VLSI System Design (Graduate Level)

Fall 2023

HOMEWORK I

REPORT

Must do self-checking before submission:

Compress all files described in the problem into one tar

All SystemVerilog files can be compiled under SoC Lab environment

All port declarations comply with I/O port specifications

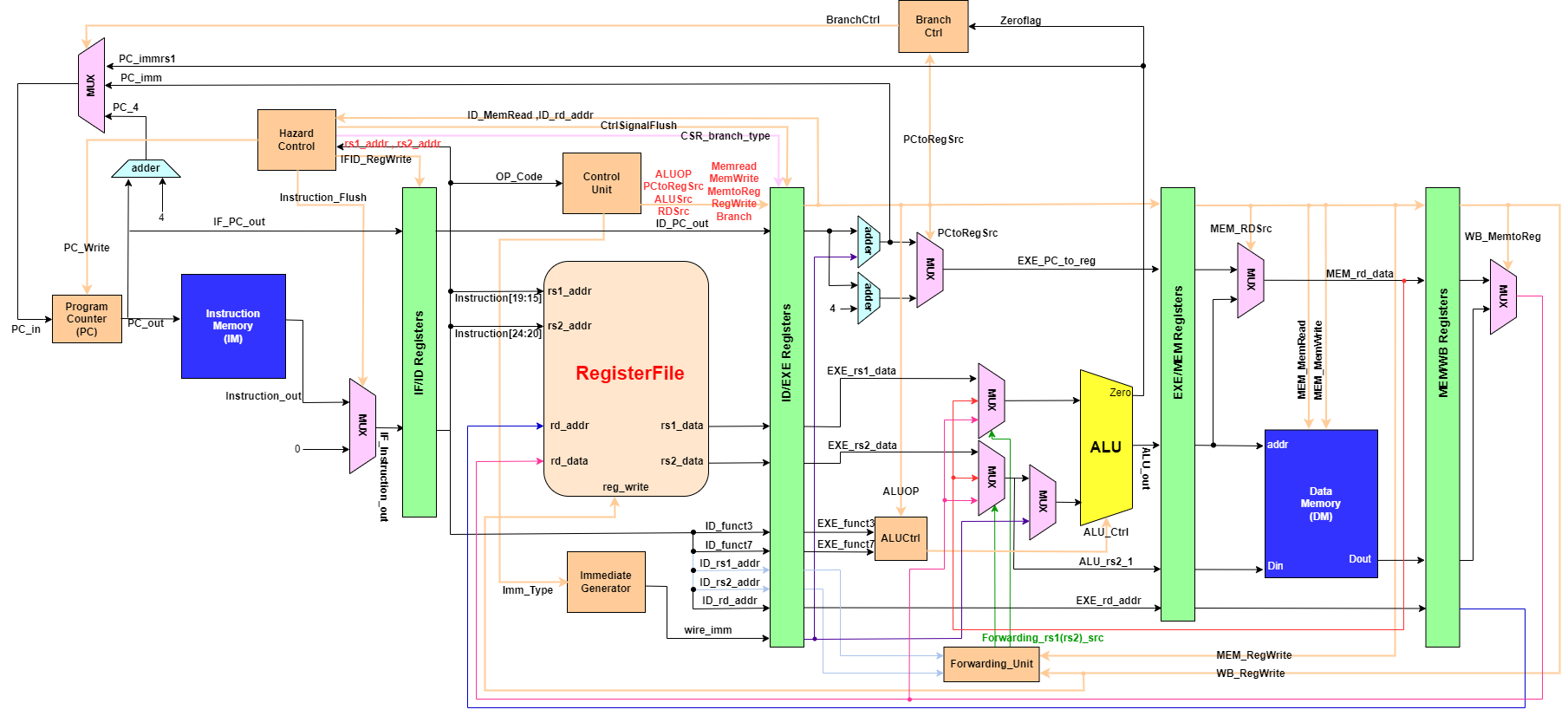
Organize files according to File Hierarchy Requirement

No any waveform files in deliverables

Student name: \_\_陳彥佑\_\_\_\_\_\_\_\_\_

Student ID: \_\_\_P76121411\_\_\_\_\_\_\_

1. My pipeline CPU’s block diagram



1. Summary:

In this homework, we need to complete the following goals.

(1). Use system Verilog to implement 5-stage pipeline RISC-V ISA pipeline CPU.

(2). Complete **45** instruction on my RISC-V pipeline CPU.

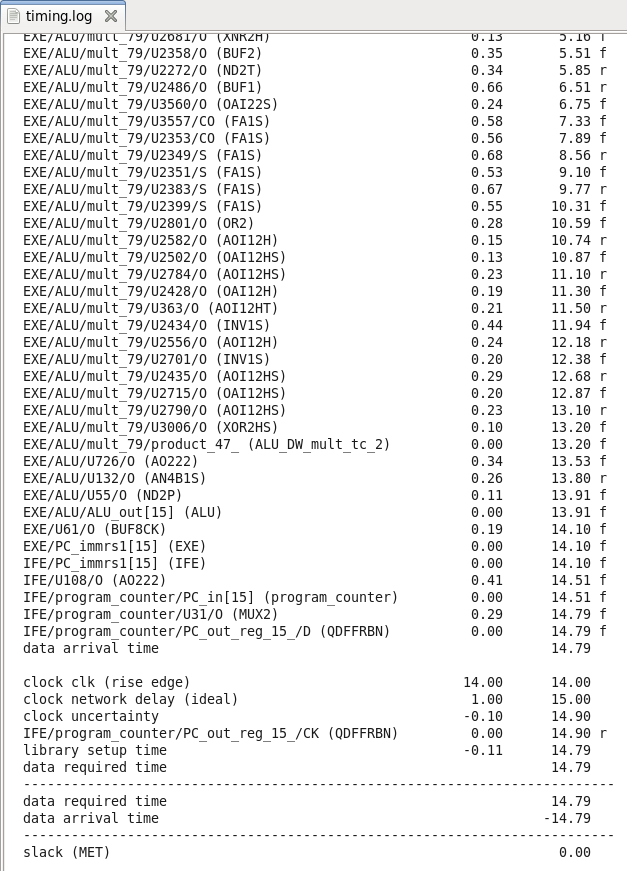
(3). Pass all RTL simulation and all gate-level simulation for **prog0, prog1, prog2, prog3, prog4, prog5**.

(4). Write the testing program by myself for **prog1, prog3**

(5). Use jaspergold superlint to check any **WARNING** and **ERROR** in my design.

