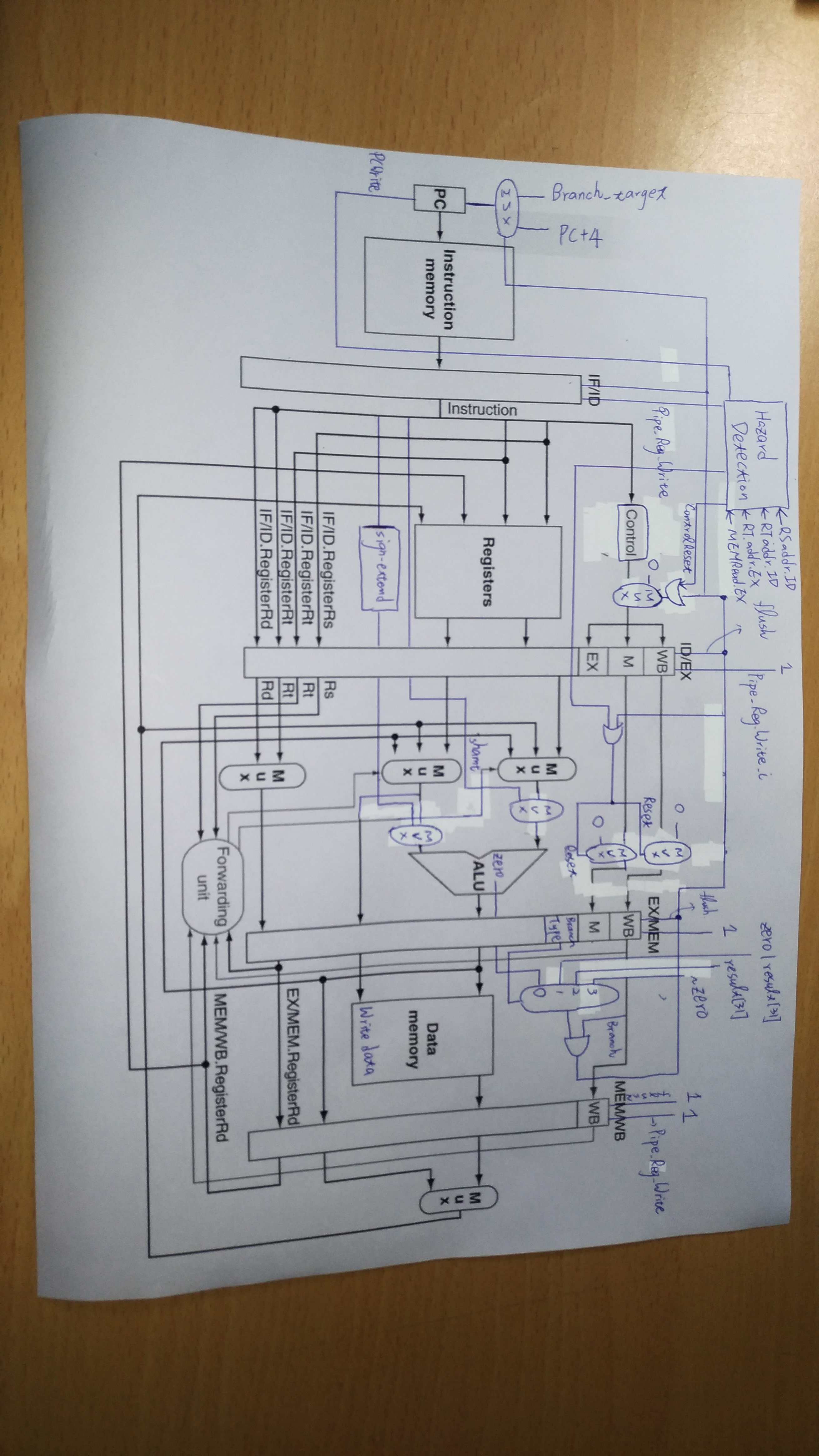
0416315 王定偉、0416005 張彧豪

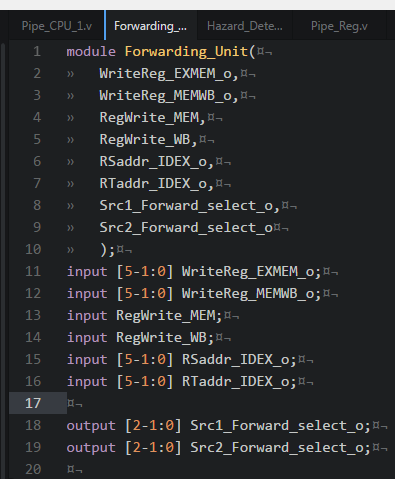
**Computer Organization Lab4**

**Architecture diagram:**

****

**Detailed description of the implementation:**

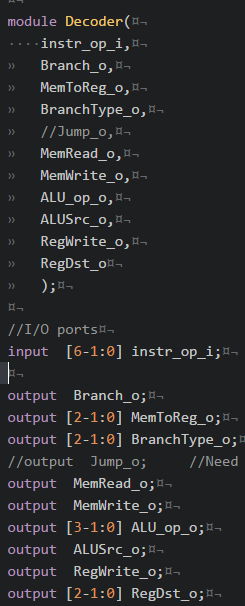
1. 以Pipe\_CPU\_1為主要的模組，此模組會去呼叫它所需要的所有模組，包含PC、IM、RG、Decoder、ALU\_Ctrl、Sign\_Extend、MUX、ALU、Data\_Memory、PipeReg、Forwarding\_Unit、Hazard\_Detection\_Unit等模組。此CPU為pipeline版本的CPU。
2. 這次大部分的模組都跟上一次lab一樣，唯獨新增了pipeline所需要的Pipe\_Reg、Forwarding\_Unit、Hazard\_Detection\_Unit這三個模組。而主要模組也與上次大同小異，只差在這次的主模組須呼叫Pipe\_Reg等模組。
3. Forwarding\_Unit模組說明：
4. 輸入輸出：



1. 此模組的功能就是判斷需不需要forwarding需要的值給前面的stage，判斷方法是講義上那種判斷方法。
2. 輸出對應說明：

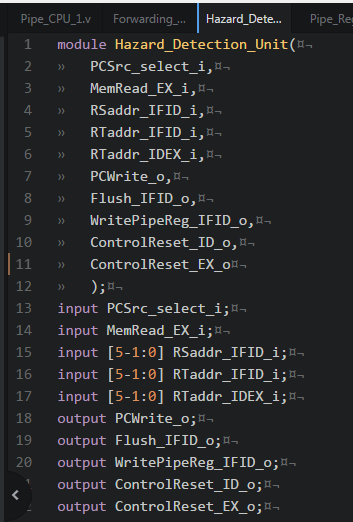
|  |  |
| --- | --- |
| Src1\_Forward\_select\_o,  Src2\_Forward\_select\_o | 運算值來源 |
| 00 | 原本register讀取的值 |
| 01 | EXMEM pipeline register傳回來的值 |
| 10 | MEMWB pipeline register傳回來的值 |

1. Decoder模組說明：
2. 輸入輸出：



