**CA Final Report**

B05901074陳泓均

B05901158林緯瑋

**CPU Architecture:**

CPU的架構基本上參考Final project slide上面的架構，有PC\_CAL、Control、ALU、ALU\_Control、Imm\_Gen、RegWrite\_mux還有multDiv這幾個架構。以下分各個架構介紹。此部分僅介紹基本功能，關於新加入的instruction例如jal, jalr等，請參考後面的部分有詳細說明。

1. PC\_CAL: 主要input是PC，output是PC\_nxt，是為了計算下一個clk的PC是多少。須由一個mux去控制，有jump就是jump到PC+immediate，沒有jump就是+4。
2. Control: 基本上就是圖片上的這7個signal，接線方法就不詳述。其中，我們將MemtoReg改為RegWrite\_src[1:0]，因為之後加上jal, jalr等instruction之後，寫進rd的data source不只兩種，所以需兩個bit控制。最後會拿RegWrite\_src去控制RegWrite\_mux。
3. ALU: 根據ALU control signal，來做對應的ALU operation。我們的CPU有支援+、-、&、|這4個operation。另外，為了要支援SLTI、SRAI與SLLI，必須另外加上compare、logical left shift和arithmetic right shift等operation。
4. ALU\_Control: 利用control的ALUOp signal，還有funct3, opcode以及funct7的field判斷ALU會用到哪一種operation。要注意ADD和SUB的區別會用到funct7，但是ADDI沒有funct7 field，所以要先判斷是否是ADDI，再區別ADD和SUB。最後輸出一個4-bit的control signal，用來控制ALU的運作。
5. Imm\_Gen: 根據不同instruction，去拿取對應的immediate field，生成相對應的immediate output。要注意此處output也是32 bit而非64 bit。
6. RegWrite\_mux: 決定要write到rd的data source要從哪裡拿取的mux，原先的CPU只會有兩個source (memory和alu\_out)，但後面支援其他instruction之後，就不只一個source，因此需要額外新增一個module來處理。

**Datapath of jal, jalr, auipc:**

因為三種情況都會有不同的做法，所以需要jal, jalr和auipc三個額外的control signal。以下分三種instruction的情況討論。

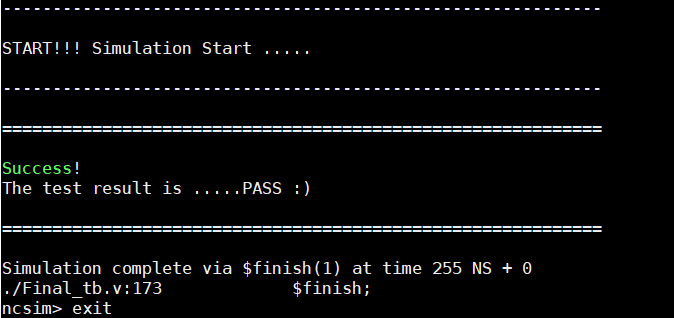
1. AUIPC: 在auipc當中，PC\_nxt = PC +4，但又需要將PC+immediate存到register，所以前者利用PC\_adder，後者利用ALU完成，所以alu\_in\_1 = PC，因此alu\_in\_1前面須加上mux，即alu\_in\_1 = auipc ? PC : rs1\_data。另外auipc的immediate比較不同，所以imm\_gen要另外判斷如果是auipc，imm\_gen = I\_in & {20'b1, 12'b0}。另外因為auipc是做加法，所以ALU control要判斷輸出相對應的control，且最後寫回regiser時是從ALU拿。(即regWrite\_mux要判斷)
2. JAL: 在jal當中，PC也要jump，所以PC會jump的條件變成 (branch & zero ) | jal。和auipc不同，jal不需要用到ALU，所以ALU相關的control和output都不須理會；只需要注意最後write register的時候，write的data是PC+4，所以regWrite\_mux要額外拿PC的數值，+4之後寫到rd。(不要不小心拿到ALU或memory的值)
3. JALR: 最後是jalr，它做的事情是要先把要jump的address從memory當中load出來，所以ALU做的是load的事情，只是這裡load的數值要寫到PC，而寫到rd的數值則是PC+4，所以在PC的mux要加上一個jalr的情況，是從mem\_rdata\_D去拿data；而regWrite\_mux和jal一樣，是存PC+4。

