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: \_\_黃于睿\_\_

Student ID: \_\_M16121158\_\_

## Summary

* **RISC-V Pipelined CPU 設計：**

實作一個5 Stage pipeline CPU，支援 49 條 RISC-V 指令，並使用 32x32 bit的整數與浮點數暫存器檔案。指令與資料記憶體各有 16Kx32 bit大小，CSR暫存器大小為2x64 bit。

* **指令集：**

實作多種類型的指令，包括 R-type、I-type、S-type、B-type、U-type及J-type，並支援 M Extension（整數乘法）與F-type浮點數操作。

* **測試與模擬：**

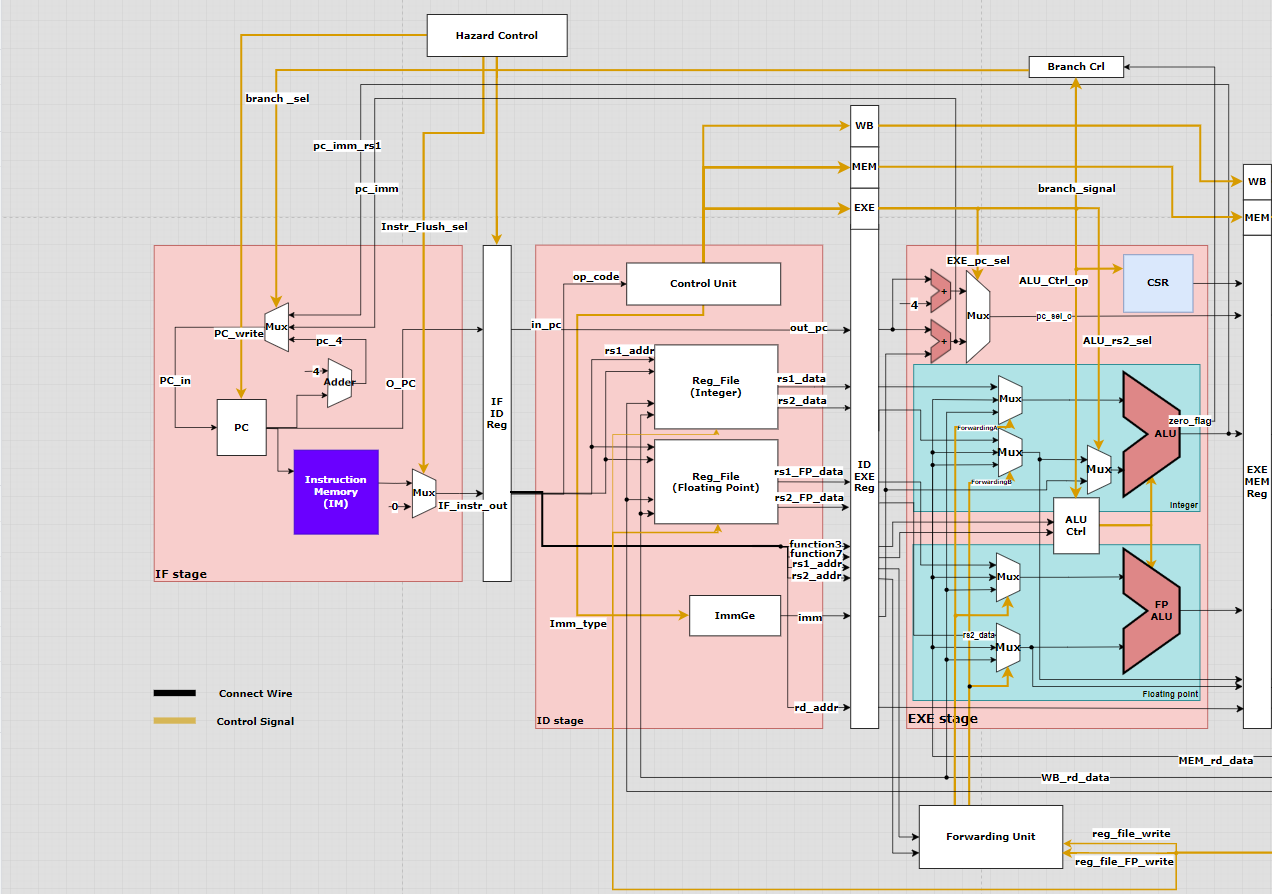
透過實際程式及組合語言代碼，進行全面驗證，包含基礎指令測試、排序算法、以及不使用乘法指令的乘法實現，及浮點數測試。

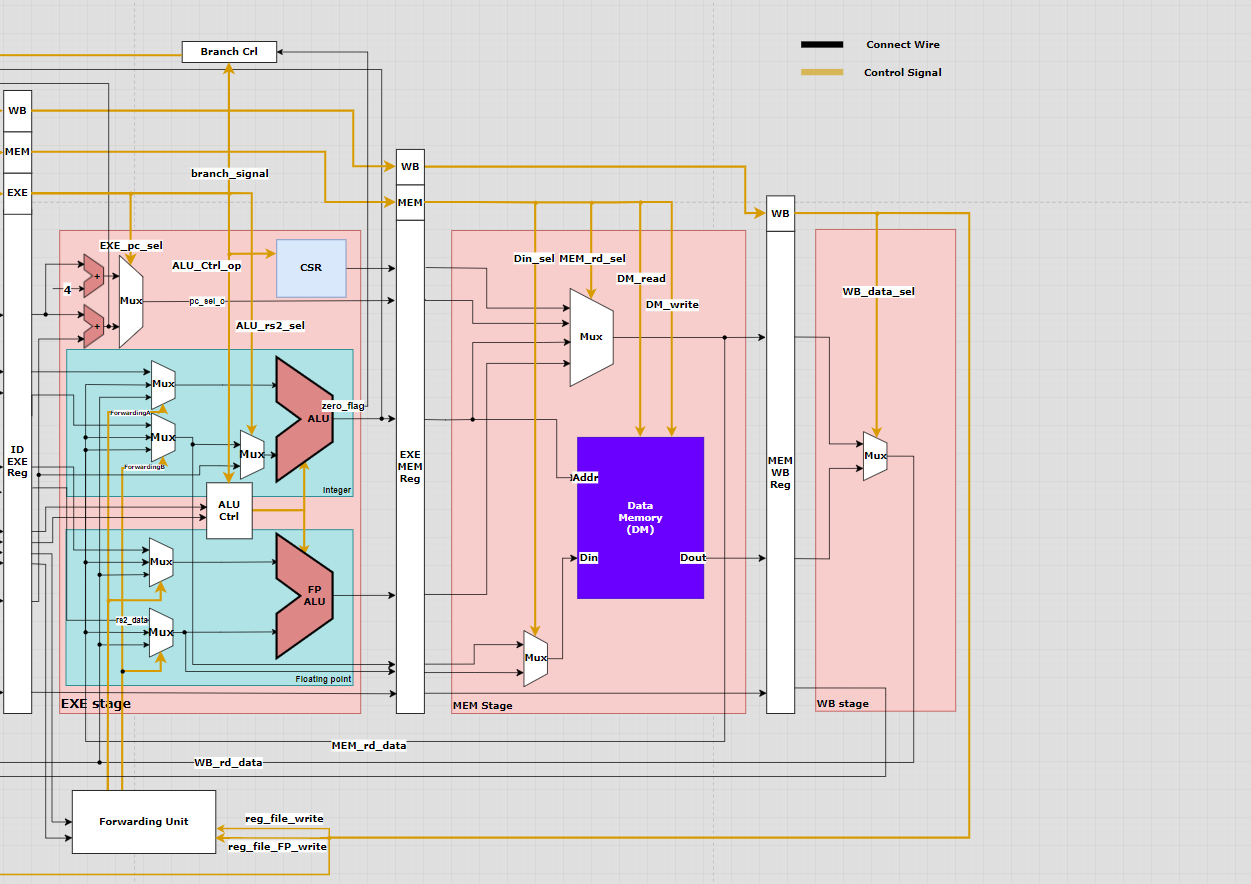
## Design Report

|  |  |
| --- | --- |
| **RTL Simulation** | |
| Program 0 | Program 1 |
| Program 2 | Program 3 |
| Program 4 | Program 5 |
| Program 6 |  |
| **Gate Level Simulation** | |
| Program 0 | Program 1 |
| Program 2 | Program 3 |
| Program 4 | Program 5 |
| Program 6 |  |
| **PA Report** | |
| Cycle Period | 2ns |
| Cell Area : 18477.8690 |  |
| **Superlint Coverage: 96.90%** | |
|  |  |

