VLSI System Design (Graduate Level)

Fall 2021

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

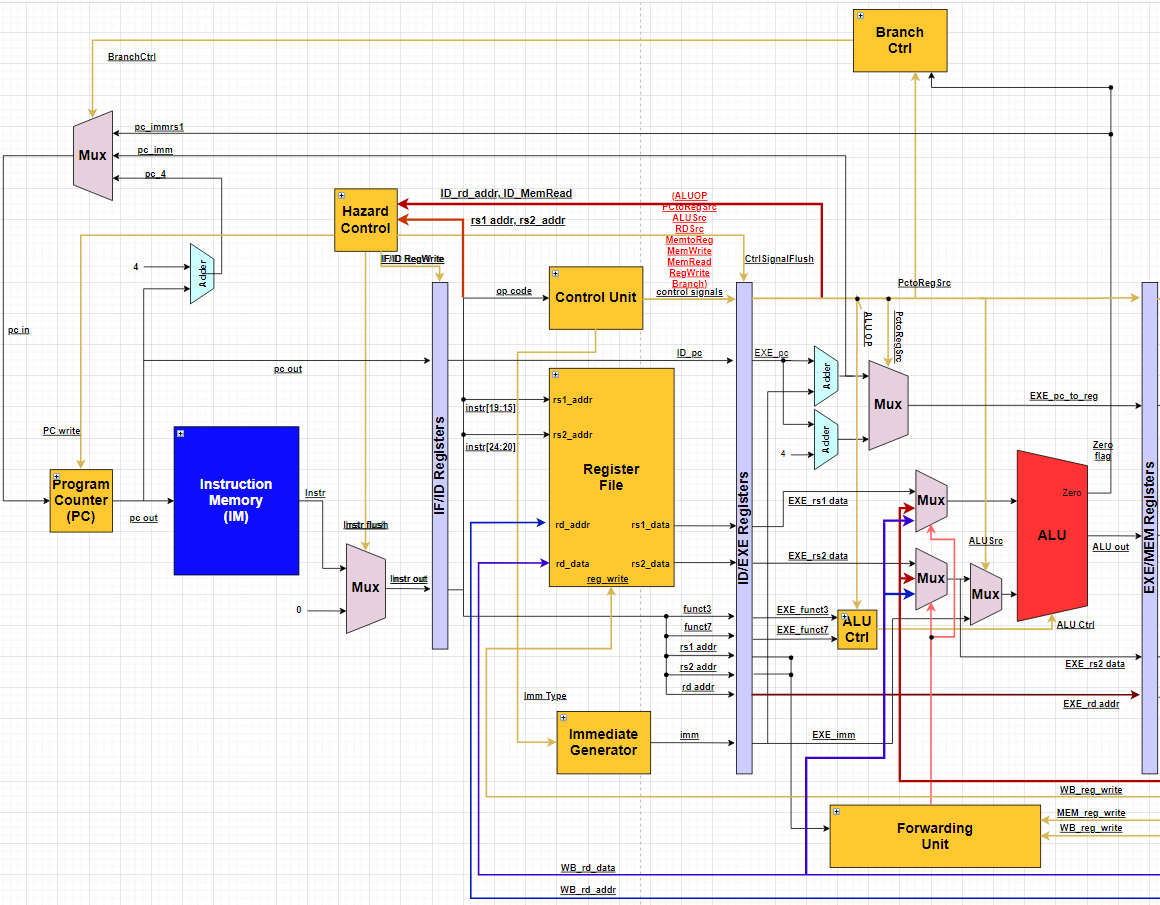
Student name: \_陳柏廷\_

Student ID: \_P76091284\_

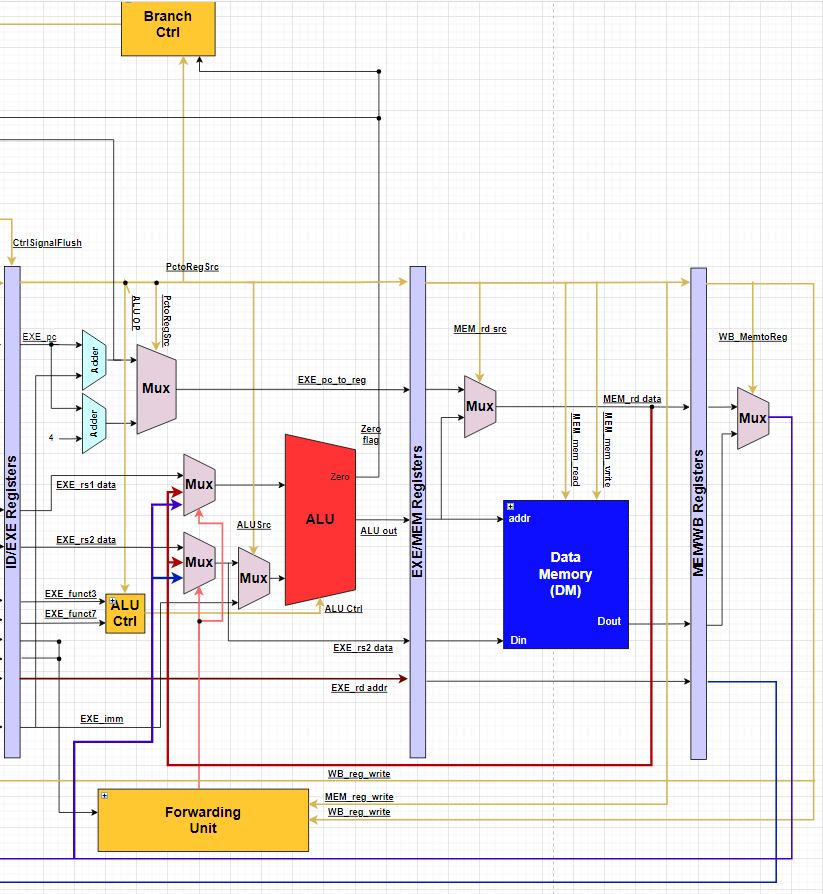