輸出的控制訊號一個14 bit。

1. Hazard\_Detection\_Unit模組說明：
2. 輸入輸出：



1. 此模組的功能是判斷有無load-use data hazard發生。
2. 當有Load-use data hazard發生時，PCwrite\_o要為0，因為要stall一個cycle；WritePipeReg\_IFID\_o也要為0，重新decode一次指令；Flush\_IFID\_o要為1，因為要將原本指令變為bubble指令。
3. Pipe\_Reg模組說明：
4. 輸入：clk\_i, rst\_i, data\_i, Pipe\_Reg\_Write\_i, Flush\_i，輸出：data\_o
5. Flush\_i的用意在於判斷是否要將pipeline register裡面的重設為0，在control hazard時會需要用到；Pipe\_Reg\_Write\_i的用意在於當Flush\_i為0時是否要寫入新值進入pipeline register，這在需要stall cycle的時候會用到。
6. 因為有control hazard，所以有可能要將某些stage的控制訊號reset成0。在這次lab中，若branch發生了，則需將ID、EX這兩個stage的控制訊號reset成0，這部分的工作是交給兩個MUX來做，分別是Mux\_ControlReset\_ID及Mux\_ControlReset\_EX，經過處理的控制訊號分別為Real\_Control\_IDEX\_i及Real\_Control\_EXMEM\_i。
7. 各階段Pipeline register說明：
8. IF/ID pipeline register
9. 資料量：64 bit。
10. 資料：PC+4(32bit)、instruction(32bit)。
11. Pipe\_Reg\_Write\_i是由Hazard\_Detection\_Unit的WritePipeReg\_IFID\_o傳來的。
12. ID/EX pipeline register
13. 資料量：189 bit
14. 資料：