**Report**

(1). Timing report



Clock cycle = 14

(2). Area

一張含有 文字, 螢幕擷取畫面, 軟體, 數字 的圖片

自動產生的描述

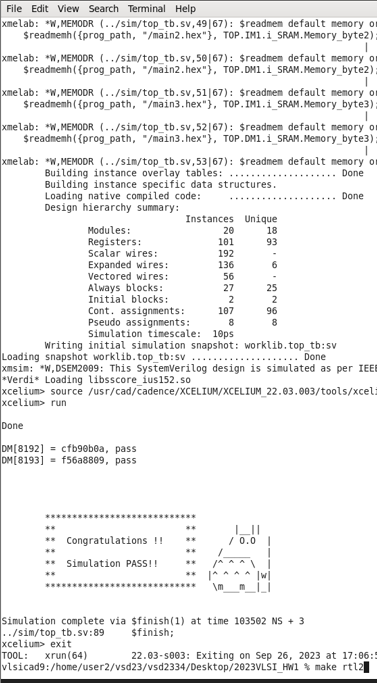
Area = 5750884.115251

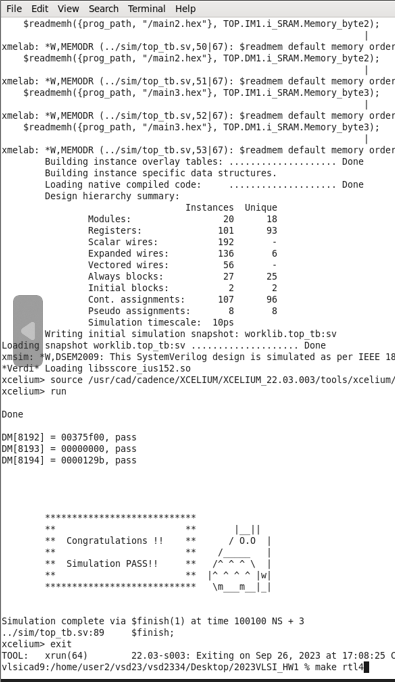
(3). Make rtl

一張含有 文字, 螢幕擷取畫面, Rectangle 的圖片

自動產生的描述一張含有 文字, 螢幕擷取畫面, Rectangle, 樣式 的圖片

自動產生的描述



一張含有 文字, 螢幕擷取畫面, 收據, 字型 的圖片

自動產生的描述

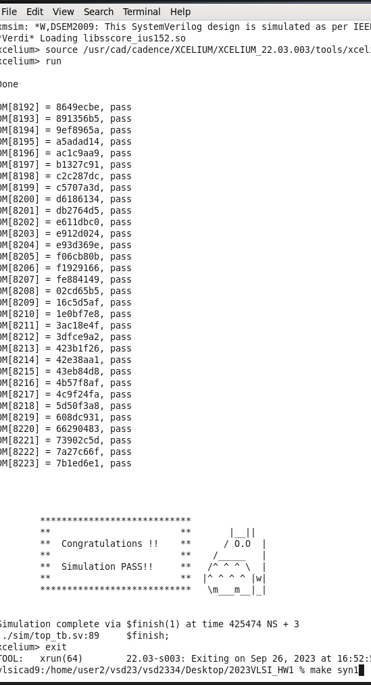
RTL results

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Prog0 | Prog1 | Prog2 | Prog3 | Prog4 | Prog5 |
| RTL | Pass | Pass | Pass | Pass | Pass | Pass |

ALL PASS

(4). Make syn

一張含有 文字, 螢幕擷取畫面, 文件, 樣式 的圖片

自動產生的描述

一張含有 文字, 螢幕擷取畫面, 收據 的圖片

自動產生的描述一張含有 文字, 收據, 螢幕擷取畫面, 文件 的圖片

自動產生的描述

一張含有 文字, 螢幕擷取畫面, 文件, 收據 的圖片

自動產生的描述一張含有 文字, 螢幕擷取畫面, 收據 的圖片

自動產生的描述

SYN results

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Prog0 | Prog1 | Prog2 | Prog3 | Prog4 | Prog5 |
| SYN | Pass | Pass | Pass | Pass | Pass | Pass |

ALL PASS

(5). Jaspergold superlint

(A). Number of lines of RTL code: 1382.

(B). Final Result of superlint is 11.

一張含有 文字, 螢幕擷取畫面, 軟體, 電腦 的圖片

自動產生的描述

1. Core detail

(1). In-order 5-stage pipeline CPU with Forwarding Unit and Hazard control.

(2). Architecture with separate instruction and in/output.

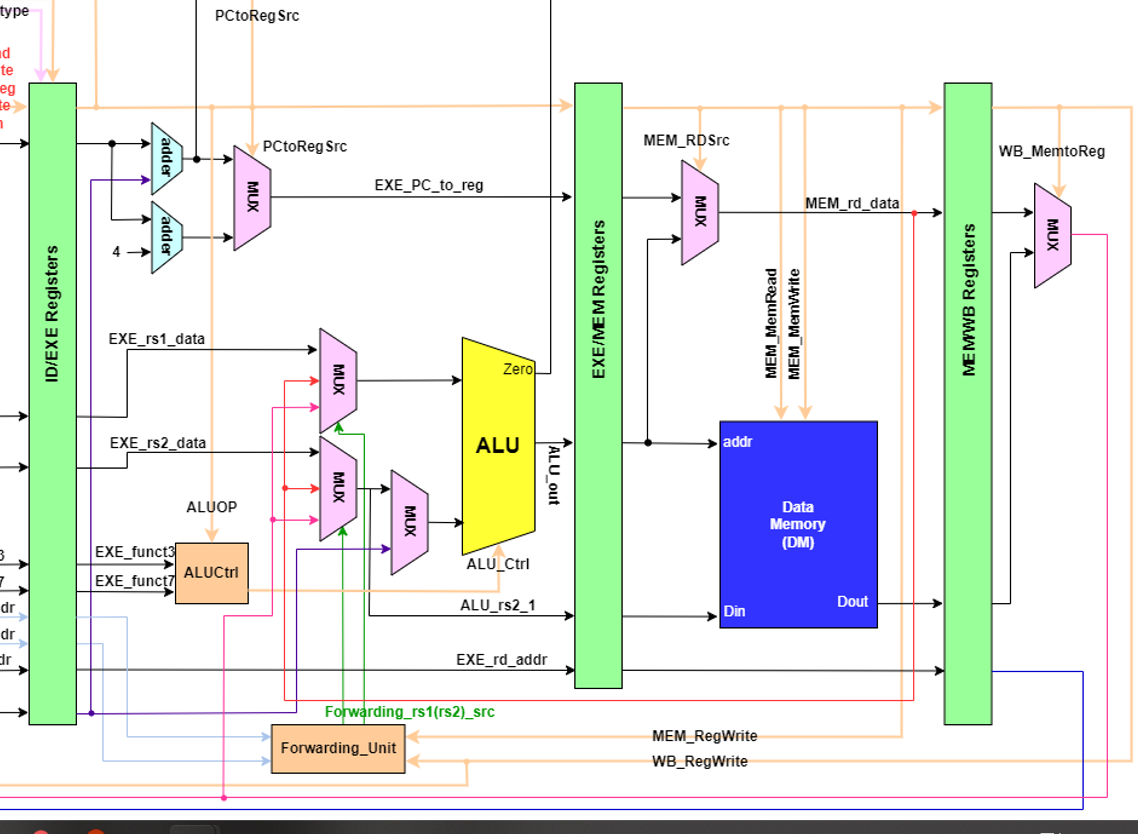
(3). **45** instructions should complete.

1. I have found Some issue in implementation and **Lessons learned**:

(1). **Data Hazard** : There are usually have three type of data hazard. **Normal data hazard**, **load use data hazard** and **control hazard**.

(I). ***Normal data hazard***:

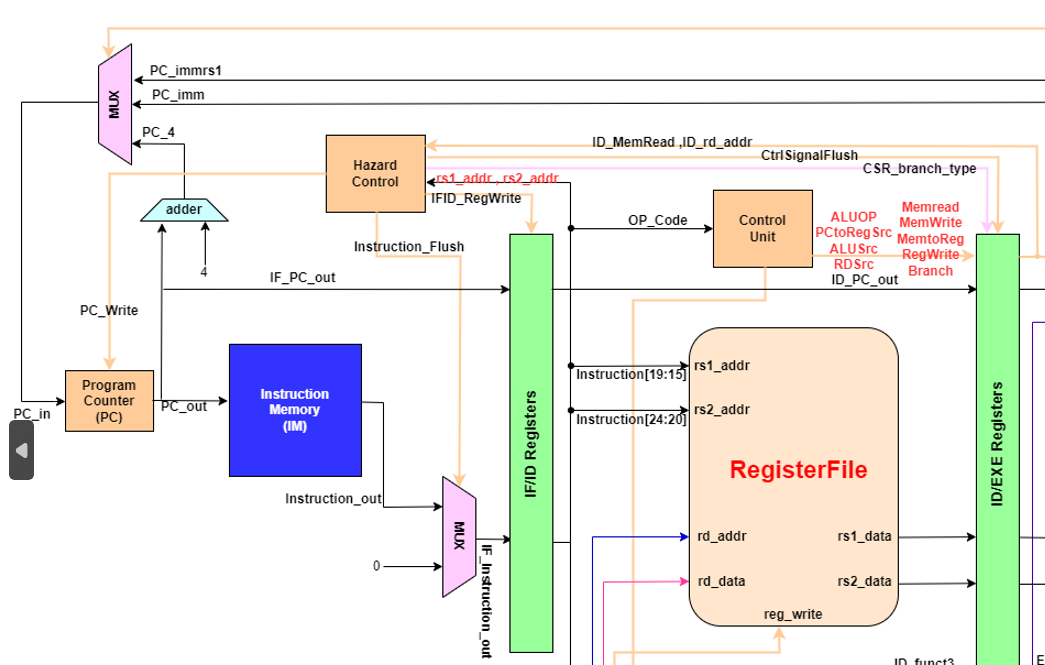
Slove: we can easily add a forwarding unit to overcome this problem.



