**Computer Architecture Project 1 Report**

*B08902083 謝鈺嘉*

**I. Modules Explanation**

**I.1 Adder.v**

Adder.v takes two 32-bit inputs, data1\_in and data2\_in. It has one 32-bit output, data\_o. Two Adders are initialized in CPU.v, Add\_PC and Add\_Branch. For Add\_PC, o. Adder takes the current PC, PC\_now, as data1\_i and 4 as data2\_i, the output is stored in PC\_Four which refers to the next line of instructions. For Add\_Branch, Adder takes the immediate as data1\_i and the current PC from Register\_IFID, IFID\_PC, as data2\_i, the output is stored in PC\_Branch which refers to the jump address in BEQ. Both outputs will be used in MUX\_PC to determine the next instruction.

**I.2 ALU.v**

ALU.v takes two 32-bit inputs, data1\_i and data2\_i, and one 4-bit input, ALUCtrl\_i. It has one 32-bit output (a register of the same size is also declared), data\_o. By using the case statements, ALU.v performs different operations corresponding to ALUCtrl\_i as shown 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. It is used as ALU\_Res\_i in Register EXMEM. For the BEQ instruction, there is no operation needed so data\_o is given the value zero.

**I.3 ALU\_Control.v**

ALU\_Control.v takes one 10-bit input, funct\_i, and one 2-bit input, ALUOp\_i. It has one 4-bit output (a register of the same size is also declared), ALUCtrl\_o. funct\_i is a concatenation of instruction [31:25] and instruction [14:12]. ALUCtrl\_o is used in ALU.v to choose which operation is performed. By using case statements, ALU\_Control.v assign different values to ALUCtrl\_o based on the value of funct\_i and ALUOp\_i as shown below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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 |

**I.4 AND.v**

AND.v takes two 1-bit inputs, data1\_in and data2\_in. It has one 1-bit output, data\_o. The two inputs undergo the operation “&” and the result is assigned to data\_o. The output is used as select\_i in MUX\_PC and as Flush\_i in Register\_IFID.

**I.5 Control.v**

Control.v takes one 7-bit input, Op\_i, and one 1-bit input, NoOp\_i. Op\_i corresponds to the opcode of the instruction and NoOp\_i is used for hazard control. Control.v has one 2-bit output (a register of the same size is also declared), ALUOp\_o, and seven 1-bit outputs (a register of the same size is also declared), RegWrite\_o, MemtoReg\_o, MemRead\_o, MemWrite\_o, ALUSrc\_o and Branch\_o. In Control.v, Op\_i and NoOp\_i are used to determine the value of seven outputs based on the table below:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **NoOp** | **Op** | **RegWrite** | **MemtoReg** | **MemRead** | **MemWrite** | **ALUOp** | **ALUSrc** | **Branch** |
| 1 | X | 0 | 0 | 0 | 0 | 10 | 0 | 0 |
| 0 | 0110011 | 1 | 0 | 0 | 0 | 10 | 0 | 0 |
| 0 | 0010011 | 1 | 0 | 0 | 0 | 00 | 1 | 0 |
| 0 | 0000011 | 1 | 1 | 1 | 0 | 00 | 1 | 0 |
| 0 | 0100011 | 0 | X | 0 | 1 | 01 | 1 | 0 |
| 0 | 1100011 | 0 | X | 0 | 0 | 11 | 0 | 1 |

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 data1\_in to determine if the instruction should jump (because 1 & X = X).

**I.6 CPU.v**

CPU.v takes three 1-bit inputs: clk\_i, clock of the cycle datapath, rst\_i, used in PC.v as parameter to reset PC to zero, and start\_i, used in PC.v as parameter if next PC is pc\_i. For those modules in CPU.v, the input ports and output ports are connected as shown in the datapath from the Project assignment by a wire declared with corresponded size between two ends. Some reoccurring variables are given a prefix, like Ctrl\_ALUOp and IDEX\_ALUOp, to differentiate them. Some modules are also used several times like Adder.v is used twice: Add\_PC and Add\_Branch.

