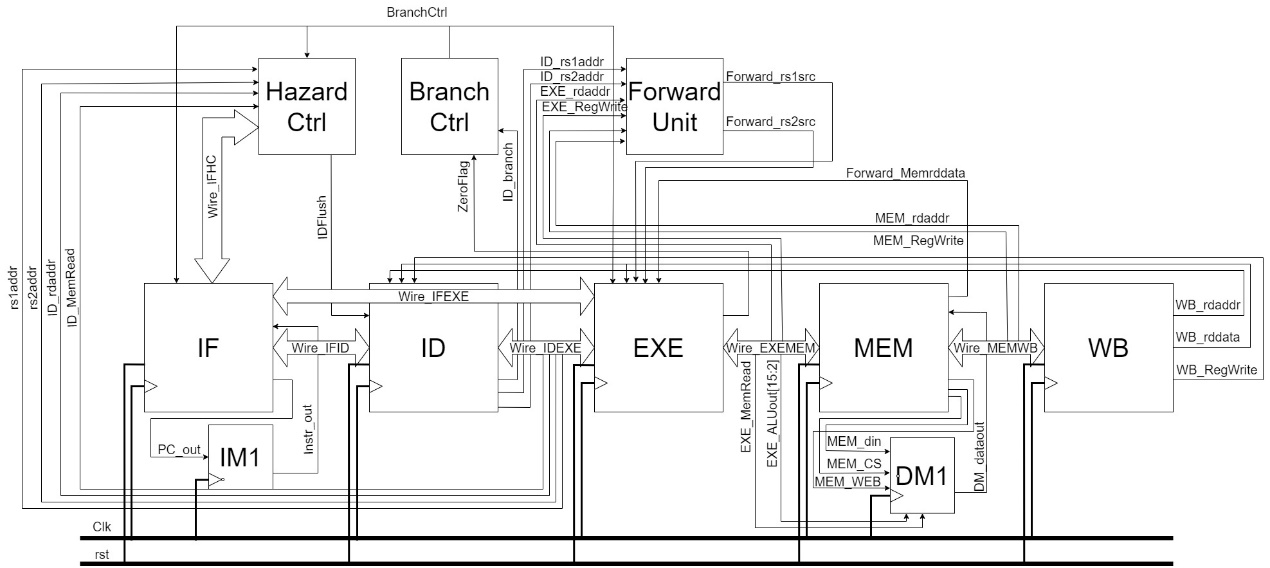
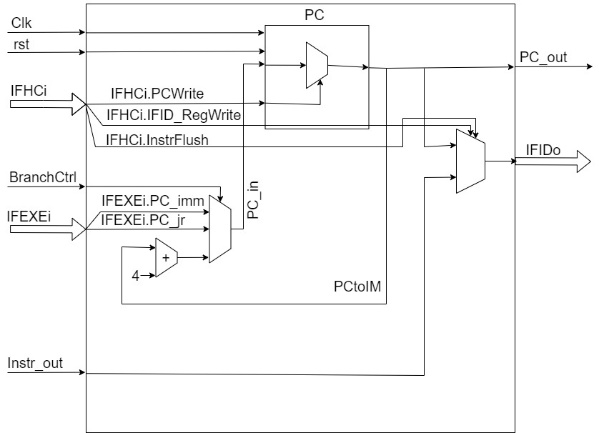
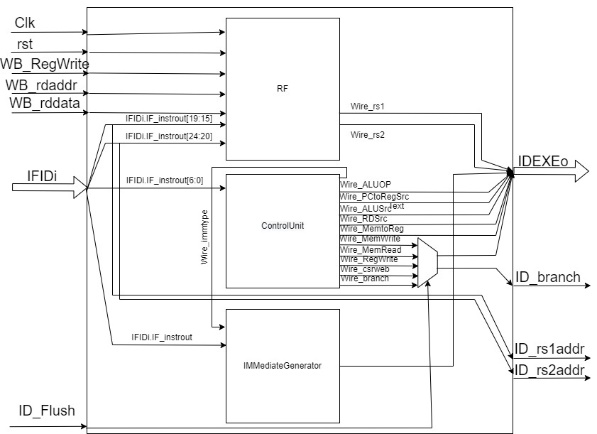
**Explanation & Depict of Design**

1. **Full Design Overview**  
     
   我把CPU分成五個stage，分別是IF、ID、EXE、MEM、WB。且因為目前是使用System Verilog來撰寫程式碼，所以在接線的部分使用了Interface的功能來做嘗試。因為以前在接module時很常因為多接、少接或是更改wire名稱時有缺漏造成debug耗時，因此想藉由此HW學習Interface的使用方式。  
   「正方形」即為此次HW的module，共有10個，其餘module則是接在5個stage之中，在第二部分會仔細說明。  
   「空心雙箭號」即為這次所使用的Interface，內部宣告寫在interface.sv之中，其中在top.sv中實例化作為wire。在各module之間由modport分辨in/output。  
   「實心單箭號」即為單向的資料傳輸，因為只有特定一條或是要輸出給多個module，單獨接線才不會使整體架構太過簡略，造成解讀的困難
2. **Block Detail**  
   **1. IF**