We can see the orange arrow (from MEM stage) and the green arrow (form WB stage) point to. Those two data and normal data come from ID stage can use forwarding unit I wrote to determine the register in instruction are depended (EXE/MEM or MEM/WB), and choose the correct register data to EXE stage. This can prevent the error occurs.

(II). ***Load use data hazard***:

Want to finish this question, in addition to Forwarding unit, I also need to have a new unit, Hazard control unit.



The Hazard control unit will receive the signal which come from the ID stage and EXE stage. The green arrow is come from ID stage (rs1\_addr, rs2\_addr) and the orange arrow is come from EXE stage (MemRead, rd\_addr).

We see the MemRead is wrote to hazard control unit, because it can through the addr to determine the data we want to push into register is depended on the other register which other instruction used.

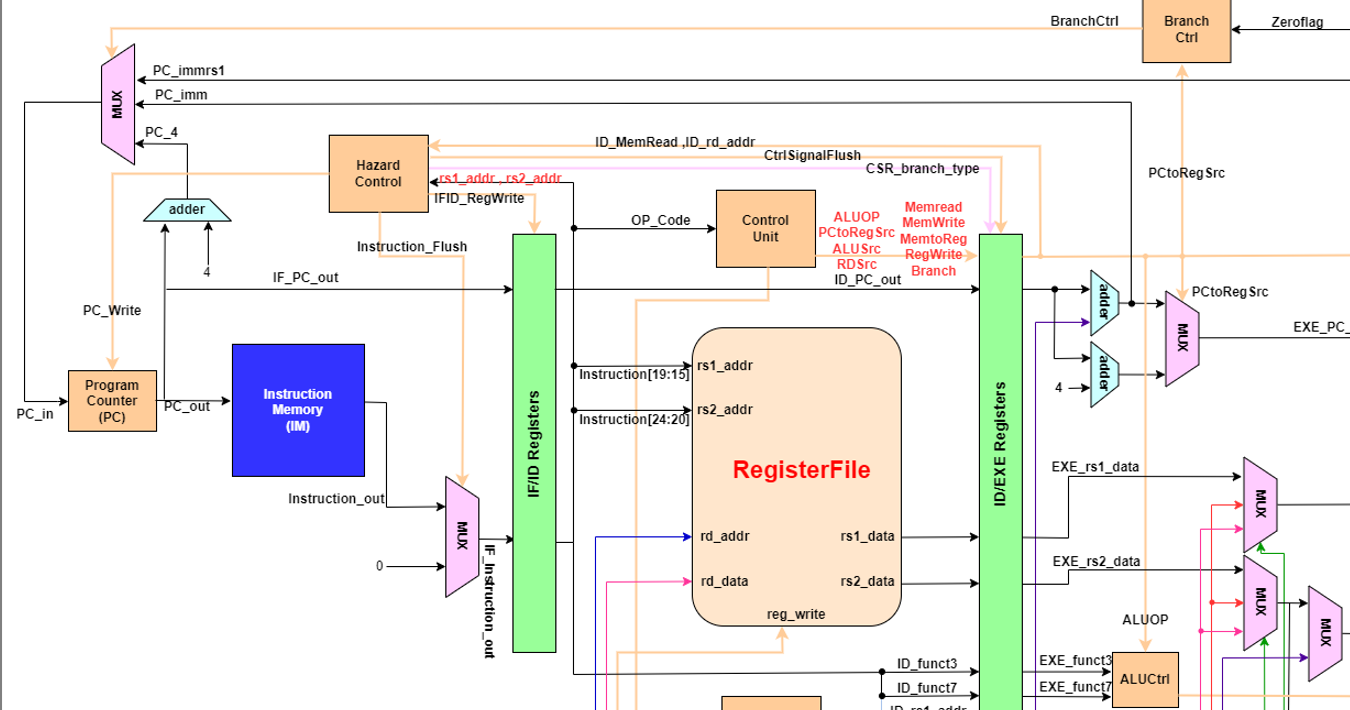
When load use data hazard occurs, we will keep the instruction and those data in IF/ID register until next cycle and flush some signal (MemRead, MemWrite, Breanch, RegWrite) in ID/EXE register, so the next cycle the instruction send to EXE stage will become **NOP**, and an important detail is our Program Counter can’t refresh, need to stay in same place.

Usually, this way we still need to stall 2 cycles, but we can add the above method, like normal data hazard, use Forwarding unit forward the data WB stage have to EXE stage. This can reduce a stall cycle.

(III). ***Control Hazard***:

If Branch condition is active, we need to flush the instruction in IF, ID stage to void the error happen.

I use the same Unit, Hazard control unit to solve this problem.



When we notice that we need to Branch, we can through instruction\_Flush and CtrlSignalFlus signal to flush the instruction in ID, EXE stage. That will avoid the instruction error.

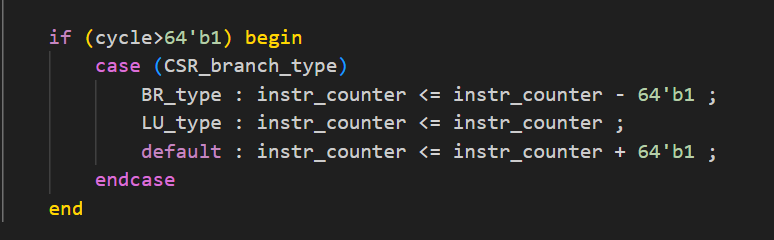
(2). Posedge clk and negedge clk problem:

Register File active need to be negedge or it will get the wrong data.

(3). The CSR instruction – instruction count:

The CSR instruction have some problem when I first write.

Because I write the decide logic in Hazard control unit, so we can know the instruction count will not match the actual instruction count. When I send the **CSR\_type** to EXE stage to determine the number of instruction, will shift **2** away.



If there come from a Branch type it will calculate instr\_counter more than once that our need, so we need to minus one.

If come from a Load use type it actual equals to the counter in EXE stage, so we don’t need to change.

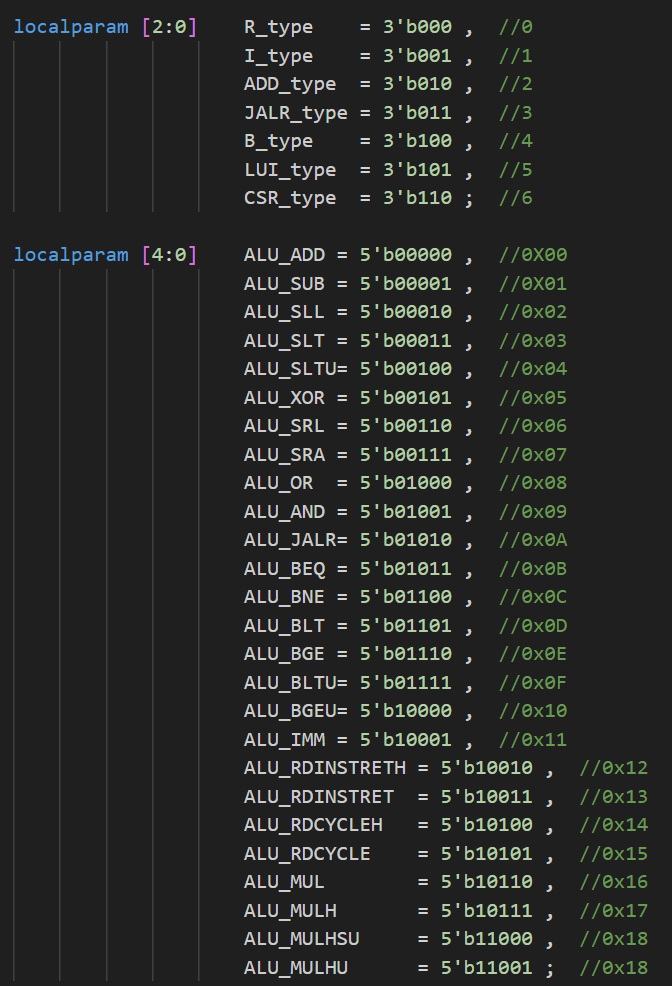
And default (normal instruction) that is add one to instr\_counter.

