**Computer Architecture Project 1 Report**

**Team: 單生狗聯盟(WannaLaugh)**

**2.1 Module Explanation**

|  |
| --- |
| **Adder.v**  **Adder.v** module have two inputs, **data1\_in** and **data2\_in**, both are 32 bits, one output, **data\_o** which is also 32 bits.  In **Adder.v**, **data1\_in** and **data2\_in** will be added and the result is assigned to **data\_o**.  **Adder.v** is used twice for the operation **PC + 4** and **PC + immediate** in instruction ‘**beq**’.  For **PC + 4**, the module is called **Add\_PC** in **CPU.v**.  The **data\_o** is used in **MUX\_PC** as **data1\_i** which contain the value of current **PC + 4**.  For **PC + immediate**, the module is called **Add\_Branch** in **CPU.v**.  The **data\_o** is used in **MUX\_PC** as **data2\_i** which contain the value of current **PC + immediate**. |
| **ALU.v**  **ALU.v** module have three inputs, **data1\_i** and **data2\_i** are 32 bits while **ALUCtrl\_i** is 4 bits, one outputs, **data\_o** which is 32 bits.  A register of size 32 bits is declared for **data\_o**.  By using the case statements, **ALU.v** perform different operation corresponded to **ALUCtrl\_i** as shown in the table below:   |  |  |  | | --- | --- | --- | | **ALUCtrl\_i** | Instr | Operation | | 0000 | and | **data\_o** <= $signed(**data1\_i**) & $signed(**data2\_i**); | | 0001 | xor | **data\_o** <= $signed(**data1\_i**) ^ $signed(**data2\_i**); | | 0010 | sll | **data\_o** <= $signed(**data1\_i**) << **data2\_i**; | | 0011 | add | **data\_o** <= $signed(**data1\_i**) + $signed(**data2\_i**); | | 0100 | sub | **data\_o** <= $signed(**data1\_i**) - $signed(**data2\_i**); | | 0101 | mul | **data\_o** <= $signed(**data1\_i**) \* $signed(**data2\_i**); | | 0110 | addi | **data\_o** <= $signed(**data1\_i**) + $signed(**data2\_i**); | | 0111 | srai | **data\_o** <= $signed(**data1\_i**)>>> **data2\_i[4:0]**; | | 1000 | lw | **data\_o** <= $signed(**data1\_i**) + $signed(**data2\_i**); | | 1001 | sw | **data\_o** <= $signed(**data1\_i**) + $signed(**data2\_i**); | | 1010 | beq | **data\_o** <= 0 |   The result of the operation is assigned to **data\_o**.  **data\_o** is the result of an instruction and use as **ALU\_Result\_i** in **Register\_EXMEM.v**.  For **ALUCtr\_i** = 1010, which is instruction ‘**beq**’, no operation needed in **ALU.v** thus assign 0 to **data\_o**. |
| **ALU\_Control.v**  **ALU\_Control.v** module have two inputs, **funct\_i** which is 10 bits and **ALUOp\_i** which is 2 bits, one output**, ALUCtrl\_o** which is 4 bits.  **funct\_i** is a concatenation of instruction [31:25] and instruction [14:12].  A register of size 4 bits is declared for **ALUCtrl\_o**.  By using case statements and if-else statements, **ALU\_Control.v** assign different value to **ALUCtrl\_o** based on the value of **funct\_i** and **ALUOp\_i.**   |  |  |  |  |  | | --- | --- | --- | --- | --- | | **ALUOp\_i** | **funct\_i** | | **Instruction** | **ALUCtrl\_o** | | **funct\_i [9:3]** | **funct\_i [2:0]** | | 10 | 0000000 | 111 | and | 0000 | | 0000000 | 100 | xor | 0001 | | 0000000 | 001 | sll | 0010 | | 0000000 | 000 | add | 0011 | | 0100000 | 000 | sub | 0100 | | 0000001 | 000 | mul | 0101 | | 00 | x | 000 | addi | 0110 | | x | 101 | srai | 0111 | | x | 010 | lw | 1000 | | 01 | x | x | sw | 1001 | | 11 | x | x | beq | 1010 |   **ALUCtrl\_o** is used in **ALU.v** to determine which operation to be performed. |
| **And.v**  **And.v** Module have two inputs, **data1\_i** and **data2\_i** which both 1 bit, one output **data\_o** which also 1 bit.  In **And.v**, the **data1\_i** and **data2\_i** undergo operation of ‘**&**’ and assign the result to **data\_o**.  **data\_o** is used to determined which PC is used in **MUX\_PC** as **select\_i** and also as **Flush\_i** in **Register\_IFID.v.** |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Control.v**  **Control.v** module have two inputs, **Op\_i** which is 7 bits and **NoOp\_i** which is 1 bit, 7 outputs, **ALUOp\_o** which is 2 bits, **RegWrite\_o**, **MemtoReg\_o**, **MemRead\_o**, **MemWrite\_o**, **ALUSrc\_o** and **Branch\_o** which all 1 bit.  