PC主要負責將從PC\_in所得數值傳給PC\_out，由PCWrite來操控PC\_in的值是否可以通過。PC\_in的來源主要有三種，分別是一般、jump以及branch的位置，再由當時BranchCtrl來判斷是不是要跳。

**2. ID**

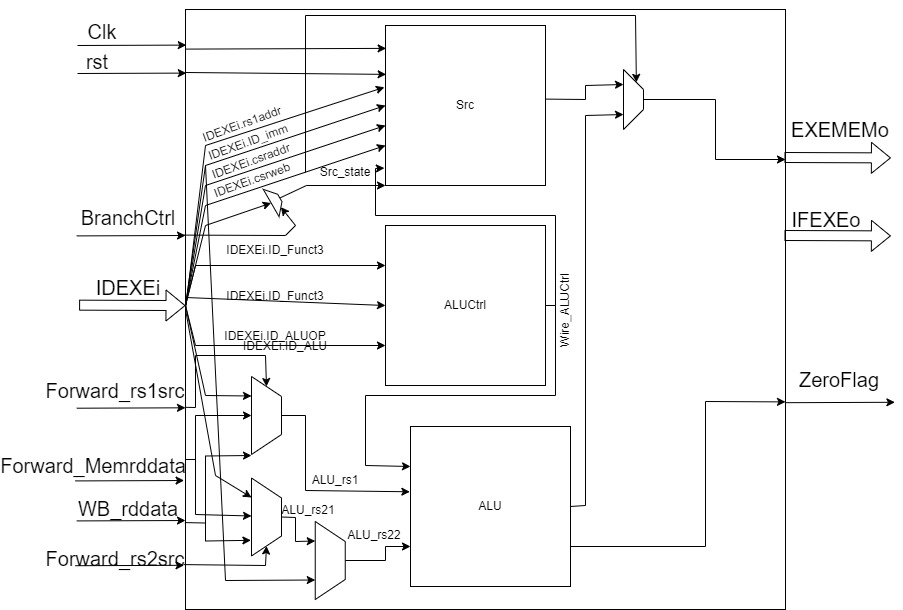


RF用來存值與取值，由data傳到WB時寫回，或是取rs1、rs2的data

CU用來解構instruction，藉由Opcode來決定後續stage的一些function unit signal。且今天如果有Flush時，將相關賦值給予0

IG用來做資料數值擴充，依據CU給的type，來給予不同的擴充方式

**3. EXE**

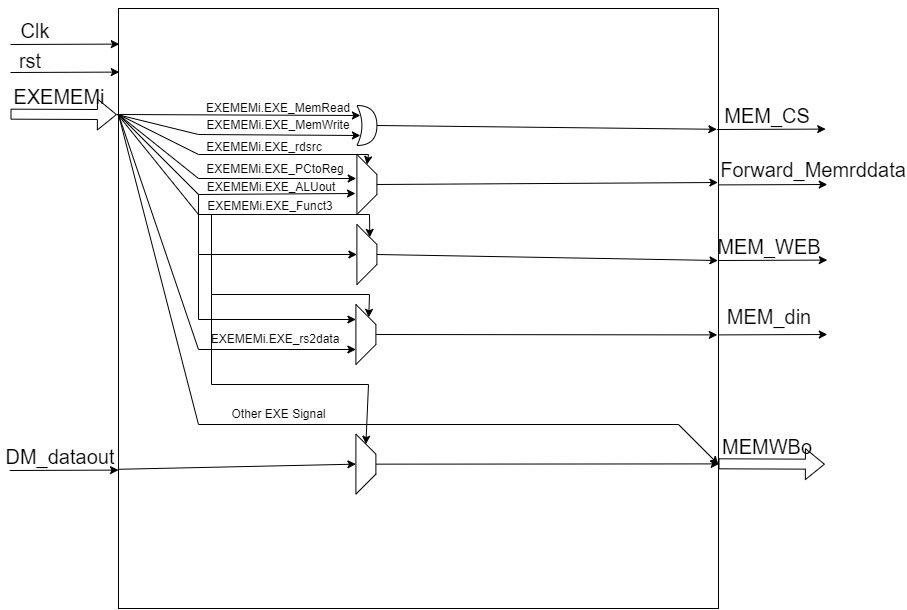


Src主要是計算RDCYCLE以及RDINSTRET，藉由Src\_state來判斷現在的狀態是甚麼，藉此來控制instret的數目。Cycle的部分只要隨著clk做+1即可。而instret又分為branch Flush、Load use以及正常instruction。分別是做-1(因為flush IF&ID)、維持原樣(一加一減不動)以及+1。最後再根據Csrweb來決定是否可以輸出

