Computer Architecture Lab 3 Report

B09202004 物理四 程意絜

1. Modules Explanation
   a. Branch_Predictor.v

   The Branch_Predictor is a 2-bit dynamic branch predictor module that predicts whether a conditional branch will be taken or not. The module uses a 2-bit state machine with four states to predict branch behavior:

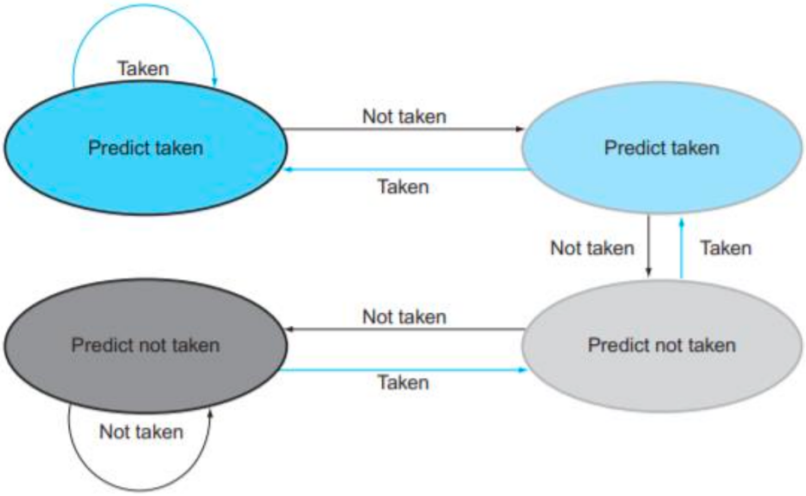
State Code	State
00	strong not taken
01	weak not taken
10	weak taken
11	strong taken

predict_o	Prediction
0	not taken
1	taken

   Initially, the state is set to 11 (strong taken), and predict_o is set to 1 (taken). The state changes based on ifTaken_i, which indicates if the branch was actually taken (0 for taken, 1 for not taken). and the current state. The logic can be illustrated with a state transition table:

Current State	ifTaken_i	Next State	predict_o (Prediction)
00	0	01	0
00	1	00	0
01	0	10	0
01	1	00	0
10	0	11	1
10	1	01	1
11	0	11	1
11	1	10	1

This logic satisfy the FSM for Branch predictor on the spec:



b. Control.v

The control module takes an opcode (opcode_i) and NoOp (NoOp_i) as inputs. It generates control signals, ALUOp, ALUSrc, Branch, MemRead, MemWrite, RegWrite, and MemtoReg for the CPU. The module uses a register control (8 bits) to store the control signals. The format of control is as following chart:

Control (8-bits)						
[7]	[6]	[5]	[4]	[3:2]	[1]	[0]
RegWrite	MemtoReg	MemRead	MemWrite	ALUOp	ALUSrc	Branch

And here's a chart summarizing the logic I use in this module:

Opcode	RegWrite	MemtoReg	MemRead	MemWrite	ALUOp	ALUSrc	Branch
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	1	00	1	0
1100011	0	X	0	0	01	0	1
NoOp / Other	0	0	0	0	11	0	0

Note that if NoOp_i is true, the module sets the control register to 8'b00001100, disabling most operations except for ALU control and ALU source selection; while if NoOp_i is false, the control signals are set based on the opcode.

c. ALU_Control.v

The ALU_Control module in Verilog is designed to generate control signals for the Arithmetic Logic Unit (ALU) based on the instruction's function codes (funct3_i and funct7_i) and the ALU operation code (ALUOp_i). Here's a chart summarizing the logic of the ALU_Control module:

ALUOp_i	funct3_i	funct7_i[5]	funct7_i[0]	ALUCtrl_o
10	111	X	X	111
	100			100
	001			001
	101			101
	000	1		010
		0	1	011
			0	000
00	101	X	X	101
	other			000
01	X			010
other				110

d. ALU.v

ALU module is responsible for performing arithmetic and logic operations. It takes two data inputs (data1_i and data2_i), an ALU control (ALUCtrl_i), and calculates the result (data_o) of the specified operation. It also determines if the result is zero (zero_o). The operations corresponding to different ALUCtrl_i is as follow:

ALUCtrl_i	Operations
000	data1_i + data2_i
010	data1_i - data2_i
011	data1_i * data2_i
111	data1_i & data2_i
100	data1_i ^ data2_i
001	data1_i << data2_i
101	data1_i >> data2_i[4:0]

After the operation's result is stored in data_reg, assign the result of data_reg to data_o. Then, check if the value of data_reg is 0. If it is, set the value of zero_o to 1; otherwise, set it to 0. If the zero_o is set to 1 and it is a branch instruction, we know that the branch instruction should actually be taken. Thus, we can decide whether we should change the state of the branch predictor.

e. IF_ID.v, ID_EX.v, EX_MEM.v, MEM_WB.v

The IF_ID, ID_EX, EX_MEM, and MEM_WB modules are components in a pipelined processor architecture, specifically designed to handle the interface between different stages.