The **Op\_i** is the opcode of an instruction.  The **NoOp\_i** is used for the hazard control.  Seven registers are declared for seven outputs with corresponding size.  In **Control.v**, **Op\_i** and **NoOp\_i** are used to determine the value of seven outputs based on the table below:   |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | | **NoOp\_i** | **Op\_i** | **RegWrite\_o** | **MemtoReg\_o** | **MemRead\_o** | **MemWrite\_o** | **ALUOp\_o** | **ALUSrc\_o** | **Branch\_o** | | 1 | x | 0 | 0 | 0 | 0 | 10 | 0 | 0 | | 0 | 0110011 | 1 | 0 | 0 | 0 | 10 | 0 | 0 | | 0010011 | 1 | 0 | 0 | 0 | 00 | 1 | 0 | | 0000011 | 1 | 1 | 1 | 0 | 00 | 1 | 0 | | 0100011 | 0 | x | 0 | 0 | 01 | 1 | 0 | | 1100011 | 0 | x | 0 | 0 | 11 | 0 | 1 | | 0000000 | 0 | 0 | 0 | 0 | 00 | 0 | 0 |   For **Op\_i == 7’b0000000**, it is a special case which no operation needed for this instruction.  **RegWrite\_o** is used in **Register\_IDEX.v** as **RegWrite\_i**.  **MemtoReg\_o** is used in **Register\_IDEX.v** as **MemtoReg\_i.**  **MemRead\_o** is used in **Register\_IDEX.v** as **MemRead\_i.**  **MemWrite\_o** is used in **Register\_IDEX.v** as **MemWrite\_i.**  **ALUOp\_o** is used in **Register\_IDEX.v** as **ALUOp\_i.**  **ALUSrc\_o** is used in **Register\_IDEX.v** as **ALUSrc\_i.**  **Branch\_o** is used in **And.v** as one of the inputs for determination in instruction ‘**beq**’. |
| **Equal.v**  **Equal.v** module have two inputs, **data1\_i** and **data2\_i** which both are 32 bits, one output **equal\_o** which is 1 bit.  In **Equal.v**, if **data1\_i** and **data2\_i** are same, assign 1 to **equal\_o**.  If **data1\_i** and **data2\_i** are not same assign 0 to **equal\_o**.  **equal\_o** is used as one of the inputs in **And.v** for determination in instruction ‘**beq**’. |
| **MUX32.v**  **MUX32.v** module have 3 inputs, **data1\_i** and **data2\_i** which both 32 bits and **select\_i** which is 1 bits, one output, **data\_o** which is 32 bits.  In **MUX32.v**, the **select\_i** is used to decide which input data is assign to **data\_o**.   |  |  | | --- | --- | | **select\_i** | **data\_o** | | 0 | **data1\_i** | | 1 | **data2\_i** |   **MUX\_32** is used three times which is as shown below in **CPU.v**:  **MUX\_PC** on determination of **pc\_i** in **PC.v**  **MUX\_ALUSrc** on determination of **data2\_i** in **ALU.v**  **MUX\_MemtoReg** on determination of **RDdata\_i** in **Registers.v** and **WB\_WriteData\_i** in  module **Forward\_MUX\_A** and **Forward\_MUX\_B** in **CPU.v**. |
| **Register\_IFID.v**  **Register\_IFID.v** module have  Six inputs:  **clk\_i**, **start\_i**, **Stall\_i** and **Flush\_i** which are 1 bit  **pc\_i** and **instr\_i** which both 32 bits  Two outputs:  **pc\_o** and **instr\_o** which both 32 bits  Two register of size 32 bits was declare for **pc\_o** and **instr\_o**.  On every posedge of the **clk\_i**, **Register\_IFID.v** determine **pc\_o** and **instr\_o** as shown below:   |  |  |  |  | | --- | --- | --- | --- | | **Flush\_i** | **Stall\_i** | **pc\_o** | **instr\_o** | | 1 | x | **pc\_o <= 32’b0** | **instr\_o <= 32’b0** | | 0 | 1 | **pc\_o <= pc\_o** | **instr\_o <= instr\_o** | | 0 | 0 | **pc\_o <= pc\_i** | **instr\_o <= instr\_i** |   **pc\_o** is used as one of the inputs in mmodule **Add\_Branch** in **CPU.v**.  **instr\_o** is used as the 32 bits instruction, **instr** saved in 32 bits wire in **CPU.v** which will then send to **Control.v**, **Registers.v**, **Hazard\_Detection.v**, **Sign\_Extend.v** and **Register\_IDEX.v** |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Register\_IDEX.v**  **Register\_IDEX.v** module have  Fifteen inputs:  **clk\_i**, **start\_i** which both 1 bit  **RS1Data\_i**, **RS2Data\_i** and **SignExtended\_i** which are 32 bits  **funct\_i** which is 10 bits  **RS1Addr\_i**, **RS2Addr\_i** and **Rd\_Addr\_i** which are 5 bits  **RegWrite\_i**, **MemtoReg\_i**, **MemRead\_i**, **MemWrite\_i** and **ALUSrc\_i** which all 1 bit  **ALUOp\_i** which is 2 bits  Thirteen outputs:  **RS1Data\_o**, **RS2Data\_o** and **SignExtended\_o** which are 32 bits  **funct\_o** which is 10 bits  **RS1Addr\_o**, **RS2Addr\_o** and **Rd\_Addr\_o** which are 5 bits  **RegWrite\_o**, **MemtoReg\_o**, **MemRead\_o**, **MemWrite\_o** and **ALUSrc\_o** which all 1 bit  **ALUOp\_o** which is 2 bits  Thirteen registers are declared for each output with correspond size.  