ALUCtrl 主要是由ALUOP、Funct3以及Funct7來做function的細分，再給ALUCtrl賦值，以利Csr或是ALU做判斷

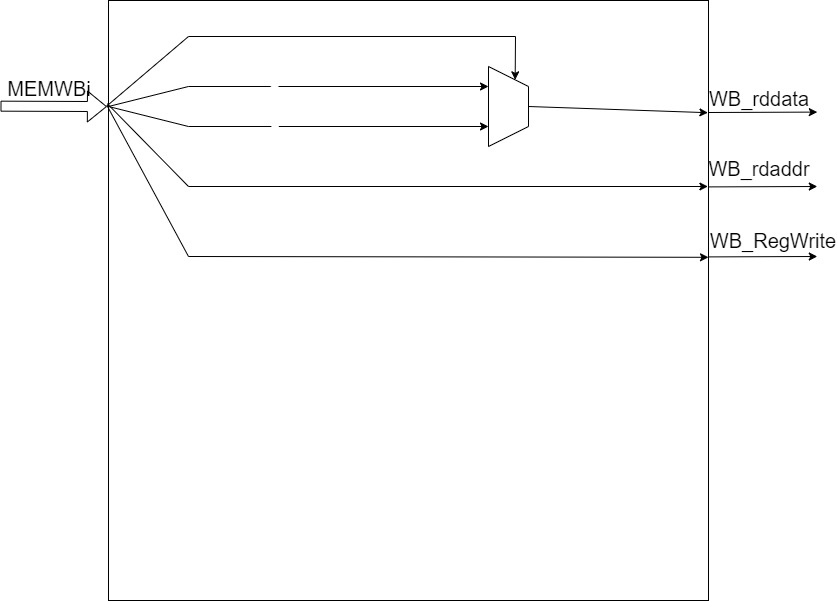
ALU主要是做算術運算(EXE\_ALUout)以及檢查是否有ZeroFlag

**4.MEM**



MEM中剩下的主要是一些資料的傳遞以及對DM的lw與sw

**5.WB**



MEMWBi.MEM\_MemtoReg

MEMWBi.MEM\_rddata

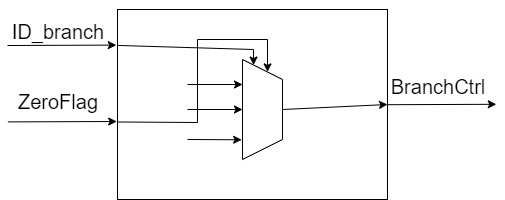
MEMWBi.MEM\_dout

MEMWBi.MEM\_rdaddr

MEMWBi.MEM\_RegWrite

WB也是一樣做資料的傳遞

**6.BC**



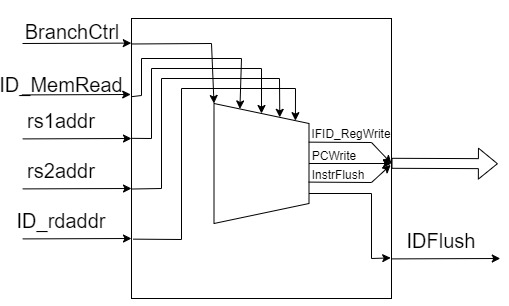
PC4

PCB

PCJR

BC主要是藉由當下指令狀態，給BranchCtrl做賦值，使得PC中的MUX可以知道下一個指令位置是PC+4或是branch或是jump

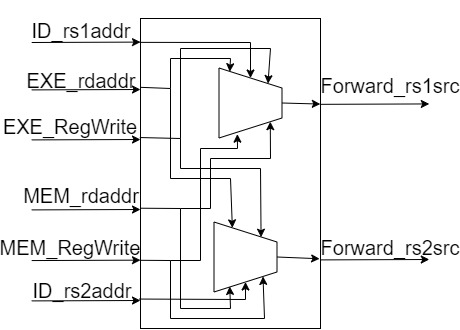
**7.HC**



IFHCo

HC主要是用來看是否需要Flush指令，或是因為有load use，所以要把PC卡住，才可以避免產生錯誤的結果

**8.FU**



FU主要是用來解決一般的data hazard，只要是從EXE算完後馬上要用的值，或是從DM讀出的後也需要做計算時(不包括load use)，在跟ID\_rs1addr以及 ID\_rs2addr做比較，在輸出是要從哪個state中取值(Forward\_rs1,2src)