**I.7 Equal.v**

Equal.v takes two 32-bit inputs, data1\_in and data2\_in. It has one 1-bit output, data\_o. If data1\_in and data2\_in have the same value, then data\_o will be set to one. It will be set to zero otherwise. The output is used in AND.v as data2\_in and the determining factor of the BEQ instruction.

**I.8 Forwarding\_Unit.v**

Forwarding\_Unit.v takes four 5-bit inputs, EX\_RS1\_i, EX\_RS2\_i, MEM\_Rd\_i, and WB\_Rd\_i, two 1-bit inputs, MEM\_RegWrite\_i and WB\_RegWrite\_i. It has two 2-bit outputs (a register of the same size is also declared), Forward\_A\_o and Forward\_B\_o. There is also two additional 1-bit register declared, flag\_A and flag\_B. Forward\_A\_o and Forward\_B\_o are used in ForwardA\_MUX and ForwardB\_MUX, respectively, as Forward\_i. It is first assigned the value 00, meaning there is no need for forwarding. We also assign the value zero to flag\_A and flag\_B. Next, we handle the two possibility cases.

The first case happens if there exist two data dependent consecutive instructions where instruction two needs the result of instruction one. In this case, we will take that value from Register\_EXMEM (before it is written in the register) and forward it so that instruction two can continue to run without stalling. We will assign the value 10 to Forward\_X\_MUX (X can either be A or B, depending on the case). The example of the first case can be seen below:

*add x30 x28 x29*

*add x31 x30 x0*

The second case happens if there exist three consecutive instructions where instruction three need the result of instruction one. In this case, we will take the value from Register\_MEMWB (before it is written in the register) and forward it so that instruction three can run without having to stall. We will assign the value 01 to Forward\_X\_MUX (X can either be A or B, depending on the case). The example of the second case can be seen below:

*add x30 x28 x29*

*add x5 x6 x7*

*add x31 x30 x0*

**I.9 Hazard\_Detection.v**

Hazard\_Detection.v takes one 1-bit input, MemRead\_i, and three 5-bit input, RDaddr\_i, RS1addr\_i and RS2addr\_i. It has three 1-bit outputs, PCWrite\_o, Stall\_o, and NoOp\_o. IFID\_instr[19:15] is assigned to RS1addr\_i, IFID\_instr[24:20] is assigned to RS2addr\_i, IDEX\_MemRead is assigned to MemRead\_i and IDEX\_RDaddr is assigned to RDaddr\_i. The main purpose of this module is to handle the data hazards that arise from the LW instruction. When (IDEX) MemRead has the value of one, meaning (to Register\_IFID) that last instruction was a LW instruction, thus if any of RS1addr\_i or RS2addr\_i have the same value as RDaddr\_i, a data hazard has occurred. Therefore, we will set Stall\_o to one, meaning we are telling Register\_IFID that we are going to perform stall cycle, and NoOp\_o to one, signaling we have a data hazard, and PCWrite\_o to zero, meaning that PC will not fetch the next instruction. Otherwise, we will set Stall\_o and NoOp\_o to zero and PCWrite\_o to one.

**I.10 MUX32.v**

MUX32.v takes two 32-bit inputs, data1\_i and data2\_i, and one 1-bit input, select\_i. It has one 32-bit output, data\_o. MUX32 chooses data which data to output based on select\_i. If the value of select\_i is zero, then data1\_i would be returned. Likewise, if select\_i is one, data2\_i would be assigned as output.

There are three occasions in which MUX32.v is used. First, MUX\_PC takes PC-four as data1\_i and PC\_branch as data2\_i and Flush as select\_i. It is used to determine pc\_i in PC.v Second, MUX\_ALUSrc takes the result of ForwardB\_MUX and the result of SignExtend.v with IDEX\_ALUSrc as select\_i to determine the value of data2\_i in ALU.v. Lastly, MUX\_MemtoReg takes MEMWB\_ALU\_Res as data1\_i and MEMWB\_MemRead\_Data as data2\_i with MEMWB\_MemtoReg as select\_i. It is used to determine WB\_WriteData\_i in module ForwardA\_MUX and ForwardB\_MUX in CPU.v and to determine RDdata\_i in Registers.v

