VLSI System Design (Graduate Level)

Fall 2022

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

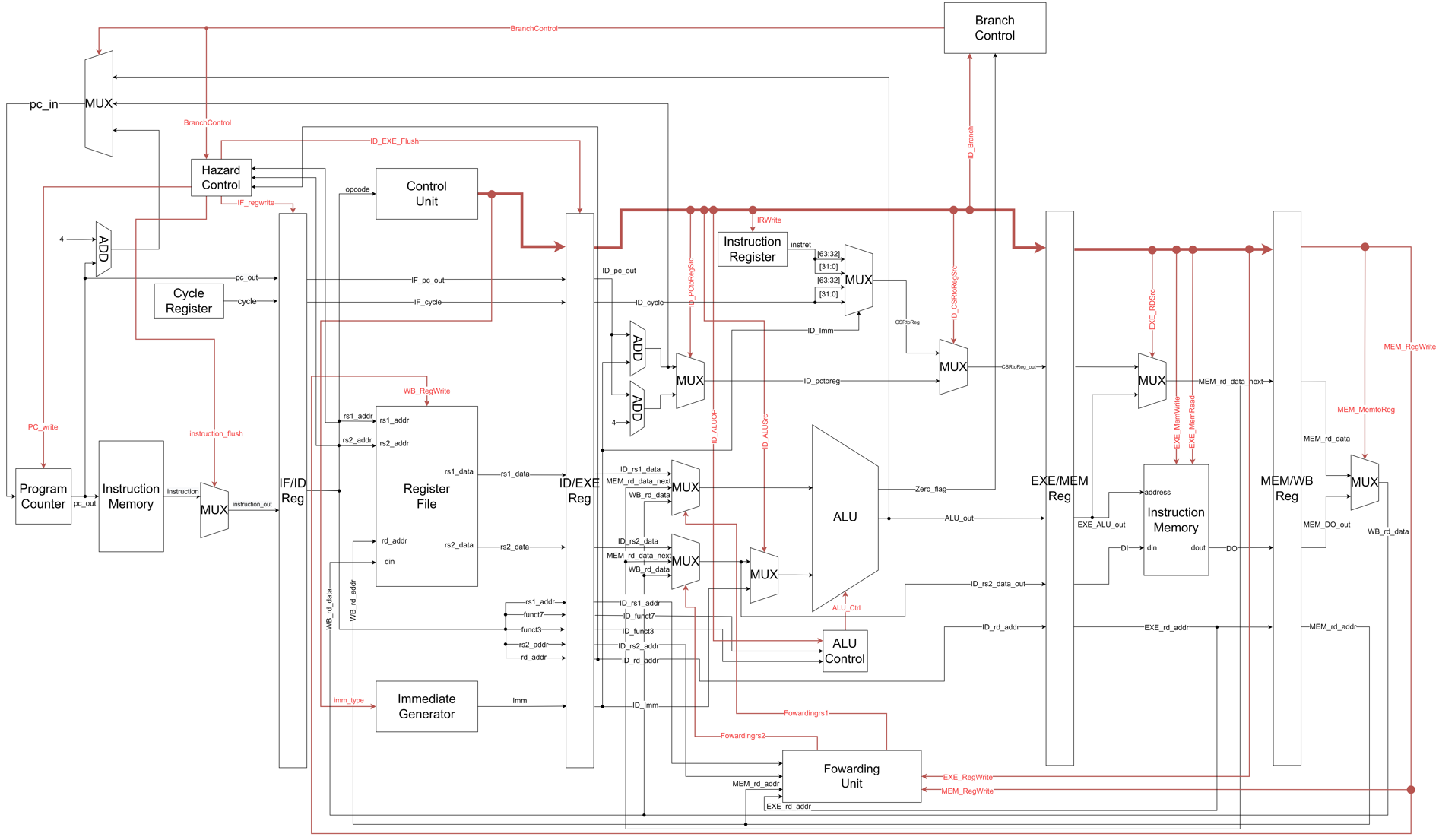
Student name: 黃冠予

Student ID: M16111064

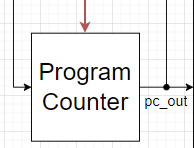
1. **Summary**

在過去的練習中，都未曾寫過如此大範圍的題目，所以在寫這次作業時感覺到了前所未有的挑戰。而在寫CPU過程中可以發現，許多地方都是過去曾經學過的觀念，但在實作時卻還是會遇到許多的問題，像是發生Branch Hazard時所需要應對的方法，以及遇到Data Hazard時該如何去解決。在最後完成作業時，可以感覺到對CPU的了解更加深入，並且提升了許多寫verilog的能力，獲益良多。

1. **Blockdiagram**

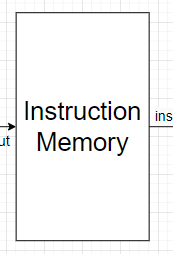
****

1. ProgramCounter



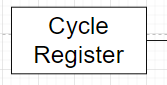
計數instruction adderss的counter,每經過一個clk會栓鎖輸入,可以使用PC\_write訊號來控制是否繼續栓鎖。

1. InstructionMemory



此Memory為一SRAM，根據不同的address給出相對應的instruction。

1. CycleRegister



計數當前Cycle數量

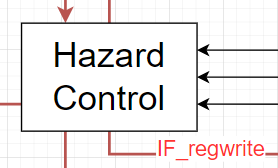
1. IF/ID Reg

**一張含有 文字, 拉門 的圖片

自動產生的描述**

負責把IF階段的訊號栓鎖至ID階段，並且使用IF\_Regwrite來控制是否暫停訊號傳遞

1. HazardControl



判斷是否為Branch Hazard或是Data Hazard的load-use情況，如果為Branch Hazard(即發生Branch/Jump),則使IF/ID Reg中的instruction變成0以及讓ID/EXE Reg裡的控制訊號同樣轉變為0，使發生Branch後的原本後2clk的指令為NOP。

如果為load-use(即在EXE階段發現有讀取DM的訊號以及EXE階段準備寫回register的位址和ID階段使用register的位址相同),則使ProgramCounter和IF/ID Reg暫停並讓ID/EXE Reg裡的控制訊號轉變為0使其為NOP。

1. ControlUnit

一張含有 桌 的圖片

自動產生的描述

根據不同Type的opcode給出不同的控制訊號。

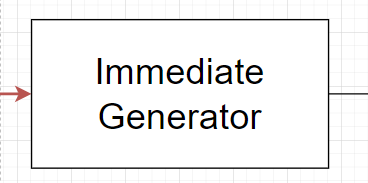
1. RegisterFile

一張含有 桌 的圖片

自動產生的描述

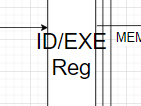
為32個32bits的Register，可以根據rs1,rs2位址給出相對應的資料，並且接收來自WB階段的register位址以及資料。

1. ImmediateGenerator



根據ControlUnit所分出的type，使Immediate對該type進行排列並SignExtension。

1. ID/EXE Reg



負責把ID階段的訊號栓鎖至EXE階段，並且使用Flush訊號來控制其中的訊號是否轉變為NOP。

1. InstructionRegister



計數當前Instruction執行的次數，只要instruction不為NOP就會持續遞增。

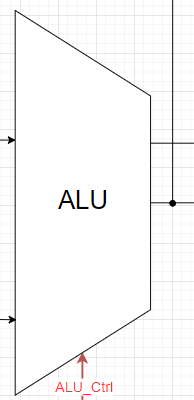
1. ALUControl

一張含有 桌 的圖片

自動產生的描述

根據不同的type以及funtc3,funct7來控制ALU的運算。

1. ALU



跟據ALU\_Ctrl使ALU對輸入進行不同的運算，其中在B type instruction成立時，會拉起Zeroflag代表發生Branch。

1. FowardingUnit

一張含有 桌 的圖片

自動產生的描述

負責解決Data Hazard，如果發生Data Hazard(即在MEM階段發現有寫入Registerfile的訊號，以及MEM階段準備寫回register的位址和EXE階段使用register的位址相同)，則使在EXE階段所使用的Rs1/Rs2轉換成在MEM階段準備寫回Registerfile的數值。