On every posedge of **clk\_i**, **Register\_IDEX.v** determine its outputs as shown below:   |  |  | | --- | --- | | **start\_i** | outputs | | 1 | **RS1Data\_o <= RS1Data\_i;**  **RS2Data\_o <= RS2Data\_i;**  **SignExtended\_o <= SignExtended\_i;**  **RS1Addr\_o <= RS1Addr\_i;**  **RS2Addr\_o <= RS2Addr\_i;**  **funct\_o <= funct\_i;**  **Rd\_Addr\_o <= Rd\_Addr\_i;**  **RegWrite\_o <= RegWrite\_i;**  **MemtoReg\_o <= MemtoReg\_i;**  **MemRead\_o <= MemRead\_i;**  **MemWrite\_o <= MemWrite\_i;**  **ALUOp\_o <= ALUOp\_i;**  **ALUSrc\_o <= ALUSrc\_i;** | | 0 | **RS1Data\_o <= RS1Data\_o;**  **RS2Data\_o <= RS2Data\_o;**  **SignExtended\_o <= SignExtended\_o;**  **RS1Addr\_o <= RS1Addr\_o;**  **RS2Addr\_o <= RS2Addr\_o;**  **funct\_o <= funct\_o;**  **Rd\_Addr\_o <= Rd\_Addr\_o;**    **RegWrite\_o <= RegWrite\_o;**  **MemtoReg\_o <= MemtoReg\_o;**  **MemRead\_o <= MemRead\_o;**  **MemWrite\_o <= MemWrite\_o;**  **ALUOp\_o <= ALUOp\_o;**  **ALUSrc\_o <= ALUSrc\_o;** |   Overall,  If **start\_i** == 1, update the all outputs to correspond inputs  If **start\_i** == 0, all the outputs remain the previous value  **RS1Data\_o** is used as **EX\_RS\_Data\_i** in module **Forward\_MUX\_A** in **CPU.v**  **RS2Data\_o** is used as **EX\_RS\_Data\_i** in module **Forward\_MUX\_B** in **CPU.v**  **SignExtended\_o** is used as one of the inputs in module **MUX\_ALUSrc** in **CPU.v**.  **RS1Addr\_o** is used as **EX\_Rs1\_i** in **Forwarding\_Unit.v** for the determination of data forwarding  **RS2Addr\_o** is used as **EX\_Rs2\_i** in **Forwarding\_Unit.v** for the determination of data forwarding  **funct\_o** is used as **funct\_i** in **ALU\_Control.v** to determine which operation will the **ALU.v** perform  **Rd\_Addr\_o** is used as **Rd\_Addr\_i** in **Register\_EXMEM.v**.  **RegWrite\_o** is used as **RegWrite\_i** in **Register\_EXMEM.v**.  **MemtoReg\_o** is used as **MemtoReg\_i** in **Register\_EXMEM.v**.  **MemRead\_o** is used as **MemRead\_i** in **Register\_EXMEM.v**.  **MemWrite\_o** is used as **MemWrite\_i** in **Register\_EXMEM.v**.  **ALUOp\_o** is used as **ALUOp\_i** in **ALU\_Control.v** to determine which operation will the **ALU.v** perform  **ALUSrc\_o** is used as **select\_i** in module **MUX\_ALUSrc** in **CPU.v** to determine either **data1\_i** or **data2\_i** is choose. |
| **Register\_EXMEM.v**  **Register\_EXMEM.v** module have  Nine inputs:  **clk\_i**, **start\_i** which both 1 bit  **ALU\_Result\_i** and **MemWrite\_Data\_i** which both 32bits  **Rd\_Addr\_i** which is 5 bits  **RegWrite\_i**, **MemtoReg\_i**, **MemRead\_i** and **MemWrite\_i** which all 1 bit  Seven outputs:  **ALU\_Result\_o** and **MemWrite\_Data\_o** which both 32bits  **Rd\_Addr\_o** which is 5 bits  **RegWrite\_o**, **MemtoReg\_o**, **MemRead\_o** and **MemWrite\_o** which all 1 bit  Seven registers are declared for each output with correspond size.  On every posedge of **clk\_i**, **Register\_EXMEM.v** determine its outputs as shown below:   |  |  | | --- | --- | | **start\_i** | outputs | | 1 | **ALU\_Result\_o <= ALU\_Result\_i;**  **MemWrite\_Data\_o <= MemWrite\_Data\_i;**  **Rd\_Addr\_o <= Rd\_Addr\_i;**  **RegWrite\_o <= RegWrite\_i;**  **MemtoReg\_o <= MemtoReg\_i;**  **MemRead\_o <= MemRead\_i;**  **MemWrite\_o <= MemWrite\_i;** | | 0 | **ALU\_Result\_o <= ALU\_Result\_o;**  **MemWrite\_Data\_o <= MemWrite\_Data\_o;**  **Rd\_Addr\_o <= Rd\_Addr\_o;**  **RegWrite\_o <= RegWrite\_o;**  **MemtoReg\_o <= MemtoReg\_o;**  **MemRead\_o <= MemRead\_o;**  **MemWrite\_o <= MemWrite\_o;** |   Overall,  If **start\_i** == 1, update the all outputs to correspond inputs  If **start\_i** == 0, all the outputs