**I.11 MUX32\_4i.v**

MUX32\_4i takes one 2-bit input, Forward\_in, and three 32-bit inputs, EXRS\_Data\_in which is RSdata from IDEX, MEM\_ALU\_Res\_in, and WB\_WriteData\_in. It has one 32-bit output (a register of the same size is also declared), MUX\_Res\_o. MUX32\_4i is capable of choosing from four different data, but we are only using it with three input data. MUX32\_4i chooses data which data to output based on Forward\_i. If the value of Forward\_i is 00, meaning there is no need to forward, then EXRS\_Data\_in would be returned. If the value of Forward\_i is 01, then WB\_WriteData\_in would be returned. If the value of Forward\_i is 10, then MEM\_ALU\_Res\_in would be returned.

There are two occasions in which MUX32\_4i.v is used, ForwardA\_MUX and ForwardB\_MUX, both are similar. ForwardA\_MUX takes IDEX\_RS1Data as EXRS\_Data\_in and ForwardA as Forward\_in. Likewise, ForwardB\_MUX takes IDEX\_RS2 Data as EXRS\_Data\_in and ForwardB as Forward\_in. Both take EXMEM\_ALU\_Res as MEM\_ALU\_Res\_in and MUX\_MemtoReg\_Res as WB\_WriteData\_in.

**I.12 Register\_IFID.v**

Register\_IFID.v takes two 32-bit inputs, instr\_i and pc\_i, and four 1-bit inputs, clk\_i, start\_i, Stall\_i and Flush\_i. It has two 32-bit output (a register of the same size is also declared), instr\_o and pc\_o. If Flush\_i has the value of one, then both pc\_o and instr\_o will have the value 32’b0 regardless of Stall\_i’s value. Otherwise, the output value depends on Stall\_i’s value. If Flush\_i has the value of zero and Stall\_i has the value of one, meaning there is a stall, then the value of pc\_o and instr\_o remains unchanged. If Flush\_i has the value of zero and Stall\_i has the value of zero, then the value of pc\_o and instr\_o is updated to pc\_i and instr\_i. The output, instr\_o, is important as it is sent to Control.v, Registers.v, Hazard\_Detection.v, Sign\_Extend.v and Register\_IDEX.v as input. Meanwhile, pc\_o is used as data2\_in in Add\_Branch.

**I.13 Register\_IDEX.v**

Register\_IDEX.v takes three 32-bit inputs, SignExtend\_Res\_i, RS1Data\_i and RS2Data\_i, one 10-bit input, funct\_i, three 5-bit inputs, RDaddr\_i, RS1Addr\_i and RS2Addr\_i, one 2-bit input, ALUOp\_i, and seven 1-bit inputs, clk\_i, start\_i, RegWrite\_i, MemtoReg\_i, MemRead\_i, MemWrite\_i and ALUSrc\_i. It has three 32-bit outputts, SignExtend\_Res\_o, RS1Data\_o and RS2Data\_o, one 10-bit output, funct\_o, three 5-bit outputs, RDaddr\_o, RS1Addr\_o and RS2Addr\_o, one 2-bit output, ALUOp\_o, and five 1-bit outputs, RegWrite\_o, MemtoReg\_o, MemRead\_o, MemWrite\_o and ALUSrc\_o. All of the outputs have a register of the same size declared and assigned to them. If start\_i has the value of one, then all of the outputs will be updated to the corresponding input value. Otherwise, the value of the outputs remains the same.

RS1Data\_o is used as EXRS\_Data\_in in ForwardA\_MUX. RS2Data\_o is used as EXRS\_Data\_i in ForwardB\_MUX. SignExtend\_Res\_o is used as data2\_i in MUX\_ALUSrc. RS1Addr\_o and RS2Addr\_o are used as EX\_RS1\_i and EX\_RS2\_i in Forwarding\_Unit. funct\_o and ALUOp\_o are used as funct\_i and ALUOp\_i in ALU\_Control.v to determine which operation ALU should perform. RDaddr\_o is used as RDaddr\_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. ALUSrc\_o is used as select\_i in MUX\_ALUSrc as the determining factor.