如果為DataHazard的load-use(即在WB階段發現有寫入Registerfile的訊號，以及WB階段準備寫回register的位址和EXE階段使用register的位址相同)，則使在EXE階段所使用的Rs1/Rs2轉換成在WB階段準備寫回Registerfile的數值。

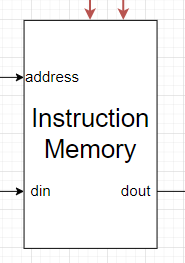
1. EXE/MEM Reg

一張含有 文字, 拉門 的圖片

自動產生的描述

負責把EXE階段的訊號栓鎖至MEM階段。

1. InstructionMemory



根據Read/Write來決定讀取或是寫入，在讀取時可以根據需要的型態來進行SignExtension。而在寫入時，需要給定WEB來控制word中那些byte需要被寫入。

1. MEM/WB Reg

一張含有 文字, 拉門 的圖片

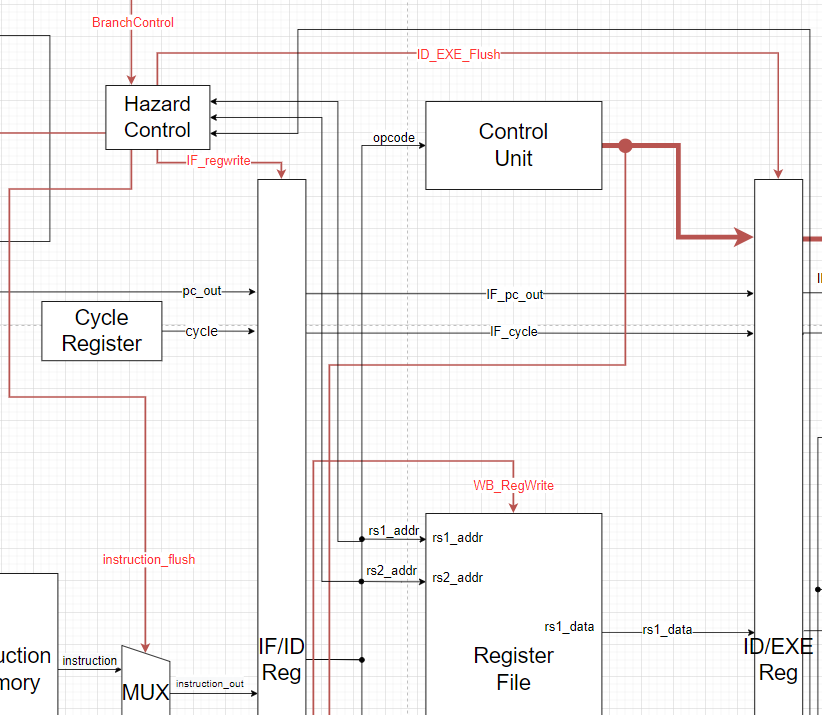
自動產生的描述

負責把MEM階段的訊號栓鎖至WB階段。

1. **Lesson learned**

在這次作業中，會了解到CPU內的結構，並且在實作面中臨到兩種Hazard的問題，分別是Branch Hazard以及Data Hazard。

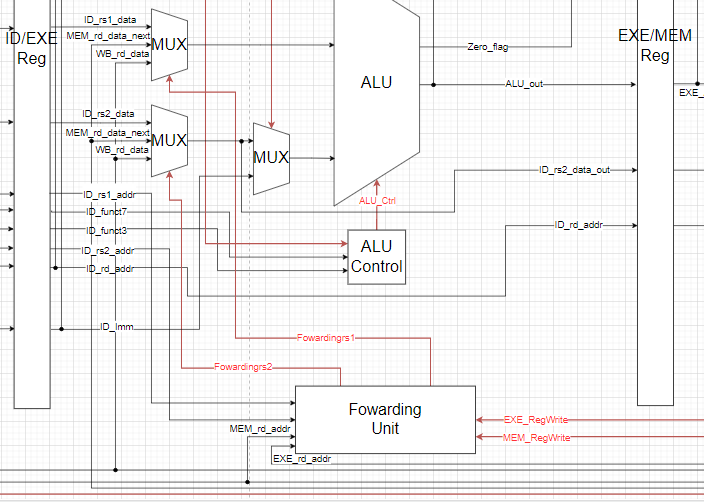
Branch Hazard:由於Branch的判定是在EXE階段被執行，因此當發生Branch時，在IF,ID不需要的指令會繼續被執行。

解決方法: 

在電路中增加HazardControl,每當發生Branch時instruction\_flush以及ID\_EXE\_Flush會把在IF/ID,ID/EXE等register中的instruction改為NOP，使執行結果正確。

DATA Hazard:

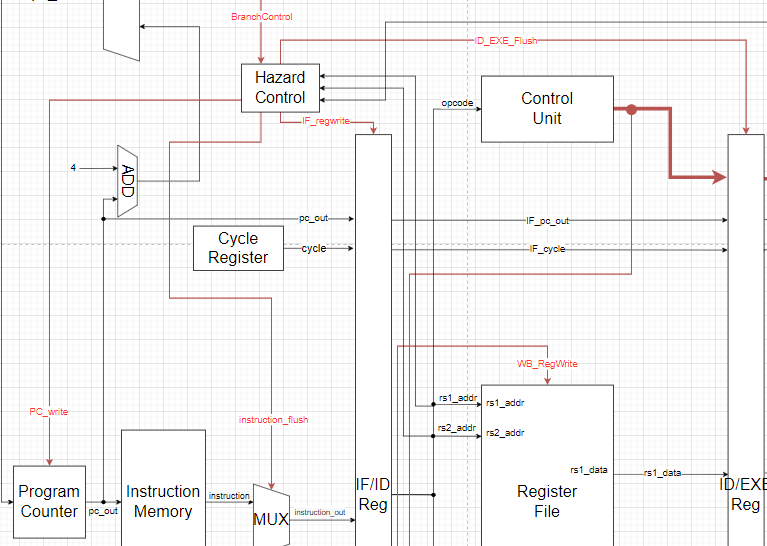
(1)在EXE階段算完的值，需要再2個CLK才能把值寫回Register file，在這段期間若有指令需要用到該值(即Data Dependency)，將會發生錯誤。

解決方法: 

增加Fowarding，比對在EXE階段所使用的rs1及rs2之register address(圖中ID\_rs1\_addr,ID\_rs2\_addr)是否為在MEM或WB中要寫回之address(EXE\_rd\_addr,MEM\_rd\_addr)。

(2)load-use:跟上述情況相似，差別在於此時寫回的值來自於Data Memory，因此跟一般情況相比會需要多一個clk。

解決方法:



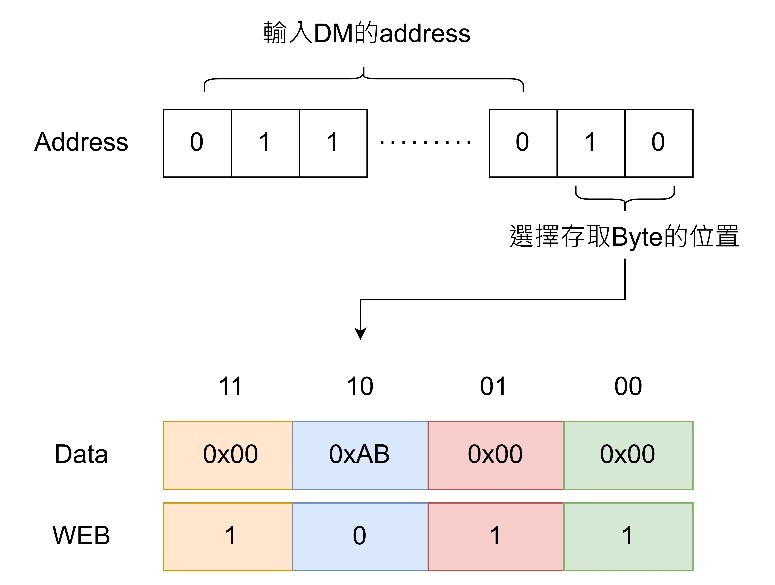
使用HazardControl ，比對在ID階段所使用到register address是否為EXE階段所寫回去之address(EXE\_rd\_addr)，如果成立，把PC以及IF/ID Register暫停一次，並且把ID/EXE中的instruction改為NOP以此多增加1個CLK時間，之後再使用Fowarding即可解決。

除了了解到Hazard如何解決，也了解到如何解決SB 中Byte address轉換到Word address的問題。

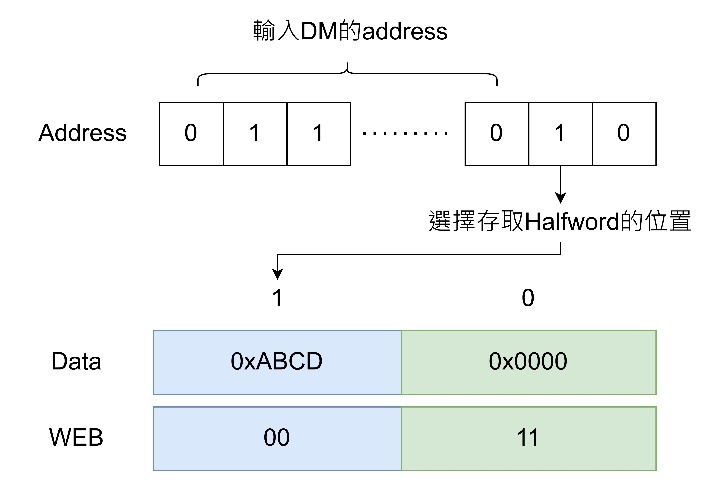
由於Memory為4個Byte 成一個單位，因此輸入DM的address並不會看見單個Byte的address

，藉此須利用Word address 加上 WEB來轉換相對應的 Byte address。

Save Byte:把在register中的Byte取出，並且把向DM輸入的位址前兩位元轉為0，而後使用被捨棄的2位元選擇填入Word中哪一個位置之Byte，最後在該Word中把剩下的部分填入0。



Save Halfword:把在register中的Halfword取出，並且向DM放入位址所指定的Halfword，在該Word中把剩下的部分填入0。

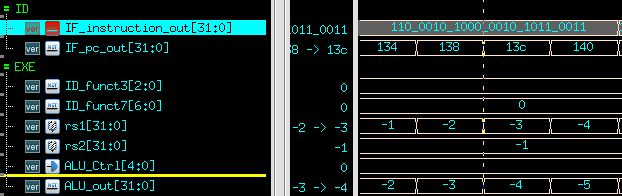


Load Byte:把在指定位址DM中的Byte取出，並且向register放入Byte，如果為LB就進行SignExtension，如果為LBU則剩下的bit全填入0。

Load Halfword: 把在指定位址DM中的Halfword取出，並且向register放入Halfword，如果為LH就進行SignExtension，如果為LHU則剩下的bit全填入0。

1. **Instruction Waveform**
2. R\_type

1.ADD

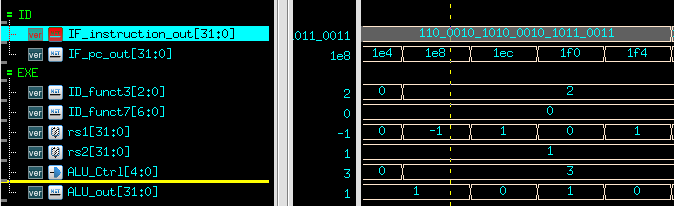


(1)Opcode=0110011,funct3=0,funct7=0(ADD)

(2)ALU\_Ctrl=0(ALU執行ADD)

(3)rs1=-3 , rs2=-1 , ALU\_out=rs1+rs2=-4

2.SLT

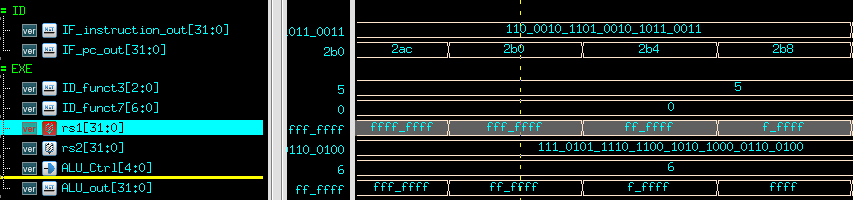


(1)Opcode=0110011,funct3=2,funct7=0(SLT)

(2)ALU\_Ctrl=3(ALU執行SLT)

(3)rs1=-1 , rs2=1 , ALU\_out=rs1<rs2=1 //比較rs2是否大於rs1

3.SLL

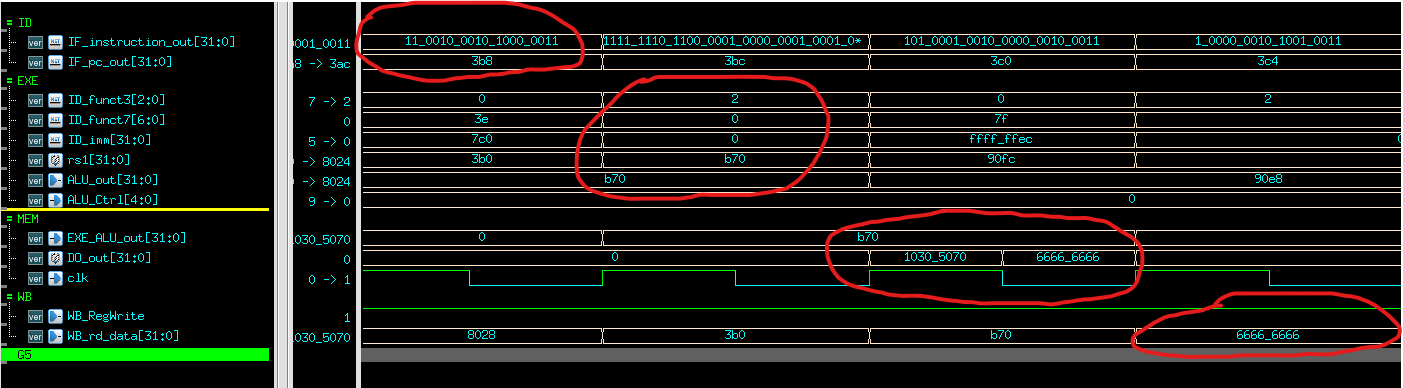


(1)Opcode=0110011,funct3=5,funct7=0(SRL)

(2)ALU\_Ctrl=6(ALU執行SRL)

(3)rs1=0xFFFFFFF , rs2[4:0]=4 ,

ALU\_out=rs1>>rs2=0xFFFFFF //把rs1右移一個byte

1. I\_type
2. LW

(1)Opcode=0000011,funct3=2 (LW)

(2)ALU\_Ctrl=0(ALU執行ADD)