## Design Architecture

下圖為一5-stage pipeline CPU 完整架構圖。此CPU以 RISC-V 為基礎，總共切為5個stage，分別為IF、ID、EXE、MEM以及WB





## Sub-module Functional Explanation

### https://www.draw.io/?open=G1L_ngzNXUjKbvokIGh9GLqUMypqsMWFCY&local-data=%7B%22type%22%3A%22Drive%22%2C%22id%22%3A%221L_ngzNXUjKbvokIGh9GLqUMypqsMWFCY%22%2C%22lastModifiedDate%22%3A%222024-10-10T13%3A54%3A16.280Z%22%2C%22pageId%22%3A%22yNgi87nR175QiDtrsDPd%22%2C%22layers%22%3A%5B0%5D%7DIF stage

下圖為IF stage 局部設計圖，當中可分為2個submodule：

**Program Counter (PC)更新：**

**PC**為每一條要執行的指令的地址。每執行一次 IF 階段，PC 的值會根據目前的指令類型進行更新，通常是自增來指向下一條指令地址，但在某些情況下（如跳轉指令）會更新為一個新的目標地址。

**Instruction Memory讀取Instruction：**

PC 更新後，CPU 使用 PC 的值作為地址，從Instruction Memory(IM)中讀取對應地址的指令。這個操作在一個時鐘週期內完成。

### https://www.draw.io/?open=G1L_ngzNXUjKbvokIGh9GLqUMypqsMWFCY&local-data=%7B%22type%22%3A%22Drive%22%2C%22id%22%3A%221L_ngzNXUjKbvokIGh9GLqUMypqsMWFCY%22%2C%22lastModifiedDate%22%3A%222024-10-10T09%3A10%3A32.828Z%22%2C%22pageId%22%3A%222K5ftU8uuCKKKdErGrP8%22%2C%22layers%22%3A%5B0%5D%7DID stage

右圖為ID stage 局部設計圖，當中可分為4個submodule，分別為Control Unit、Register File(Integer)、Register File(Floating Point)、 Immediate Generator

**指令解碼：**

從 IF 階段傳遞來的指令會在這裡被解碼。通常，指令的不同部分（例如操作碼、來源寄存器、目的寄存器、立即數等）會被提取出來，以便後續處理。

**控制訊號產生：**

根據op\_code，ID階段會產生一組對應操控的控制訊號，對後續控制後續階段中的數據流與操作。控制信號通常用來決定 ALU 要執行的運算類型、是否訪問記憶體，以及是否需要寫回寄存器等。

下表為本次所設計的控制訊號總表:

|  |  |  |
| --- | --- | --- |
| Signal | Destination | Explanation |
| ALU\_Ctrl\_op | EXE – ALU ctrl | 選擇為哪一種Type進行計算 |
| Imm\_type | ID – ImmGe | 產生對應Immediate Vlue |
| ALU\_rs2\_sel | EXE | 選擇rs2進入ALU的值(imm) |
| EXE\_pc\_sel | EXE | 選擇pc為pc+4 or pc+imm |
| MEM\_rd\_sel | MEM | 選擇rd\_data為pc or ALU的值 |
| Din\_sel | MEM-DM | 選擇進入DM的address為何 |
| DM\_read | MEM-DM | 選擇進入DM的是否為load狀態 |
| DM\_write | MEM-DM | 選擇進入DM的是否為store狀態 |
| WB\_data\_sel | WB | 選擇rd\_data為rd\_data or DM送出的值 |
| reg\_file\_write | ID - reg\_file | 選擇rd進入的是否至 reg\_file |
| reg\_file\_FP\_write | ID- reg\_file\_FP | 選擇rd進入的是否至 reg\_file\_FP |
| branch\_signal | Branch Ctrl | 給Branch type |

**暫存器讀取：**

指令需要使用暫存器中的資料，ID 階段會根據解碼出的rs1、rs2 addr來讀取對應的暫存器數值。

### https://www.draw.io/?open=G1L_ngzNXUjKbvokIGh9GLqUMypqsMWFCY&local-data=%7B%22type%22%3A%22Drive%22%2C%22id%22%3A%221L_ngzNXUjKbvokIGh9GLqUMypqsMWFCY%22%2C%22lastModifiedDate%22%3A%222024-10-10T09%3A10%3A32.828Z%22%2C%22pageId%22%3A%229md-5IGasgJbhcH-B0wv%22%2C%22layers%22%3A%5B0%5D%7DEXE stage

右圖為EXE stage 局部設計圖，當中可分為7大部分，分別為pc值選擇、Integer ALU、Floating Point ALU、CSR、Forwarding Unit、Hazard、Branch Ctrl。

將實際的rs1、rs2及imm依照所要需求進行ALU運算，當中rs1、rs2需考量到指令的前後關係，是否進行forwarding 操作。運算後的結果依照需求可能被視為data或address傳輸到MEM、WB Stage運用。

下表為ALU Ctrl對Integer ALU、Floating Point ALU總代碼表:

|  |  |  |  |
| --- | --- | --- | --- |
| **Parameter** | **Value** | **Parameter** | **Value** |
| ALU\_add | 5’d0 | ALU\_mulhsu | 5’d12 |
| ALU\_sub | 5’d1 | ALU\_mulhu | 5’d13 |
| ALU\_sll | 5’d2 | ALU\_jalr | 5’d14 |
| ALU\_slt | 5’d3 | ALU\_beq | 5’d15 |
| ALU\_sltu | 5’d4 | ALU\_bne | 5’d16 |
| ALU\_xor | 5’d5 | ALU\_blt | 5’d17 |
| ALU\_srl | 5’d6 | ALU\_bge | 5’d18 |
| ALU\_sra | 5’d7 | ALU\_bltu | 5’d19 |
| ALU\_or | 5’d8 | ALU\_bgeu | 5’d20 |
| ALU\_and | 5’d9 | ALU\_imm | 5’d21 |
| ALU\_mul | 5’d10 | ALU\_FP\_add | 5’d22 |
| ALU\_mulh | 5’d11 | ALU\_FP\_sub | 5’d23 |

**CSR:**

在本實驗當中著重於計算整個CPU執行時花費的時間(cycle)以及所執行的指令總數。

**Forwarding Unit:**

當pipeline中的多個指令同時執行，有時會發生data dependency 中的Read after write(RAW)問題，這些算結果尚未寫入暫存器。為了避免多餘的stall，forwarding unit 可以直接從MEM Stage及WB Stage取得資料，不用等到暫存器檔案更新。

**Hazard: (分為Data Hazard及 Control Hazaad)**

* **Data Hazard**

發生在當前指令需要使用的資料尚未準備好。例如，如果一個指令在 EX 階段需要使用一個還沒被寫入的資料，這就會導致Data Hazard。包含lw use指令。

* **Control Hazard**

發生在 (branch) 指令後，因為在決定跳躍的目標地址之前，CPU 不確定接下來應該執行哪個指令。這會導致 pipeline 停頓，直到跳躍是否發生的結果確定。需要丟棄正在pipeline中處理的指令，並執行正確指令，稱為**刷新（flush）**。將會導致pipeline stall。