For example, the IF_ID module handles the interface between the Instruction Fetch (IF) and Instruction Decode (ID) stages. The module captures and temporarily stores the instruction (instr_i) and its associated PC value (PC_i) fetched in the IF stage. It then passes these values to the ID stage for decoding. The other three modules also perform similar functions.

Different from Lab 2, we should additionally transfer PC_next through IF/ID and Predict_Branch, PC_next, and PC_Branch through ID/EX in Lab 3. Note that we should send a flush signal to IF/ID and ID/EX when we have wrong prediction.

f. MUX32.v

The MUX32 acts as a 32-bit multiplexer. It selects one of the two input data values (data1_i and data2_i) based on the input select signal (select_i). The selected data is output as data_o. If select_i is 0, the value of data_o is assigned as data1_i. If select_i has a value of 1, the value of data_o is assigned as data2_i.

g. Adder.v

The Adder module performs a simple addition operation. It takes two input data values (data1_i and data2_i) and produces the sum as the output (data_o).

h. Imm_Gen.v

The Imm_Gen module is designed to generate immediate values from. It takes a 32-bit instruction as input (instr_i) and outputs a 32-bit immediate value (imm_o). Based on the opcode (instr_i[6:0]) and funct3 (instr_i[14:12]), the module uses a combination of the instruction's bits to form the immediate value. This process varies depending on the type of instruction, so I use the following rule to distinguish them:

<table>
<tr><th>function</th><th>opcode</th><th>funct3</th><th>imm_o</th></tr>
<tr><td>and</td><td rowspan="6">0110011</td><td rowspan="6">X</td><td rowspan="6">32'b0</td></tr>
<tr><td>xor</td></tr>
<tr><td>sll</td></tr>
<tr><td>add</td></tr>
<tr><td>sub</td></tr>
<tr><td>mul</td></tr>
<tr><td>addi</td><td rowspan="2">0010011</td><td>000</td><td>20*instr_i[31] + instr_i[31:20]</td></tr>
<tr><td>srai</td><td>101</td><td>27*instr_i[24] + instr_i[24:20]</td></tr>
<tr><td>lw</td><td>0000011</td><td>X</td><td>20*instr_i[31] + instr_i[31:20]</td></tr>
<tr><td>sw</td><td>0100011</td><td>X</td><td>20*instr_i[31] + instr_i[31:25] + instr_i[11:7]</td></tr>
<tr><td>beq</td><td>1100011</td><td>X</td><td>21*instr_i[31] + instr_i[7] + instr_i[30:25] + instr_i[11:8]</td></tr>
</table>

i. Hazard_Detection.v

The Hazard_Detection module is designed to detect and manage data hazards in a pipelined processor. Data hazards occur when instructions that are close together in program order depend on each other's data. This module helps to ensure correct program execution by detecting these situations and controlling the processor's behavior.

The module checks if the current instruction in the Execute stage is reading from memory (EX_MemRead_i is high) and if its destination register (EX_RDaddr_i) matches either of the source registers (ID_RS1addr_i or ID_RS2addr_i) of the instruction in the Decode stage. If such a condition is detected, it implies a data hazard. To handle this, the module set Stall_o to 1 to stall the pipeline. It also prevents updating the Program Counter (set PCWrite_o to 0) to pause fetching new instructions, and activates NoOp_o to indicate that no operation should be processed in the subsequent pipeline stage.

The logic I use is as the following table:

Condition	PCWrite_o	Stall_o	NoOp_o
Hazard Detected	0	1	1
No Hazard Detected	1	0	0

j. Forwarding_Unit.v

The Forwarding_Unit module in Verilog is designed to manage data hazards in a pipelined processor architecture, specifically forwarding hazards. Forwarding (or data bypassing) is a technique used to deal with situations where an instruction depends on the result of a previous instruction that has not yet written its result back to a register. This module determines if and from where operand data should be forwarded to the current instruction in the Execute (EX) stage of the pipeline.

The logic I use for this module follows the suggestion on the spec. The module checks if the operands of the current EX stage instruction (EX_RS1addr_i and EX_RS2addr_i) are being written to by the previous instructions in the pipeline.
If the MEM stage instruction is writing to a register (MEM_RegWrite_i is high) and the destination register of MEM stage matches the source register of EX stage (EX_RS1addr_i or EX_RS2addr_i), then the output (select_A_o or select_B_o) is set to indicate that the data should be forwarded from the MEM stage.

