EE312 Lab #4 Report

20180610 정의준

1. Introduction

In Lab 4, we design a multi-cycle CPU. By maintaining the same microarchitecture design that we have made in Lab 3, we modify it to run instructions in multiple cycles.

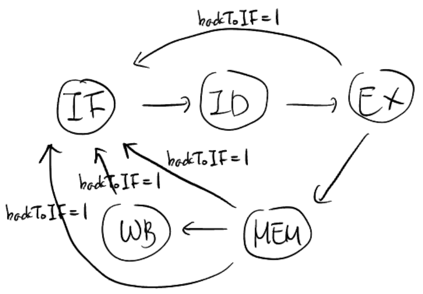
For the data path, I used multiple pipeline registers, or D-flip flops that passes the data at every rising edge. The registers prevent the data to flow through the whole uarch at once, and makes sure that the data stops at each stage. Then, we can make sure that the data goes to the next stage at every rising edge.

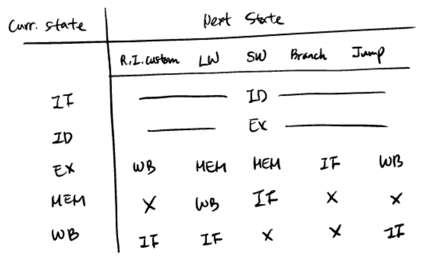
For the control path, I attached the clock to the control module, and let it switch its state at every rising edge. I used the Mealy machine to implement the FSM.

Since we are not doing pipelining yet, I didn't put any pipeline registers for the control signals. But, for the next lab, we should put the pipeline registers for the control signal as well, and let the control signal flow along the data signal.

2. Design

- FSM for the control module



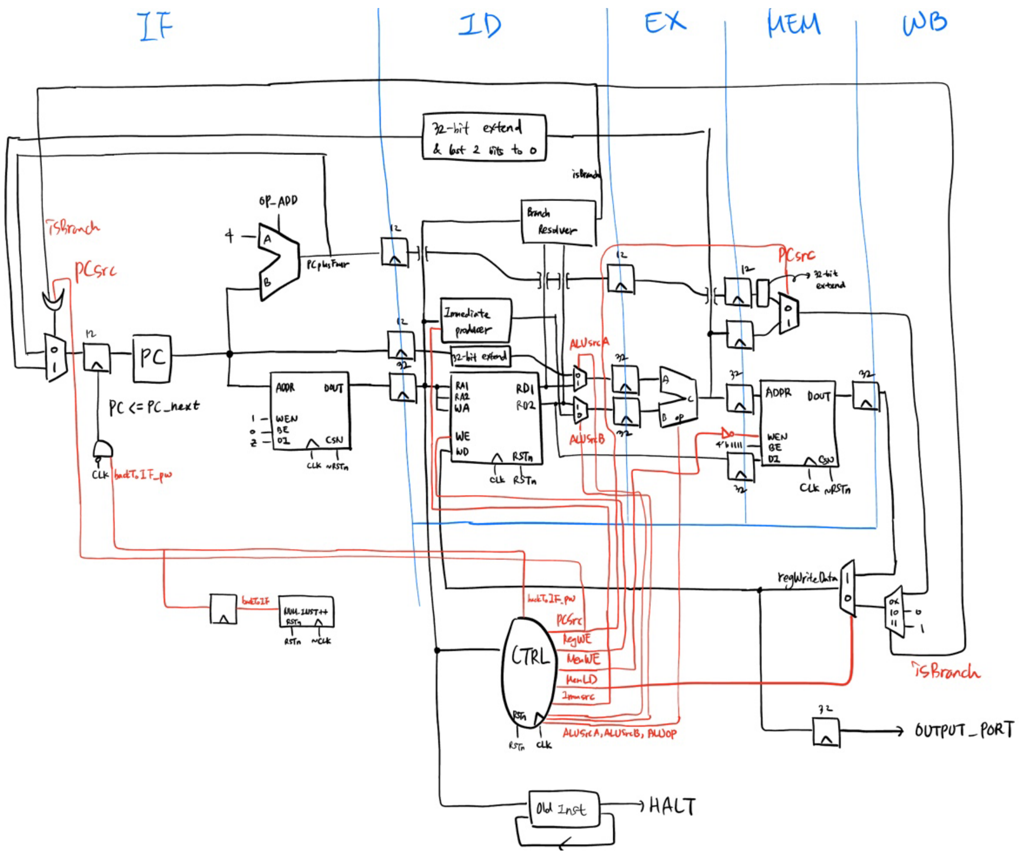


Curr. state	Next State				
	R,I,custom	LW	SW	Branch	Jump
IF	———	———	ID	———	———
ID	———	———	EX	———	———
EX	WB	MEM	MEM	IF	WB
MEM	X	WB	IF	X	X
WB	IF	IF	X	X	IF

First, we should design an FSM for the control module. This is because each instructions have different sequence of stages, and at each stage we should produce different control signals. For instance, MemWE and RegWE should be enabled only at the MEM and WB stage respectively. The diagram and the table for the FSM is shown as above. Based on this table, we create a Mealy machine in code and assign proper control signals for each state. In the state diagram, I have not stated how the control signals should be because there are too many of them. Check RISCV_CTRL.v for every control signal for each state in the state diagram.

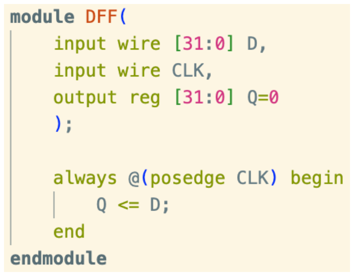
Also, since we should increment NUM_INST only when the instruction is finished, I created a new control signal called backToIF, and this becomes 1 only when the state returns "back to IF stage." Therefore, we will let a new instruction to come (=update PC_next) in only when this value is 1. I also used a negedge clock at the PC_next pipeline register to prevent data hazards.