RISC-V pipeline CPU overview

下面兩張圖是依據我所設計的pipeline所繪製

IF-EXE stage

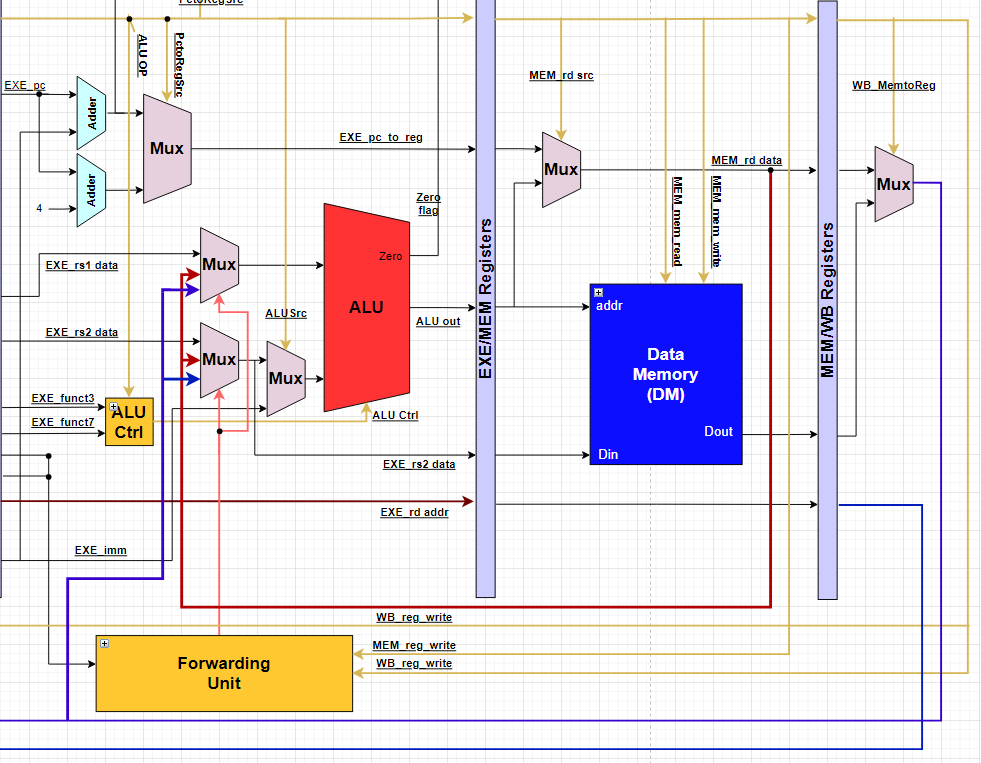
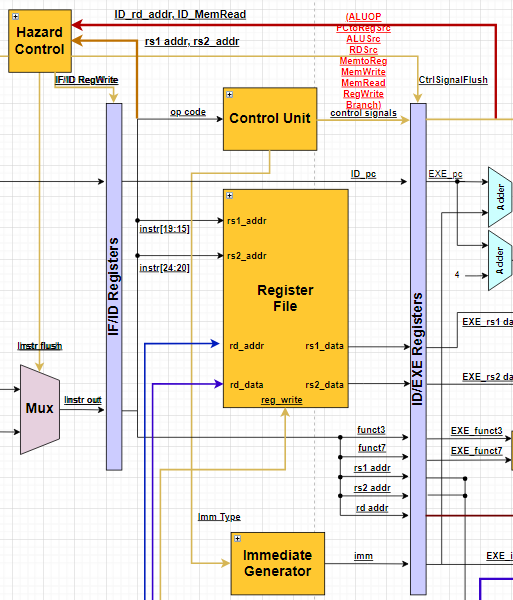
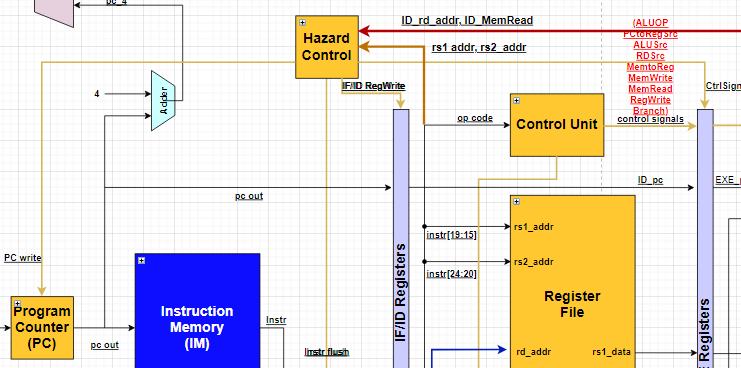
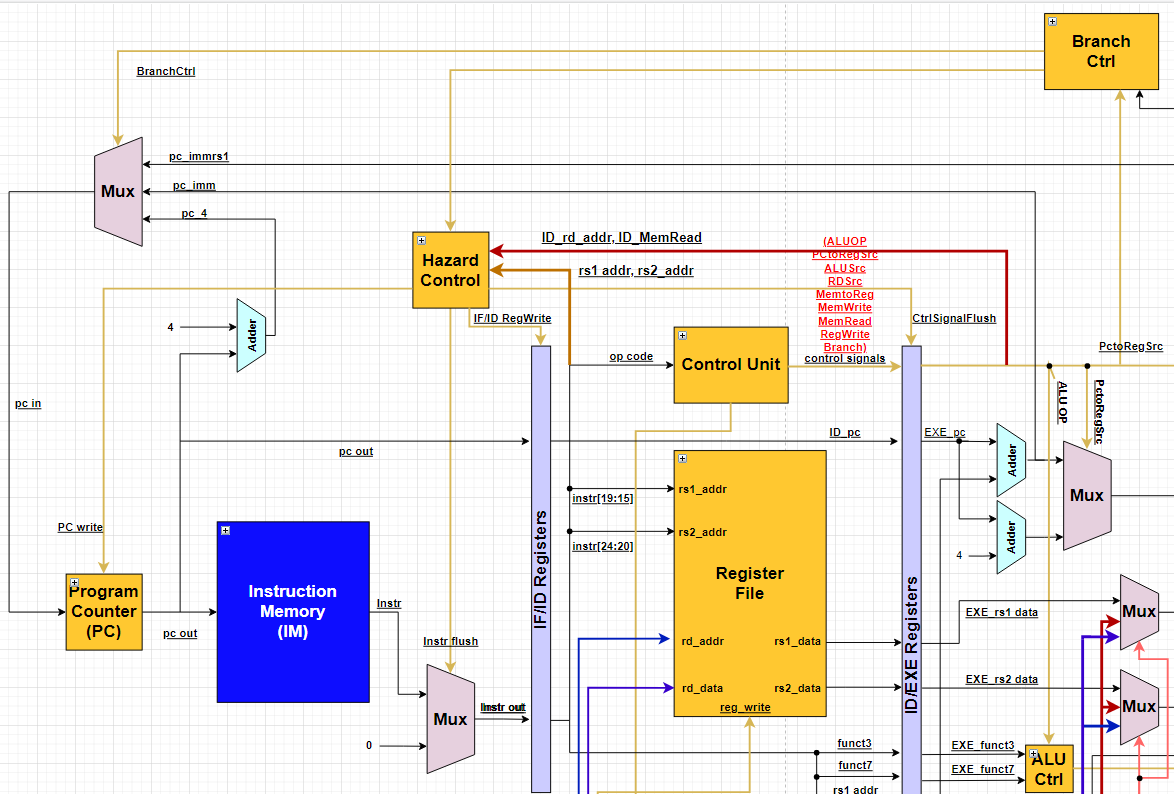