If the above condition is not met but the WB stage instruction is writing to a register (WB_RegWrite_i is high) and its destination register matches the source register of EX stage, then the output is set to indicate that the data should be forwarded from the WB stage.If neither condition is met, no forwarding is needed.

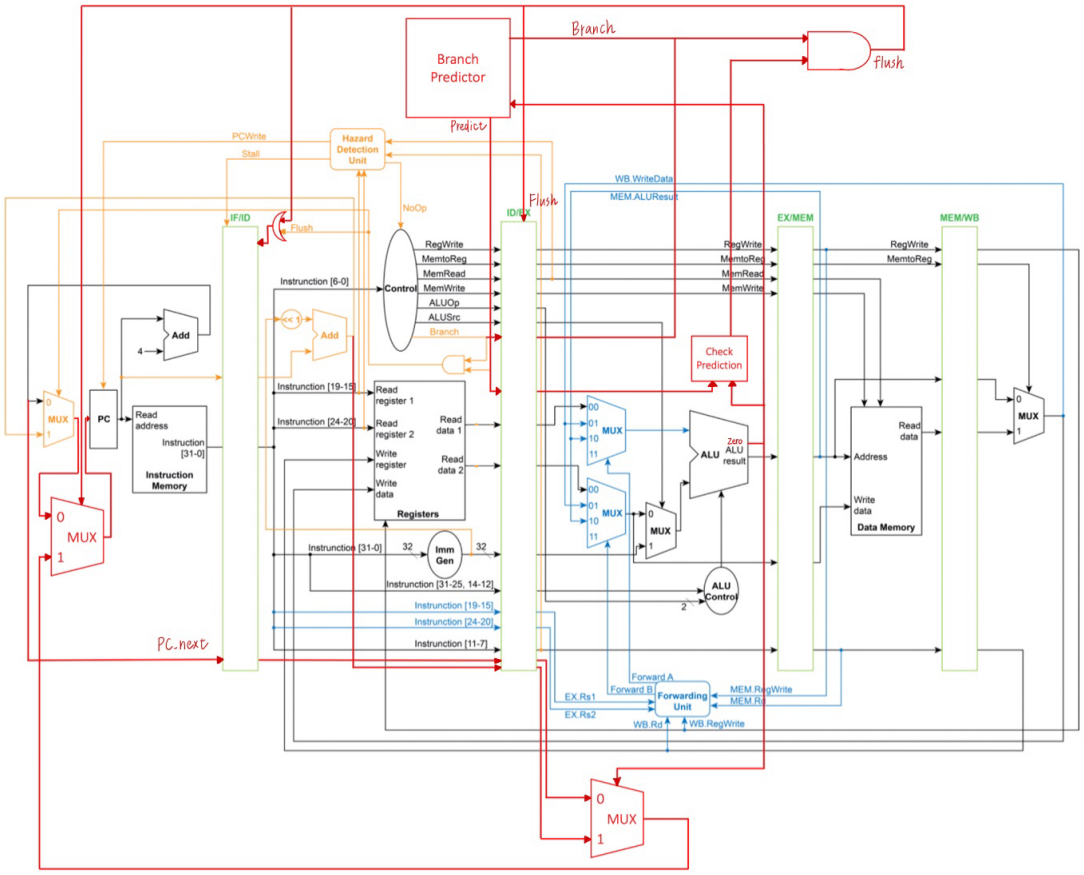
Here’s a chart summarizing the logic of the Forwarding_Unit module:

Condition	select_A_o	select_B_o
MEM stage writes to EX stage RS1	10	X
WB stage writes to EX stage RS1 (and not MEM)	01	

No forwarding needed for RS1	00	
MEM stage writes to EX stage RS2	X	10
WB stage writes to EX stage RS2 (and not MEM)		01
No forwarding needed for RS2		00

k. CPU.v

The top-level module, CPU, integrates all the components. It connects and controls the flow of data and instructions in the CPU. The module takes clock (clk_i) and reset (rst_i) signals as inputs. I connect the inputs and outputs of various modules according to the following datapath.



2. Difficulties Encountered and Solutions in This Lab
    a. Initially, I didn't realize that after a branch is taken, it needs to be passed on to the next stage. This oversight led to a situation where, if the branch prediction was incorrect, the original Program Counter (PC) location couldn't be retrieved.

       Later, I adjusted the design to pass both the original PC location and the PC location of the branch to subsequent stages, continuing up to the Execution (EX) stage. This way, if a prediction error is discovered, it's possible to recover.

3. Development Environment

   OS: MacOS
   Compiler: iverilog