**How to handle multi-cycle operation - mul:**

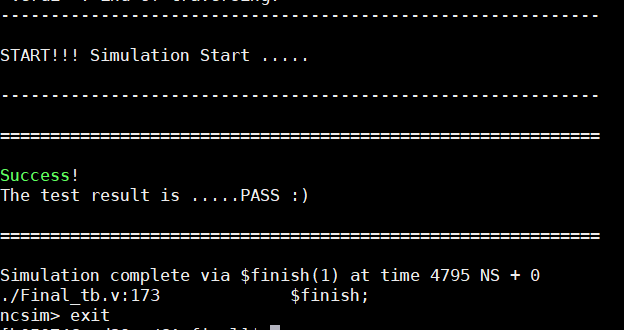
我們直接使用HW3的multDiv module，沒有改動內部的module(因此乘法器內部怎麼運作就不再贅述)。外面control的部分，我們為了讓整個CPU知道現在是在做mul，所以control signal多了一個mul。為了達到multi-cycle的功能，只要是在做mul的時候 (mul=1’b1)，PC就不加4 (PC\_nxt = PC)，這樣才不會fetch到下一個instruction。直到mul的ready拉起，代表mul做完之後，PC才會+4，fetch到下一個instruction。最後在要寫回register的時候，要額外判斷如果是mul，write的data要從multDiv的output拿。

**Total simulation time:**

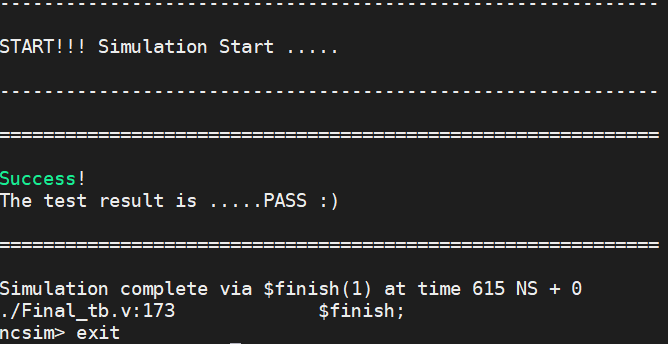
* Leaf:



* Fact:



* HW1:



**Observation:**

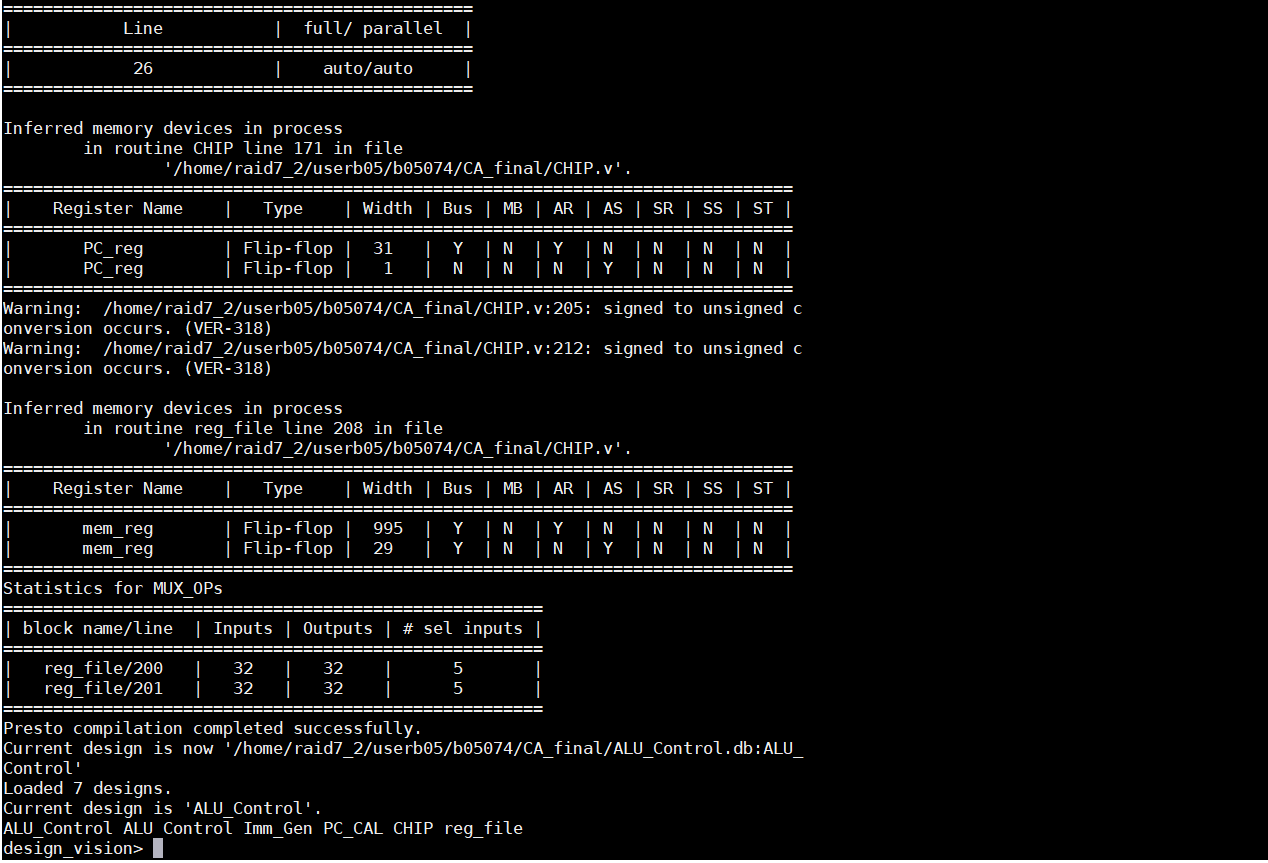
Non-pipelined的CPU實作起來相對簡單，因為幾乎使用的都是combinational circuit，且memory部分都由助教完成，因此實作上沒有太大的困難；只是需要額外處理multDiv的multi-cycle問題。原本以為必須像作業三一樣寫一個FSM、定義兩種state、加上state以及state\_nxt等參數，後來發現只需一個mul signal紀錄state即可。

一開始遇到的困難是在debug的時候，因為只要有一個部分有錯，例如PC，instruction的順序就會亂掉，所以要一個一個去trace回去找到源頭；只要一個指令有錯，整個程式執行的情形就會完全亂掉，所以debug上不是非常容易。後來就發現，debug的時候一定要看著要執行的assembly，一行一行去看有沒有執行正確，找出開始錯誤的地方下去修正。

實作了processor後體會到risc-v 指令集的分類邏輯非常清楚好懂，依照不同類型的指令分為大類別再往下細分，也盡量將相同的東西放在 instruction中相同的位置，因此 parse instruction 方面並沒有很困難。像是 SRAI、SLLI等就是在I-type下的special case，因此在處理上不需要額外判斷，而是按照I-type的方式判斷、計算immediate，最後在 ALU中取出 immediate需要的部分實現功能即可。

另外在實作auipc, jalr等指令的時候發現，如果設計得宜，其實可以有效節省硬體，例如auipc就不用再多加一個adder，直接用ALU的adder就可以(因為PC的adder是+4，和運算需要用到的不同)，只是需要另外用control訊號處理接線的問題。

**Register Table:**



**Work Distribution Table:**

|  |  |
| --- | --- |
|  | Work |
| 陳泓均 | 架構：PC\_CAL、Control、Imm\_Gen、RegWrite\_mux、multDiv  額外指令：auipc、jal、jalr、mul、slti |
| 林緯瑋 | 架構：ALU、ALU\_Control  額外指令：slli、srai |