**Branch:**

如果為一Branch指令，Branch Control根據條件來決定是否修改 PC（程序計數器）的值，影響後續指令的取得進行跳轉。

### https://www.draw.io/?open=G1L_ngzNXUjKbvokIGh9GLqUMypqsMWFCY&local-data=%7B%22type%22%3A%22Drive%22%2C%22id%22%3A%221L_ngzNXUjKbvokIGh9GLqUMypqsMWFCY%22%2C%22lastModifiedDate%22%3A%222024-10-10T13%3A54%3A16.280Z%22%2C%22pageId%22%3A%22be-SBJn2g4H9DDypS6W1%22%2C%22layers%22%3A%5B0%5D%7DMEM stage

右圖為MEM stage 局部設計圖，當中address 及 DM為設計主軸:

**Data Memory：**

MEM階段主要使用Data Memory來進行讀寫操作，Data Memory為SRAM Module，其中分為Load word /Store word 解釋 :

**Load word：**

從記憶體讀取數據，MEM階段會根據EX（Execute）階段計算出來的rd\_address，從DM中讀取資料，並將資料存入暫存器中。

**Store word：**

將數據寫入記憶體，MEM階段將EX階段產生的data寫入計算出的target address中。

**LW/SW控制訊號：**

對於load指令，控制訊號會啟動記憶體讀取操作，將資料回傳給後續的Write-back（WB）階段。對於store指令，控制訊號會啟動寫操作，將資料寫入到指定的記憶體位置。

### https://www.draw.io/?open=G1L_ngzNXUjKbvokIGh9GLqUMypqsMWFCY&local-data=%7B%22type%22%3A%22Drive%22%2C%22id%22%3A%221L_ngzNXUjKbvokIGh9GLqUMypqsMWFCY%22%2C%22lastModifiedDate%22%3A%222024-10-10T13%3A54%3A16.280Z%22%2C%22pageId%22%3A%22ugObICs5xghxOWmt38SC%22%2C%22layers%22%3A%5B0%5D%7DWB stage

右圖為WB stage 局部設計圖，在 WB 階段，CPU 的主要工作是將來自EXE stage的rd\_data或MEM Stage的結果寫回register file 。

**Data 選擇:**

* + - 1. 指令是算術或邏輯運算，結果會來自 ALU（算術邏輯單元）。
      2. 指令是 Load 指令（如 lw），結果來自Data Memory讀取。
      3. pc計算結果亦有可能在MEM Stage被選擇為rd\_data。

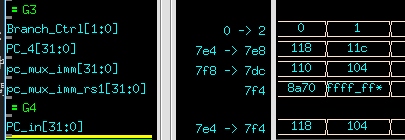
## 波形圖驗證

### IF stage

由Program Counter 產生所要的地址，並至對應的Instruction Memory 內送出所需完整Instruction。波形如下圖所示。

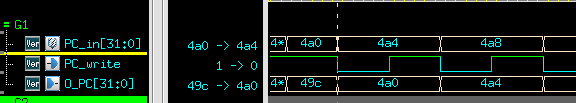
送入PC的值依照不同狀態先透過Mux 進行選擇，共分為三種。

* PC = PC + imm
* Pc = imm + rs1
* Pc = pc + 4

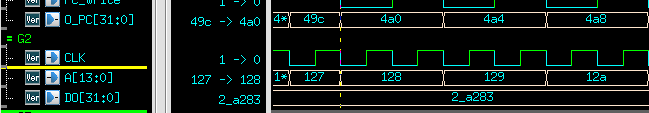


下圖以4a0, 4a4舉例🡺選擇後，透過PC\_in 送入Program counter

此處要送出的值，由PC\_write 決定是否要將值由O\_PC送至IM

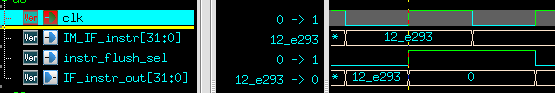


IM 收到對應的Address(0x4a0, 0x4a4)，當中有記憶體bias，所以最後2 bit 不須採用，只需要採用[15:2] 🡺(0x128, 0x129)，會送出Instruction



Mux (Instruction)負責是否因為Hazard問題，而將Instruction 進行清除

下圖當從IM讀取出的Instruction🡺IM\_IF\_instr(12\_e293)

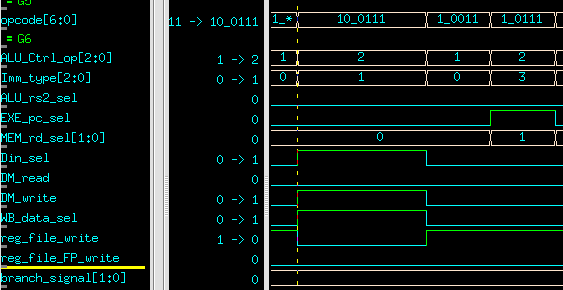


### ID stage- Control Unit

Op-code 送入 Control Unit 後，根據不同型態指令給予不同控制訊號。

以下用Floating point store word進行解釋，相對應送出訊號如下:

|  |  |  |
| --- | --- | --- |
| Control Signal | State | Value |
| Op\_code | F\_type | 010\_0111 |
| ALU\_Ctrl\_op | ADD\_type | 3’b010 |
| Imm\_type | Imm\_S | 3’b001 |
| ALU\_rs2\_sel | IMM (default) | 1’b0 |
| EXE\_pc\_sel | pc+4 (default) | 1’b0 |
| MEM\_rd\_sel | From ALU (Integer) | 2’d0 |
| Din\_sel | FP\_data | 1’b1 |
| DM\_read | Not read mode | 1’b0 |
| DM\_write | Write mode | 1’b1 |
| WB\_data\_sel | From IM (default) | 1’b1 |
| reg\_file\_write | Not use | 1’b0 |
| reg\_file\_FP\_write | Not use | 1’b0 |
| branch\_signal | N\_Branch | 2’b00 |

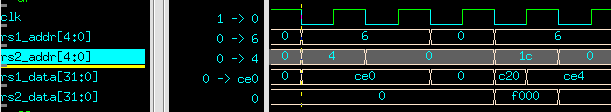
波行對應圖如下所示: 

### ID stage- Register File (Integer/Floating Point)

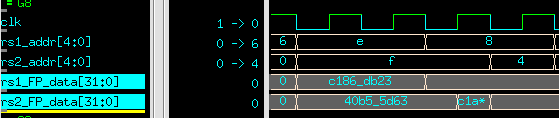
Register file 主要分為取出Instruction 所指定rs1及rs2的data。

此CPU設計採用Integer及Floating Point 各自讀取，最後在MEM stage 才選擇何者為所要計算的值。

下圖為reg\_file(integer)實際操作結果，



下圖為reg\_file(Floating Point)實際操作結果，

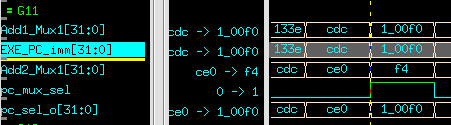


要將rd寫回時，會搭配選擇reg\_write (Integer) 或者 reg\_FP\_write 來判斷寫入哪一個暫存器。(註: 須注意Floating Point 0x0 可以存值，Integer 0x0 不可存值)

### EXE stage

* **pc:**

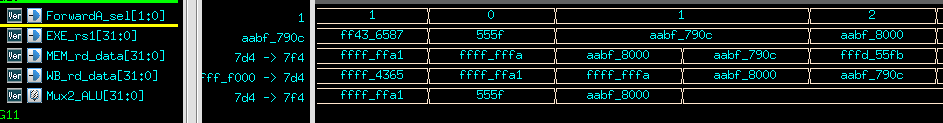
EXE階段中pc需依照執行控制訊號pc\_mux\_sel進行選擇。  
一為計算pc+imm (Add1\_Mux1/EXE\_pc\_imm)，  
一為計算pc + 4 (Add2\_Mux1) 並送出。



