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