [get_ports {PO_reg_m_out_alu_out_15_0[11]}}
35 set_load -pin_load 0.002323 [get_ports {PO_reg_m_out_alu_out_15_0[10]}}
36 set_load -pin_load 0.002323 [get_ports {PO_reg_m_out_alu_out_15_0[9]}}
37 set_load -pin_load 0.002323 [get_ports {PO_reg_m_out_alu_out_15_0[8]}}
38 set_load -pin_load 0.002323 [get_ports {PO_reg_m_out_alu_out_15_0[7]}}
39 set_load -pin_load 0.002323 [get_ports {PO_reg_m_out_alu_out_15_0[6]}}
40 set_load -pin_load 0.002323 [get_ports {PO_reg_m_out_alu_out_15_0[5]}}
Normal text file
```

SDC Constraints

1	Version: 1		
2	Pad: CORNER0	NW	PCORNER
3	Pad: PAD_IOVDD1	N	PVDD2DGZ
4	Pad: PAD_COREVDD1	N	PVDD1DGZ
5	Pad: PAD_clk	N	
6	Pad: PAD_rst	N	
7	Pad: PAD_im_inst	N	
8	Pad: PAD_COREVSS1	N	PVSS1DGZ
9	Pad: PAD_IOVSS1	N	PVSS2DGZ
10			
11	Pad: CORNER1	NE	PCORNER
12	Pad: PAD_IOVDD2	E	PVDD2DGZ
13	Pad: PAD_COREVDD2	E	PVDD1DGZ
14	pad: PAD_current_pc_15_0	E	
15	Pad: PAD_COREVSS2	E	PVSS1DGZ
16	Pad: PAD_IOVSS2	E	PVSS2DGZ
17			
18	Pad: CORNER2	SE	PCORNER
19	Pad: PAD_IOVDD3	S	PVDD2DGZ
20	Pad: PAD_COREVDD3	S	PVDD1DGZ
21	Pad: PAD_F_im_w_en	S	
22	Pad: PAD_M_dm_w_en	S	
23	Pad: PAD_reg_w_in_ld_data	S	
24	Pad: PAD_COREVSS3	S	PVSS1DGZ
25	Pad: PAD_IOVSS3	S	PVSS2DGZ
26			
27	Pad: CORNER3	SW	PCORNER
28	Pad: PAD_IOVDD4	W	PVDD2DGZ
29	Pad: PAD_COREVDD4	W	PVDD1DGZ
30	Pad: PAD_reg_m_out_alu_out_15_0		W
31	Pad: PAD_reg_m_out_rs2_data		W
32	Pad: PAD_COREVSS4	W	PVSS1DGZ
33	Pad: PAD_IOVSS4	W	PVSS2DGZ
34			

IO Constraints


```
==== CALIBRE::DRC-H SUMMARY REPORT
====
Execution Date/Time:  Fri Jan  6 10:12:29 2023
Calibre Version:      v2020.2.14.12   Thu Apr 2 15:39:27 PDT 2020
Rule File Pathname:  Calibre-drc-cur
Rule File Title:
Layout System:       GDS
Layout Path(s):      chip_netlist.gds
Layout Primary Cell:  chip
Current Directory:    /Users/Working/23-1-6_pj1212_CALIBRE_st6912_101225/calibrerun.5480
User Name:           quser
Maximum Results/RuleCheck:  ALL
Maximum Result Vertices:  4096
DRC Results Database:  DRC_RES.db (ASCII)
Layout Depth:        ALL
Text Depth:          PRIMARY
Summary Report File:  DRC.rep (REPLACE)
Geometry Flaggings:  ACUTE = YES  SKEW = YES  ANGLED = NO  OFFGRID = YES
                     NONSIMPLE POLYGON = YES  NONSIMPLE PATH = NO