* **rs1(integer)/rs1\_FP (floating):**

rs1 進入時需經過一層Mux，將值跟Forwarding回來進行的選擇，以解決掉data dependency 中的Read after write(RAW)問題。在rs1\_FP下作法相同。  
下圖同時展示了三種可能狀態

|  |  |  |  |
| --- | --- | --- | --- |
| ForwardA\_sel | 0 | 1 | 2 |
| 狀態 | 直接使用 EXE\_rs1 | 使用 MEM\_rd\_data | 使用WB\_rd\_data |
| Value | 555f | aabf\_8000 | aabf\_8000 |



* **rs2(integer)/rs2\_FP (floating):**

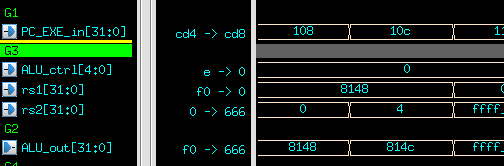
rs2再進行值的選擇時與rs1做法相同，rs2 進入時需經過一層Mux，將值跟Forwarding回來進行的選擇，以解決掉data dependency 中的Read after write(RAW)問題。

在rs2當中只有(integer)需要注意處理選擇IMM的mux

* **ALU R-type:** 
  + - 1. **ADD <當下執行的PC(PC\_EXE\_in) 🡺0X10C>**

ALU\_ctrl 設為 5’d0，rs1為0x8148 , rs2 為 0x4

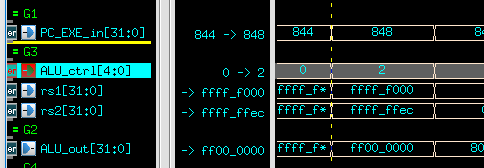
相加後透過ALU\_out 送出，為0x814c



* + - 1. **SLL<當下執行的PC(PC\_EXE\_in) 🡺0X848>**

ALU\_ctrl 設為 5’d2，rs1為0xffff\_f000 , rs2 為 0xffff\_ffec

對rs1左移rs2[4:0] 🡺 0x0c(12)後，透過ALU\_out 送出，為0xff00\_0000



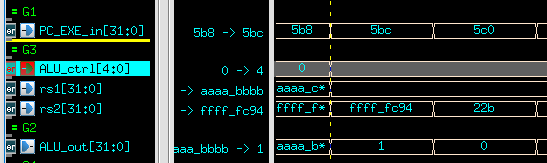
* + - 1. **SLTU<當下執行的PC(PC\_EXE\_in) 🡺0X5bc、0X5c0>**

ALU\_ctrl 設為 5’d4，rs1為0xaaa\_bbbb ，rs2 為 0xffff\_fc94

SLTU主要對unsigned 做處理，所以以此例為rs2 > rs1，

透過ALU\_out 送出，為1。

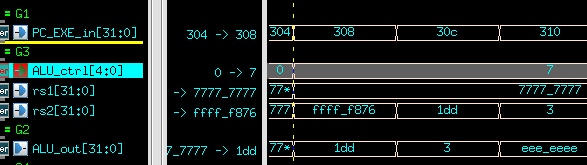
反之，執行0X5c0 rs2<rs1，透過ALU\_out 送出，為0。



* + - 1. **SRA<當下執行的PC(PC\_EXE\_in) 🡺0X308>**

ALU\_ctrl 設為 5’d7，rs1為0x7777\_7777，rs2 為 0xffff\_f876

對rs1右移rs2[4:0] 🡺 0x16(22)後，須注意為signed，以此例MSG為0，過右移時也都是補0，透過ALU\_out 送出，為0x1dd

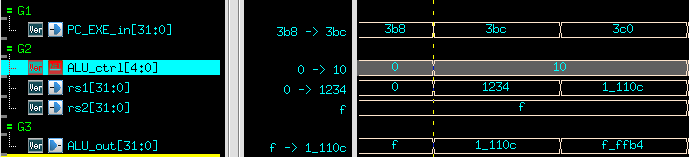


* **ALU M-extension** 
  + - 1. **MUL<當下執行的PC(PC\_EXE\_in) 3bc>**

ALU\_ctrl 設為 5’d10，rs1為0x1234 , rs2 為 0xf

MULH 為抓取2個signed 乘法(MUL)，並只抓取lower 32bit

乘法計算完為lower 32bit 為0x1\_111c ，並透過ALU\_out 送出。

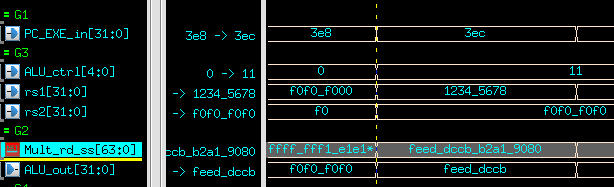


* + - 1. **MULH<當下執行的PC(PC\_EXE\_in) 🡺0x3ec>**

ALU\_ctrl 設為 5’d11，rs1為0x1234\_5678 , rs2 為 0xf0f0\_f0f0

MULH 為抓取2個signed 乘法(MUL)，並只抓取upper 32bit

乘法計算完為upper 32bit 為0xfeed\_dccb ，並透過ALU\_out 送出。

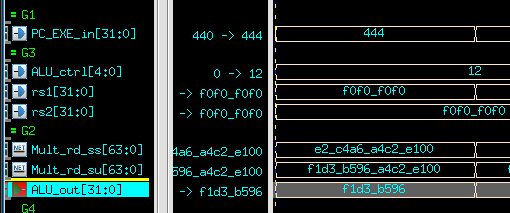


* + - 1. **MULHSU<當下執行的PC(PC\_EXE\_in) 🡺0x444>**

ALU\_ctrl 設為 5’d12，rs1為0xf0f0\_f0f0 , rs2 為 0xf0f0\_f0f0

MULH 為抓取rs1🡺 signed 、rs2🡺 unsigned乘法(MUL)，並只抓取upper 32bit，算完的值為Mult\_rd\_su，與Mult\_rd\_ss不同

乘法計算完為upper 32bit 為0xf1d3\_b596，並透過ALU\_out 送出。



, rs2 為

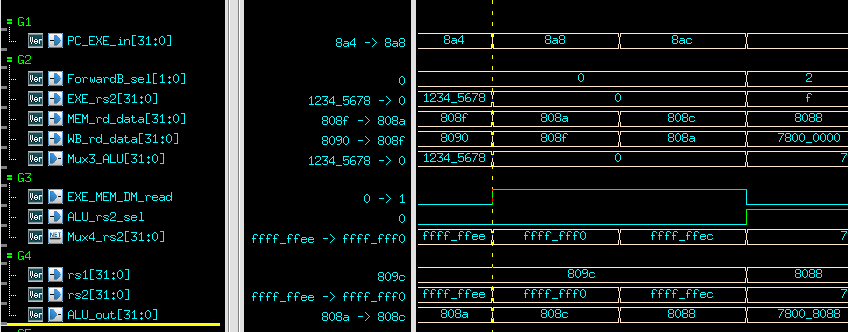
#### ALU I-type

下列(1)(2) ALU\_Ctrl\_op在從ControlUnit產生時為ADD\_type，其目標是將rs1值與IMM值去進行相加，當作從Data Memory 讀值出來的address。

1. **LW<當下執行的PC(PC\_EXE\_in) 🡺0x8a8>**

ALU\_ctrl 設為 5’d0(採用加法運算)，目標為rs1 +imm。

rs1為0x809c，rs2 為 0xffff\_fff0(由ALU\_rs2\_sel🡺 IMM mode 為0進行選擇)。LW(在EXE Stage的Control signal 🡺 EXE\_MEM\_DM\_read為1)作法為抓取Memory 中 rs1 + imm 的全部32 bit 值。(搭配MEM Stage解釋) ，計算出來ALU\_out結果為0x808c。