remain the previous value  **ALU\_Result\_o** is used as **MEM\_ALU\_Result\_i** in **Forward\_MUX\_A** and **Forward\_MUX\_B** in **CPU.v**  **MemWrite\_Data\_o** is used as **data\_i** in **Data\_Memory.v** which is the data waiting to write to memory  **Rd\_Addr\_o** is used as **MEM\_Rd\_i** in **Forwarding\_Unit.v** for the determination of data forwarding  **RegWrite\_o** is used as **RegWrite\_i** in **Register\_MEMWB.v** and **MEM\_RegWrite\_i** in **Forwarding\_Unit.v**  **MemtoReg\_o** is used as **MemtoReg\_i** in **Register\_MEMWB.v**  **MemRead\_o** is used as **MemRead\_i** in **Data\_Memory.v** to determine whether the Data in Memory of address **addr\_i** is read and assign to **data\_o**.  **MemWrite\_o** is used as **MemWrite\_i** in **Data\_Memory.v** to determine whether Data, **data\_i** is written to memory of address **addr\_i.** |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Register\_MEMWB.v**  **Register\_MEMWB.v** module have  Seven inputs:  **clk\_i**, **start\_i** which both 1 bit  **ALU\_Result\_i** and **MemRead\_Data\_i** which both 32 bits  **Rd\_Addr\_i** which is 5 bits  **RegWrite\_i** and **MemtoReg\_i** which both 1 bit  Five outputs:  **ALU\_Result\_o** and **MemRead\_Data\_o** which both 32 bits  **Rd\_Addr\_o** which is 5 bits  **RegWrite\_o** and **MemtoReg\_o** which both 1 bit  Five registers are declared for each output with correspond size.  On every posedge of **clk\_i**, **Register\_MEMWB.v** determine its outputs as shown below:   |  |  | | --- | --- | | **start\_i** | outputs | | 1 | **ALU\_Result\_o <= ALU\_Result\_i; MemRead\_Data\_o <= MemRead\_Data\_i; Rd\_Addr\_o <= Rd\_Addr\_i;**    **RegWrite\_o <= RegWrite\_i;**  **MemtoReg\_o <= MemtoReg\_i;** | | 0 | **ALU\_Result\_o <= ALU\_Result\_o;**  **MemRead\_Data\_o <= MemRead\_Data\_o; Rd\_Addr\_o <= Rd\_Addr\_o;**    **RegWrite\_o <= RegWrite\_o;**  **MemtoReg\_o <= MemtoReg\_o;** |   Overall,  If **start\_i** == 1, update the all outputs to correspond inputs  If **start\_i** == 0, all the outputs remain the previous value  **ALU\_Result\_o** is used as one of the inputs in module **MUX\_MemtoReg** in **CPU.v**  **MemRead\_Data\_o** is used as one of the inputs in module **MUX\_MemtoReg** in **CPU.v**  **Rd\_Addr\_o** is used as **RDaddr\_i** in **Registers.v** which decide where the **RDdata\_i** is write to register with address **RDAddr\_i.**  **RegWrite\_o** is used as **RegWrite\_i** in **Registers.v** to decide whether Write Data, **RDdata\_i** is written to register of address **RDaddr\_i.**  **MemtoReg\_o** is used as **select\_i** in module **MUX\_MemtoReg** in **CPU.v** to determine either **data1\_i** or **data2\_i** is choose. |
| **Shift\_Left.v**  **Shift\_Left.v** module have one input, **data\_i** and one output **data\_o** which both are 32 bits.  In **Shift\_Left.v**, **data\_i** is shift left for 1 bit and assign the result to **data\_o**.  **data\_o** is used in **Add\_Branch**as one of the inputs for determination in instruction ‘**beq**’. |
| **Sign\_Extend.v**  **Sign\_Extend.v** module have one inputs, **data\_i** which is 32 bits and one output, **data\_o** which is 32 bits.  Two registers of size 32 bits is declare for **data\_i** and **data\_o**.  The immediate in the 32 bits instruction is signed extended by using the concatenation operator of Verilog as shown in table below:   |  |  | | --- | --- | | **data\_i[6:0]** | **data\_o** | | 0110011 | **data\_o**[31:0] = 32'b0; | | 0010011 | **data\_o**[31:0] = {{20{**data\_i**[31]}},**data\_i**[31:20]} | | 0000011 | **data\_o**[31:0] = {{20{**data\_i**[31]}},**data\_i**[31:20]}; | | 0100011 | **data\_o**[31:12] = {20{**data\_i**[31]}};  **data\_o**[11:5] = **data\_i**[31:25];  **data\_o**[4:0] = **data\_i**[11:7]; | | 1100011 | **data\_o**[31:11] = {21{**data\_i**[31]}};  **data\_o**[10] = **data\_i**[7];  **data\_o**[9:4] = **data\_i**[30:25];  **data\_o**[3:0] = **data\_i**[11:8]; | | 0000000 | **data\_o**[31:0] = 32'b0; |   The **data\_o** is used in **Register\_IDEX.v** as **SignExtended\_i**. |