(3)rs1=0xb70 , Imm=0 ,ALU\_out=0xb70

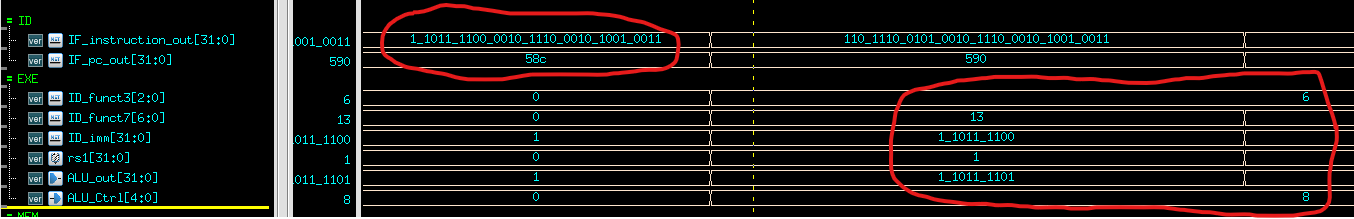
ALU\_out=rs1+Imm=0xb70

(4)EXE\_ALU\_out = 0xb70 //準備送入DM的address為0xb70

(5)DO\_out = 0x66666666 //DM在負緣給出的 DM[0xb70] 值為0x66666666

(6)WB\_rd\_data=0x66666666 //在WB階段把值給回傳給Registerfile

1. ORI



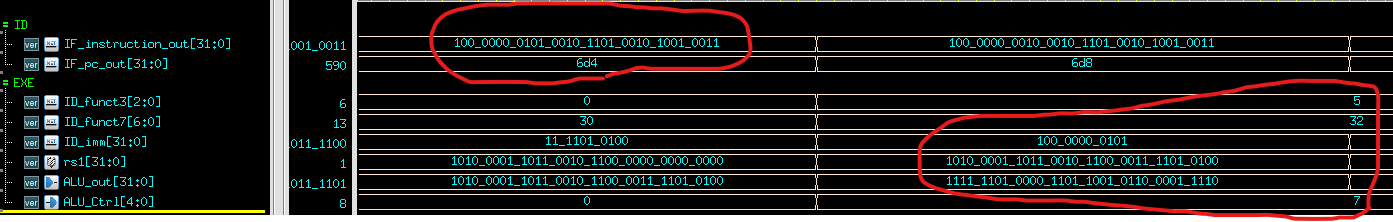
(1)Opcode=0010011,funct3=6 (ORI)

(2)ALU\_Ctrl=8(ALU執行OR)

(3)rs1=000000001 , Imm=110111100 ,

ALU\_out=rs1|rs2=110111101 //把rs1跟Imm進行OR

1. SRAI



(1)Opcode=0010011,funct3=5 (SRAI)

(2)ALU\_Ctrl=7(ALU執行SRA) ,

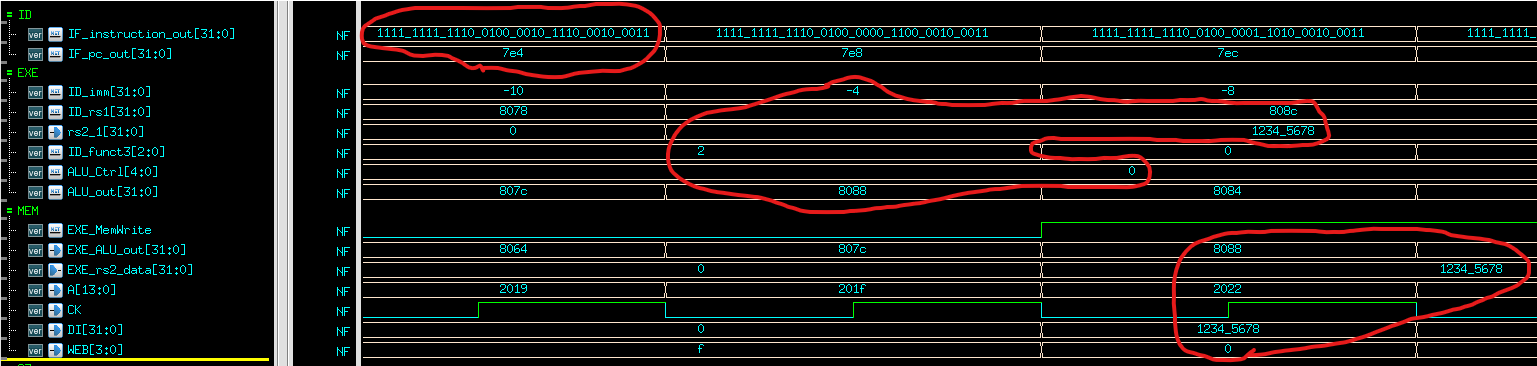
rs1=0xA1B2C3D4, Imm[4:0]=5 ,

(3)ALU\_out=rs1s>> Imm[4:0]

= 0xA1B2C3D4>>5 =0xFD0D961E

//用Imm[4:0]對rs1進行有號數右移

1. S\_type
2. SW



ID階段:

Opcode=0100011,funct3=2(SW)

EXE階段:

ALU\_Ctrl=0(ALU執行ADD) ,

ALU\_out=rs1+Imm=0x808C+(-4)=0x8088

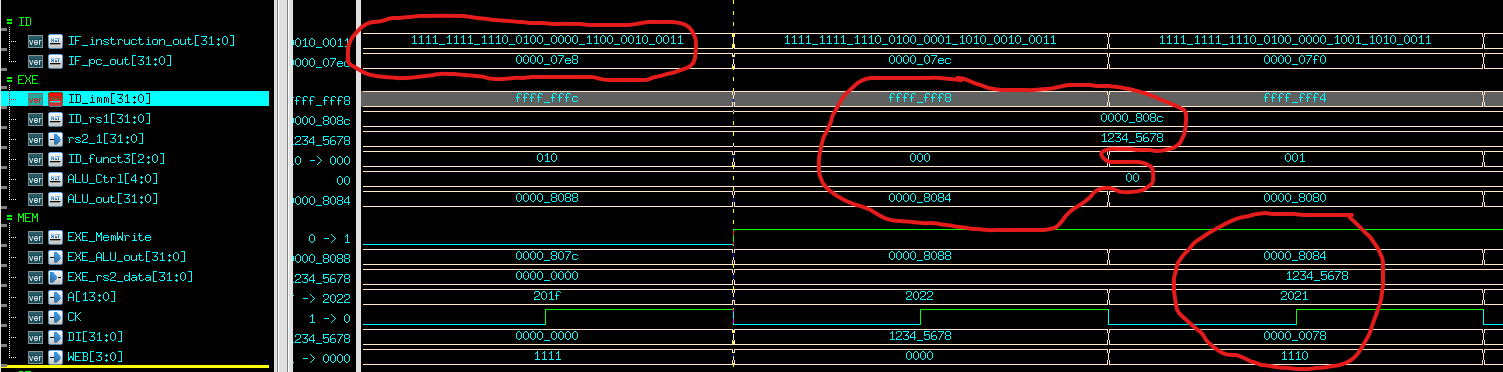
0x12345678為rs2\_1準備輸入DM之值

MEM階段:

A為EXE\_ALU\_out[15:2],即0x2022。

由於存入DM之值為一個word，因此WEB=4’b0000讓DI輸入MEM的值為rs2\_1，即0x12345678。

1. SB



ID階段:

Opcode=0100011,funct3=000(SB)

EXE階段:

對instruction中的imm進行sign\_extension得到0xFFFFFFF8即-8。

ALU\_Ctrl=0(ALU執行ADD) ,

ALU\_out=rs1+Imm=0x808C+(-8)=0x8084

0x12345678為rs2\_1準備輸入DM之值。

MEM階段:

A為EXE\_ALU\_out[15:2],即0x2021。

由於存入DM之值為一個Byte且EXE\_ALU\_out[1:0]為0，因此WEB=4’b1110讓DI輸入MEM的值為rs2\_1最右側的byte，即0x78。

1. SH

ID階段:

Opcode=0100011,funct3=001(SH)

EXE階段:

對instruction中的imm進行sign\_extension得到0xFFFFFFF4即-12。

ALU\_Ctrl=0(ALU執行ADD) ,

ALU\_out=rs1+Imm=0x808C+(-12)=0x8080

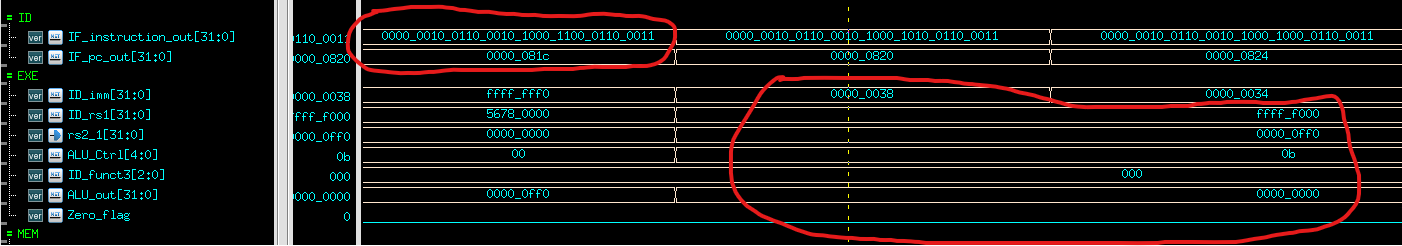
0x12345678為rs2\_1準備輸入DM之值

MEM階段:

A為EXE\_ALU\_out[15:2],即0x2020。

由於存入DM之值為一個halfword且EXE\_ALU\_out[1]為0，因此WEB=4’b1100讓DI輸入MEM的值為rs2\_1最右側的halfword，即0x5678。

1. B-type
2. BEQ



ID階段:

Opcode=1100011,funct3=000(BEQ)

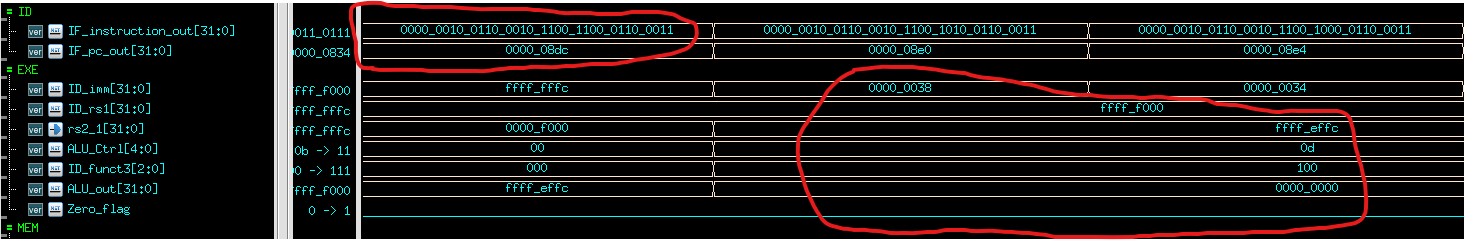
EXE階段:

ALU\_Ctrl=0xb(ALU執行BEQ,對rs1和rs2進行比較是否相等，如果成立則Zero\_flag拉起)

由於rs1不等於rs2(0xFFFFF000!=0x00000FF0)

因此Zero\_flag不會拉起，使此instruction不會發生Branch

1. BLT



ID階段:

Opcode=1100011,funct3=100(BLT)

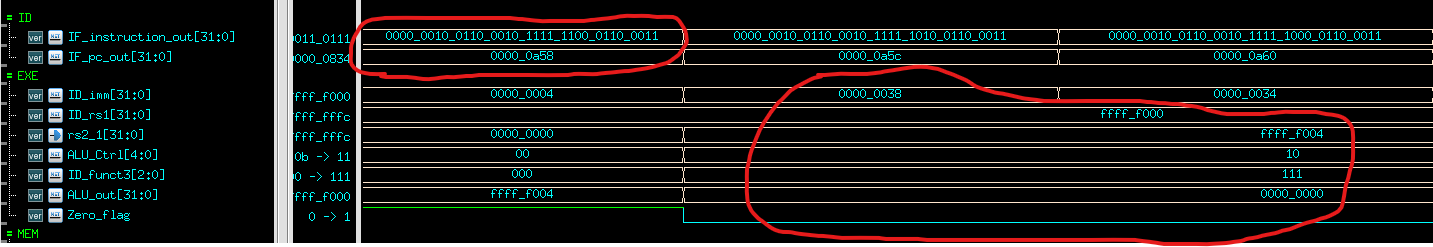
EXE階段:

ALU\_Ctrl=0xd (ALU執行BLT,比較有號數rs1是否小於 有號數rs2，如果成立則Zero\_flag拉起)

由於rs1大於rs2(0xFFFFF000(-4096)>0xFFFFEFFC(-4100))

因此Zero\_flag不會拉起，使此instruction不會發生Branch

1. BGEU



ID階段:

Opcode=1100011,funct3=111(BGEU)

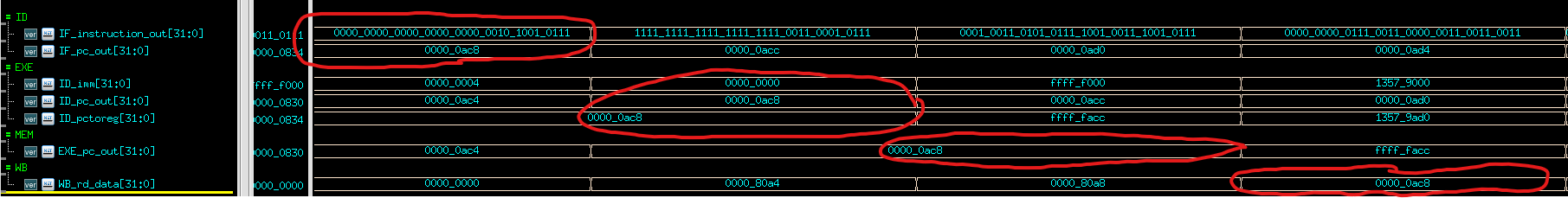
EXE階段:

ALU\_Ctrl=0x10 (ALU執行BGEU,比較無號數rs1是否大於等於無號數rs2，如果成立則Zero\_flag拉起)

由於rs1小於rs2(0xFFFFF000<0xFFFFF004)

因此Zero\_flag不會拉起，使此instruction不會發生Branch

1. U\_type
2. AUIPC



ID階段:

Opcode=0010111 (AUIPC)

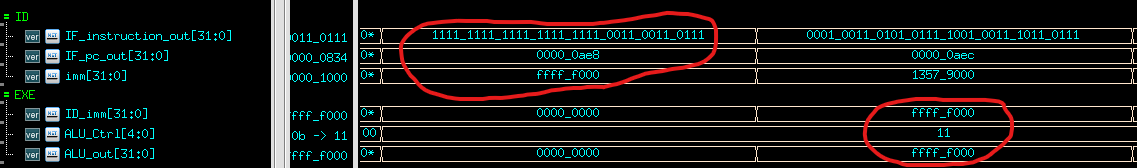
EXE階段:

EXE中的加法器使ID\_imm+ID\_pc\_out相加成為ID\_pctoreg(0x0+0xac8=0xac8)