1. **LH**

ALU\_ctrl 設為 5’d0(採用加法運算)，目標為rs1 +imm。

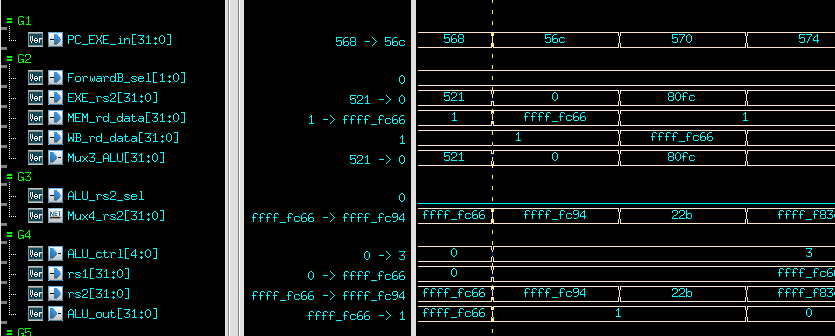
其於EXE Stage 作法完全與LW相同，剩下如何處理資料再搭配MEM Stage解釋。

1. **SLTI<當下執行的PC(PC\_EXE\_in) 🡺0x56c>**

ALU\_ctrl 設為 5’d3(SLT type)，目標為rs1 +imm。

rs1為0xffff\_fc66 , rs2 為0xffff\_fc94 (由ALU\_rs2\_sel進行選擇🡺 IMM mode 為0進行選擇)。可清楚看見，Mux4\_rs2並未選擇Mux3\_ALU(forwarding mux)結果，而是採用IMM結果。

SLTI會進行比較大小，並且是有signed。可見，IMM(rs2)在負值時比rs1大，故ALU\_out結果為1。

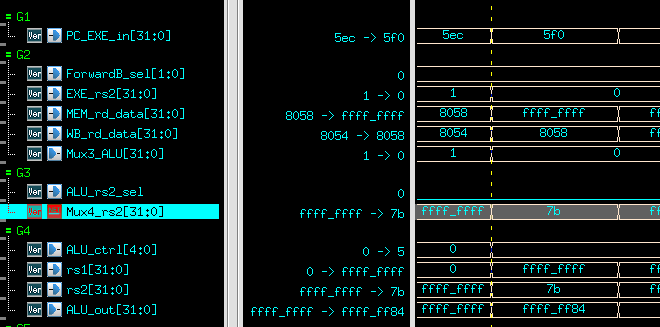


1. **XORI<當下執行的PC(PC\_EXE\_in) 🡺0x5fc>**

ALU\_ctrl 設為 5’d5(xor type)，目標為rs1 +imm。

rs1為0xffff\_ffff , rs2 為0x7b (由ALU\_rs2\_sel進行選擇🡺 IMM mode 為0進行選擇)。可清楚看見，Mux4\_rs2並未選擇Mux3\_ALU(forwarding mux)結果，而是採用IMM結果。

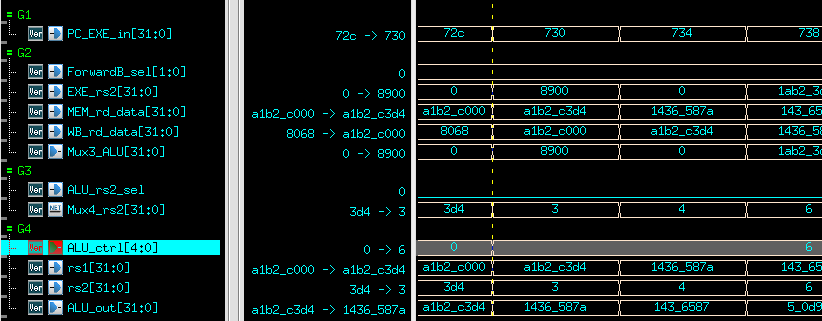
XORI會進行xor計算，故ALU\_out結果為0xffff\_ff84。



1. **SRLI<當下執行的PC(PC\_EXE\_in) 🡺0x730>**

ALU\_ctrl 設為 5’d6，rs1為0xa1b2\_c3d4，rs2 為 0x3(由ALU\_rs2\_sel進行選擇🡺 IMM mode 為0進行選擇)。可清楚看見，Mux4\_rs2並未選擇Mux3\_ALU(forwarding mux)結果，而是採用IMM結果。

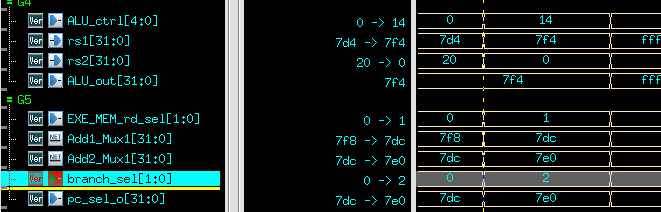
對rs1右移(shamt) rs2[4:0] 🡺3後，須注意為Logic shift，故為unsigned，以此例MSG為1，過右移時只需補0，透過ALU\_out 送出，為0x1436\_587a



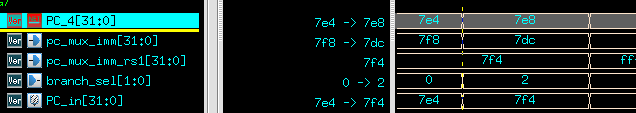
1. **JALR**

ALU\_ctrl 設為 5’d14，rs1為0x7f4，rs2 為 0x0

其rd值需要更新為PC+4，此時Control Signal 🡺MEM\_rd\_sel 在EXE Stage 裡為2’d1，用來將rd\_data選為PC+4(0x7e0)。



而PC值更新imm+rs1，可看到這為Branch Ctrl進行控制。當branch\_sel送給IF Stage 中的MUX進行選擇，2’d2代表選擇imm+rs1並將其LSB設為0，其值為0x7f4



#### ALU S-type

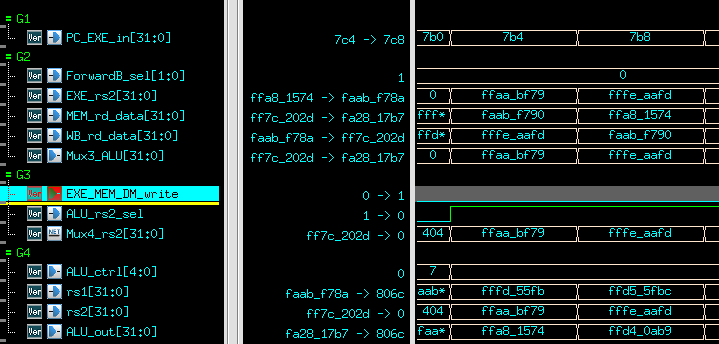
ALU\_Ctrl\_op在從ControlUnit產生時為ADD\_type，其目標是將rs1值與IMM值去進行相加，當作寫入Data Memory 的address。

IMM\_type在從ControlUnit產生時為 Imm\_S，其組成的IMM為**{{20{Instr\_in[31]}}, Instr\_in[31:25], Instr\_in[11:7]}**

**SW<當下執行的PC(PC\_EXE\_in) >**

ALU\_ctrl 設為 5’d0(採用加法運算)，目標為rs1 +imm。

rs1為0xfffd\_55fb，rs2 為 0xffaa\_bf79(由ALU\_rs2\_sel🡺IMM mode 為0進行選擇)。SW(在EXE Stage的Control signal 🡺EXE\_MEM\_DM\_write為1)作法為抓取Memory 中 rs1 + imm 的全部32 bit 值。(搭配MEM Stage解釋) ，計算出來ALU\_out結果為0xffa8\_1574。



