VLSI System Design (Graduate Level)
Fall 2024

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: _____王華昀_____
Student ID: __N26134308_____

一. 系統架構圖

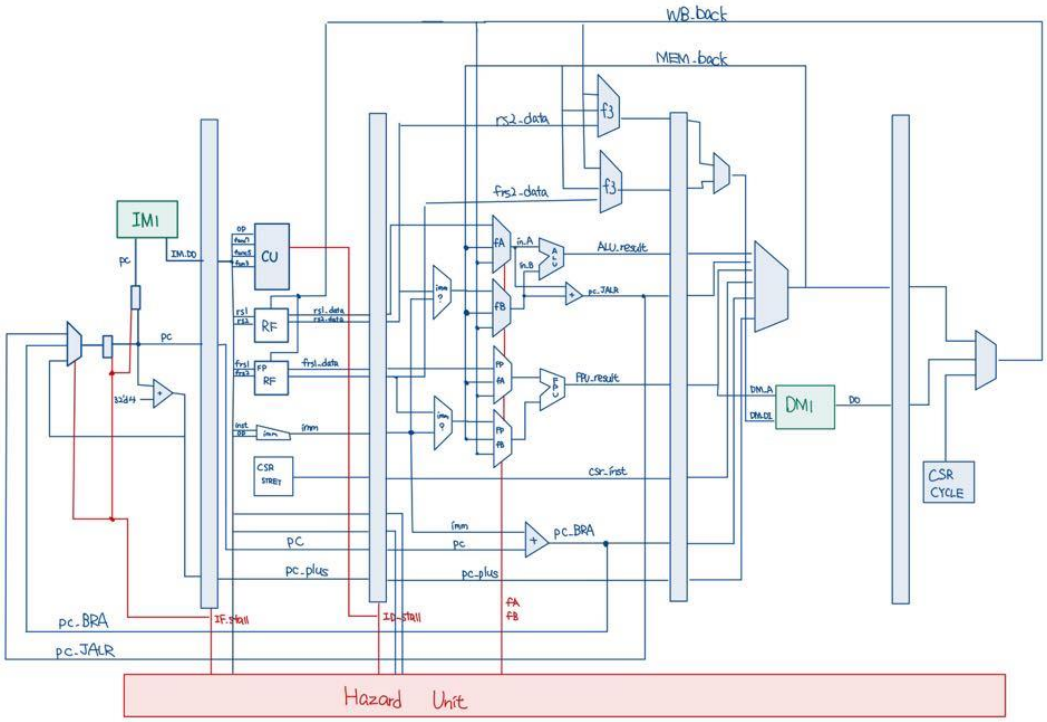


圖 1 系統架構圖

上圖為本次作業設計之五階 RISCV 架構圖，主要分為 IF、ID、EXE、MEM、WB 以及 Hazard Unit，以下分別介紹各個階級的架構設計及更詳細的 I/O。

橘色: Input　綠色: Output　紫色:Hazard Unit Signal　紅色: Control signal

1. IF(Instruction Fetch)

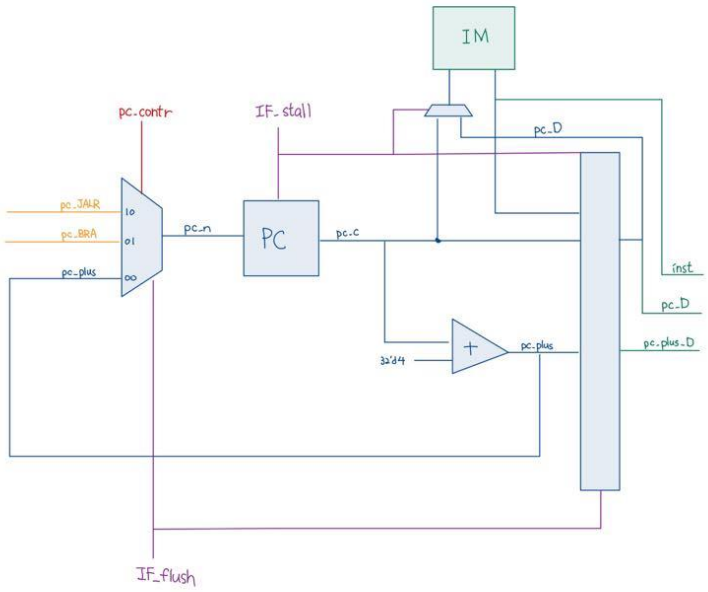


圖 2 IF 階級架構圖

IF 階級主要是將記憶體(IM)之地址傳送至 IM，並讀取該指令，再將該指令傳送至 ID 階級進行解碼，這階級我首先用一個 PC MUX 單元來做 PC 的選擇，當 EXE 階級接收到 Branch 及 Jump 發生時，會將其 PC 傳送回 IF 階級進行 PC 的變動。而本次作業中較為特別的是每當有 rst 或是 PC 跳動後的讀取，都會延遲一個 clk cycle 才會將記憶體之指令讀取出來，解決的方法是加上一個 IF stall 訊號用來控制送入 IM 的地址，若上述狀況發生時，會有 IF flush 訊號將暫存重製，並且 IM 讀取出的指令就直接傳送至 ID 端，並同時傳送至 IFID 暫存做備份，讓傳送出來的指令不會因為延遲而影響到 ID 的解碼。