**I.14 Register\_EXMEM.v**

Register\_EXMEM.v takes two 32-bit inputs, ALU\_Res\_i and MemWrite\_Data\_i, one 5-bit input, RDaddr\_i, and six 1-bit inputs, clk\_i, start\_i, RegWrite\_i, MemtoReg\_i, MemRead\_i, and MemWrite\_i. It has two 32-bit outputs, ALU\_Res\_o and MemWrite\_Data\_o, one 5-bit output, RDaddr\_o, and four 1-bit outputs, RegWrite\_o, MemtoReg\_o, MemRead\_o, and MemWrite\_o. All of the outputs have a register of the same size declared and assigned to them. If start\_i has the value of one, then all of the outputs will be updated to the corresponding input value. Otherwise, the value of the outputs remains the same.

ALU\_Res\_o is used as MEM\_ALU\_Res\_i in ForwardA\_MUX and ForwardB\_MUX. MemWrite\_Data\_o is used as data\_i in Data\_Memory.v which is the data to-be written onto the memory. RDaddr\_o is used as MEM\_Rd\_i in Forwarding\_Unit.v to determine the 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 if the data in memory address addr\_i is to be read and assign to data\_o. MemWrite\_o is used as MemWrite\_i in Data\_Memory.v to determine if data\_i is tobe written to memory address addr\_i.

**I.15 Register\_MEMWB.v**

Register\_MEMWB.v takes two 32-bit inputs, ALU\_Res\_i and MemRead\_Data\_i, one 5-bit input, RDaddr\_i, and four 1-bit inputs, clk\_i, start\_i, MemtoReg\_i, and RegWrite\_i. It has two 32-bit outputs, ALU\_Res\_o and MemRead\_Data\_o, one 5-bit output, RDaddr\_o, and two 1-bit outputs, MemtoReg\_o, and RegWrite\_o. All of the outputs have a register of the same size declared and assigned to them. If start\_i has the value of one, then all of the outputs will be updated to the corresponding input value. Otherwise, the value of the outputs remains the same.

ALU\_Res\_o is used as data1\_i in MUX\_MemtoReg, while MemRead\_Data\_o is used as data2\_i. RDaddr\_o is used as RDaddr\_i in Registers.v. RegWrite\_o is used as RegWrite\_i in Registers.v to decide whether Write Data, RDdata\_i is written to register address RDaddr\_i. MemtoReg\_o is used as select\_i in MUX\_MemtoReg.

**I.16 Shift\_Left.v**

Shift\_Left.v takes one 32-bit inputs, data\_i. It has one 32-bit output, data\_o. The input is shifted left by one bit and the result is assigned to data\_o. The output is used in Add\_Branch as data1\_in to help determine the jump address in the BEQ instruction.

**I.17 Sign\_Extend.v**

Sign\_Extend.v takes one 32-bit inputs, data\_i. It has one 32-bit output (a register of the same size is also declared), data\_o. The immediate in the 32-bit instruction is signed extended by using the concatenation operator of Verilog as shown 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]; |

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

**II.1 Sign\_Extend.v**

I thought I could reuse the Sign\_Extend.v from Homework 4 and just change the register size to 32-bit. However, I saw that the output was totally off. I realize from instruction\_4 of the first testcase that PC did not jump to the right place during the loop. I changed the implementation completely to handle the input separately.

**II.2 CPU.v**

I write the code with the naming from the pipeline to make things easy to visualize, but I soon found out that it was more confusing. Because of the added five registers to save the same value, there are several wires with the same name. The control signals are especially hard to keep track off. The number of wires needed itself hard to handle. I started adding the register name in front of the wire name. For example, Ctrl\_RegWrite for the RegWrite signal from Control.v, IDEX\_RegWrite for the RegWrite that has been saved in Register\_IDEX and EXMEM\_RegWrite for the RegWrite signal that was saved in EXMEM.

**II.3 Miscellaneous**

I think there are many more problems that I encountered and have forgotten. However, I believe the hardest part of debugging is tracking the values and actually pin-pointing where is the problem located. There are too many values to keep track and it is stressful to go through them one by one.

**III. Development Environment**

III.1 Operating System: Linux (CSIE Workstation)

III.2 Compiler : iverilog