1. nWave verify:

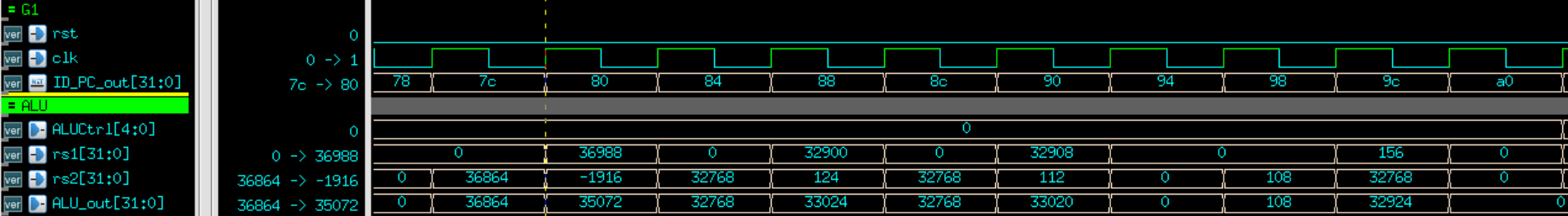
Next, I will categorize and explain some instruction in different type.

(1). R-type:

We can see following picture, the correspond ALU operation is based on the signal which is sent by ALUCtrl.



(A). ADD



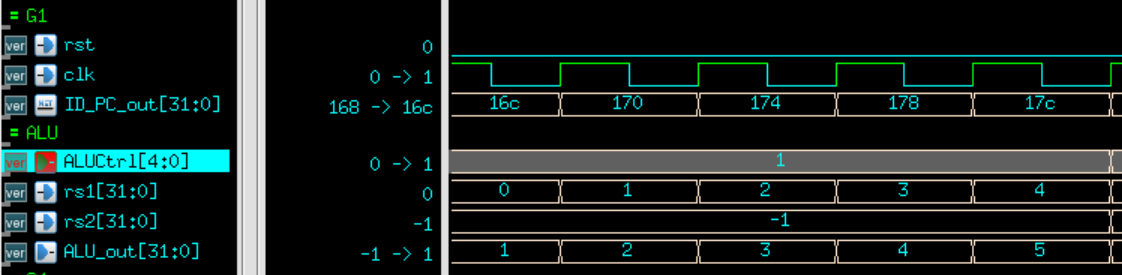
We can see that rs1 and rs2 are showed in decimal.

ALUCtrl is 00000, it means ADD.

For example: 36988 + -1916 = 35072 that correct.

And 32908 + 112 = 33020 is also correct.

(B). SUB

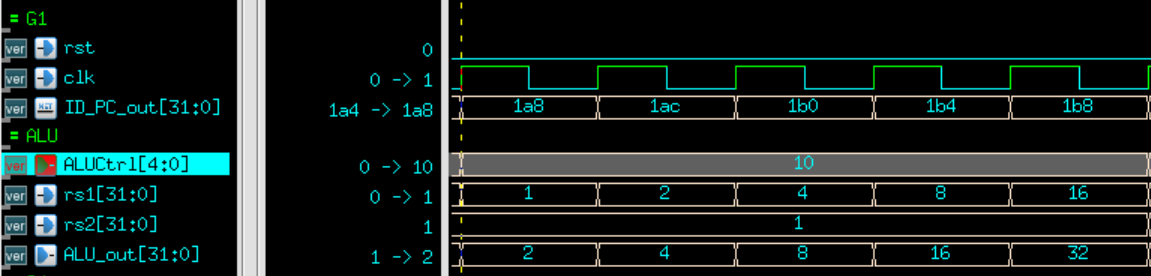
 We can see that rs1 and rs2 are showed in decimal.

ALUCtrl is 00001, it means SUB.

For example: 1 – (-1) = 2 , correct.

And 2 – (-1) = 3 , correct.

(C). SLL



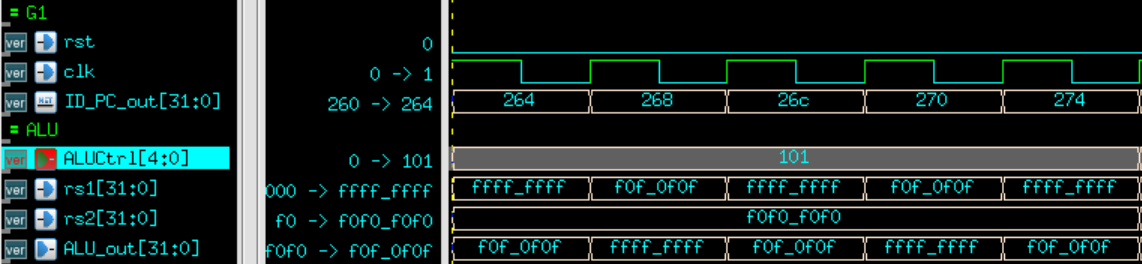
We can see that rs1 and rs2 are showed in decimal.

ALUCtrl is 00010, it means SLL.

For example: 1 after SLL rs2 is 2, correct.

And 4 after SLL rs2 is 8, correct.

(D). XOR

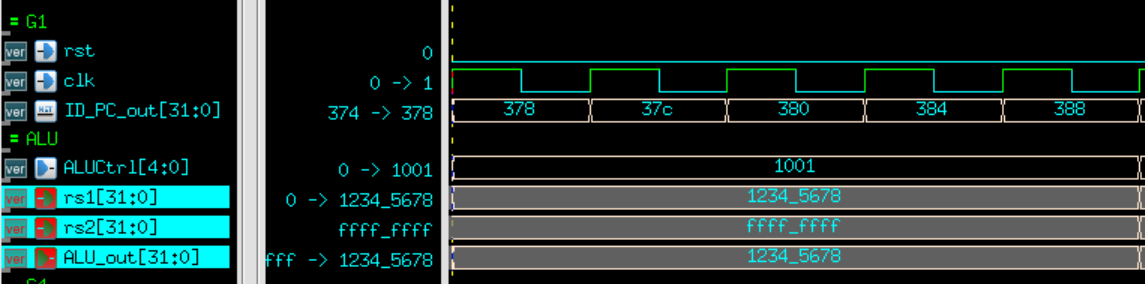


We can see that rs1 and rs2 are showed in hex.

ALUCtrl is 00101, it means XOR.

0f0f0f0f ^ f0f0f0f0 = ffffffff, correct.