|  |
| --- |
| **CPU.v**  CPU.v **connects all the modules together via wires** to make the logic in the datapath works.  In the project, some of the modules need to be reused. For example, MUX32 and MUX32\_4. We can reuse those modules with different names. MUX32 are reused with names of MUX\_ALUSrc, MUX\_MemtoReg and MUX\_PC. MUX32\_4 are reused with names of Forward\_MUX\_A and Forward\_MUX\_B. |
| **Hazard\_Detection.v**  這個模組主要是實作當**處理lw指令時所有產生的data hazard**。 這個模組有四個inputs分別是RS1addr\_i, RS2addr\_i,MemRead\_i,RdAddr\_i, 三個outputs分別是PCWrite\_o, Stall\_o, NoOp\_o。  當**ID/EX** 這個register的**MemRead\_i為1**（也就是說相對IF/ID這個register來說上一個指令是lw）的時候，如果ID/EX register的**rd address**和IF/ID的**rs1 或 rs2**任一個address**相同**的話，這時候便會**發生data hazard**。因此，**Hazard Detection Unit**便會使：-   1. **Stall\_o = 1**告訴IF/ID 這個register要stall一個cycle 2. **PCWrite\_o = 0**告訴PC這個模組不要fetch下一個instruction 3. **NoOp = 1**，告訴Control.v這個模組這個回合不做任何操作 |
| **Forwarding\_Unit.v**  由於有**pipeline**，CPU被分為5段，在某些情形下，**為了不要等到結果寫到register才取得資料**，會**直接從pipeline**不同區**取得結果**進行計算，稱之為**forward**  **實際作法**是使用**Forwaring Unit進行判斷**後，**傳訊息到MUX32\_4**選取正確資料，**再把資料傳輸到ALU**等需要相關資料的unit中  **狀況一**  **instr1**與**instr2**相連，而**instr2.rs會使用到instr1.rd**時，須**從EX/MEM取出instr1.rd**，不要等到它真的到寫入register才拿資料（此外也要確認instr1確實有要寫入register，且寫入目標不是x0）  此時，**傳訊息到MUX32\_4為“10”**，選擇**從EX/MEM拿資料**  **範例**    **狀況二**  **instr1, instr2, instr3依次相連**，而**instr3.rs會使用到instr1.rd**時，**須從MEM/WB取出instr1.rd**，不要等到它真的到寫入register才拿資料（此外也要確認instr1確實有要寫入register，且寫入目標不是x0）  此時，**傳訊息到MUX32\_4為“01”**，選擇**從MEM/WB拿資料**  **範例**    此處有個例外情形，**在instr2.rs會使用到instr1.rd**時，即符合狀況一的情形時，**應從EX/MEM取出instr1.rd**，所使用的結果才會是正確的 此時，**傳訊息到MUX32\_4為“10”**，選擇**從EX/MEM拿資料** **範例** |
| **MUX32\_4**  本運算單元是在**接收Forwarding\_Unit的結果**之後，**選擇欲運算的rs1, rs2**要從何處取得  **若Forwarding\_Unit結果為00**，表示沒有forward的需求，則**直接從ID/EX拿資料即可**  **若Forwarding\_Unit結果為10**，**從EX/MEM取得資料**  **若Forwarding\_Unit結果為01**，**從MEM/WB取得資料** |