**9. Interface**

**一張含有 桌 的圖片

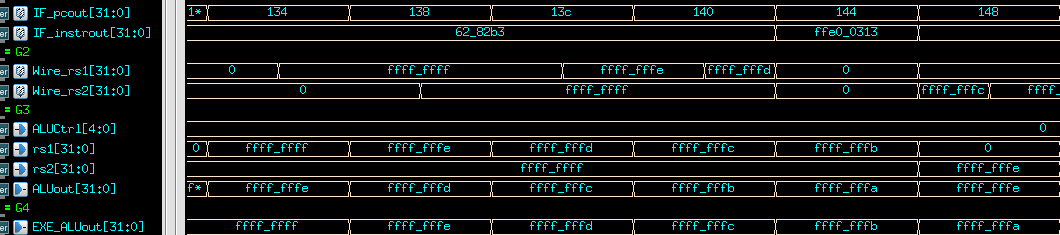
自動產生的描述**

**三、Lessons learned & Summary** 這次的HW讓我第一次真正的有在做一個project的感覺，因為之前頂多是200~300行可以解決，但是現在做每一個module時都要事先想好每個線路要怎麼佈，然後每個控制訊號是如何互相影響的，另外還要考慮到forwarding、data hazard..等。所以我覺得這次做完這份作業對於RISC-V有更深的了解，雖然還有很多細節是這次沒有的，但我也因為這份作業去翻了RISC-V的規格書，了解到原來之前大學所學不過是簡單中的簡單版本，但是我也收穫良多。其中最有收穫的就屬我會使用interface了，雖然在module的特定output port 要拆線時會比較麻煩，還要另外花時間去網路上學。但是大體上來說節省我很多要接線或是debug的時間，而且學會了以後我覺得對於之後更大型的project時，也更能夠讓其他人一目了然。

**Waveform Verification**

**一、R-type**

1. ADD (ALUCtrl == 0)



圖中紅框處可以發現，rs1與rs2都是-1，之後持續用rs1-rs2再存回rs1位置，如圖中ffff\_ffff ADD ffff\_ffff，其ALUout為ffff\_fffe。

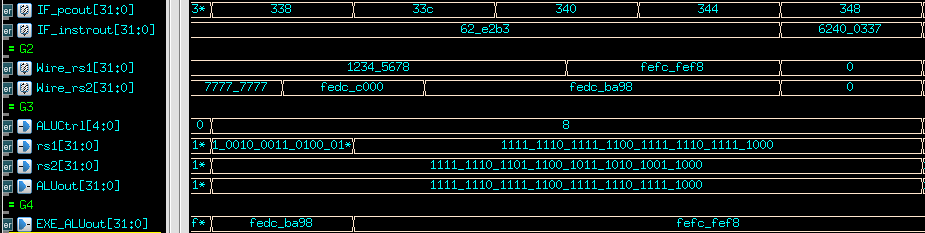
2.SLL (ALUCtrl == 2)

一張含有 文字, 室內 的圖片

自動產生的描述

圖中紅框處可以發現，rs1為2的幂次方，因為rs2每次都會使得rs1向左移一位，也就是不斷地乘以2。

3.OR (ALUCtrl == 8)



圖中紅框處可以發現，

rs1為 1111\_1110\_1111\_1100\_1111\_1110\_1111\_1000，

rs2為 1111\_1110\_1101\_1100\_1011\_1010\_1001\_1000，最後做OR

結果 1111\_1110\_1111\_1100\_1111\_1110\_1111\_1000。

**二、I-type**

1.LW (ALUCtrl == 0)

一張含有 文字, 監視器, 螢幕, 陳列 的圖片

自動產生的描述

圖中紅框處可以發現，因為lw是要將Reg位置將上偏移位置，所以ALUCtrl為0，即是作加法。然後在圖中藍框處可以發現，有確實Chip Select後從DM讀取90f4的值出來，另外因為希望在一個clk之前能夠把數值給讀出來，所以有把DM的clk做反向，因此會有綠框的部份，最後可以在黃框的部份看到有被WB回去Reg中。

2.ADDI (ALUCtrl == 0)

