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

<u>狀況一</u>

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**

<u>範例</u>

```
add x10, x8, x9
add x11, x10, x1
```

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

```
add x10, x8, x9
add x8, x0, x0
add x11, x10, x1
```

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

```
add x10, x8, x9
add x10, x10, x1
add x11, x10, x1
```

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