MEM階段:把EXE階段之ID\_pctoreg值傳給WB階段

WB階段:可以從上圖中得知imm+pc從WB\_rd\_data傳回Registerfile

1. LUI



ID階段: Opcode=0110111 (LUI)

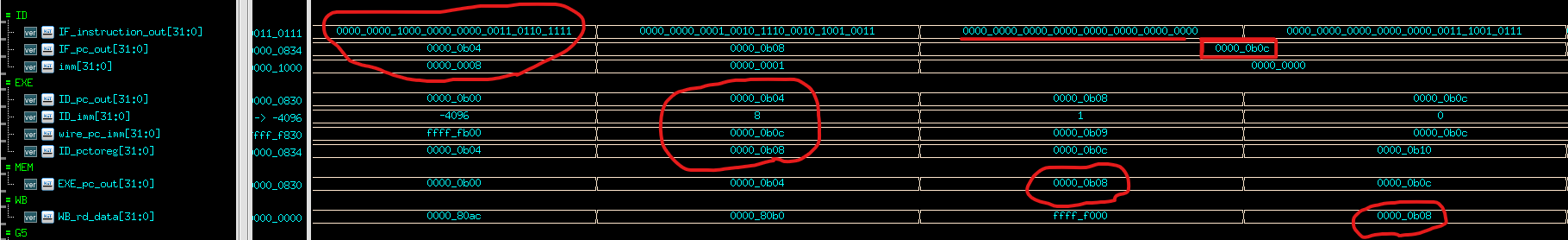
EXE階段:

ALU\_Ctrl=0x11(ALU輸出等於Imm)

ALU\_out=Imm=0xFFFFF000

1. J\_type

JAL:



ID階段: Opcode=1101111 (JAL)

Instruction中的Imm經過sign\_extension後為0x00000008

EXE階段:

1.EXE中的加法器使ID\_imm+ID\_pc\_out相加成為wire\_pc\_imm(0xb04+8=0xb0C)

2.EXE中的加法器使ID\_pc\_out+4相加成為ID\_pctoreg (0xb04+4=0xb08)

MEM階段:

1.把ID\_pctoreg傳去給EXE\_pc\_out

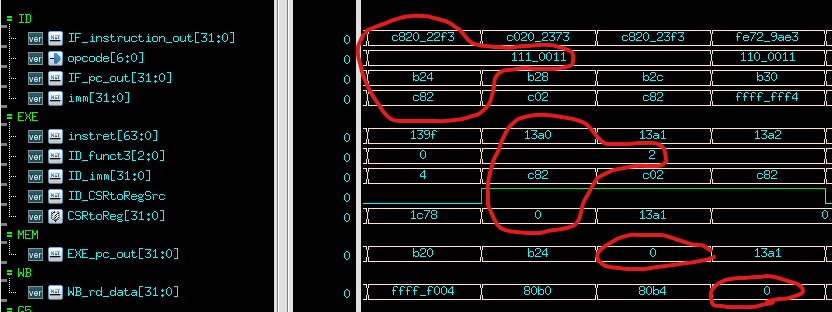
2.此時往圖上方可發現ID階段Instruction為0，這是因為Hazard\_control使用Instruction\_flush清空了原本的指令使其成為NOP

WB階段:

WB\_rd\_data把EXE\_pc\_out傳回給Registerfile

1. CSR\_type

1.RDINSTRETH



ID階段:

1. OPCODE= 1110011
2. funct3=0x2
3. imm[11:0]=0xC82(RDINSTRETH)

EXE階段:

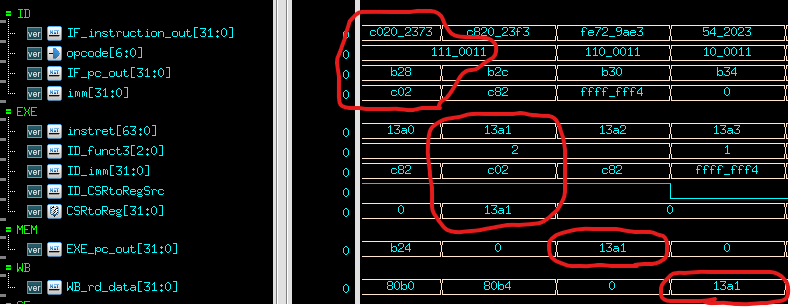
1. 由於instruction為RDINSTRETH,因此只輸出instret[63:32],即CSRtoREG為instret[63:32]=0x00

MEM階段:

1. EXE\_pc\_out把CSRtoREG傳給WB階段

WB階段:

1. WB\_rd\_data成功把值0x00回傳給Registerfile
2. RDINSTRET



ID階段:

* 1. OPCODE= 1110011
  2. funct3=0x2
  3. imm[11:0]=0xC02(RDINSTRET)

EXE階段:

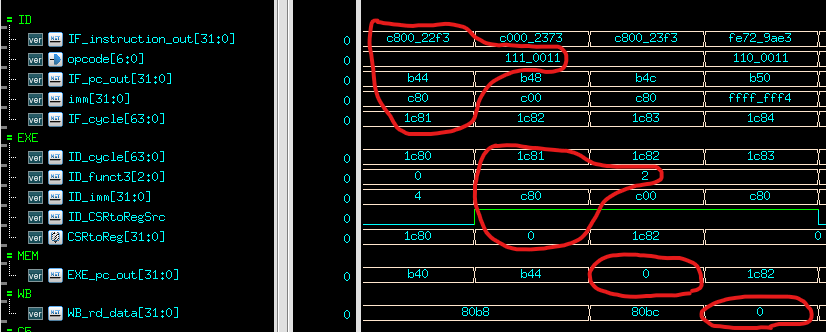
* 1. 由於instruction為RDINSTRET,因此只輸出instret[31:0],即CSRtoREG為instret[31:0]=0x13a1

MEM階段:

1. EXE\_pc\_out把CSRtoREG傳給WB階段

WB階段:

1. WB\_rd\_data成功把值0x13a1回傳給Registerfile
2. RDCYCLEH



ID階段:

1.OPCODE= 1110011

2.funct3=0x2

3.imm[11:0]=0xC80(RDCYCLEH)

EXE階段:

由於instruction為RDCYCLEH,因此只輸出 ID\_CYCLE[63:32],即CSRtoREG為instret[63:32]=0x00

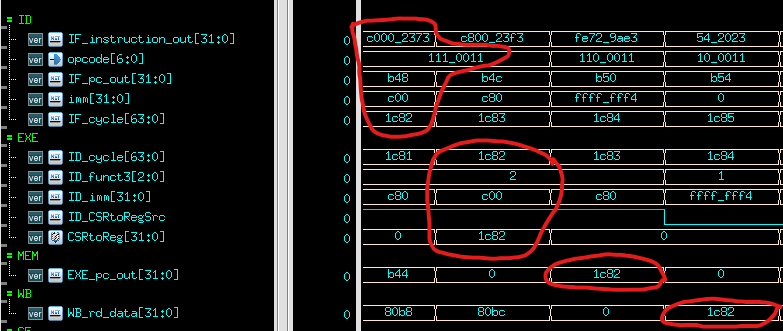
MEM階段:

EXE\_pc\_out把CSRtoREG傳給WB階段

WB階段:

WB\_rd\_data成功把值0x00回傳給Registerfile

4.RDCYCLE



ID階段:

1.OPCODE= 1110011

2.funct3=0x2

3.imm[11:0]=0xC00(RDCYCLE)

EXE階段:

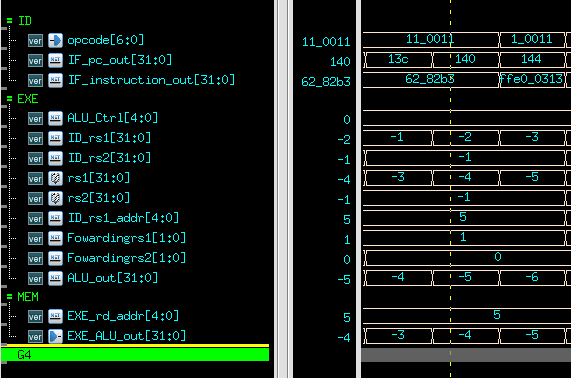
1.由於instruction為RDCYCLE,因此只輸出 ID\_CYCLE[31:0],即CSRtoREG為instret[31:0]=0x1C82

MEM階段:

1.EXE\_pc\_out把CSRtoREG傳給WB階段

WB階段:

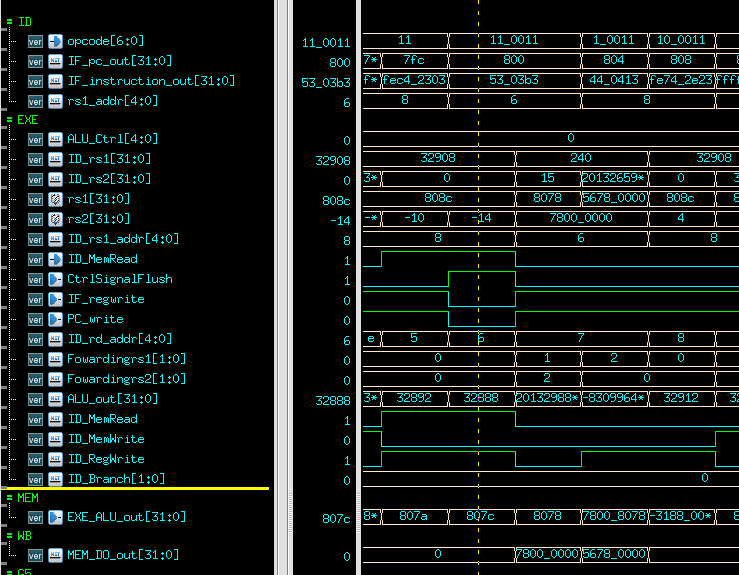
1. WB\_rd\_data成功把值0x1C82回傳給Registerfile
2. DataHazard



此instruction為ADD。

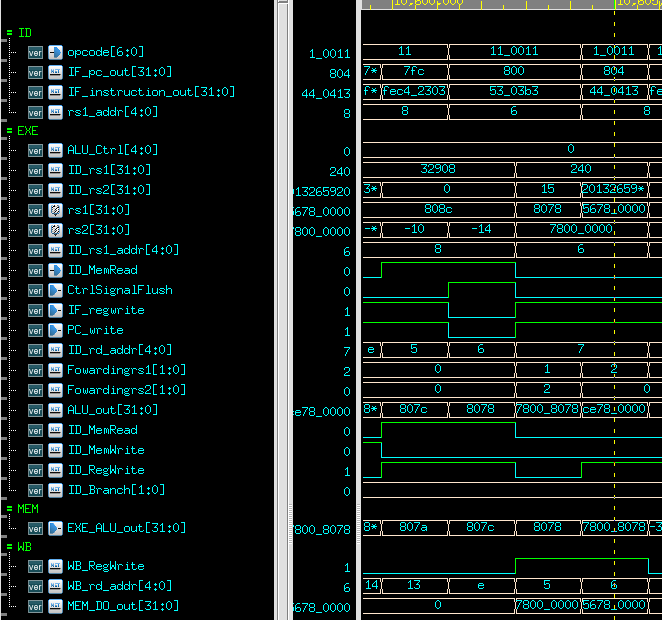
在上圖中黃色虛線的位置可以看見EXE的rs1\_addr和在MEM的rd\_addr相同，即發生了DataHazard，因此Fowardingrs1為1 (發生DataHazard時會為1,沒發生為0) ，直接把在MEM階段的ALU\_out (即上圖EXE\_ALU\_out) 直接取代原本在EXE階段的rs1(即上圖ID\_rs1),傳給準備進入ALU的rs1。如果沒有Forwarding的幫助，則會使輸出變成ID\_rs1+ID\_rs2=(-2)+(-1)=-3,而有著Forwarding的幫助，使輸出變成rs1+rs2=(-4)+(-1)=-5，成功輸出正確答案。

\*load-use情況



此指令為ADD

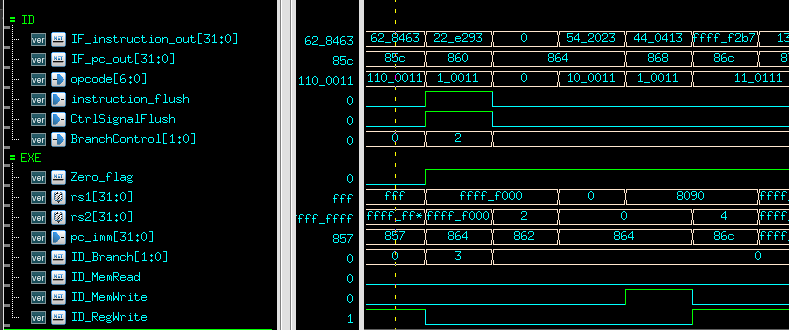
在上圖中黃色虛線的位置可以看見ID階段的rs1\_addr和在EXE的rd\_addr(ID\_ra\_addr)相同並且EXE階段中對DM的Read為1(ID\_MemRead)，因此可以知道接下來會發生 DataHazard中的load-use，而利用此方法來使HazardControl發出訊號線CtrlSignalFlush使在下個clk中ID/EXE Reg裡面的控制訊號(MemRead,MemWrite,RegWrite,Branch)都轉為0，並讓IF\_regwrite,PC\_write變成0，使PC,IF/ID Reg暫停，由上圖最上方可以發現位址為0x800的instruction持續了2個clk，



而在黃線處為暫停完且DM已經讀出資料，並發現WB\_ra\_addr和ID\_rs1\_addr相同，即發生了load-use，因此可以看到forwardingrs1為2(發生load-use時會為2)，而把在WB階段準備寫回Registerfile的值(MEM\_DO\_out)傳給rs1，因此ALU輸出(ALU\_out)為rs1(0x56780000)+rs2(0x78000000)=0xCE780000。

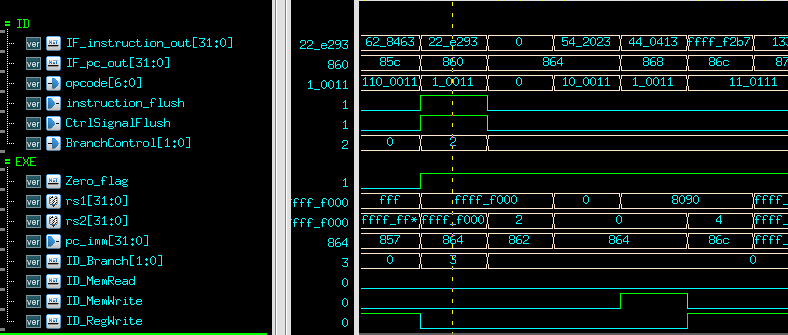
1. BranchHazard

a.



上圖黃色虛線中的ID階段instruction為BEQ

b.



而再經過一個clk來到上圖的黃色虛線，並且剛剛在ID階段的指令來到EXE階段，從圖中可以知道 rs1==rs2,因此ALU的Zero\_flag拉起來，準備把pc\_imm(0x864)回傳給PromgramCounter。由於發生Branch後，還在IF/ID Reg,ID/EXE Reg的instruction就不會被執行，因此在當中增加了flush來讓兩個Reg的instruction變為NOP

c.