一張含有 文字, 監視器, 陳列, 組 的圖片

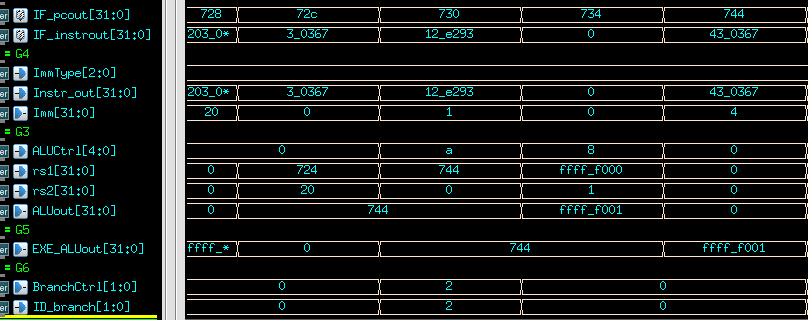
自動產生的描述

圖中紅框處可以發現，ALUCtrl一樣是0，然後rs2為擴充過的數字，

加完即為結果。如圖中ffff\_fffe ADDI 765，其ALUout為763。

3.JALR (ALUCtrl == a)

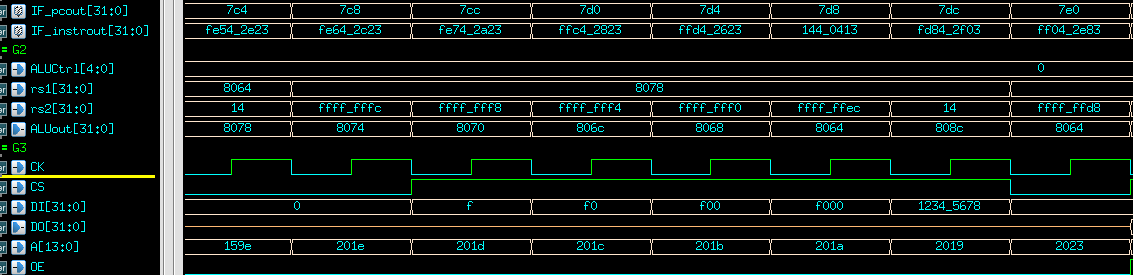
一張含有 文字 的圖片

自動產生的描述

圖中紅框處可以發現，因為ImmType是I-type，所以說在使用Instr\_out做擴充後會得到Imm為0。接著在圖中藍框處可以發現，主要是作加法，且把LSB設為0，只是此處的數值剛好為0，所以與單純相加無差別。最後可以再綠框的部份看到，此時ID\_branch已設為相對應之值且賦值BranchCtrl為PCJR

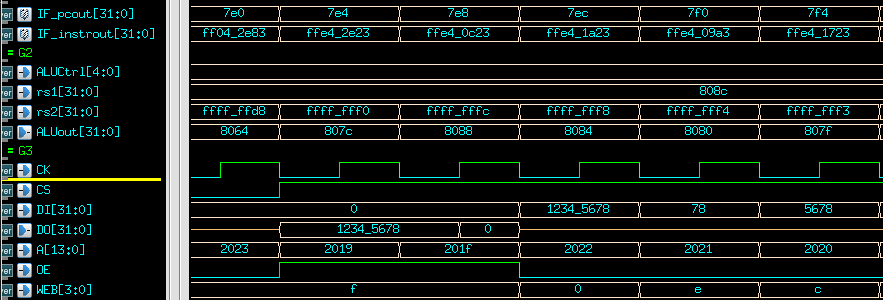
**三、S-type**

1.SW (ALUCtrl == 0)



圖中紅框處可以發現，ALUCtrl一樣是0，和lw一樣是要將Reg位置將上偏移位置。而藍框處可以發現，是ALU在做運算，直到結果為808c後，黃框處可以發現存入1234\_5678在2019位置。

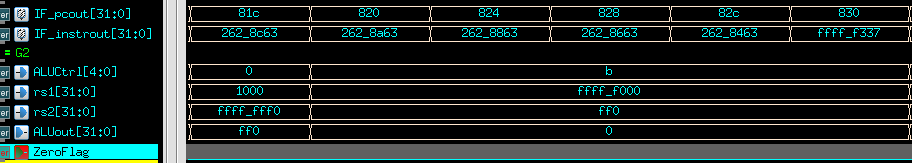
2.SB & SH



圖中紅框處可以發現，從2019位置存取值1234\_5678，接著在藍框處可以發現，和sw一樣是要將Reg位置加上偏移位置，最後在黃框處可以發現存入78在2021位置，存入5678在2020位置。

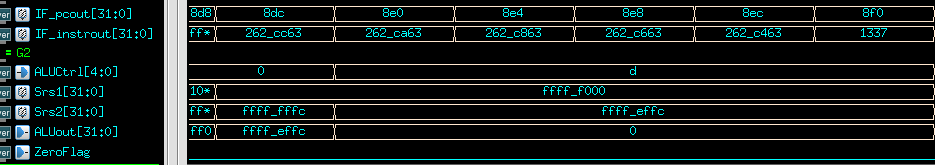
**四、B-type**

1.BEQ (ALUCtrl == b)