Excluded Cells:
CheckText Mapping:   ALL TEXT
Layers:
Keep Empty Checks:   YES

--- RUNTIME WARNINGS
---
DISCONNECT operations are present without DRC INCREMENTAL CONNECT YES specified.
Layer 16 contains unmapped objects and is the source layer of LAYER MAP == 16 DATATYPE == 1.
Layer 16 contains unmapped objects and is the source layer of LAYER MAP == 16 DATATYPE == 2.
Layer 18 contains unmapped objects and is the source layer of LAYER MAP == 18 DATATYPE == 1.
Layer 18 contains unmapped objects and is the source layer of LAYER MAP == 18 DATATYPE == 2.
Layer 28 contains unmapped objects and is the source layer of LAYER MAP == 28 DATATYPE == 1.
Layer 28 contains unmapped objects and is the source layer of LAYER MAP == 28 DATATYPE == 2.
Layer 31 contains unmapped objects and is the source layer of LAYER MAP == 31 DATATYPE == 1.
Layer 31 contains unmapped objects and is the source layer of LAYER MAP == 31 DATATYPE == 2.
Missing connections STAMPING layer DNM by layer NWELL1.

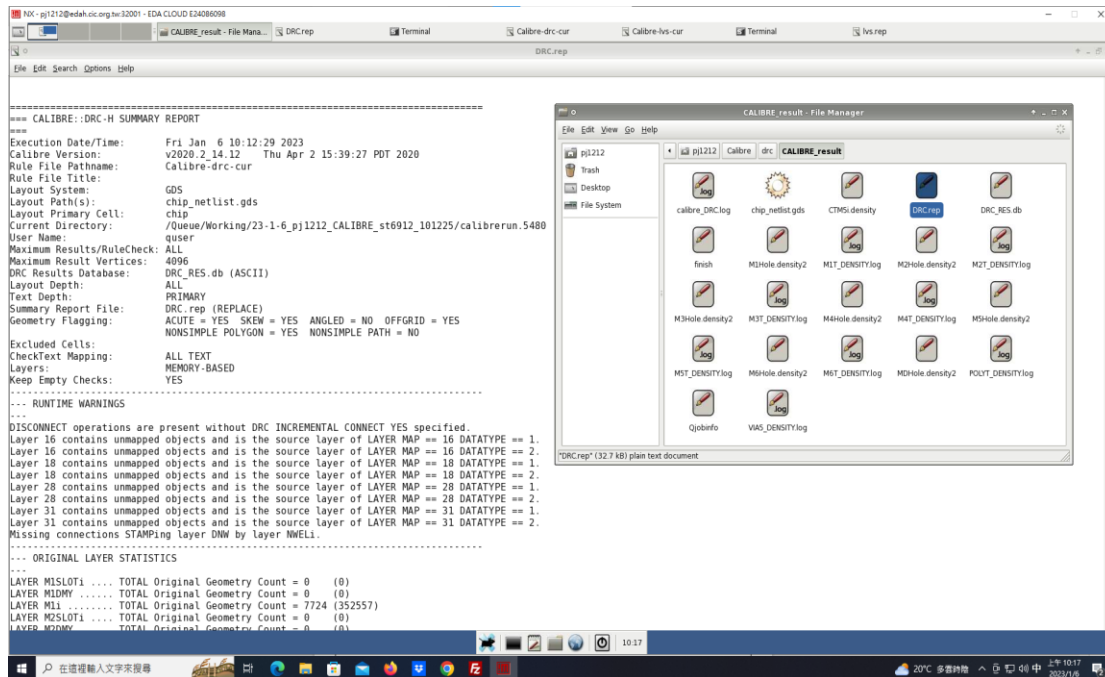
--- ORIGINAL LAYER STATISTICS
---
LAYER MISLOT1 .... TOTAL Original Geometry Count = 0 (0)
LAYER MIDW1 .... TOTAL Original Geometry Count = 0 (0)
LAYER M11 .... TOTAL Original Geometry Count = 7724 (352557)
LAYER M2SL01 .... TOTAL Original Geometry Count = 0 (0)
LAYER MIDW2 .... TOTAL Original Geometry Count = 0 (0)
```

Design Rule Check (DRC) (1/3)

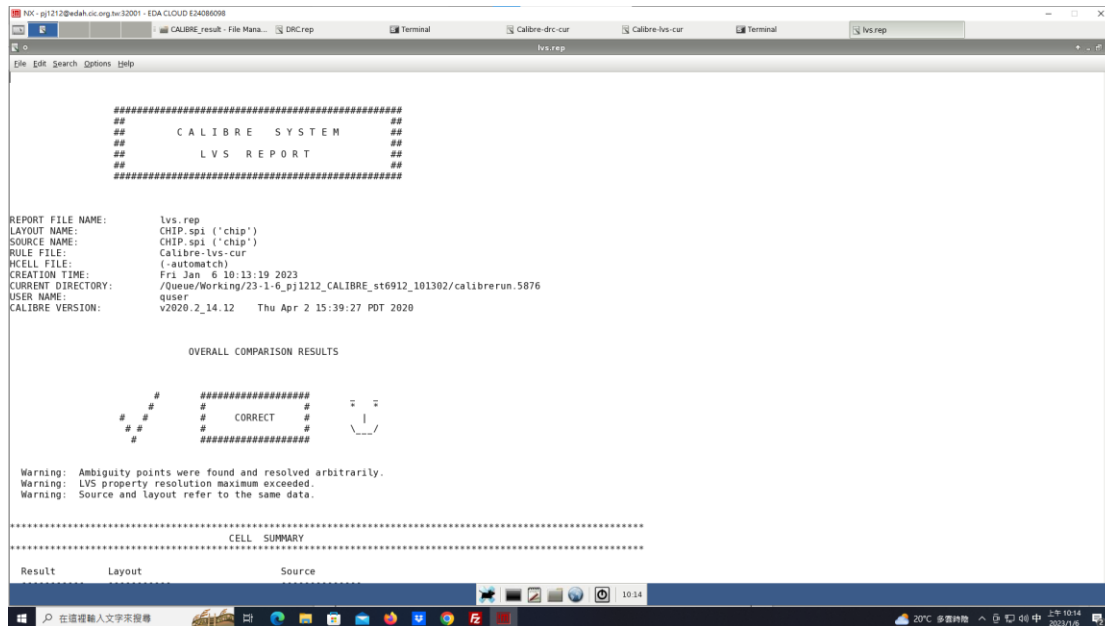
```
==== SUMMARY
====
TOTAL CPU Time:      4
TOTAL REAL Time:     5
TOTAL Original Layer Geometries: 24208 (536520)
TOTAL DRC RuleChecks Executed: 366
TOTAL DRC Results Generated: 85937 (149981)