EXE-WB stage



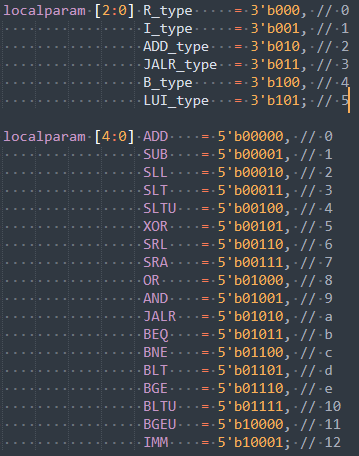
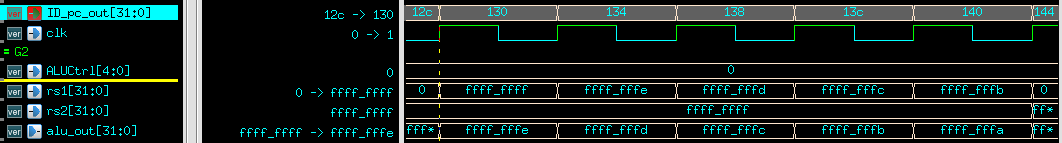
Lessons learned

以下列出一些在實作RISC-V pipeline CPU時比較會遇到的一些問題及想法。

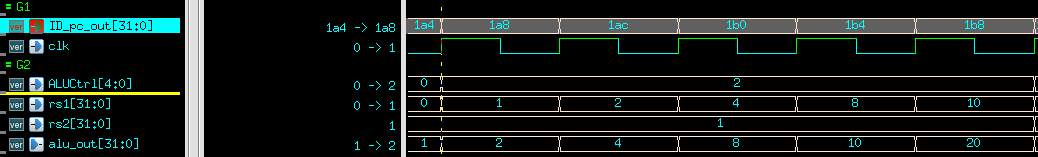
1. Data Hazard  
   問題描述:
   1. 一般的Data hazard
      1. 新增一個Forwarding Unit即可解決
      2.   
         如上圖，可以看到紅色(from MEM stage)以及藍色(from WB stage)的線，可以藉由Forwarding unit判斷(EXE, MEM) 或(EXE, WB)中指令的reg是否有相依，再即時的將正確資料送到EXE的指令，避免產生錯誤的結果。
   2. Load use data hazard
      1. 除了原本的Forwarding unit以外，還需要新增**Hazard Control Unit**
      2.   
         如上圖，從左上的Hazard Control送進來的兩條橘色(ID stage的 rs1\_addr, rs2\_addr)以及紅色(EXE stage 的 rd\_addr, MemRead)訊號線，可以藉由MemRead判斷是否為Load type指令以及addr來判斷Registers是否有相依性。
      3. 若有Load Use發生，則此電路會將IF/ID 的資料保留至下一個cycle並清除ID/EXE之部分控制訊號線(MemRead, MemWrite, Branch, RegWrite)，使得下一個cycle進入EXE stage的指令變為NOP，同時ProgramCounter也不會去做更新。
      4. 透過iii.之步驟做stall後，我們就可以前一個解決一般data hazard的方法，也就是透過Forwarding unit將WB stage的data forward給EXE stage的指令。
2. Control Hazard  
   問題描述: 由於Branch的條件達成，因此必須將ID, IF中的指令Flush，以避免不必要的錯誤。
   1. 同樣可以透過先前的Hazard Control unit來解決
   2. 

如上圖，當判斷到Branch Ctrl為需要branch時，透過instr\_flush以及CtrlSignalFlush將即將進入ID及EXE之指令清除，就可以避免錯誤的指令執行而發生錯誤。