橘色: Input　綠色: Output　紫色:Hazard Unit Signal　紅色: Control signal

2. ID(Instruction Decode)

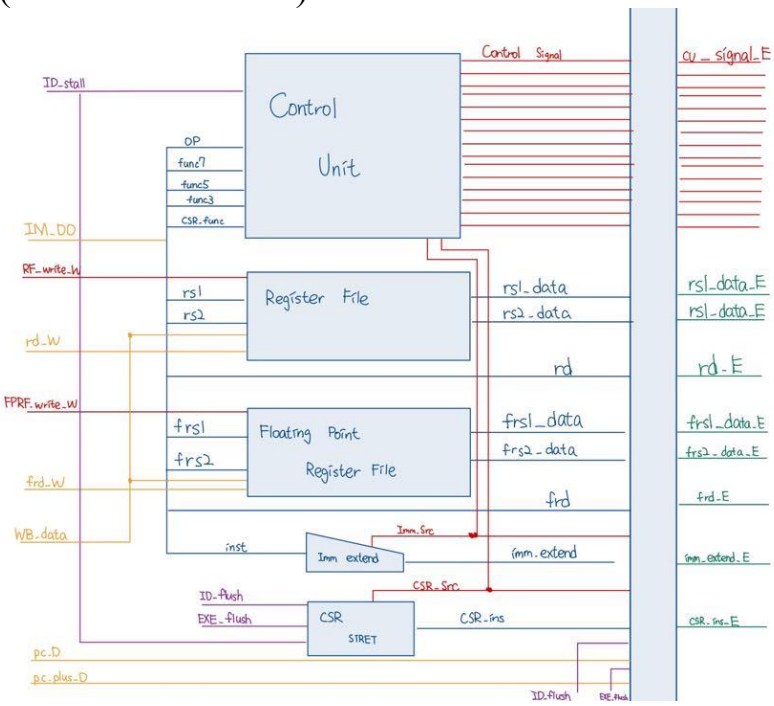


圖 3 ID 階級架構圖

ID 階段主要是解碼從 IF 階段傳來的指令，並讀取對應的寄存器值。其中可以將 ID 階段分為四個部分。第一部分為 CU(control Unit)，CU 主要是將從 IF 傳送過來的指令進行解碼，並根據指令的不同送出特定控制訊號。第二部分為 RF(Register File)、FPRF(Floating Point Register File)，這兩個暫存器主要是要接收從 WB 階級傳送回來的資料，並且依照指令對應的地址讀取出暫存資料供 EXE 階級進行計算，這部分由於會有同時寫跟讀的事件發生，因此在設計暫存器時我是將讀取設在 posedge clk 而寫入設在 negedge clk 作為區分。第三部分為 Imm extend Unit，這部分是依照指令的不同來決定 imm 用不同的方式擴展。第四部份為 CSR STRET，這部分是用來計算系統執行的總指令數量，會放在這階段是因為由於在 EXE 後的階段會因為有 Branch 及 Jump 的發生，而導致有 NOP 的產生，而當有上述情況發生時，由於 IM 會延遲一個 clk 才將指令傳出，因此計算時同樣會延遲一個 clk 才繼續做計算。

橘色: Input　綠色: Output　紫色:Hazard Unit Signal　紅色: Control signal

3. EXE(Execute)

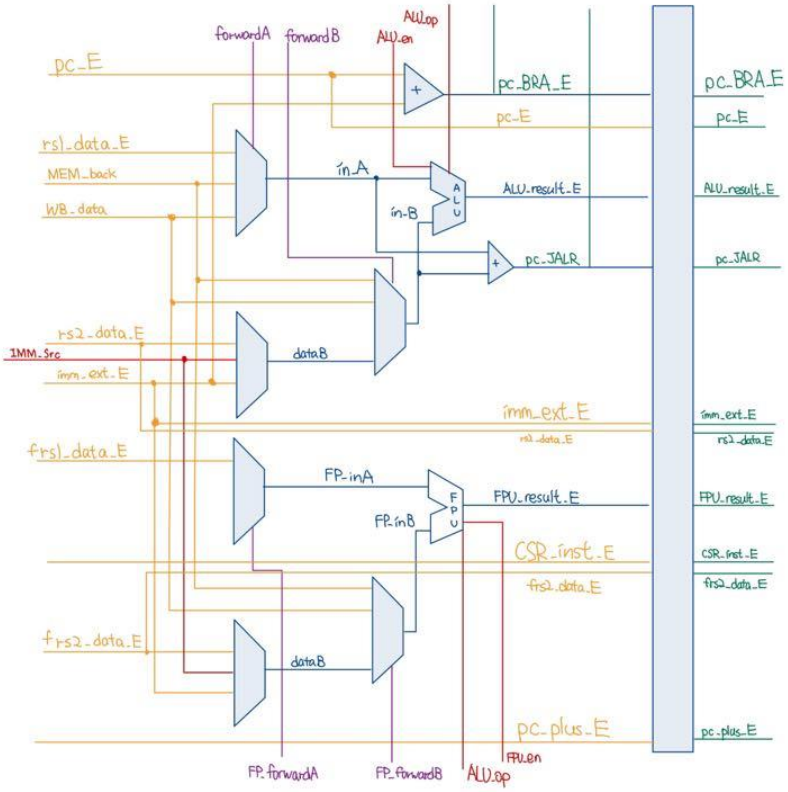


圖 4 EXE 階級架構圖

EXE 階段主要是在運算單元(ALU、FPU)中執行操作，根據 ID 階段解碼出的控制訊號，做出不同的運算。為了避免 Hazard 的發生，再送入 ALU 及 FPU 計算的 inputA 及 inputB 會有 forward 訊號進行控制，確保 pipeline 持續運行。原先設計想將 Branch 及 Jump 的 PC 同樣由 ALU 進行計算來節省面積，但實現上導致控制訊號過於複雜，因此最後將 Branch 及 Jump 的 PC 分別用加法器直接計算，再以 ALU 判斷是否有達成 Branch 的條件決定要不要進行 PC 的更動，這樣做的好處是讓設計較為簡單，但同時由於不管甚麼指令每次都會進行計算，犧牲掉一些系統的面積及效能。