--- SUMMARY
---
TOTAL CPU Time:      4
TOTAL REAL Time:     5
TOTAL Original Layer Geometries: 24208 (536520)
TOTAL DRC RuleChecks Executed: 366
TOTAL DRC Results Generated: 85937 (149981)
```

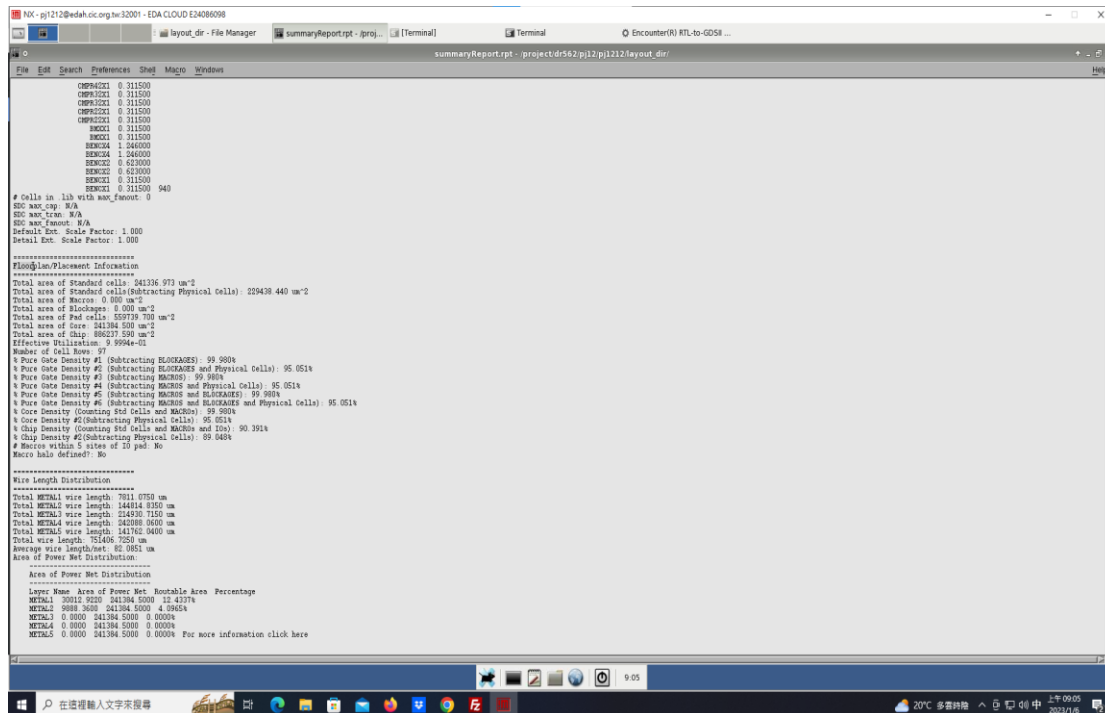
Design Rule Check (DRC) (2/3)



Design Rule Check (DRC) (3/3)



Layout Versus Schematic (LVS)



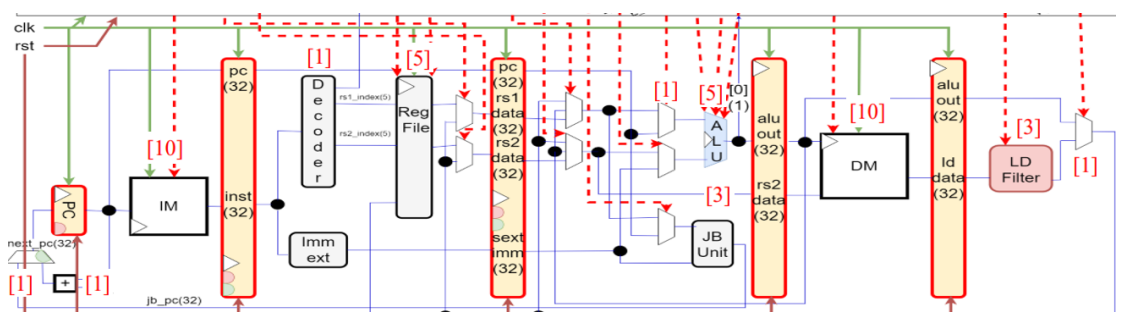
Layout area

Total area of chip : 886237 um²

Total wire length : 751406 um

7. Pipeline

- Computational Instruction : 10 + 1 + 5 + 1 + 5 + 1 + 1(Setup time) = 24
- Load : 10 + 1 + 5 + 1 + 5 + 10 + 3 + 1 + 1(Setup time) = 37
- Store : 10 + 1 + 5 + 1 + 5 + 1(Setup time) = 23
- Jump : 10 + 1 + 5 + 1 + 5 + 1 + 1(Setup time) = 24
- Branch : 10 + 1 + 5 + 1 + 5 + 1 + 1(Setup time) = 24
- => Critical Path : 37 (Load) => clock period need to ≥ 37 => max frequency = $\frac{1}{37}$

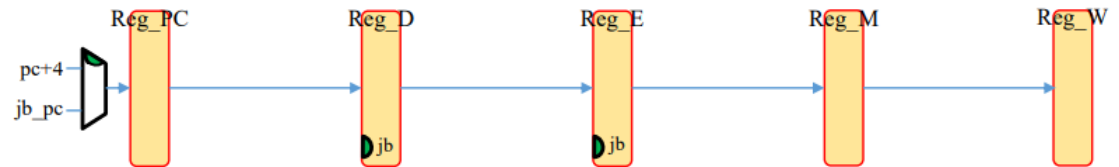


1. hazard

Control Hazard

- Problem: There is an indeterminate instruction flow in the pipeline

- Problem in Pipeline CPU: The subsequent instructions have entered the pipeline before the jump or branch is determined
- Solution: We implement a flush signal in the Controller to flush the wrong instructions in pipeline register(jb signal high for flushing)



- Example:

```

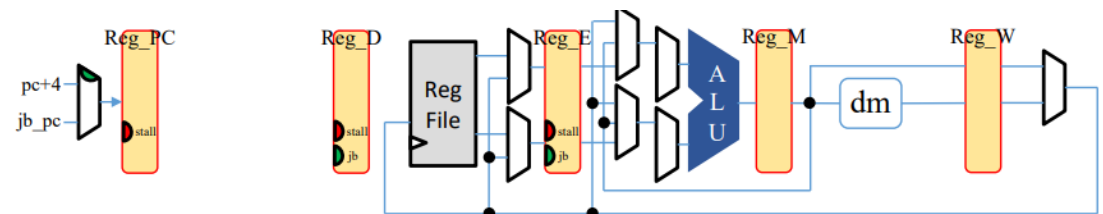
0 : li t0, 1
4 : li t1, 1
8 : beq t0, t1, equal
c : li s0, 4
10: jal x0, exit
equal:
14: li t2, 3
18: li t3, 4
1c: beq t2, t3, end
20: li s0, 5
24: jal x0, exit
end:
28: li s0, 6
exit:
2c: ret

```

cycle	IF	ID	EX	MEM	WB
0	li t0, 1				
1	li t1, 1	li t0, 1			
2	beq t0, t1, equal	li t1, 1	li t0, 1		
3	li s0, 4	beq t0, t1, equal	li t1, 1	li t0, 1	
4	jal x0, exit	li s0, 4	beq t0, t1, equal	li t1, 1	li t0, 1
5	li t2, 3	nop	nop	beq t0, t1, equal	li t1, 1
6	li t3, 4	li t2, 3	nop	nop	beq t0, t1, equal
7	beq t2, t3, end	li t3, 4	li t2, 3	nop	nop
8	li s0, 5	beq t2, t3, end	li t3, 4	li t2, 3	nop
9	jal x0, exit	li s0, 5	beq t2, t3, end	li t3, 4	li t2, 3
10	li s0, 6	jal x0, exit	li s0, 5	beq t2, t3, end	li t3, 4
11	ret	li s0, 6	jal x0, exit	li s0, 5	beq t2, t3, end
12	ret	nop	nop	jal x0, exit	li s0, 5

Data Hazard

- Problem: Unable to get the latest data for calculations
- Problem in Pipeline CPU: If the result data of an instruction needs to be writeback, subsequent instructions cannot get it until it completes
- Solution: We forwards the writeback data from MEM stage & WB stage to ID stage & EX stage



- Example:


```

0 : li t0, 1
4 : li t1, 1
8 : add t2, t0, t1
c : li t2, 3
10 : add t3, t2, t1
14 : lw t4, 0(t3)
18 : addi t5, t4, 1
1c : ret

```

cycle	IF	ID	EX	MEM	WB
0	li t0, 1				
1	li t1, 1	li t0, 1			
2	add t2, t0, t1	li t1, 1	li t0, 1		
3	li t2, 3	add t2, t0, t1	li t1, 1	li t0, 1	
4	add t3, t2, t1	li t2, 3	add t2, t0, t1	li t1, 1	li t0, 1
5	lw t4, 0(t3)	add t3, t2, t1	li t2, 3	add t2, t0, t1	li t1, 1
6	addi t5, t4, 1	lw t4, 0(t3)	add t3, t2, t1	li t2, 3	add t2, t0, t1
7	ret	addi t5, t4, 1	lw t4, 0(t3)	add t3, t2, t1	li t2, 3
8	ret	addi t5, t4, 1	nop	lw t4, 0(t3)	add t3, t2, t1
9	-	ret	addi t5, t4, 1	nop	lw t4, 0(t3)

Structure Hazard

- Problem: Hardware resources are not enough
- Problem in Pipeline CPU: Accessing memory at the same time by fetching instructions and loading data
- Solution: We duplicate SRAM as im & dm to solve memory access problem of simultaneous instruction fetch and load data