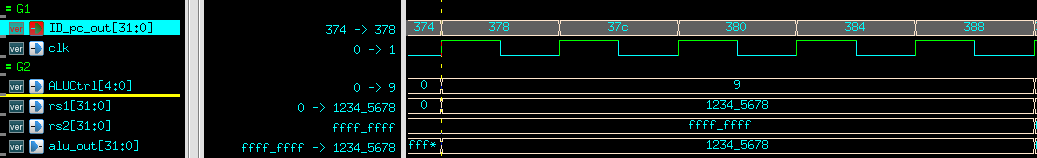
波型圖驗證

1. R-type，下面這個table是此電路之ALU根據送進來的ALU Ctrl做出相對應的運算  
   
   1. ADD
      1. 

如上圖，紅框處為連續ADD的運算

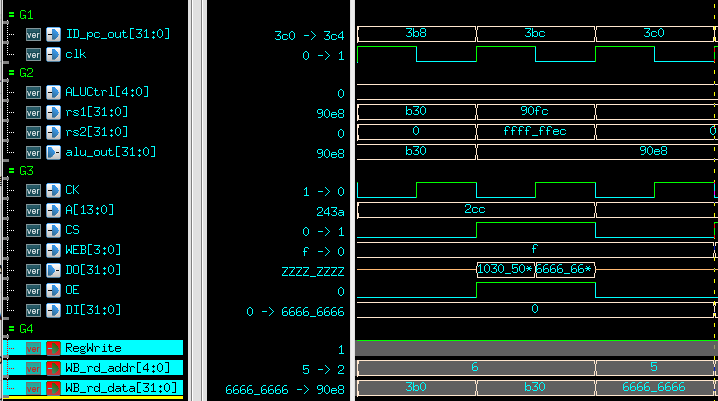
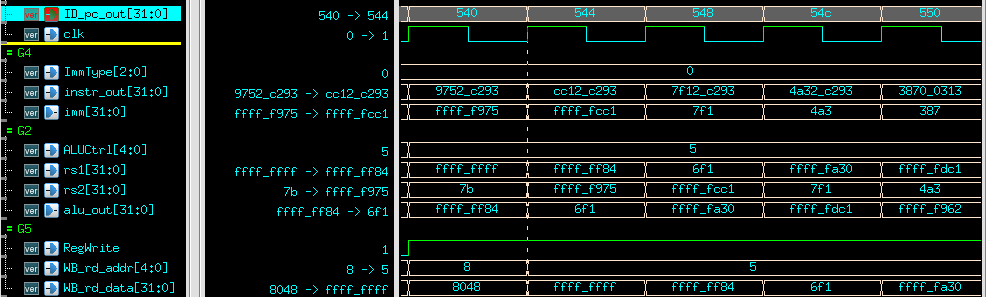
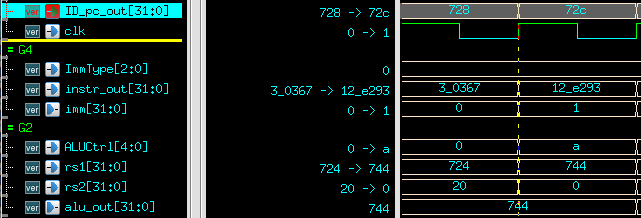
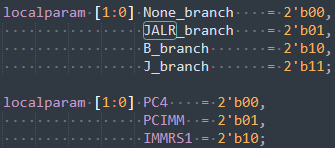
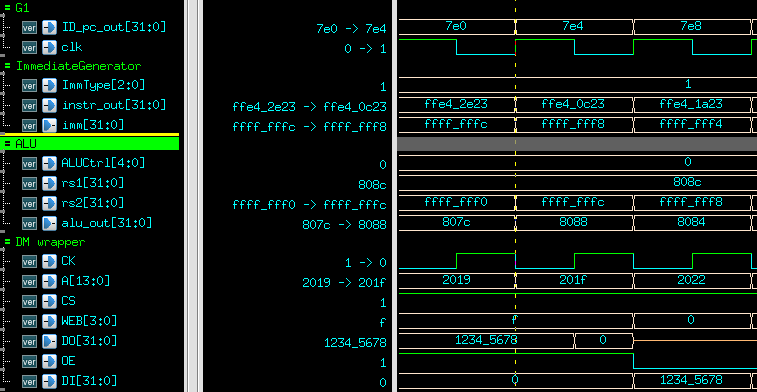
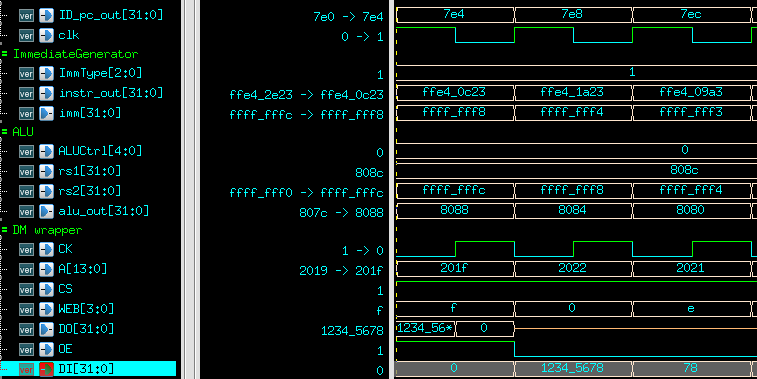
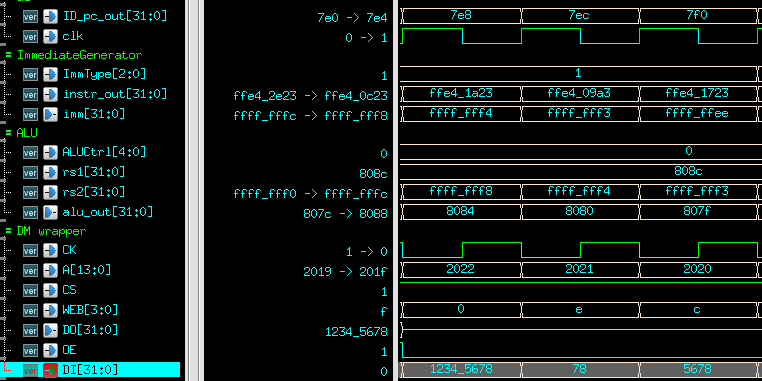