圖中紅框處可以發現ALUCtrl為b，所以在ALU中他會判斷(rs1==rs2)?，在此例子中，因為ffff\_f000 != ff0，所以說ZeroFlag沒有被觸發

2.BLT (ALUCtrl == d)



圖中紅框處可以發現ALUCtrl為d，所以在ALU中他會判斷(Srs1 < Srs2)?，在此例子中，因為ffff\_f000 (-4096)> ffff\_effc(-4100)，所以說ZeroFlag沒有被觸發

3.BGEU (ALUCtrl == 10)

一張含有 文字, 監視器, 室內, 螢幕 的圖片

自動產生的描述

圖中紅框處可以發現ALUCtrl為10，所以在ALU中他會判斷(rs1 > rs2)?，在此例子中，因為ffff\_f000 < ffff\_f004，所以說ZeroFlag沒有被觸發

**五、U-type**

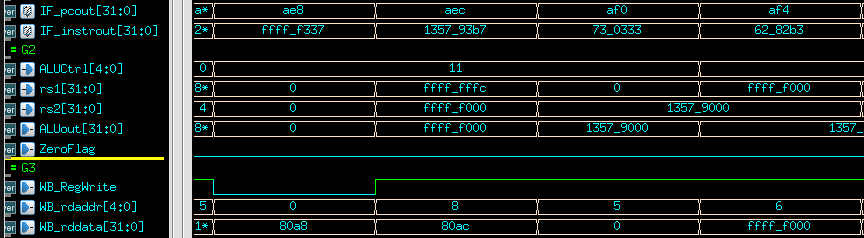
1.AUPIC

一張含有 文字, 監視器, 螢幕, 電視 的圖片

自動產生的描述

圖中紅框處可以發現PC\_imm為ac8，且在圖中藍框處可以發現，WB有寫回對應位置。

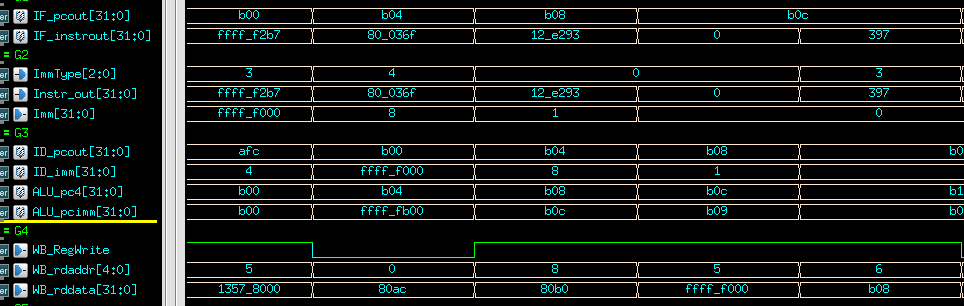
2.LUI (ALUCtrl == 11)



圖中紅框處可以發現ALUCtrl == 11，即會直接把rs2傳給alu\_out，且在圖中藍框處可以發現WB有寫回對應位置。

**六、J-type**

1.JAL



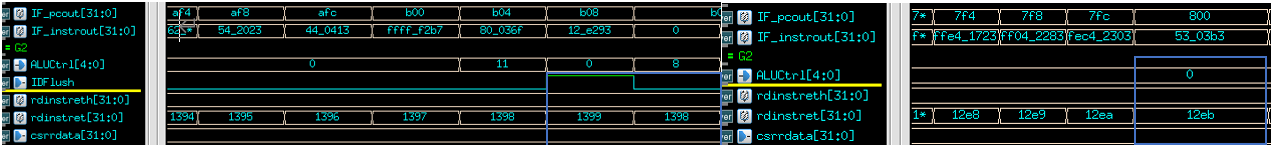
圖中紅框處可以發現ImmType為J\_imm(4)，所以80\_036f就會變成8，且在圖中藍框處可以發現，有分別計算出PC4以及PCimm，最後在黃框處可以發現，PC4被WB寫回對應位置

**七、Csr-type**

1.RDINSTRET[H]

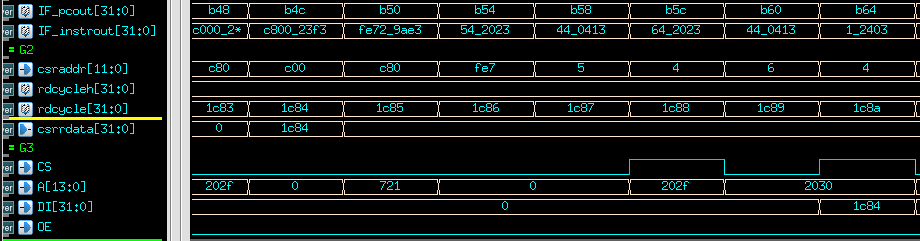
一張含有 文字, 監視器, 電視, 螢幕 的圖片

自動產生的描述



圖中紅框處可以發現Csraddr = c02,c82，所以ALU會將存在{rdinstreth, rdinstret}之值給csrrddata，且在藍框處可以發現，遇到loaduse時+0，Flush時-1，其餘加一最後在圖中黃框處可以發現有寫入13a1在202e位置