(E). AND



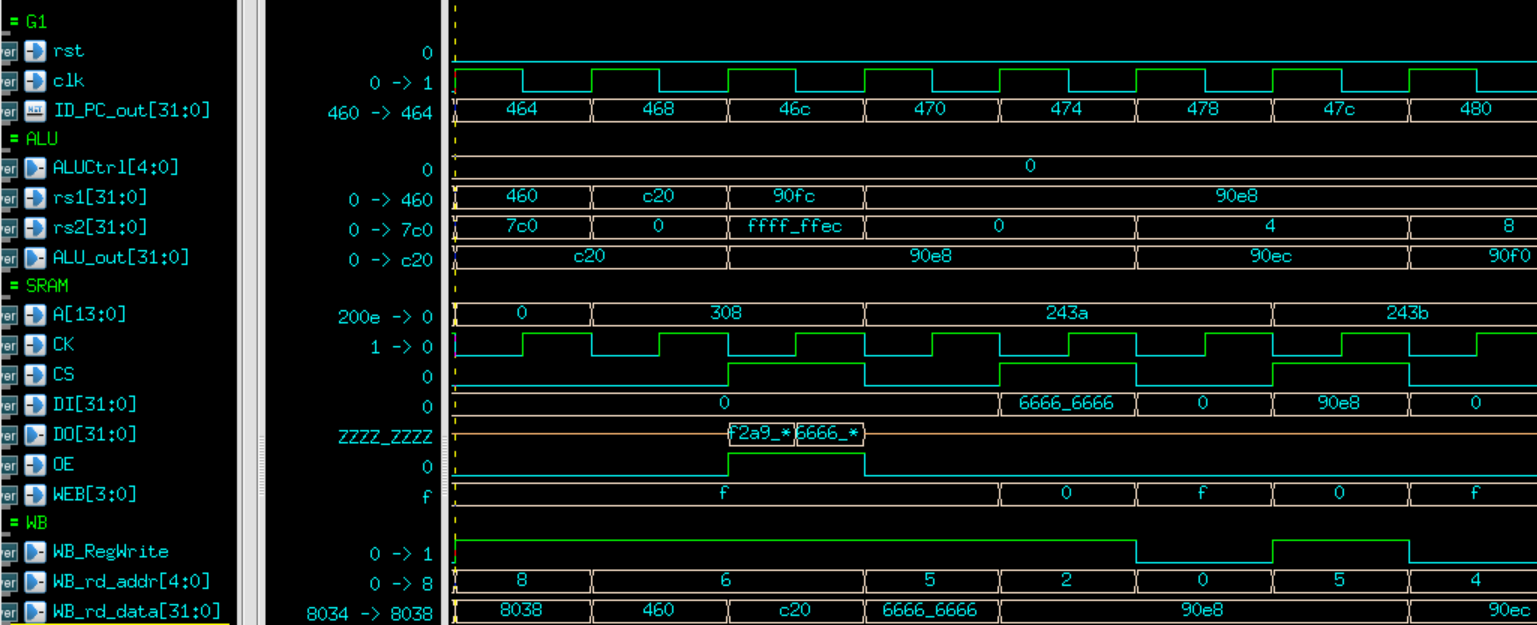
We can see that rs1 and rs2 are showed in hex.

ALUCtrl is 01001, it means AND.

1234\_5678 & ffff\_ffff = 12345678 is clearly correct.

(2). I-type

(A). LW



Rs1 is 3104 and rs2 is 0, so we will access data in address 3104 on data memory.

The yellow circle represents the LW instruction in MEM stage.

The A[13:0] is taken by the previous clock, ALU\_out[15:2] is equal to 308.

Because we need to read, so we set CS to high, WEB set to f.

An important setting is that the CK is the inverse of clk, this setting is used to take the correct data in data memory.

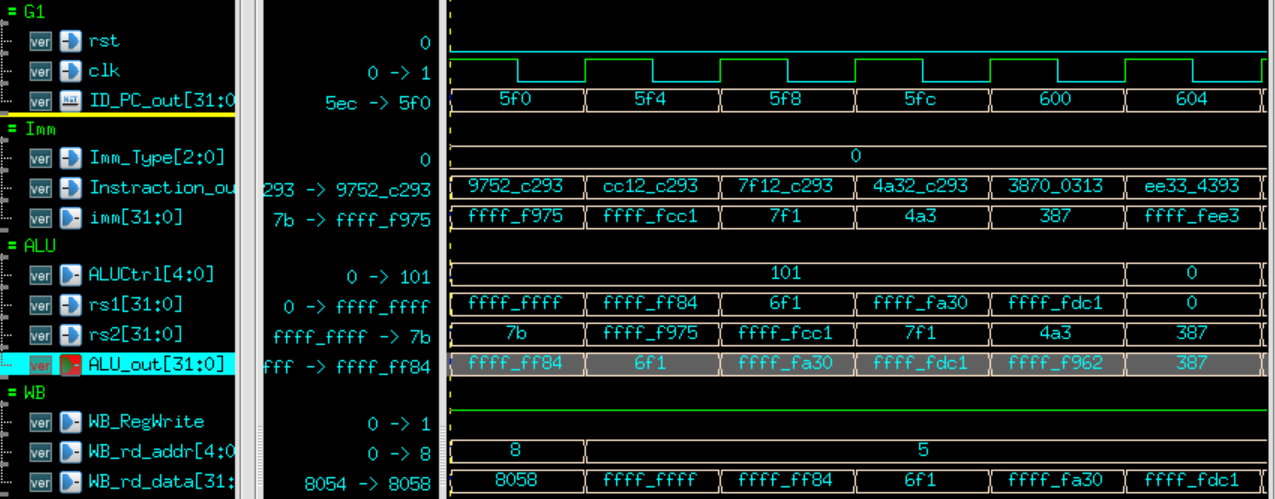
We can see the light blue circle, we get the data 6666\_6666.

And set OE to high let our result send back to CPU.

Finally, the purple circle is LW instruction in WB stage, and write back 6666\_6666 to register.

We need to active RegWrite to high and also send back rd\_addr and re\_data..

(B). XORI



We can see the red circle, XORI instruction is in ID stage, because ALUCtrl is 00101.

And Imm\_Type is 0, we can know this is I-type, so imm will be extension.

imm = {{20{instruction\_out[31]}},instruction\_out[31:20]}.

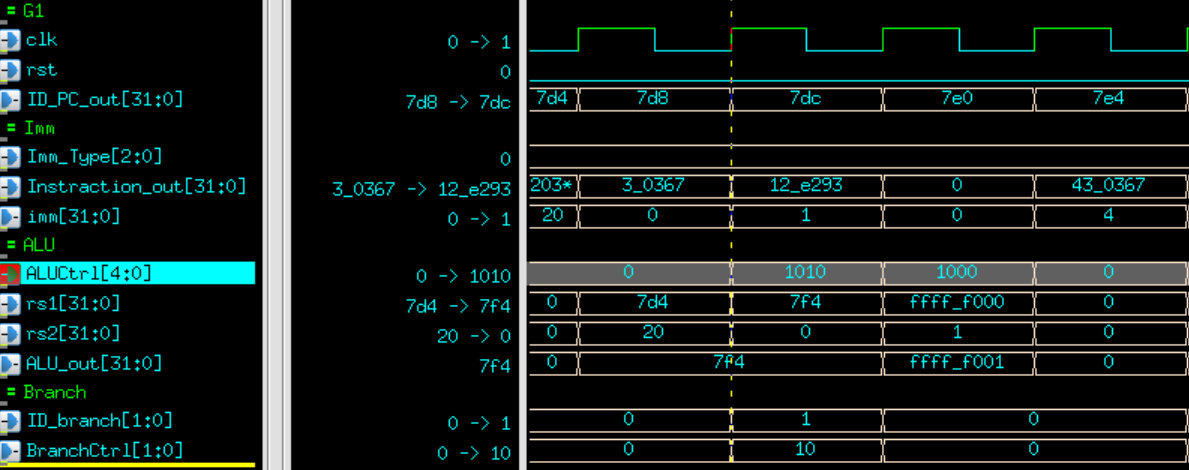
So imm become ffff\_f975.

See the yellow circle, rs1 is ffff\_ff84, rs2 is ffff\_f975.