**SH**  
ALU\_ctrl 設為 5’d0(採用加法運算)，目標為rs1 +imm。

其於EXE Stage 作法完全與SW相同，剩下如何處理資料再搭配MEM Stage解釋。

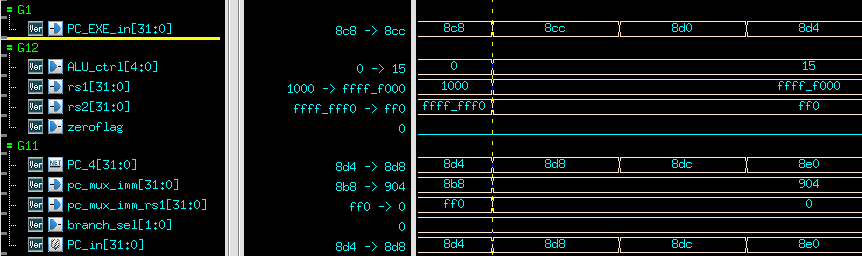
#### ALU B-type

Branch 狀態需要依照rs1 rs2關係，計算其zero\_flag，並將值送入Branch Ctrl進行計算。下列用3例子進行說明:

1. **BEQ<當下執行的PC(PC\_EXE\_in) 🡺0x8cc>**

ALU\_ctrl 設為 5’d15，rs1為0xffff\_f000，rs2 為 0xff0。

Beq指令用來確認rs1、rs2是否相同，如果相同，zero\_flag給1，就將PC 更新為PC+imm。反之，將PC更新為PC+4。此例為rs1!=rs2 ，zero\_flag 為0。而PC值更新為PC+4，可看到這為Branch Ctrl進行控制。當branch\_sel送給IF Stage 中的MUX進行選擇，2’d0代表選擇PC+4，為0x8d8。

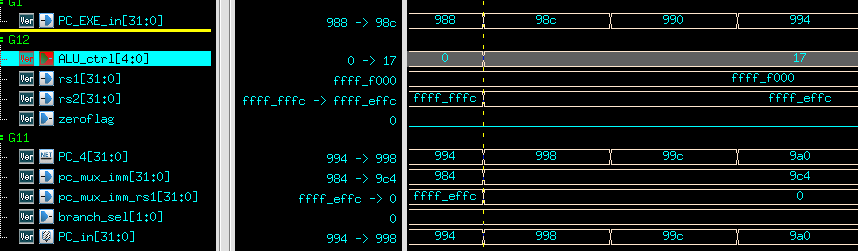


符合

1. **BLT<當下執行的PC(PC\_EXE\_in) 🡺0x98c>**

ALU\_ctrl 設為 5’d17，rs1為0xffff\_f000，rs2 為 0xffff\_effc。

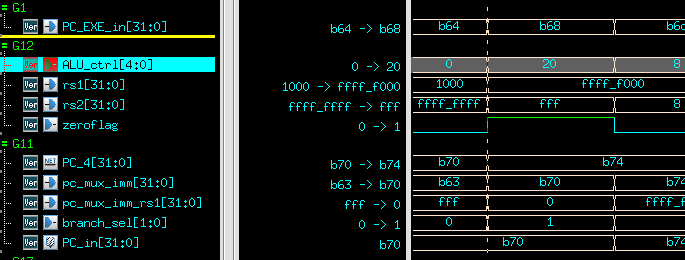
BLT指令用來確認rs1<rs2是否符合，如果符合，zero\_flag給1，就將PC 更新為PC+imm。反之，將PC更新為PC+4。此例為rs1>rs2 ，zero\_flag 為0。而PC值更新為PC+4，可看到這為Branch Ctrl進行控制。當branch\_sel送給IF Stage 中的MUX進行選擇，2’d0代表選擇PC+4，為0x998。

****

1. **BGEU<當下執行的PC(PC\_EXE\_in) 🡺0xb68>**

ALU\_ctrl 設為 5’d20，rs1為0xffff\_f000，rs2 為 0xfff。

BGEU指令用來確認rs1>=rs2(unsigned)是否符合，如果符合，zero\_flag給1，就將PC 更新為PC+imm。反之，將PC更新為PC+4。此例為rs1>=rs2 ，zero\_flag 為1。而PC值更新為PC+imm，可看到這為Branch Ctrl進行控制。當branch\_sel送給IF Stage 中的MUX進行選擇，2’d1代表選擇PC+imm，為0xb70。

****

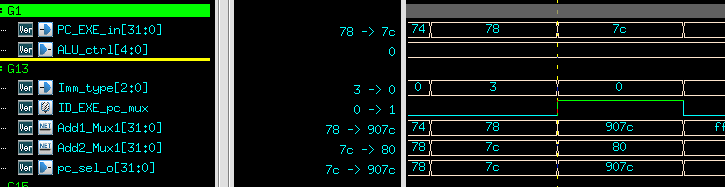
#### ALU U-type

用於處理較大的immediate運算。這類指令會將immediate儲存在指令的upper 20 bit，其與 PC（程式計數器）或直接存入暫存器中。RISC-V 中常見的 U-type 指令包括 LUI 和 AUIPC，具體波形如下：

1. **AUIPC**

ALU\_ctrl 設為 5’d0(採用加法運算)，目標為rd = pc +imm。

原pc為0x7c，IMM值為0x9000。AUIPC在EXE\_pc\_mux選擇為1’b1 (Add1\_Mux1)(pc +imm) ，計算出來ALU\_out結果為0x907c。

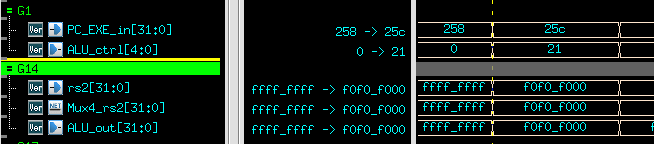
****

1. **LUI<當下執行的PC(PC\_EXE\_in) 🡺0xb94>**

ALU\_ctrl 設為 5’d21(imm\_type)，目標為rd = imm。

rs2 為0xf0f0\_f000 (由ALU\_rs2\_sel進行選擇🡺 IMM mode 為0進行選擇)。可清楚看見，Mux4\_rs2並未選擇Mux3\_ALU(forwarding mux)結果，而是採用IMM結果。

**LUI**會直接將rs2給，ALU\_out結果為0xf0f0\_f000。



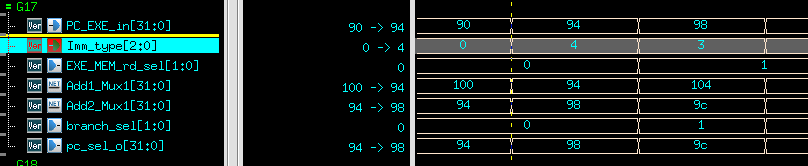
#### ALU J-type

用於無條件跳躍，主要用於跳轉到一個指定的相對位址來執行程式。

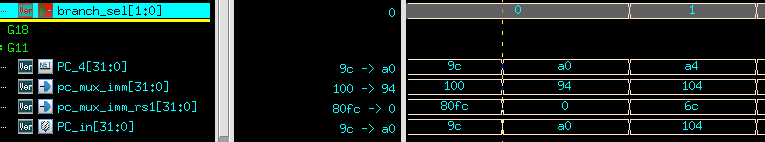
1. **JAL(作法與I\_type🡺JAL相似)** **<當下執行的PC(PC\_EXE\_in) 0x94>**

IMM在上一ID Stage，Imm\_type狀態為J\_Branch(3’b100)

ALU\_ctrl 設為 5’d0，其rd值需要更新為PC+4，此時Control Signal 🡺MEM\_rd\_sel 在EXE Stage 裡為2’d1，用來將rd\_data選為PC+4(0x9c)。



而PC值更新imm+PC，可看到這為Branch Ctrl進行控制。當branch\_sel送給IF Stage 中的MUX進行選擇，2’d1代表選擇imm+PC，其值為0x104。