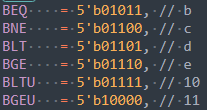
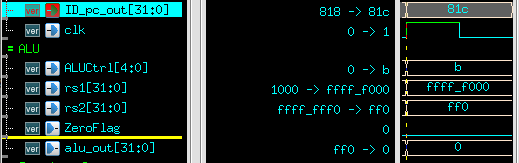
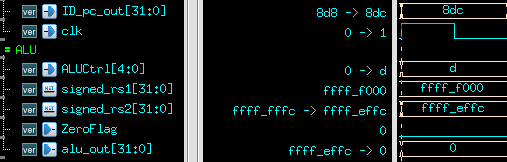
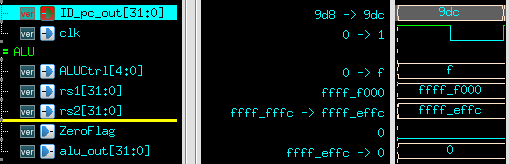
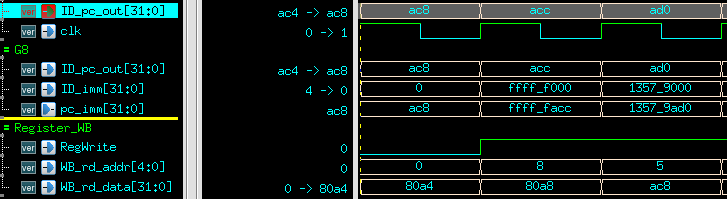
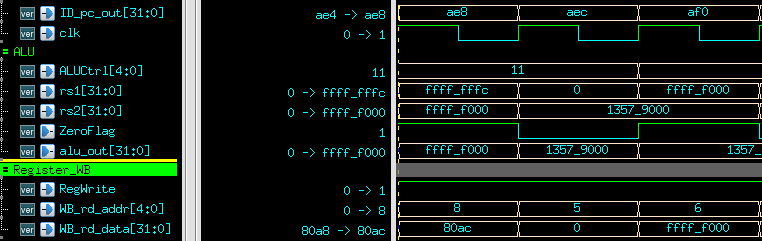
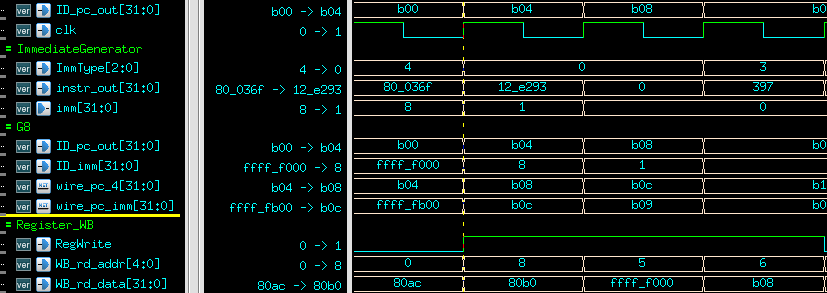
* + - 1. ALUCtrl為0表示目前ALU運作ADD的運算
      2. 以第一段做說明，rs1為ffff\_ffff, rs2為ffff\_ffff，結果為ffff\_fffe
  1. SLL
     1. 