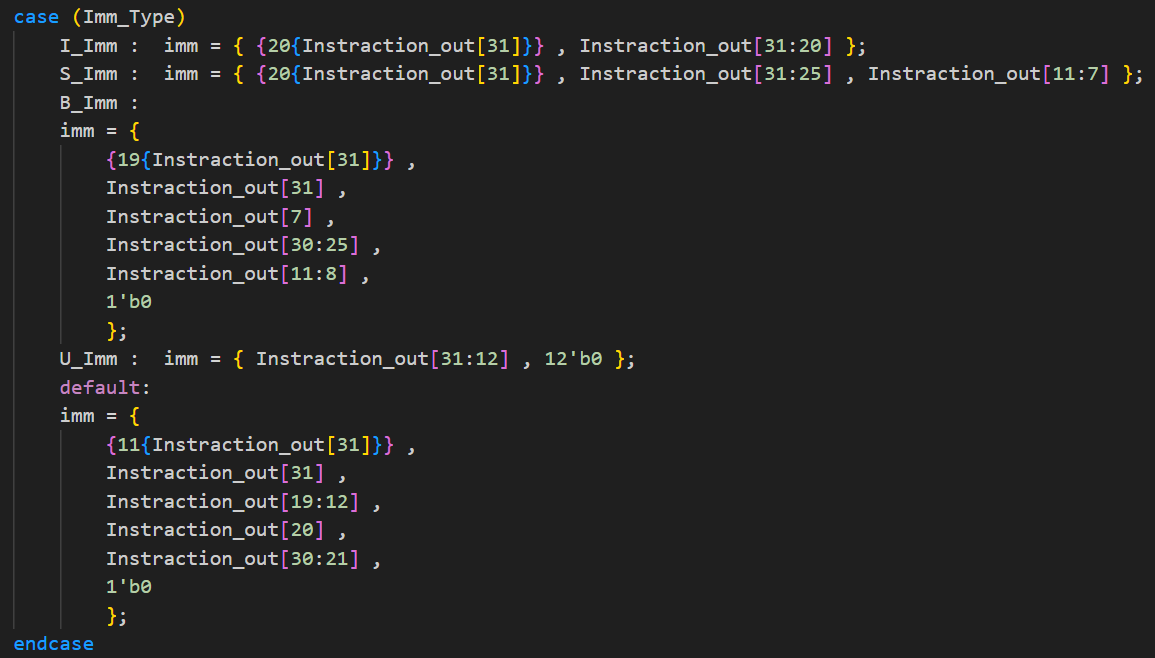
The answer is 0000\_06f1.

Finally, we can see the purple circle, it means the data in WB stage is wrote back to register.

(C). JALR







We can see red circle, it means JALR instruction is in ID stage.

By imm signed extension, imm is 0.

The yellow circle represent JALR instruction in EXE stage because ALUCtrl is 01010. And is calculating the rs1 + rs2. In this case is 7f4 + 0 = 7f4 .

Because this is JALR instruction, ID\_branch is set to 1.

BranchCtrl is show next PC should choose IMMRS1

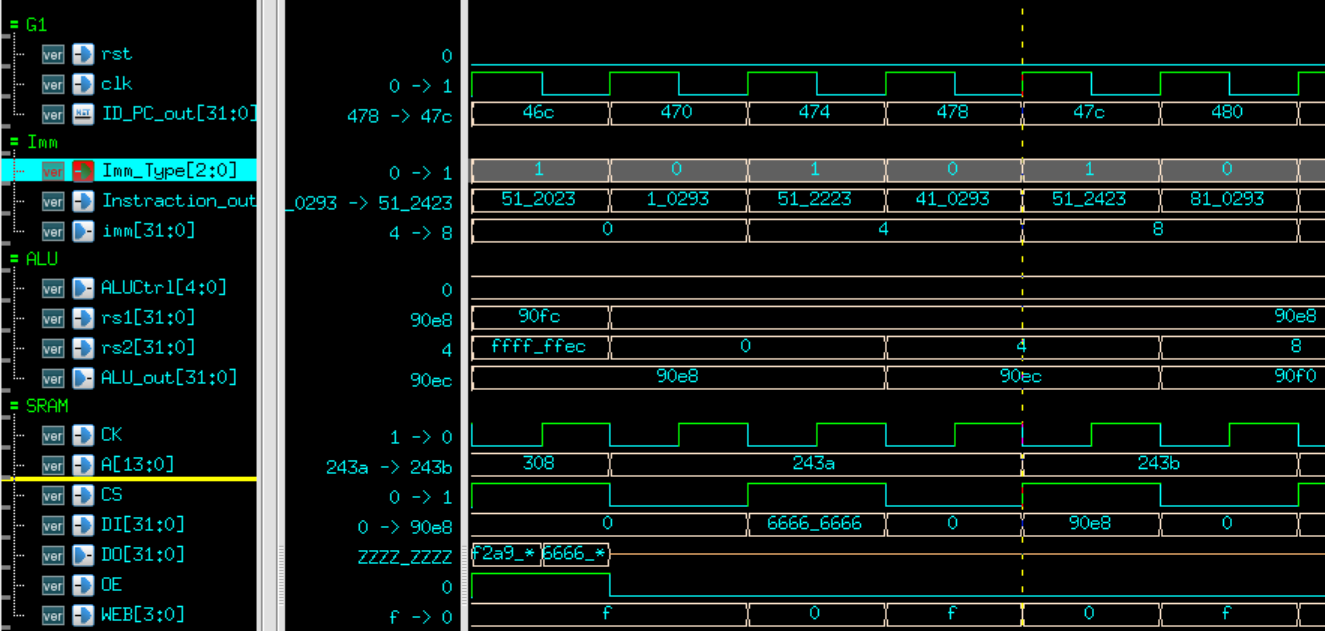
(rs1+Imm).

And the hazard control will work in the following operation.

Please reference the detail ahead in this doc.

(3). S-type

(A). SW



Red circle is SW instruction in ID stage.

Br Imm\_Type, we know it is S-type, and imm will be generate. Imm = 0.

The yellow circle represent that SW instruction is in EXE stage. Doing ADD operation. 90e8 + 0 = 90e8.

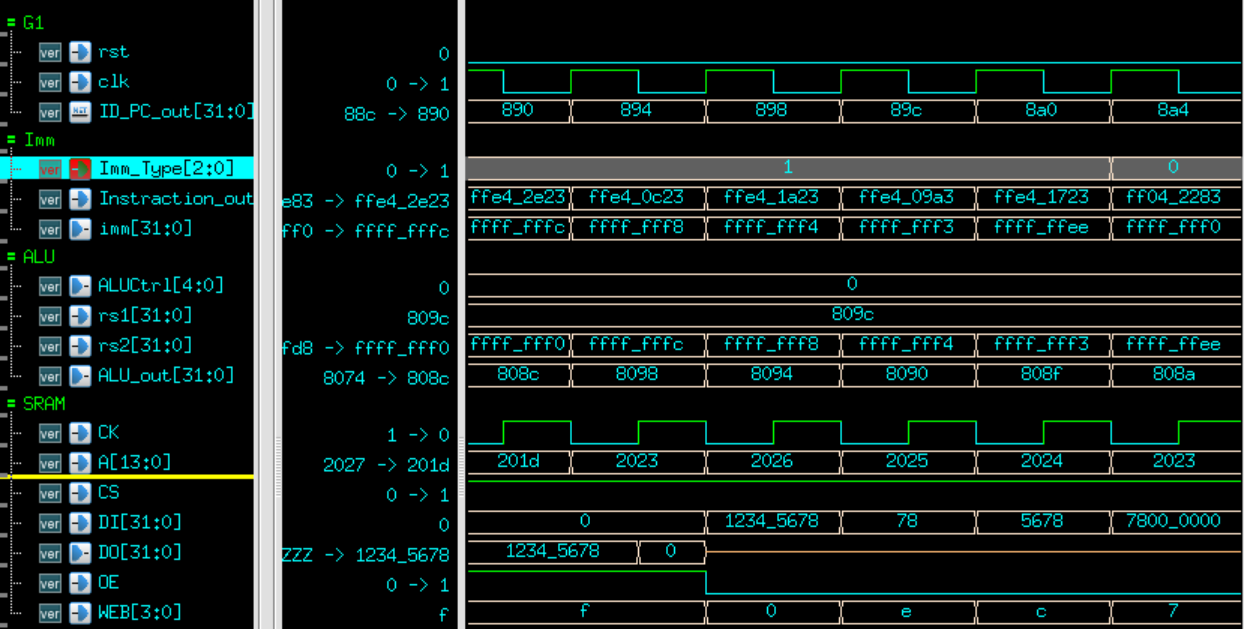
The A[13:0] is taken by the previous clock, ALU\_out[15:2] is equal to 243a. And setting CS to high, WEB is 0, OE is 0

.