橘色: Input　綠色: Output　紫色:Hazard Unit Signal　紅色: Control signal

4. MEM(Memory Access)

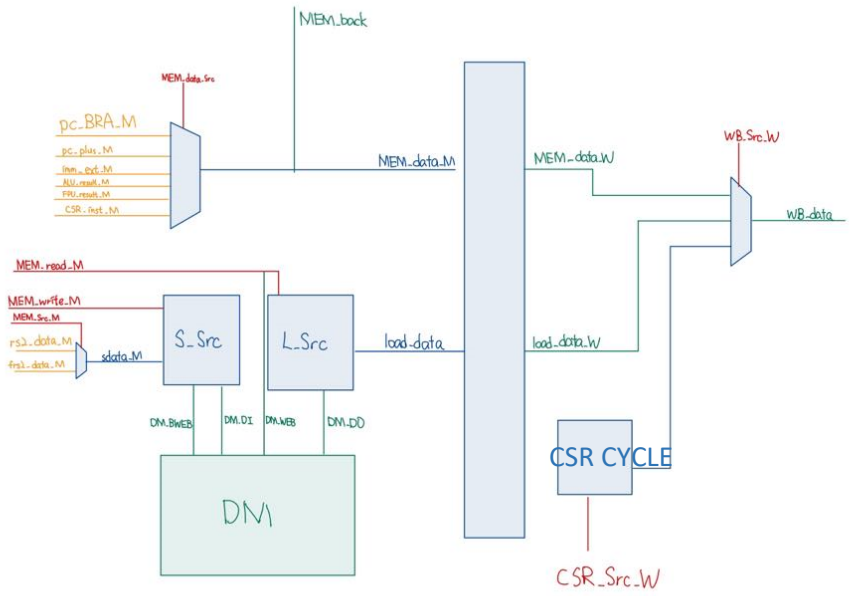


圖 5 MEM 階級架構圖

MEM 階級主要是對存取記憶體(DM)進行讀取或寫入，在這個階段我將讀取及寫入分為兩個單元進行操作，根據控制訊號來決定要執行讀取及寫入，而數值同時又分為 B(byte)、H(halfword)、W(word)，若是要儲存資料(S type)時，會將資料送入 S_Src 單元進行判斷 DM BWEB 及 DM DI；而若是要讀取資料時，則會將資料 DM DO 讀取到 L_Src 單元，再依據讀取類別將資料更改為對應的樣式傳出。而從 EXE 計算完的結果則依照不同的指令決定要傳送到 WB 及回傳回 EXE 的訊號。

5. WB(Write Back)

WB 階級主要將運算結果寫回暫存器中，透過控制訊號決定要將甚麼訊號寫回暫存器中，主要分為 MEM 傳送的 EXE 結果、DM 讀取結果、CSR CYCLE 計算結果，會將 CSR CYCLE 的放在 WB 階段進行計算是由於這樣計算出來的結果就可以直接回傳至暫存器中，不會因為需要在各階級中傳送導致計算的結果不及時還要進行修正。

二. 波型圖

- **R-type、I-type ALU 計算結果**

1. ADD

當指令為 ADD 時，可以看到:
in_A : 1000_0000_1000_1100
in_B : 111_0000
ALU_result : 1000_0000_1111_1100
結果正確。

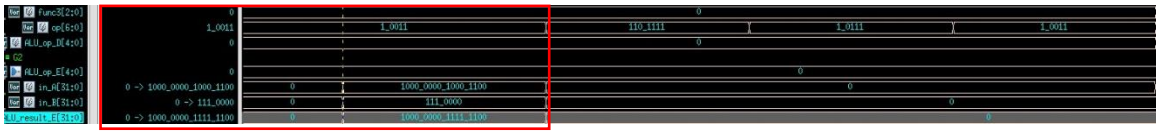


圖 6 ADD 波型圖

2. SUB

當指令為 SUB 時，可以看到:
in_A : 1111_1111_1111_1111_1111_1111_1111_1101
in_B : 1111_1111_1111_1111_1111_1111_1111_1000
ALU_result : 101
結果正確。

圖 7 SUB 波型圖

3. AND

當指令為 AND 時，可以看到:
in_A : 0001_0010_0011_0100_0101_0110_0111_1000
in_B : 1111_1111_1111_1111_1111_1111_1111_1111
ALU_result : 0001_0010_0011_0100_0101_0110_0111_1000
結果正確。