2.RDCYCLE[H]

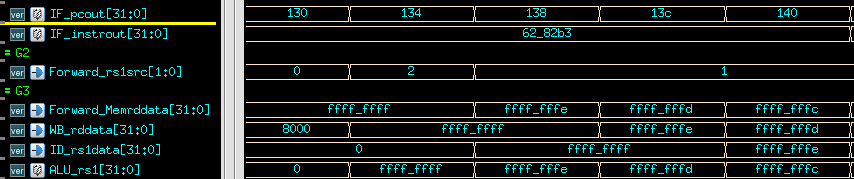


圖中紅框處可以發現Csraddr = c00,c80，所以ALU會將存在{rdcycleh, rdcycle}之值給csrrddata，且在藍框處可以發現有寫入1c84在2030位置

**七、Hazard**

1. Data hazard

i. Forwarding



對於Forwarding來說，處理方式為判斷當下stage是否要寫回，且寫回

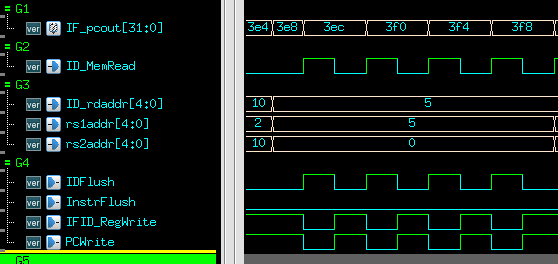
暫存器位置是否為後兩個指令之rs或rt。以rs1為例，即

EXE\_RegWrite && (EXE\_rdaddr == ID\_rs1addr)或是

MEM\_RegWrite && (MEM\_rdaddr == ID\_rs1addr)

再依據其對應狀況給予寫回的機制，例如上圖為li t1.-1以及add t0,t0,t1，圖中紅框處可以發現第一次是因為t1在EXE要值且在MEM才load，所以src為2，取WB\_rddata的值。且在藍框處可以發現因為是t0在EXE要值且在EXE完成計算，src為1，取Forward\_MEMrddata的值

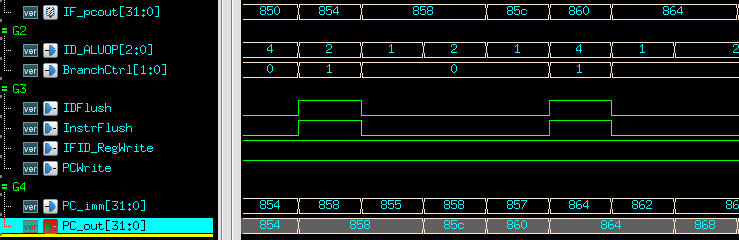
ii. Loaduse



對於loaduse來說，處理方式為判斷MEM<read && (rd=rs || rd=rt),，即

ID\_MemRead && ((ID\_rdaddr == rs1addr)||(ID\_rdaddr == rs2addr))。如果上述條件成立，則stall the PC & IFID\_reg，然後在IF的instruction不用Flush，但是在ID的控制訊號線要歸0。例如上圖為連續lw t0,0(t0)，圖中紅框處可以發現rdaddr == rs1addr && MEMread，所以在藍框處可以發現PC & IFID\_reg設0然後IDFlush設1。

2. Control Hazard



對於Control Hazard來說，處理方式為判斷是否產生branch或是jump，即BranchCtrl!=PC4。如果上述條件成立，則PC & IFID\_reg照常運作，但是要把給IFID\_reg的instruction Flush，且ID的控制訊號線要歸0 。例如上圖為j 858 <beq+0x4c>以及beq t0,t1,864 <beq+0x58>，其運算已在前述各自的type講解，此處著重於hazard處理。圖中紅框處可以發現ID\_ALUOP為2表示jump的ADD\_type且BranchCtrl為1，以及藍框處可以發現ID\_ALUOP為4表示branch的B\_type且BranchCtrl為1，皆不為PC4(設為0)。所以可以在黃框處發現，因為control hazard，所以把ID控制訊號線規0，IFID\_reg的instruction Flush。最後在綠框處可以發現，PC\_imm為所要到的位置，且PC\_out也改為跳後位置，以及因為ControlHazard造成IF\_pcout有多一個cycle去執行。