Importantly, set WEB to 0.

Finally, put the data into Din.

(B). SB



Red circle is SB instruction in ID stage.

Br Imm\_Type, we know it is S-type, and imm will be generate. Imm = ffff\_fff8.

The yellow circle represent that SB instruction is in EXE stage. Doing ADD operation. 809c + ffff\_fff = 8094.

The A[13:0] is taken by the previous clock, ALU\_out[15:2] is equal to 2025. And setting CS to high, WEB is 0, OE is 0

Importantly, set WEB to e. Because this instruction is SB.

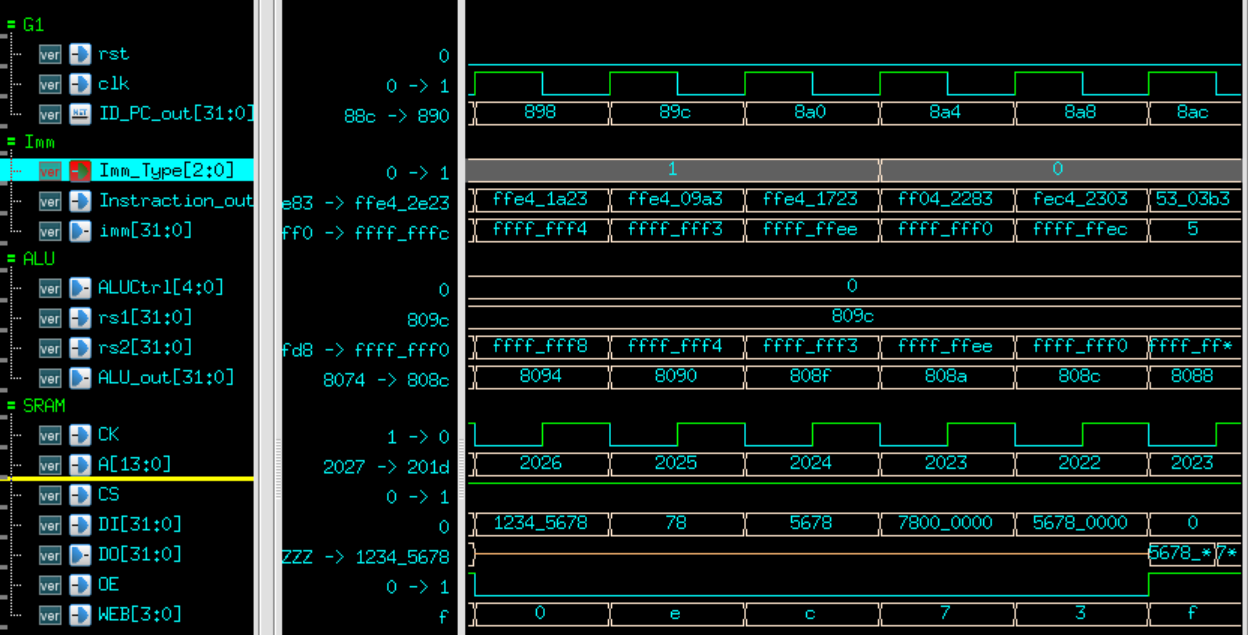
In SB instruction, WriteEnable and WriteData is based on the rightest two bits of write address.

WriteEnable [WriteAddr[1:0]] = 1’b0

WriteData[{WriteAddr[1:0], 3’b0}+:8] = rs2\_data[7:0]

Finally, put the data into Din.

(C). SH



Red circle is SH instruction in ID stage.

Br Imm\_Type, we know it is S-type, and imm will be generate. Imm = ffff\_fff4.

The yellow circle is represent that SH instruction is in EXE stage. Doing ADD operation. 809c + ffff\_fff4 = 8090.

The A[13:0] is taken by the previous clock, ALU\_out[15:2] is equal to 2024. And setting CS to high, WEB is 0, OE is 0

.

Importantly, set WEB to C. Because this instruction is SH.

In SB instruction, WriteEnable and WriteData is based on the rightest two bits of write address.

WriteEnable [WriteAddr[1],1;b0] = 2’b00

WriteData[{WriteAddr[1], 4’b0}+:16] = rs2\_data[15:0]

Finally, put the data into Din.

(4). B-type

(A). BEQ



Red circle is BEQ instruction in EXE stage.

ALUCtrl is 01011. Represent that ALU check rs1 = rs2 ?  
If equal output 1 ,else output 0.

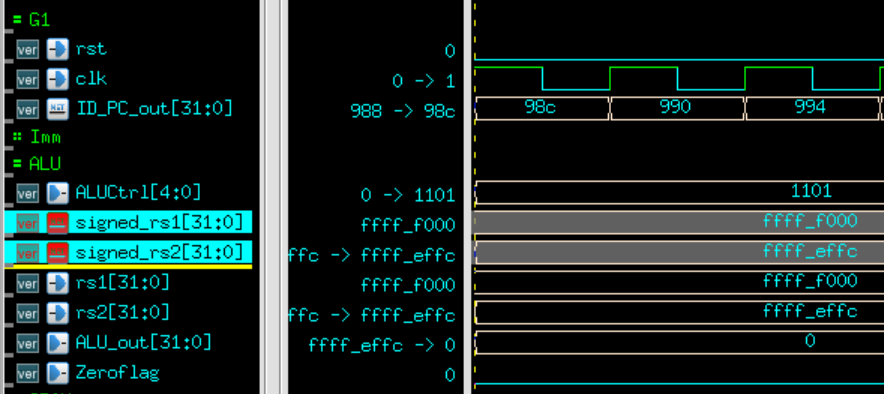
If output is 0, CPU will not be influence.

If output is 1, that cause branch will generate control hazard.

In this case, ffff\_f000 not equal to 0000\_0ff0, so output is 0

Please reference the detail ahead in this doc.

(B). BLT



Red circle is BLT instruction in EXE stage.

ALUCtrl is 01101. Represent that ALU check

Signed\_rs1 < signed\_rs2 ?  
If equal output 1 ,else output 0.

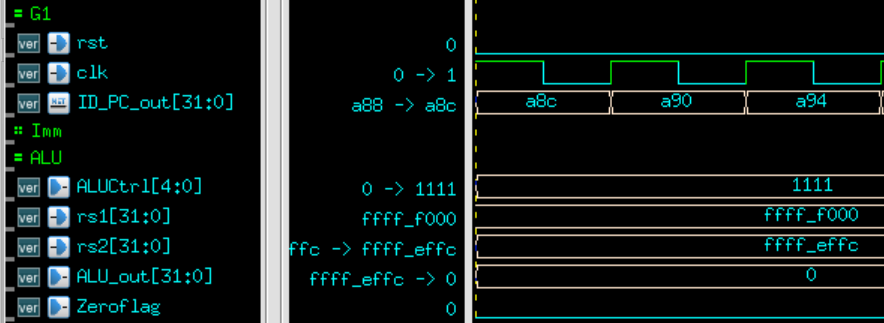
If output is 0, CPU will not be influence.

If output is 1, that cause branch will generate control hazard.