如上圖，紅框處為連續SLL的運算

* + - 1. ALUCtrl為2表示目前ALU運作SLL的運算
      2. 以第一段做說明，rs1為1, rs2為1，結果為2
  1. AND
     1. 

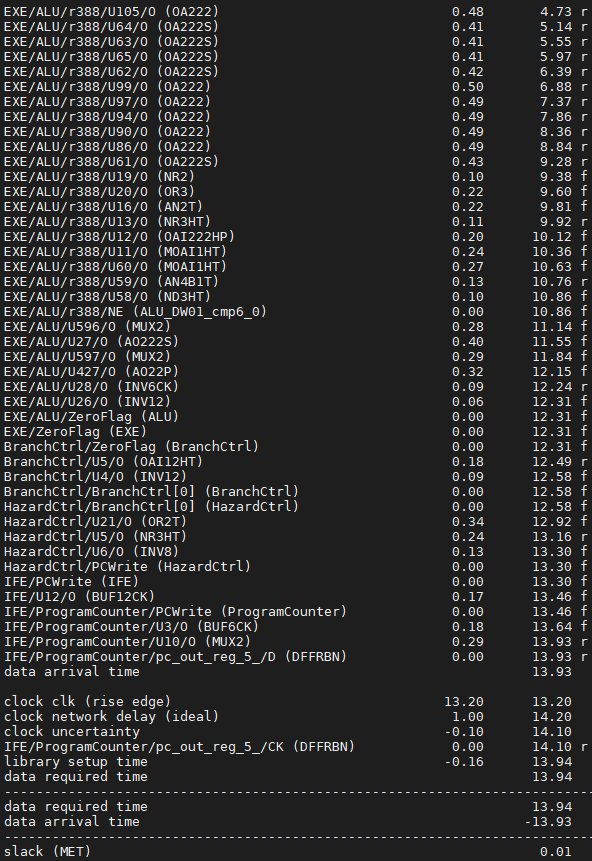
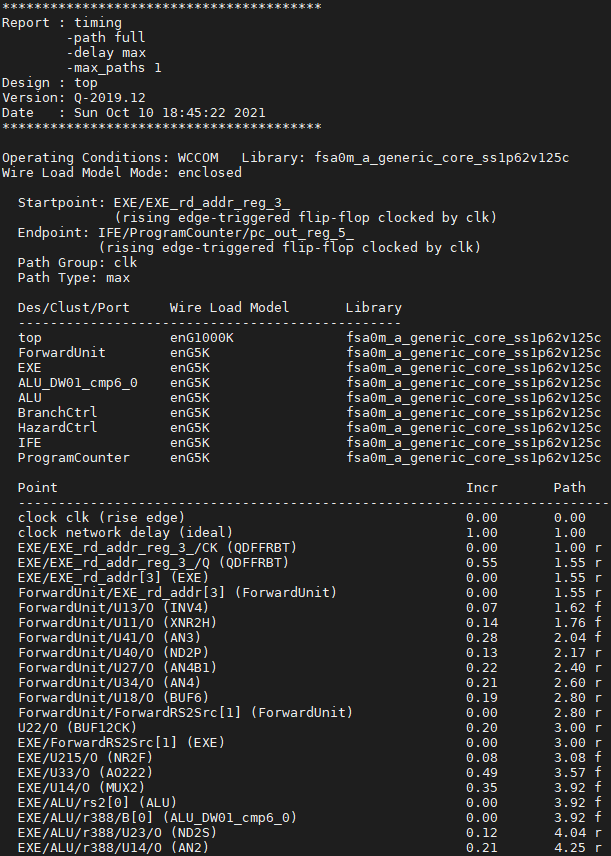
如上圖，紅框處為連續AND的運算

* + - 1. ALUCtrl為9表示目前ALU運作AND的運算
      2. 以第一段做說明，rs1為1234\_5678, rs2為ffff\_ffff，結果為1234\_5678