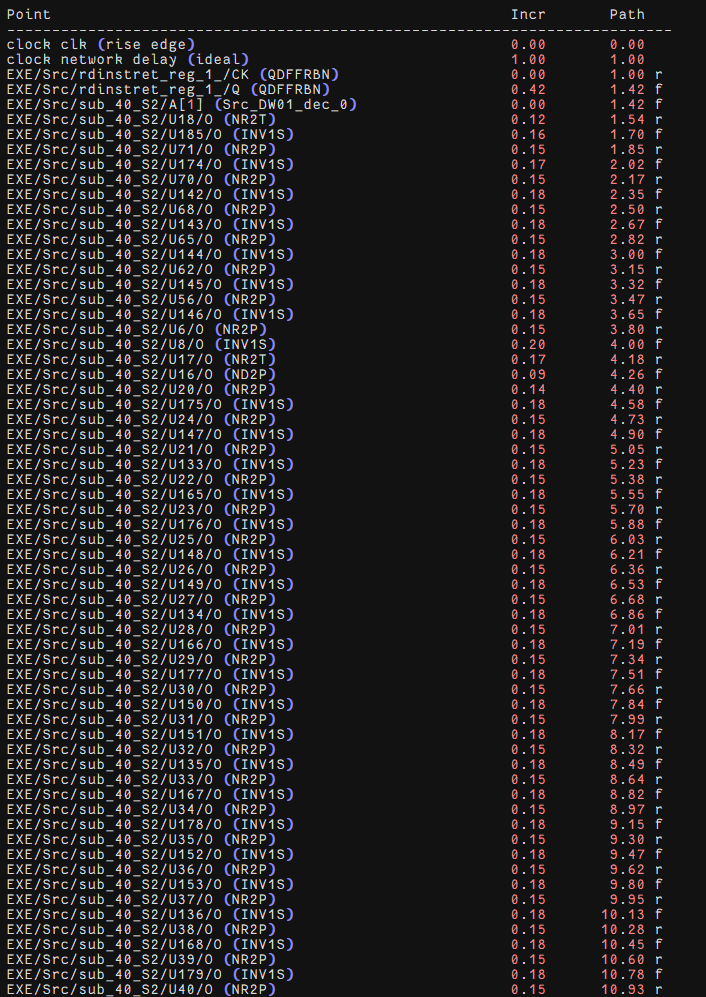
**Report**

**一、Lines of RTL code**

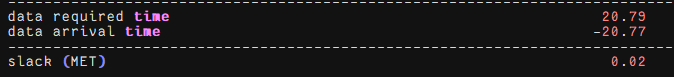
一張含有 文字 的圖片

自動產生的描述

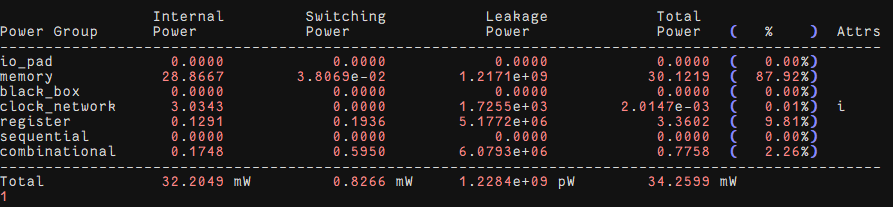
二、**Timing & Area report & Power**

一張含有 文字 的圖片

自動產生的描述



一張含有 文字 的圖片

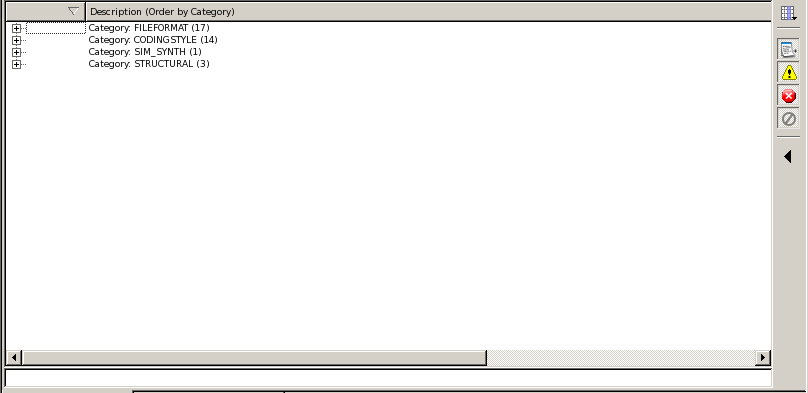
自動產生的描述

三、**Simulation & Performance**

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

四、**Result of superlint**

一張含有 文字 的圖片

自動產生的描述

**35/1357 = 2.58% ; superlint correct rate > 85%**