**2.2. Members & Teamwork**

|  |  |
| --- | --- |
| **Member** | **Works** |
| **B07902091 周俊廷** | **修改Hw4的modules**  **添加 lw, sw和beq 三個 instructions 以及這三個instructions所需的modules**  **完成4個pipeline registers 以及在testbench.v initialize pipeline registers**  **整理我所修改或增加的modules的module explanations**  **添加我所修改或增加的modules到CPU裏，並接線。** |
| **B06902098 李恩慈** | **I am in charge of collecting my teammates reports, and arrange them.** |
| **B07902089李智源** | **負責Hazard\_Detection.v和IF/ID register**  **添加我所完成的modules到CPU，並重新接線** |
| **B07902059陳君翰** | **負責 Forwarding\_Unit.v 和 MUX32\_4.v。**  **添加我所完成的modules到CPU，並重新接線** |

**2.3. Difficulties Encountered and Solutions in This Project**

|  |  |
| --- | --- |
| **B06902098**  **李恩慈** | Although I try to write the CPU.v, I found out that there are actually a lot of errors in it. The wires have confusing names and become easily connected to wrong places. But my teammates helped me debug, and they actually rewrote the whole CPU.v! Finally, the program works! |
| **B07902089**  **李智源** | **一開始 PCWrite\_o 一直為0，導致無法fetch下一個instruction,之後CPU.v接線後就完成了。Stall h和Flush的時候在IF/ID register 沒處理，導致之後的instruction都出問題。** |
| **B07902091**  **周俊廷** | **在一開始完成后對於32‘b0 的instruction在Control.v裏沒做特別處理，導致Control.v的output ports裏面的值出現錯誤的數值，直接導致了在register或memory裏出現錯誤的寫入。在加上對Op\_i[6:0] == 7’b0000000 的判斷后，把所有output ports設爲0，解決了這一問題。** |
| **B07902059**  **陳君翰** | **幾乎沒有遇到什麼問題，按照上課的slide即可完成** |

**2.4. Development Environment**

The OS used : CSIE Workstation

The Compiler used : iverilog