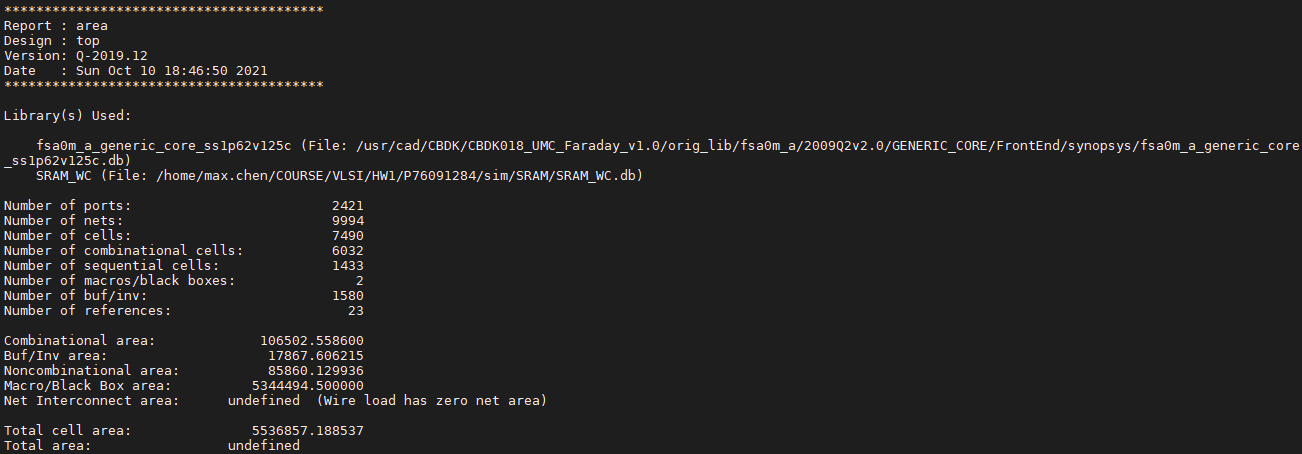
1. I-type
   1. LW
      1. 
      2. 如上圖紅框處為LW指令在EXE stage計算欲存取的address  
         rs1為b30, rs2為0，結果為b30，表示我們將存取Data memory b30的address
      3. 如上圖黃框處為LW指令在MEM stage
         1. A為將b30取[15:2]bits，得到2cc的結果
         2. 因為需要讀取，將CS設為high，WEB設為f
         3. 將CK(clk)做反向是為了能夠在同一個cycle內取得DM中的Data，如藍框處可以看出在後半cycle得到6666\_6666的結果
         4. OE設為high使得我們可以將結果傳回CPU
      4. 最後如紫框處為LW指令在WB stage將6666\_6666寫回register，因此需要將RegWrite拉為high並且將rd\_addr及rd\_data同時送給Register
   2. XORI
      1. 
      2. 如上圖紅框處為XORI指令在ID stage
         1. 根據ImmType，在此為I-type，因此將imm將會如下圖做extension，則imm = {{20{instr\_out[31]}}, instr\_out[31:20]}
         2. 以第一段為例，instr\_out為9752\_c293，根據ImmType為0(表示I-type)，Signed extension後結果為ffff\_f975
      3. 如上圖黃框處為XORI指令在EXE stage
         1. ALUCtrl為5表示目前ALU運作XOR的運算
         2. 以第一段做說明，rs1為ffff\_ffff，rs2為7b，結果為ffff\_ff84
      4. 如上圖藍圈處為XORI指令在WB stage  
         可以看出在WB stage CPU將EXE運算完的結果寫回對應的Register
   3. JALR
      1. 
      2. 如上圖紅框處為JALR指令在ID stage
         1. 根據ImmType，在此為I-type，因此將imm將會如下圖做extension，則imm = {{20{instr\_out[31]}}, instr\_out[31:20]}
         2. instr\_out為3\_0367，根據ImmType為0(表示I-type)，Signed extension後結果為0
      3. 如上圖黃框處為JALR指令在EXE stage
         1. ALUCtrl為a表示目前ALU運作JALR的加法運算，除了一般的rs1+rs2外，還需要將結果的LSB設為0
         2. 以此例做說明，rs1為744，rs2為0，結果為744
      4.   
           
         如上圖，由於是JALR，因此根據ID branch為1送出BranchCtrl為2的訊號表示下個PC需選擇IMMRS1的訊號線(imm+rs1作為下一個PC)
      5. Control Hazard的部分與參考前面Lessons Learned
2. S-type
   1. SW
      1. 
      2. 如上圖紅框處為SW指令在ID stage
         1. 根據ImmType，在此為S-type，因此將imm將會如下等式做extension，則imm = {{20{instr\_out[31]}}, instr\_out[31:25], instr\_out[11:7]}
         2. 以上圖範例做說明，instr\_out為ffe4\_2e23，根據ImmType為1(表示S-type)，Signed extension後結果為ffff\_fffc
      3. 如上圖黃框處為SW指令在EXE stage
         1. ALUCtrl為0表示目前ALU運作ADD的運算
         2. 以第一段做說明，rs1為808c，rs2為ffff\_fffc，結果為8088
      4. 如上圖藍框處為SW指令在MEM stage
         1. A為將8088取[15:2]bits，得到2022的結果
         2. 因為需要寫入DM且大小為WORD，將CS設為high，WEB設為0，OE設為0，同時將欲寫入的Data傳進Din
   2. SB
      1. 
      2. 如上圖紅框處為SB指令在ID stage
         1. 根據ImmType，在此為S-type，因此將imm將會如下等式做extension，則imm = {{20{instr\_out[31]}}, instr\_out[31:25], instr\_out[11:7]}
         2. 以上圖範例做說明，instr\_out為ffe4\_0c23，根據ImmType為1(表示S-type)，Signed extension後結果為ffff\_fff8
      3. 如上圖黃框處為SB指令在EXE stage
         1. ALUCtrl為0表示目前ALU運作ADD的運算
         2. 以第一段做說明，rs1為808c，rs2為ffff\_fff8，結果為8084
      4. 如上圖藍框處為SB指令在MEM stage
         1. A為將8084取[15:2]bits，得到2021的結果
         2. 因為需要寫入DM且大小為BYTE，將CS設為high，WEB設為e，OE設為0，同時將欲寫入的Data傳進Din
         3. SB中的WriteEnable及WriteData皆根據write address的最後兩個bits做決定，SB的決定方式請參考以下  
            WriteEnable [WriteAddr[1:0]] = 1’b0  
            WriteData[{WriteAddr[1:0], 3’b0}+:8] = rs2\_data[7:0]
   3. SH
      1. 
      2. 如上圖紅框處為SH指令在ID stage
         1. 根據ImmType，在此為S-type，因此將imm將會如下等式做extension，則imm = {{20{instr\_out[31]}}, instr\_out[31:25], instr\_out[11:7]}
         2. 以上圖範例做說明，instr\_out為ffe4\_1a23，根據ImmType為1(表示S-type)，Signed extension後結果為ffff\_fff4
      3. 如上圖黃框處為SH指令在EXE stage
         1. ALUCtrl為0表示目前ALU運作ADD的運算
         2. 以第一段做說明，rs1為808c，rs2為ffff\_fff4，結果為8080
      4. 如上圖藍框處為SH指令在MEM stage
         1. A為將8084取[15:2]bits，得到2020的結果
         2. 因為需要寫入DM且大小為BYTE，將CS設為high，WEB設為c(2’b1100)，OE設為0，同時將欲寫入的Data傳進Din
      5. SB中的WriteEnable及WriteData皆根據write address的最後兩個bits做決定，SB的決定方式請參考以下  
         WriteEnable [WriteAddr[1], 1’b0] = 2’b00  
         WriteData[{WriteAddr[1], 4’b0}+:16] = rs2\_data[15:0]
3. B-type，下面這個table為對應各B-type指令送入ALU中的ALUCtrl  
   