圖 8 AND 波型圖

4. OR

當指令為 OR 時，可以看到:

in_A : 0001_0010_0011_0100_0101_0110_0111_1000

in_B : 1111_1110_1101_1100_1011_1010_1001_1000

ALU_result : 1111_1110_1111_1100_1111_1110_1111_1000

結果正確。

圖 9 OR 波型圖

5. XOR

當指令為 XOR 時，可以看到:

in_A : 1111_1111_1111_1111_1111_1111_1111_1111

in_B : 1111_0000_1111_0000_1111_0000_1111_0000

ALU_result : 0000_1111_0000_1111_0000_1111_0000_1111

結果正確。

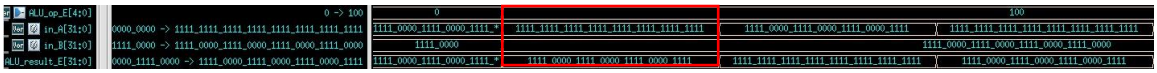


圖 10 XOR 波型圖

6. SRA

當指令為 SRA 時，可以看到:

in_A : 1000_0111_0110_0101_0100_0011_0010_0001

in_B : 100

ALU_result : 1111_1000_0111_0110_0101_0100_0011_0010

結果正確。

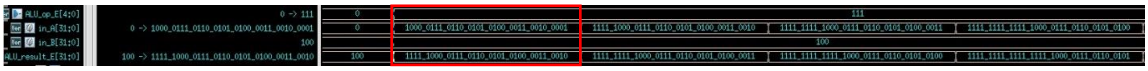


圖 11 SRA 波型圖

7. SRL

當指令為 SRL 時，可以看到:

in_A : 1111_1111_1111_1111_1111_1111_1111_1111

in_B : 0111_0101_1110_1100_1010_1000_0110_0100

ALU_result : 1111_1111_1111_1111_1111_1111_1111_1111

結果正確。

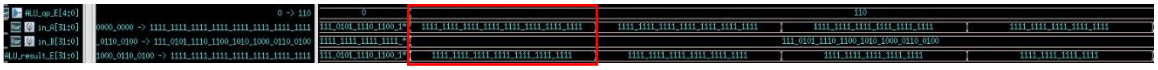


圖 12 SRL 波型圖

8. SLL

當指令為 SRL 時，可以看到:

in_A : 1

in_B : 1

ALU_result : 10

結果正確。

圖 13 SLL 波型圖

9. SLT

當指令為 SRL 時，可以看到:

in_A : 1111_1111_1111_1111_1111_1111_1111_1111

in_B : 1

ALU_result : 1

結果正確。

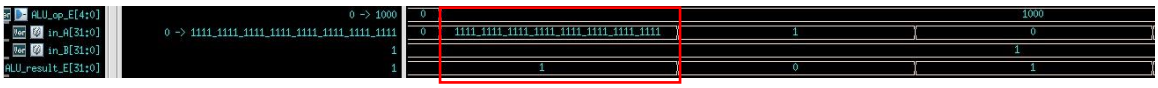


圖 13 SLT 波型圖

10. SLTU

當指令為 SLTU 時，可以看到:

in_A : 1111_1111_1111_1111_1111_1111_1111_1111

in_B : 1

ALU_result : 0

結果正確。

圖 14 SLTU 波型圖

11. MULHU

當指令為 MULHU 時，可以看到:

in_A : 0001_0010_0011_0100_0101_0110_0111_1000

in_B : 1111_0000_1111_0000_1111_0000_1111_0000

ALU_result : 0001_0001_0010_0010_0011_0011_0100_0011

結果正確。

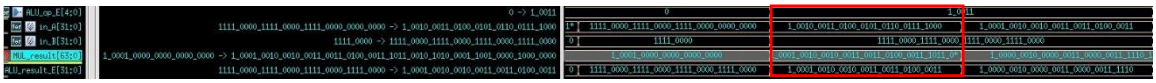


圖 15 MULHU 波型圖

12. MULHSU

當指令為 SLTU 時，可以看到:

in_A : 1111_0000_1111_0000_1111_0000_1111_0000

in_B : 1111_0000_1111_0000_1111_0000_1111_0000

ALU_result : 1111_0001_1101_0011_1011_0101_1001_0110

結果正確。

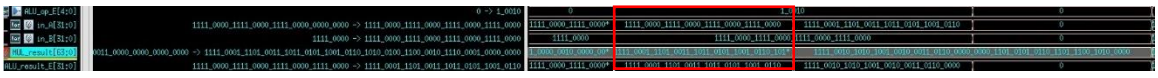


圖 15 MULHSU 波型圖

- **B-type 波型圖**

1. **BEQ**

當指令為 BEQ 但條件不符合時，可以看到:

B_Src : 1 代表 Branch 發生

in_A : 1111_1111_1111_1111_1111_0000_0000_0000

in_B : 1111_1111_0000

Zero : 0 代表條件不符合

pc_c : 沒有跳動

結果正確。

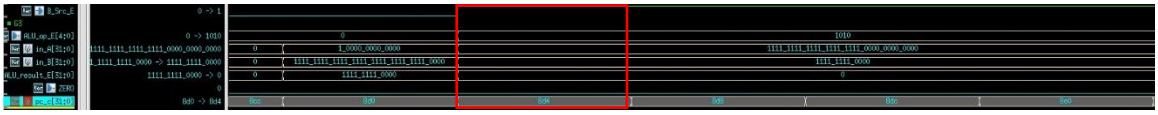


圖 16 BEQ、ZERO=0 波型圖

當指令為 BEQ 但條件符合時，可以看到:

B_Src : 1 代表 Branch 發生

in_A : 1111_1111_1111_1111_1111_0000_0000_0000

in_B : 1111_1111_1111_1111_1111_0000_0000_0000

Zero : 1 代表條件符合

pc_c :跳動 pc_BRA

結果正確。

圖 17 BEQ、ZERO=1 波型圖

2. BGE

當指令為 BGE 但條件不符合時，可以看到:

B_Src : 1 代表 Branch 發生

in_A : 1111_1111_1111_1111_1111_0000_0000_0000

in_B : 1111_1111_1111_1111_1111_0000_0000_0000

Zero : 0 代表條件不符合

pc_c : 沒有跳動

結果正確。

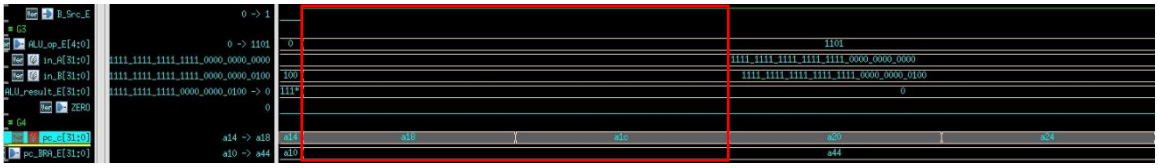


圖 18 BGE、ZERO=0 波型圖

當指令為 BGE 但條件符合時，可以看到:

B_Src : 1 代表 Branch 發生

in_A : 1111_1111_1111_1111_1111_0000_0000_0000

in_B : 1111_1111_1111_1111_1111_0000_0000_0100

Zero : 1 代表條件符合

pc_c : 跳動 pc_BRA

結果正確。

圖 19 BGE、ZERO=1 波型圖

3. BGEU

當指令為 BGEU 但條件不符合時，可以看到:

B_Src : 1 代表 Branch 發生

in_A : 1000_0001_0000_0000

in_B : 1001_0000_1111_1100

Zero : 0 代表條件不符合

pc_c : 沒有跳動

結果正確。

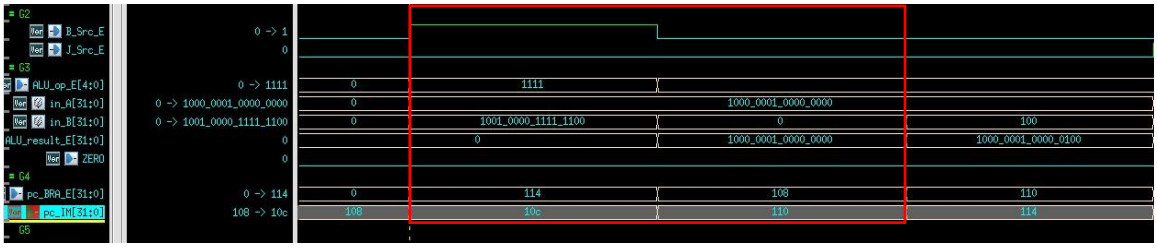


圖 20 BGEU、ZERO=0 波型圖

當指令為 BGEU 但條件符合時，可以看到:

B_Src : 1 代表 Branch 發生

in_A : 1000_0000_1111_1100

in_B : 1000_0000_1111_1100

Zero : 1 代表條件符合

pc_c : 跳動 pc_BRA

結果正確。

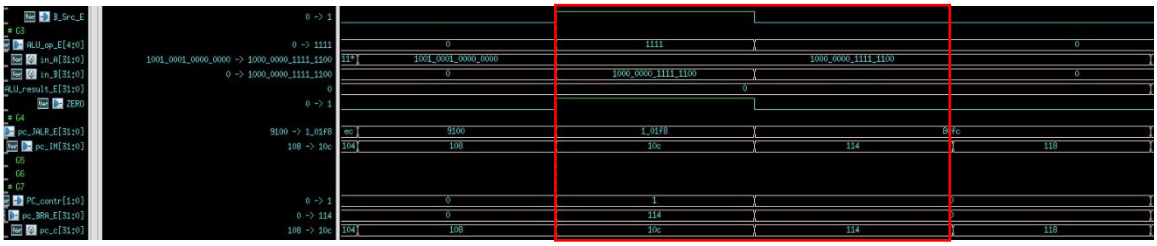


圖 21 BGEU、ZERO=1 波型圖

4. BLT

當指令為 BLT 但條件不符合時，可以看到:

B_Src : 1 代表 Branch 發生

in_A : 1111_1111_1111_1111_1111_0000_0000_0000

in_B : 1111_1111_1111_1111_1110_1111_1111_1100

Zero : 0 代表條件不符合

pc_c : 沒有跳動

結果正確。

圖 22 BLT、ZERO=0 波型圖

當指令為 BLT 但條件符合時，可以看到:

B_Src : 1 代表 Branch 發生

in_A : 1111_1111_1111_1111_1111_0000_0000_0000

in_B : 1111_1111_1111

Zero : 1 代表條件符合

pc_c :跳動 pc_BRA

結果正確。

圖 23 BLT、ZERO=1 波型圖

5. BLTU

當指令為 BLTU 但條件不符合時，可以看到:

B_Src : 1 代表 Branch 發生

in_A : 1111_1111_1111_1111_1111_0000_0000_0000

in_B : 1111_1111_1111_1111_1110_1111_1111_1100

Zero : 0 代表條件不符合

pc_c : 沒有跳動

結果正確。

圖 24 BLTU、ZERO=0 波型圖

當指令為 BLTU 但條件符合時，可以看到:

B_Src : 1 代表 Branch 發生

in_A : 1111_1111_1111_1111_1111_0000_0000_0000

in_B : 1111_1111_1111_1111_1111_1111_1111_1111

Zero : 1 代表條件符合

pc_c : 跳動 pc_BRA

結果正確。

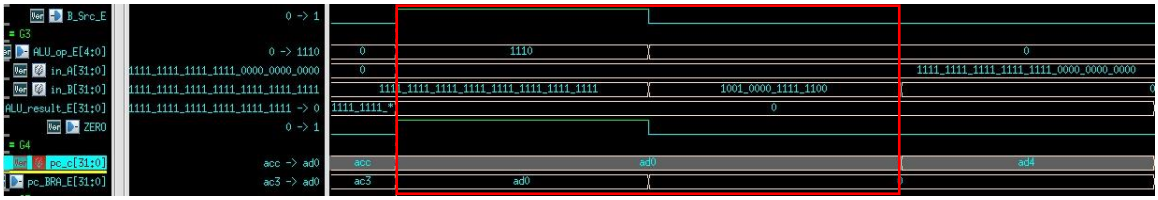


圖 25 BLTU、ZERO=1 波型圖

6. BNE

當指令為 BNE 但條件不符合時，可以看到:

B_Src : 1 代表 Branch 發生

in_A : 1111_1111_1111_1111_1111_0000_0000_0000

in_B : 1111_1111_1111_1111_1111_0000_0000_0000

Zero : 0 代表條件不符合

pc_c : 沒有跳動

結果正確。

圖 25 BNE、ZERO=0 波型圖

當指令為 BNE 但條件符合時，可以看到:

B_Src : 1 代表 Branch 發生

in_A : 1111_1111_1111_1111_1111_0000_0000_0000

in_B : 1111_1111_1111

Zero : 1 代表條件符合

pc_c : 跳動 pc_BRA

結果正確。

圖 26 BNE、ZERO=1 波型圖

➢ J-type 波型圖

1. JUMP

當指令為 JUMP 時，可以看到:
J_Src : 1 代表 JUMP 發生
pc_c : 跳動 pc_BRA(pc_JUMP)
結果正確。

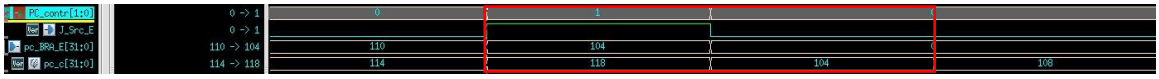


圖 27 JUMP 波型圖

2. JALR

當指令為 JUMP 時，可以看到:
J_Src : 1 代表 JUMP 發生
pc_c : 跳動 pc_JALR
結果正確。

圖 28 JALR 波型圖

➢ Load 波型圖

1. LB

當指令為 LB 時，可以看到:
MEM_read : 1 代表讀取 DM
DM_DO : 0110_0110_0110_0110_0110_0110_0110_0110
MEM_data: 0110_0110
結果正確。

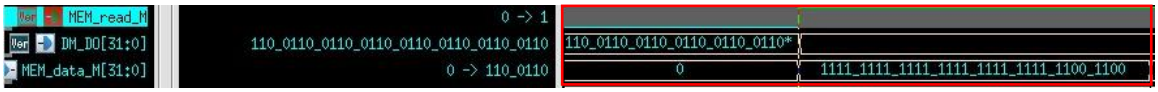


圖 29 LB 波型圖

2. LH

當指令為 LH 時，可以看到:
MEM_read : 1 代表讀取 DM
DM_DO : 1100_1100_1100_1100_1100_1100_1100_1100
MEM_data: 1100_1100_1100_1100
結果正確。

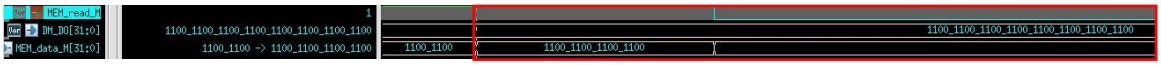


圖 30 LH 波型圖

3. LW

當指令為 LW 時，可以看到:
MEM_read : 1 代表讀取 DM
DM_DO : 0110_0110_0110_0110_0110_0110_0110_0110
MEM_data: 0110_0110_0110_0110_0110_0110_0110_0110
結果正確。

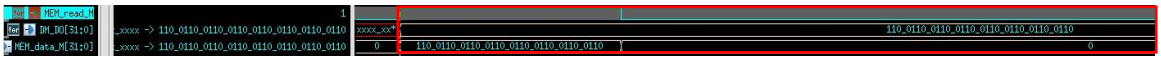


圖 31 LW 波型圖

➢ S-type 波型圖

1. SB

當指令為 SB 並寫入最後 8bit 時，可以看到:
MEM_write: 1 代表寫入 DM
sdata : 0001_0010_0011_0100_0101_0110_0111_1000
DM_DI: 1111_1111_1111_1111_1111_0111_1000
DM_BWEB: 1111_1111_1111_1111_1111_1111_0000_0000
結果正確。

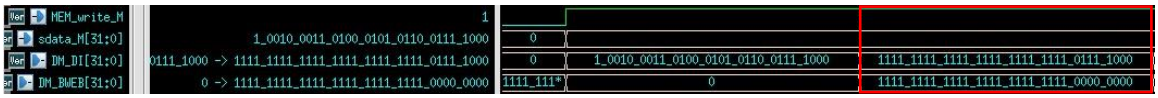


圖 32 SB(最後 8bit)波型圖

當指令為 SB 並寫入前面 8bit 時，可以看到:
MEM_write: 1 代表寫入 DM
sdata : 0001_0010_0011_0100_0101_0110_0111_1000
DM_DI: 0111_1000_0000_0000_0000_0000_0000_0000
DM_BWEB: 0000_0000_1111_1111_1111_1111_1111_1111
結果正確。