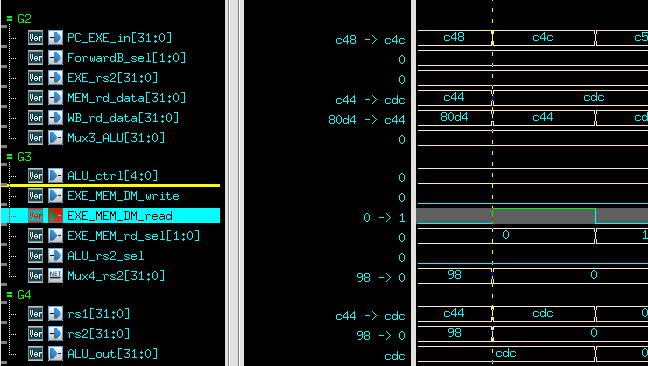
#### ALU F-type (LW/SW)

此狀態較為特殊，由於LW/SW計算地址並不受Integer 或Floating Point狀態影響，故設計上僅採原始路線，但處理值上注意會不同(MEM Stage)

1. **FLW<當下執行的PC(PC\_EXE\_in) 0xc4c>**

ALU\_ctrl 設為 5’d0(採用加法運算)，目標為rs1 +imm。

rs1為0xcdc，rs2 為 0x0 (由ALU\_rs2\_sel🡺 IMM mode 為0進行選擇)。LW(在EXE Stage的Control signal 🡺EXE\_MEM\_DM\_read為1)作法為抓取Memory 中 rs1 + imm 的全部32 bit 值。(搭配MEM Stage解釋) ，計算出來ALU\_out結果為0xcdc。



但由於讀出後要寫入register file(需要轉至Floating Point)，故reg\_file\_FP\_write 為1。



1. **FSW**

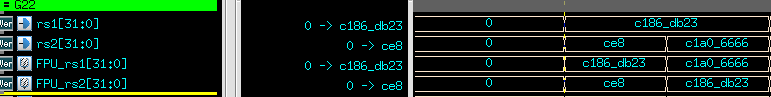
ALU\_ctrl 設為 5’d0(採用加法運算)，目標為rs1 +imm。

其於EXE Stage 作法完全與SW相同，剩下如何處理資料再搭配MEM Stage解釋。(須注意為其寫入Data\_memory中的值為Floaating point 的rs2\_FP\_Data)

#### ALU F-type

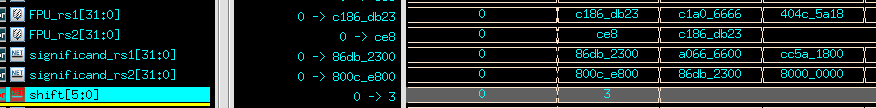
在運作當中，採用標準IEEE-754的規格，故需分為3部分，1)signed bit 2)exp (8 bit) 3)fraction (23 bit)。

運算時首要考慮2)exp (8 bit)+ 3)fraction (23 bit)。rs1為(0xc186\_db23)， rs2為(0xc1a0\_6666)🡺不signed bit rs2>rs1 ，故互換給FPU\_rs1 FPU\_rs2。



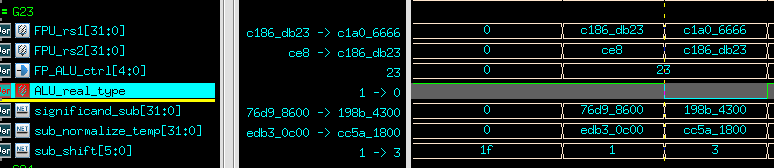
針對Fration 需復原為最大bit為1.xxxx 或0.xxxx

可看到下圖significand 為首bit🡺兩者皆為1.xxxx 後面為了預防需要round to even，皆補上8’d0。這邊由於exp已經對其，故shift為0



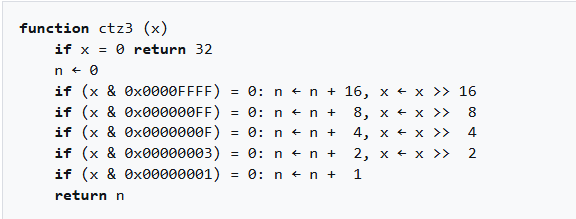
**FADD.S/FSUB.S**

下圖同時呈現ALU\_ctrl 設為 5’d23(浮點數減法運算)，但由於有大小對調換。可以看到ALU\_real\_type呈現左邊為減法(1’b1)，右邊為減法(1’b0)

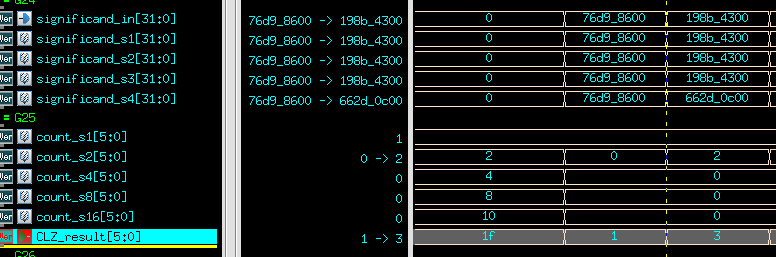


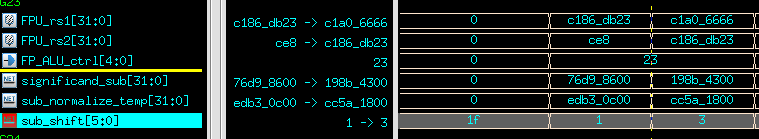
CLZ做法🡺 由於減法後可能出現exp比遠本小，變為0.xxxxxx，需要進行校正，讓Fration 校正回1.xxxxx

下圖為參考演算法

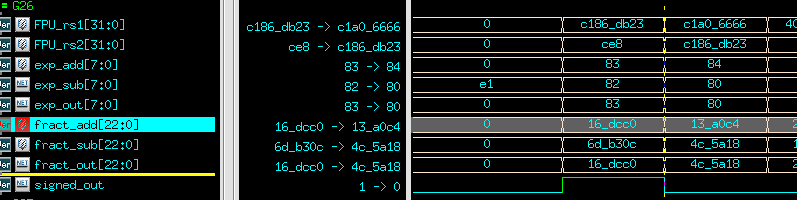


利用遮罩原理找出實際要移地位數，下圖為count後為CLZ\_result => 3



****

最後再將計算完的signed bit /exp/fraction組合起來，及為結果



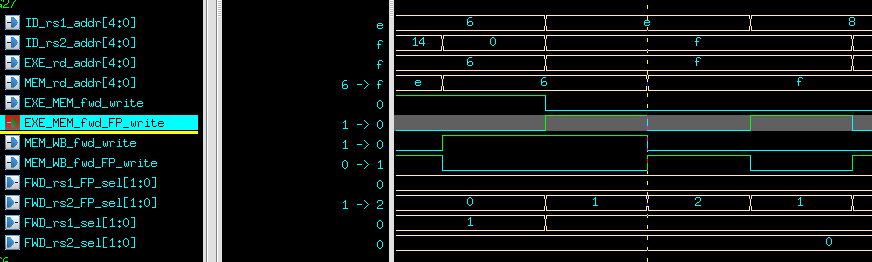
#### Control and Status Register (CSR)

在此Lab 當中，主要以計算1)執行instruction 數量 2)執行總cycle為主。

1. Instruction
2. cycle

* **Forwarding Unit**

採用ID\_rs1/rs2\_addr去跟前幾條執行到MEM/WB的指令去比較，如果符合，如下圖fwd將拉起，並會分為FP及一般Integer模式。



* **Hazard Control**

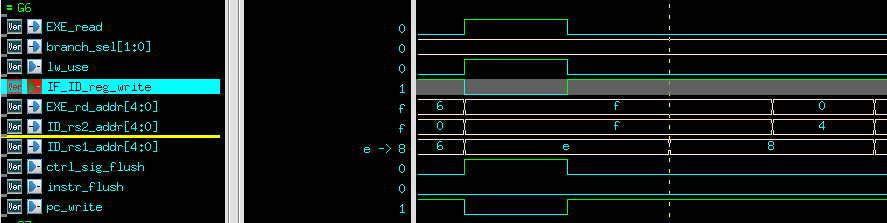
先確認branch\_sel是否為branch 狀態(2’d0為否)