   1. BEQ
      1. 
      2. 上圖紅框處為BEQ指令在EXE stage
         1. ALUCtrl 為b，表示現在ALU判斷rs1是否與rs2相同，若相同則輸出1，不同則輸出0。
         2. 以上圖做範例，rs1為ffff\_f000、rs2為ff0，因為rs1與rs2不同，因此輸出0
         3. 若輸出為0則不會對CPU有任何影響，繼續執行下一個指令；若輸出1則會因為需要branch而產生Control Hazard，詳情可參考Lessons Learn
   2. BLT
      1. 
      2. 上圖紅框處為BLT指令在EXE stage
         1. ALUCtrl 為d，表示現在ALU判斷signed\_rs1是否小於signed\_rs2，若是則輸出1，否則輸出0。
         2. 以上圖做範例，signed\_rs1為ffff\_f000(-4096)、signed\_rs2為ffff\_effc(-4100)，因為rs1沒有小於rs2，因此輸出0
         3. 若輸出為0則不會對CPU有任何影響，繼續執行下一個指令；若輸出1則會因為需要branch而產生Control Hazard，詳情可參考Lessons Learn
   3. BLTU
      1. 
      2. 上圖紅框處為BLTU指令在EXE stage(與前者不同是他為unsigned的運算)
         1. ALUCtrl 為f，表示現在ALU判斷rs1是否小於rs2，若是則輸出1，否則輸出0。
         2. 以上圖做範例，rs1為ffff\_f000、rs2為ffff\_effc，因為rs1沒有小於rs2，因此輸出0
         3. 若輸出為0則不會對CPU有任何影響，繼續執行下一個指令；若輸出1則會因為需要branch而產生Control Hazard，詳情可參考Lessons Learn
4. U-type
   1. AUIPC
      1. 
      2. 如上圖紅框處為將PC+imm的部分經過一個加法器完成  
         以上圖為範例，pc為ac8, imm為0，結果為ac8
      3. 如上圖藍框處可以看出在WB stage將結果寫回對應的register
   2. LUI
      1. 
         1. 如上圖紅框處為LUI指令在EXE stage  
            ALUCtrl 為11(hex)，表示將rs2(imm) bypass過去輸出成為結果
         2. 以上圖為範例，rs2為ffff\_f000，結果即為ffff\_f000
      2. 如上圖藍框處為LUI在WB stage將結果存回對應的register
5. J-type
   1. JAL
      1. 
      2. 如上圖紅框處為LUI指令在ID stage  
         根據ImmType為4，表示為J-type  
         依照Spec定義，J-type指令的sign extension方式為imm[20|10:1|11|19:12]  
         以上圖為範例，instruction為80\_036f，結果為8
      3. 如上圖黃框處為LUI指令在EXE stage  
         分別算出PC+4以及PC+IMM
      4. 如藍框處可以看出將pc+4存入相對應的register

Report

* Timing report



* Area report



Simulation results

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

Performance

|  |  |
| --- | --- |
| Simulation time | 13.2ns |
| Total Cell Area | 5536857.188537 |

Summary

以前在書上學過MIPS的5階pipeline CPU，學習到各個階段所負責的工作以及因為pipeline的設計會遇到的問題像是各種Hazard，但經過這次實作後才知道每個階段中的細節需要我們去仔細思考及設計，像是ALUCtrl就會因每個人的想法不同而設計及定義上也會不同，使得整個design也會不一樣。  
還有Data及Control Hazard的解決因為需stall CPU，因此常常因為某條訊號線的疏忽而使得CPU運作有錯誤，所以經過這個作業以後對CPU變得非常了解，也為後面的作業鋪路。