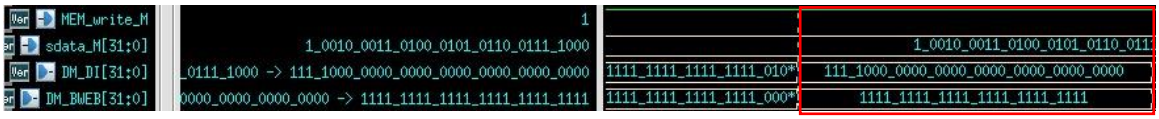


圖 33 SB(前面 8bit)波型圖

2. SH

當指令為 SH 並寫入最後 16bit 時，可以看到:
MEM_write: 1 代表寫入 DM
sdata : 0001_0010_0011_0100_0101_0110_0111_1000
DM_DI: 1111_1111_1111_1111_0101_0110_0111_1000
DM_BWEB: 1111_1111_1111_1111_0000_0000_0000_0000
結果正確。

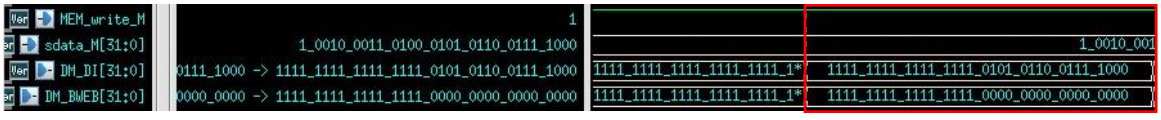


圖 34 SH(後面 16bit)波型圖

當指令為 SH 並寫入前面 16bit 時，可以看到:
MEM_write: 1 代表寫入 DM
sdata : 0001_0010_0011_0100_0101_0110_0111_1000
DM_DI: 0101_0110_0111_1000_0000_0000_0000_0000
DM_BWEB: 0000_0000_0000_0000_1111_1111_1111_1111
結果正確。

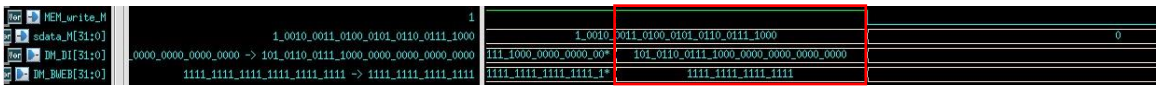


圖 35 SH(前面 16bit)波型圖

3. SW

當指令為 SW 時，可以看到:
MEM_write: 1 代表寫入 DM
sdata : 0110_0110_0110_0110_0110_0110_0110_0110
DM_DI: 0110_0110_0110_0110_0110_0110_0110_0110
DM_BWEB: 0000_0000_0000_0000_0000_0000_0000_0000
結果正確。

圖 36 SW 波型圖

- **F-type**

一. FADD

當指令為 FADD 時，可以看到:

FP_inA : 0100_0000_1011_1011_0011_0010_0111_0110

FP_inB : 0100_0000_1011_1011_0011_0010_0111_0110

FPU_result : 0100_0001_0011_1011_0011_0010_0111_0110

結果正確。

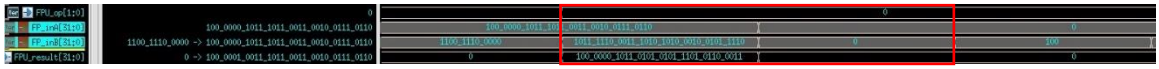


圖 37 FADD 波型圖

二. FSUB

當指令為 FSUB 時，可以看到:

FP_inA : 0100_0000_1011_1011_0011_0010_0111_0110

FP_inB : 0100_0000_1011_1011_0011_0010_0111_0110

FPU_result : 1011_0011_1000_0000_0000_0000_0000_0000

結果正確。

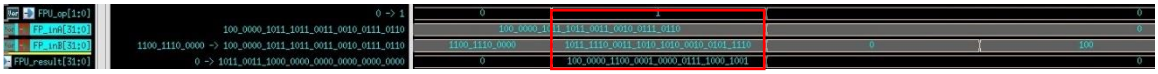


圖 38 FSUB 波型圖

三. Superlint

在使用 superlint 進行 debug 時，最常遇到的是以下兩種 warning:

1. Logic operator other than or used to describe asynchronous reset of flipflop:

   這是由於我在 IFID register 以及 IDEXE register 中，將 flush 訊號與 rst 訊號用 if (rst | flush) 數值歸零的方式描述，導致出現 warning，解決方式為我將兩者分開，用 if (rst)，else if (flush)的方式描述 warning 就消失，這在我程式碼中所有有使用到 always_ff 同時使用 rst 及其他 hazard 訊號的地方都有出現這個 error，更改完後就都消失了。

2. The latches in the always block are mixed with combinational logic

   這是我出現最多 warning 的部分，幾乎程式碼中所有的 always_comb 都有被標示，我發現到是因為我在 always_comb 中沒有給他預值導致的，被標示錯誤的都會是 always_comb begin 直接接 if (條件) begin，雖然我都有寫 else begin，但導致裡面的數值可能會有所延遲，出現 latch。再給予預值後 waring 就都消失了。

其餘部分還有一些 coding style 的問題，例如 case 全部都有但還是寫 default、LSB 與 RSB 長度不同等等，但都做小更動就可以解決。下圖為最後擷取 superlint 的結果，沒有 violation 產生。