再確認EXE\_read &&

((EXE\_rd\_addr == ID\_rs1\_addr)||(EXE\_rd\_addr== ID\_rs2\_addr)

此判斷用以決定有沒有lw\_use

下面確實發生lw use狀態，可以看到採用flush製造stall，來進行delay



### MEM stage

* rd\_data此時會由4者進行選擇，其控制訊號為MEM\_rd\_sel:
  + - 1. **(1:下圖右紅框)pc計算後的值(0xc444)**

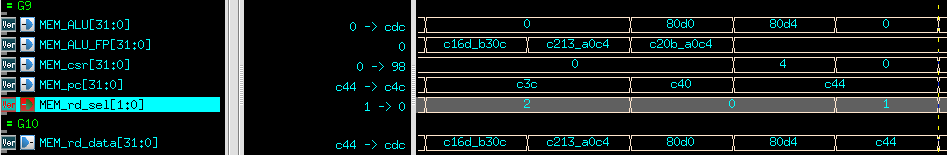
🡺當MEM\_rd\_sel=2’d1，其選擇為MEM\_pc所計算的值

* + - 1. **(0:下圖中紅框)ALU (Integer)計算後的值(0x8044)**

🡺當MEM\_rd\_sel=2’d0，其選擇為MEM\_ALU所計算的值

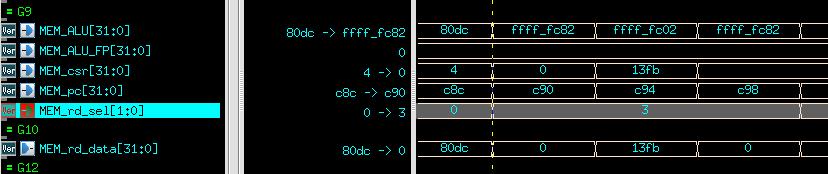
* + - 1. **(2:下圖左紅框)ALU (Floating Point)計算後的值(0xc16d\_b30c)**

🡺當MEM\_rd\_sel=2’d2，其選擇為MEM\_ALU\_FP所計算的值



* + - 1. **(3)CSR計算後的值(0x13fb)**

🡺當MEM\_rd\_sel=2’d3，其選擇為MEM\_csr所計算的值



* **Data\_Memory資料讀寫**

Chip\_select為是否執行讀寫動作(active low)，需要執行時為0

SRAM\_web 🡺 low: SW ，high: LW

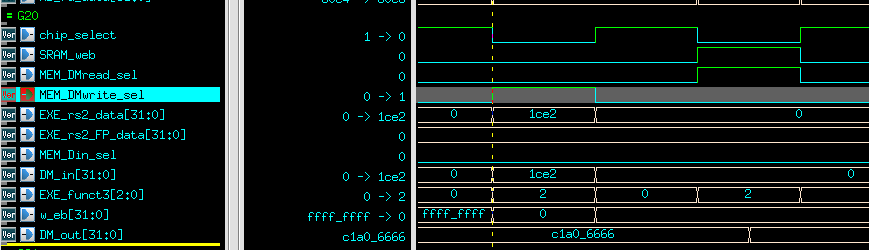
* + - 1. **Store:**

其值為從EXE Stage所送過來rs2最終結果，其值需要經過Mux進行(Integer/Floating Point)選擇。

選擇後需要注意，所要執行的是Byte Hex Full write

下圖為採用full wirte 🡺w\_eb全部active low

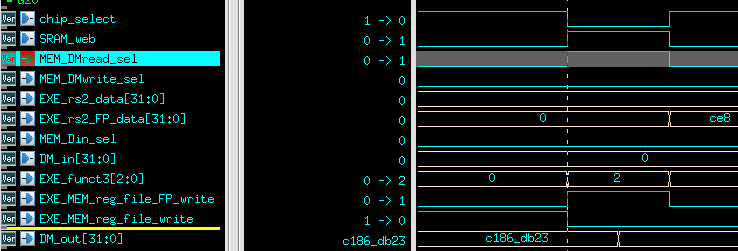
rs2🡺0x1ce2全部寫入SRAM



* + - 1. **Load:**

其值為從Data\_Memory所讀出的結果，需要進行處理

，其值需要經過Mux進行(Integer/Floating Point)選擇



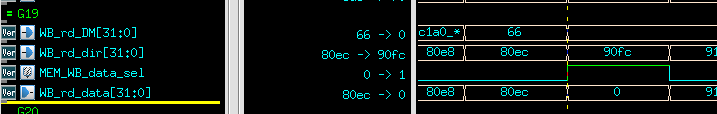
### WB stage

* WB\_rd\_data此時會由2者進行選擇，並將其送回reg\_file，其控制訊號為WB\_data\_sel:
  + - 1. MEM\_rd\_data送進的值

🡺WB\_data\_sel=0 ，採用WB\_rd\_dir中的值(0x80ec)

* + - 1. DM送至WB stage

🡺WB\_data\_sel=1 ，採用WB\_rd\_DM中的值(0x0)

****

## Lessons Learned

* **Verilog 語法:**

|  |  |
| --- | --- |
| 設計系統化: | 有用上macro 去對參數進行設計跟檔案include。  在設計上也在部分地方嘗試使用function，去對會重複使用的功能直接再利用。 |
| 語法學習: | 在課堂上有教到if 的priority 問題，並用unique if來做設計上的優化 |

* **Tool使用:**   
   這一次確切了解相關slack、timing 、critical path在設計中如何去查看這些資訊，找出設計較差的地方，重新將critical path縮短。

但有發現到一個奇特的點，不知道是不是跟製程有關。當slack已經壓到0時(cycle period為5 ns)，此時再繼續將時間壓縮，slack仍然為0，壓縮後的設計面積也未縮小，進行RTL 及Gate Level Simultion 仍然正常，這是比較特別的地方。

* **CPU設計:**  
  主要將過去計結課所學整體概念完全複習一遍，並時做出來，當中包括幾個這次比較不熟悉的地方

1. **Hazard control:**  
    在設計中，複習了過去學到的 hazard 控制技術，究竟是將指令暫停(在暫存器中)，利用從register插入stall(IF)或是利用Flush(ID\_EXE register) 來解決問題，更重要要注意採用哪個stage rd跟rs1、rs2進行設計，否則可能會發生我原先設計上比獲得值晚1 cycle。
2. **Forwarding:**以前學習計結也是只有看過這一部份內容，並未實作。 這次作業的設計Forwarding蠻常發生。透過此機制，有效減少了pipeline要暫停的次數，可以減少運算的cycle數。
3. **Floating Point 計算:**

此次作業，實作了 RISC-V 的浮點指令。由於浮點運算的特性，需要先 對格式處理，並根據所要進行的加減法重新判斷。

計算後，其中最難的部分需要進行rounding to even 部分。

當中還有要將處理後的significand進行移位，為了不要有任何case 上syntax問題，利用count leading zero(CLZ)計算32bit的移位，但需要做5次，故所花費時間相較而言比較多，因為只能採用sequential 方式。

* **線路優化設計:**   
   在Floating point 及CSR 設計上，為了避免增加過多選擇器及控制訊號中，rd部分將其與pc及Integer ALU選擇調整到同一個Mux，在控制訊號處理上要更加小心，並且要搭配register file寫到哪一個都要同步處理，但整體效果算不錯的，不用再多加很多層Mux去處理。

在Floating Point LW/SW 的address計算，由於運算上皆與原本的LW/SW相同，故僅在最後的data利用Control Signal 進行處理，這樣可以減少需要再拉線路去FP\_ALU重複的運算。