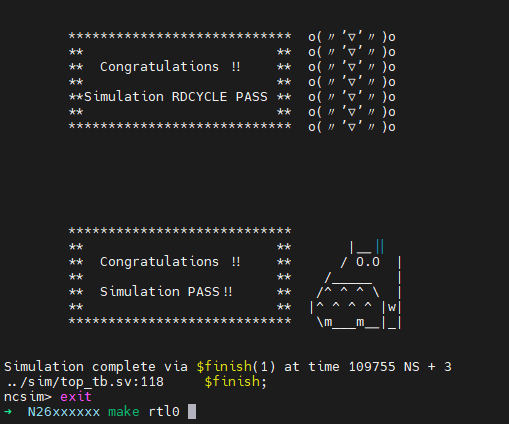
一張含有 文字, 監視器, 螢幕, 公寓 的圖片

自動產生的描述

在上圖中可以看見在Branch後ID階段的pc\_out成功跳去0x864,且flush後，ID階段的instruction為0,在EXE階段的控制訊號線(MemRead, MemWrite, RegWrite,Branch)也為0，代表成功使IF/ID Reg,ID/EXE Reg的instruction成為NOP

**V. Simulation result**

1.Prog0



2.Prog1

一張含有 文字 的圖片

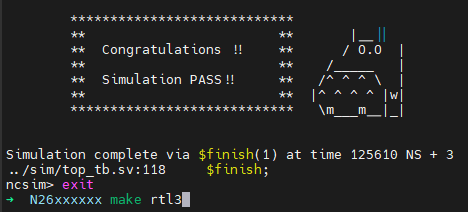
自動產生的描述

3.Prog2

一張含有 文字 的圖片

自動產生的描述

4.Prog3



5.Prog4

一張含有 文字 的圖片

自動產生的描述

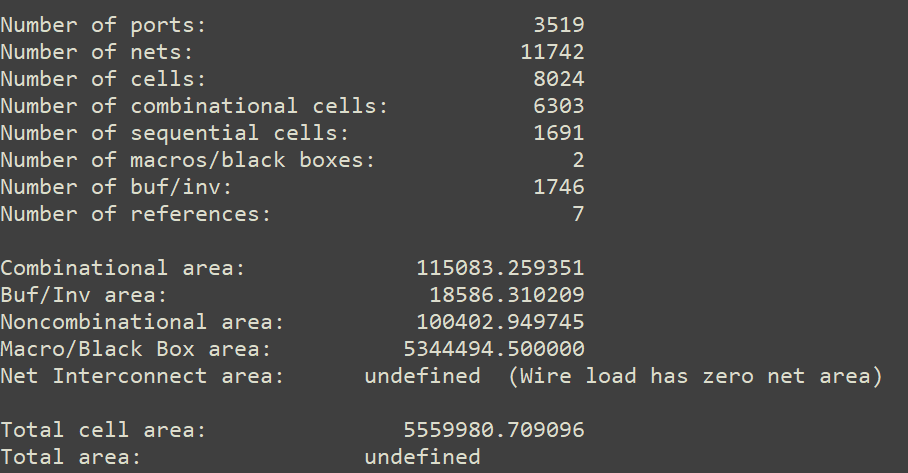
6.Results

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

7.Performance

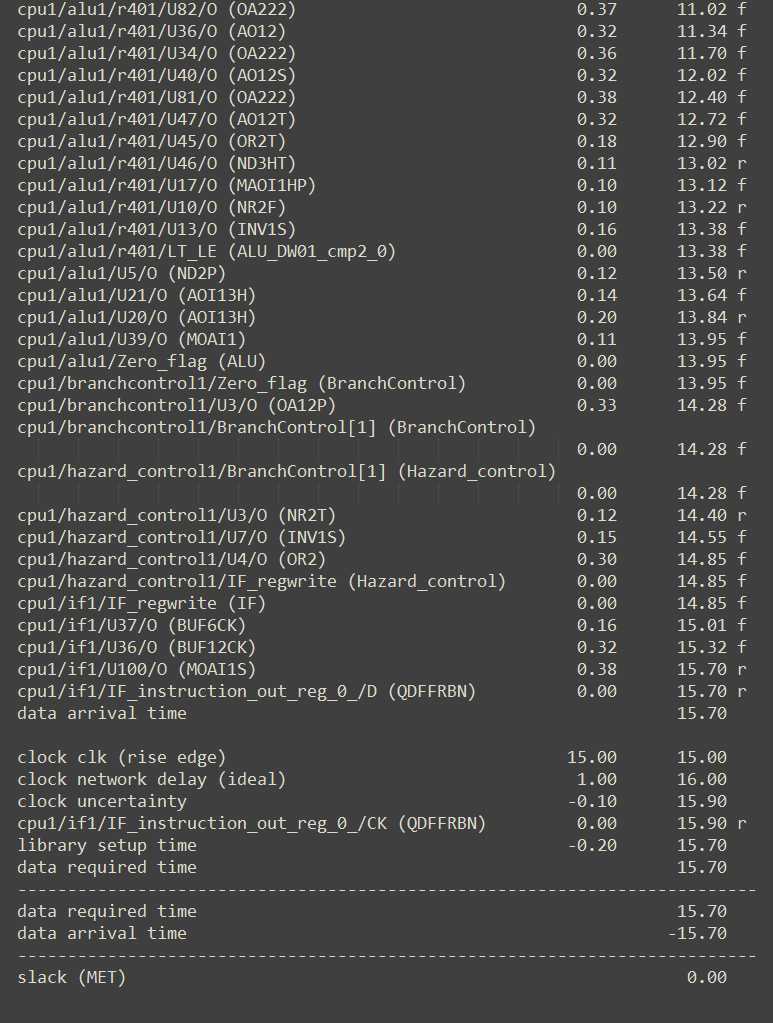
|  |  |
| --- | --- |
| CLK | 15 ns |
| Prog1 Simulation Time | 326385 ns |
| Area | 5559980.709096 |

1. Area Report



1. Timing Report

一張含有 文字 的圖片

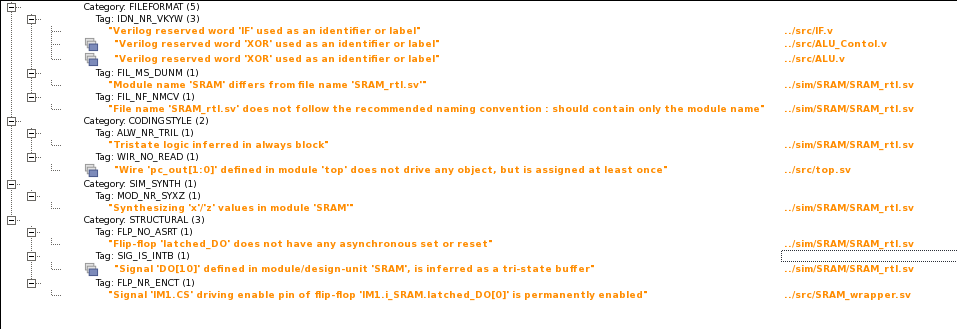
自動產生的描述

1. Power Report

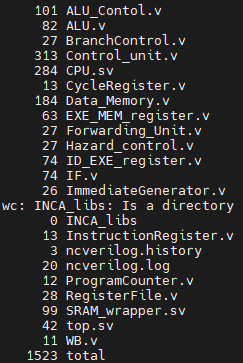
一張含有 文字 的圖片

自動產生的描述

1. **Superlint result**



Warning count=11



Total RTL code =1523

1523\*15%=229 > 11

Superlint warning is within 15% of Total RTL code