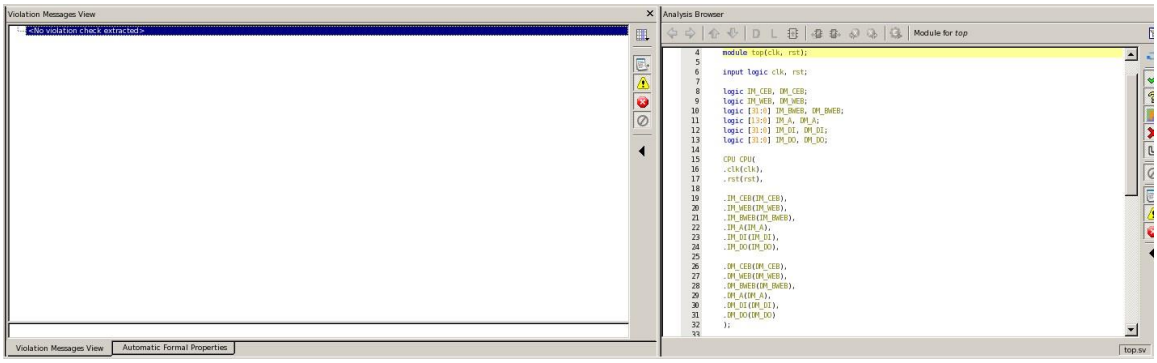
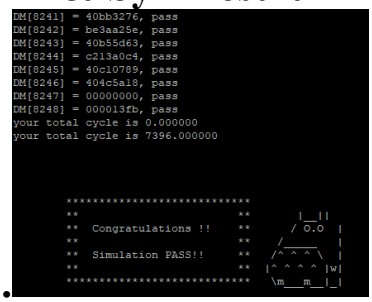
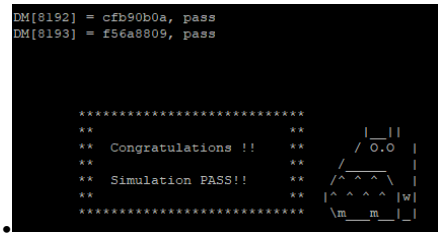


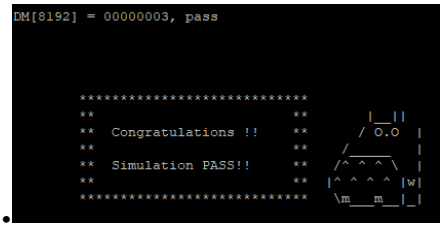
圖 39 superlint

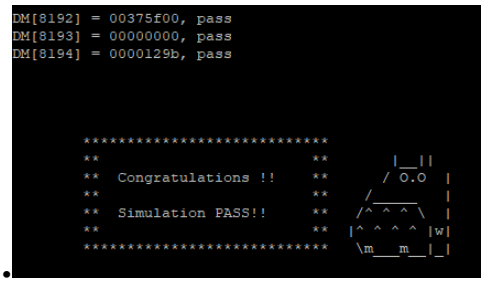
四. Sim & Syn Result

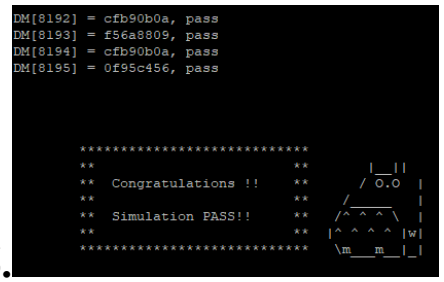
1.


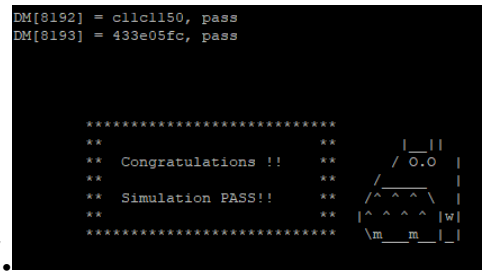
2.


3.


4.


5.


6.


7.



```
Number of ports:                         6908
Number of nets:                         31611
Number of cells:                        23900
Number of combinational cells:          20612
Number of sequential cells:              3236
Number of macros/black boxes:               2
Number of buf/inv:                       3714
Number of references:                       6

Combinational area:               8165.163075
Buf/Inv area:                      771.482897
Noncombinational area:            3697.228935
Macro/Black Box area:             9082.750000
Net Interconnect area:      undefined  (Wire load has zero net area)

Total cell area:                 20945.142010
Total area:                 undefined
1
```

圖 40 Sim & Syn Result

五. Lesson Learn

這次的作業對我而言是非常有難度的，由於沒有修過大學部的 VSD，雖然網路上有非常多的資料可以參考，但我對 riscV 的架構不夠熟悉，導致花了相當多的時間研究，透過同學及學長的幫助解惑才有較明顯的方向。其中 Hazard 訊號的設計是花我最多時間的，由於只要一個訊號不對結果就會錯，在不斷反覆觀察波型圖才找出問題所在，這次作業也讓我更加熟悉不同的 tool，以前我只使用過 vcs 進行 compile 以及用 vivado、model sim 來查看波型圖 debug，這次的 nWave 及 superlint 都是新學習的。當 sim 過後發現 rst 沒有設為 0 導致合成 error，又花費非常多的時間去修正。原先還沒有把 warning 修完時，合成的 clk period 只能到 1.5，再把 warning 修完後可以到 1。在架構上感覺仍然有一些部分可以進行優化，像是前面提到的將 branch 及 JALR 的計算併入到 ALU 中，但這樣 CU 及 MEM back 部分需要重新設計，除此之外，目前有一些 IO 是沒有用到的，在未來也需要花時間將程式碼整理一下。