In this case, ffff\_f000 > ffff\_effc , so output is 0.

Please reference the detail ahead in this doc.

(C). BLTU



Red circle is BLTU instruction in EXE stage.

ALUCtrl is 01111. Represent that ALU check rs1 < rs2 ?  
If equal output 1 ,else output 0.

If output is 0, CPU will not be influence.

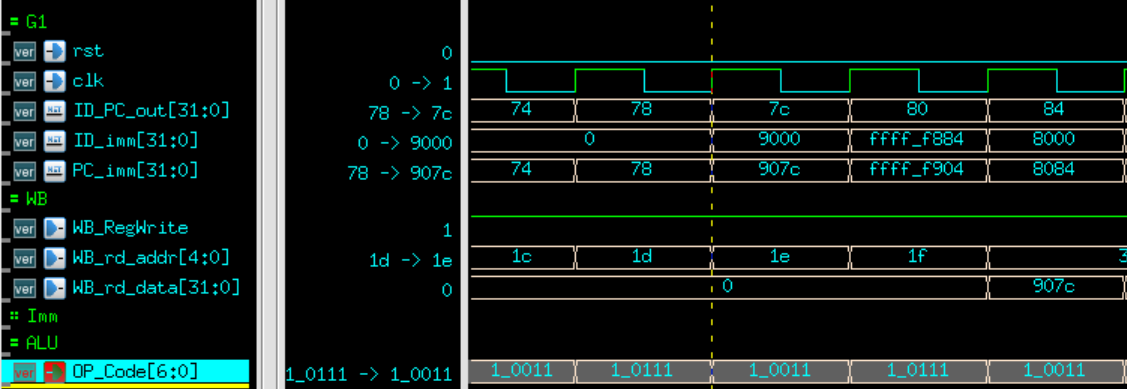
If output is 1, that cause branch will generate control hazard.

In this case, ffff\_f000 > ffff\_effc , so output is 0.

Please reference the detail ahead in this doc.

(5). U\_type

(A). AUIPC



Red circle is PC + imm.

The light blue circle can know that data write back to correspond register in WB stage.

(B). LUI



Red circle is LUI instruction in EXE stage.

OP\_Code is 011\_0111 is LUI instruction.

Represent that bypass rs2 to be result.

Light blue circle can clearly see that result.

(6). J-type

(A). JAL

一張含有 螢幕擷取畫面, 多媒體軟體, 軟體 的圖片

自動產生的描述

Red circle is JAL instruction in ID stage.

Imm\_type is 4, this is J-type.

By definition, J-type signed extension is

imm[20|10:1|11|19:12]

so imm is 8.

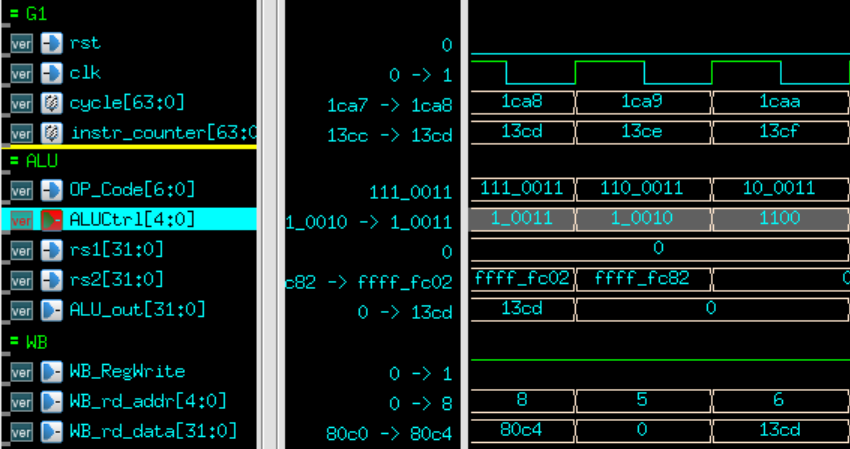
Yellow circle represent JAL instruction in EXE stage.

Wire\_PC\_4 calculate PC + 4, wire\_PC\_imm calculate PC+IMM.

Light blue circle can see PC+4 is store in register.

(7). CSR-type

(A). RDINSTRET



ALUCtrl id 10011, and this instruction, red circle, is RDINSTRET in EXE stage.

Yellow circle, rd = instr\_counter[31:0] is wrote in WB stage.

Please reference the detail ahead in my lesson learn.

(B). RDCYCLEH

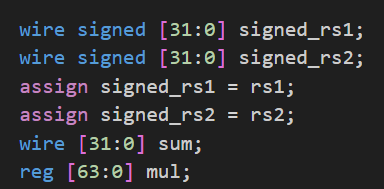


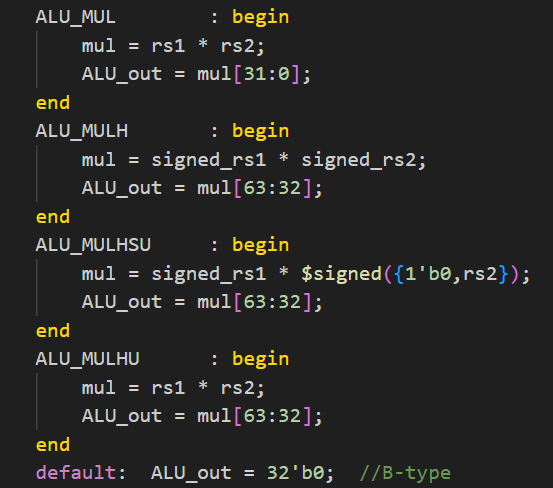
ALUCtrl id 10100, and this instruction, red circle, is RDCYCLEH in EXE stage.

Yellow circle, rd = cycle [63:32] is wrote in WB stage.

Please reference the detail ahead in my lesson learn.

(8). MUL instruction



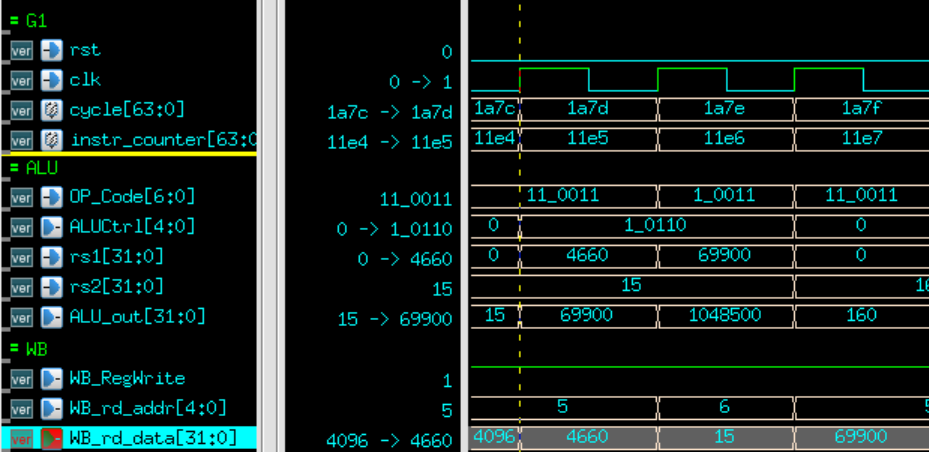


We can see the detail in ALU.sv .

How to write the correct MUL method.

Take the example in MUL.

(A). MUL



Red circle is MUL instruction in EXE stage.

For there can clear read, rs1 and rs2 are set to decimal.

We can clearly see 4660 \* 15 = 69900.

Yellow circle can see the data write back to register.

You need to pay attention to ALU\_MULHSU, because of

Signed \* unsigned will cause the signed bit error.

So, I overcome this problem by using $signed({1’b0,rs2})

to solve the bit problem.

1. Final

In this assignment, I learned how to interact with RAM module and implement 45 RISC-V instruction.

Using the verification tool is also a new experience, my design can be identified and corrected.