Real\_Control\_IDEX\_i (14bit)

PC+4 (32bit)

shift amout (32bit)

RSdata (32bit)

RTdata (32bit)

SE\_data\_o (32bit, sign-extend data)

RSaddr (5bit)

RTaddr (5bit)

RDaddr (5bit)

1. Pipe\_Reg\_Write\_i永遠是1，因為不會在這個階段有stall發生。
2. EX/MEM pipeline register
3. 資料量：142 bit
4. 資料：

Real\_Control\_EXMEM\_i

(8bit, {Branch\_o, MemtoReg\_o, BranchType\_o, MemRead\_o, MemWrite\_o, RegWrite\_o})

Branch\_target\_o (32bit)

zero\_o (1bit)

result\_o (32bit, ALU result)

RTdata\_IDEX\_o (32bit, RTdata come from IDEX pipeline register)

WriteReg (5bit, Write Register address)

SE\_data\_IDEX\_o(32bit, sign-extend data from IDEX pipeline register)

1. Pipe\_Reg\_Write\_i永遠是1，因為不會在這個階段有stall發生。
2. EX/MEM pipeline register
3. 資料量：104 bit
4. 資料：

Ori\_Control\_MEM (3bit, {MemtoReg\_o, RegWrite})

MEM\_Read\_data\_o (32bit, Date read from Memory)

ALU\_result\_EXMEM\_o (32bit, ALU result come from EXMEM pipeline register)

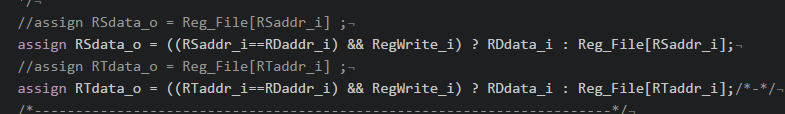
Write\_Reg\_EXMEM\_o (5bit, Write Register address come from EXMEM pipeline register)

SE\_data\_EXMEM\_o (32bit, sign-extend data from EXMEM pipeline register)

1. Pipe\_Reg\_Write\_i永遠是1，因為不會在這個階段有stall發生。
2. Flush\_i皆為0，因為branch發生時不會影像到在此stage的指令。
3. 當Branch指令在MEM stage發現要branch時，PCSrc\_select\_o會變成1，代表要選branch target當作下一個pc number，而這個PCSrc\_select\_o也是IFID、IDEX、EXMEM三個pipeline register的Flush\_i訊號來源。
4. Stall方法說明，在需要的stall的register上面我們是利用控制Pipe\_Reg\_Write\_i來控制要不要將pipeline register的值保留住，當其值為1時，代表將新的值寫入register；反之則不寫入新值，利用這種方式達到stall的效果。

**Problems encountered and solutions:**

1. Register不能同時讀寫的問題，照理說如果在ID stage要對同一個register做讀跟寫的動作應該是不會有hazard的問題，但是在測第一個測資的時候發現當現在的cycle要同時對同一個register做讀取時無法讀到正確的值出來。看了一下code發現原來是寫法造成的，因為當這個cycle要寫入register時，要下一個cycle那個新的值才會顯現在register上，這樣的現象造成讀取該register永遠讀到舊值，解決方法就是在要寫也要讀的時候，直接把要寫入的新值指派到輸出，即可解決此問題。如下圖所示：



1. Forwarding判斷的方法的流程其實一開始很困惱我們，因為一開始我們是分成四個case分開處理，可是這樣處理的結果就是select訊號會被改兩次，造成訊號錯誤。後來我們直接分成兩種case，一種是要forwarding給operand 1跟要forwarding給operand 2，至於是要從哪裡forwarding則是留作每種case內部判斷，這樣確保select訊號不會被改到兩次。

**Lesson learnt (if any):**

1. 簡易版Pipeline CPU的實作方法。
2. Control hazard的處理方法。