- Overall uArch with pipeline registers

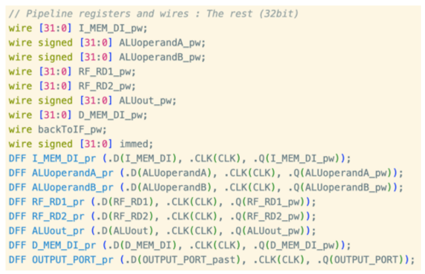


The overall design of the circuit is shown as above. Several pipeline registers are placed in the middle of the datapath's stages. Also, the clock and the RSTn signal is attached to the control module. Compared to my former single-cycle uArch, I changed the part for resolving the branch. I made a new module, RISCV_BCOND.v, which checks whether the branch is taken or not. Also, I re-drawed the diagram for the WB stage. There is a pipeline register added at the OUTPUT_PORT as well. It is added because the TB file checks the output after the NUM_INST is incremented, when the WB stage is over. Finally, I removed the parts for the byte-enable flags, because we only use LW and SW for Lab 4.

3. Implementation

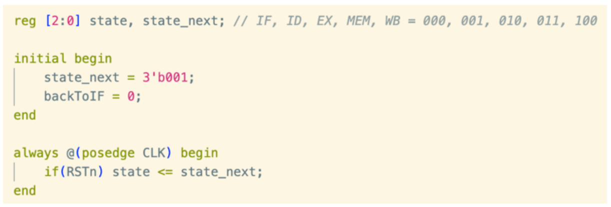


```
module DFF(
    input wire [31:0] D,
    input wire CLK,
    output reg [31:0] Q=0
    );

    always @(posedge CLK) begin
        Q <= D;
    end
endmodule
```



```
// Pipeline registers and wires : The rest (32bit)
wire [31:0] I_MEM_DI_pw;
wire signed [31:0] ALUoperandA_pw;
wire signed [31:0] ALUoperandB_pw;
wire [31:0] RF_RD1_pw;
wire [31:0] RF_RD2_pw;
wire signed [31:0] ALUout_pw;
wire [31:0] D_MEM_DI_pw;
wire backToIF_pw;
wire signed [31:0] immed;
DFF I_MEM_DI_pr (.D(I_MEM_DI), .CLK(CLK), .Q(I_MEM_DI_pw));
DFF ALUoperandA_pr (.D(ALUoperandA), .CLK(CLK), .Q(ALUoperandA_pw));
DFF ALUoperandB_pr (.D(ALUoperandB), .CLK(CLK), .Q(ALUoperandB_pw));
DFF RF_RD1_pr (.D(RF_RD1), .CLK(CLK), .Q(RF_RD1_pw));
DFF RF_RD2_pr (.D(RF_RD2), .CLK(CLK), .Q(RF_RD2_pw));
DFF ALUout_pr (.D(ALUout), .CLK(CLK), .Q(ALUout_pw));
DFF D_MEM_DI_pr (.D(D_MEM_DI), .CLK(CLK), .Q(D_MEM_DI_pw));
DFF OUTPUT_PORT_pr (.D(OUTPUT_PORT_past), .CLK(CLK), .Q(OUTPUT_PORT));
```

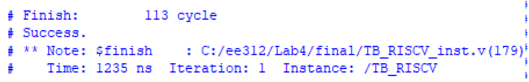
For the data path, I created a D-flipflop module and used it to implement the pipeline registers in the datapath. By creating a new module, I was able to not to write all the pipeline wires and registers on the always @(posedge CLK) statement. I also changed the code for negedge part, where we should increment NUM_INST only when the state of the control module is going back to the IF state.



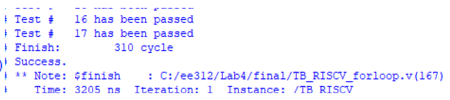
```
reg [2:0] state, state_next; // IF, ID, EX, MEM, WB = 000, 001, 010, 011, 100

initial begin
    state_next = 3'b001;
    backToIF = 0;
end

always @(posedge CLK) begin
    if(RSTn) state <= state_next;
end
```

For the control path, the above code shows the Mealy machine implementation. In always@(*) part, I used nested case() statements. First case is determined by the opcode, and the second one is for determining FSM's state. Based on the state and the opcode, control signals such as backToIF, RegWE, MemWE, MemLD are determined. For instance, backToIF and MemLD is 1 only when the next anticipated state is IF. Also, RegWE is 1 only when the state is WB. Other control signals, which do not vary respect to the state, is only assigned with a single case() statement.

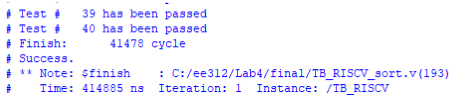
4. Evaluation



```
# Finish:        113 cycle
# Success.
# ** Note: $finish    : C:/ee312/Lab4/final/TB_RISCV_inst.v(179)
#    Time: 1235 ns  Iteration: 1  Instance: /TB_RISCV
```



```
# Test #   16 has been passed
# Test #   17 has been passed
# Finish:        310 cycle
# Success.
# ** Note: $finish    : C:/ee312/Lab4/final/TB_RISCV_forloop.v(167)
#    Time: 3205 ns  Iteration: 1  Instance: /TB RISCV
```



```
# Test #   39 has been passed
# Test #   40 has been passed
# Finish:      41478 cycle
# Success.
# ** Note: $finish    : C:/ee312/Lab4/final/TB_RISCV_sort.v(193)
#    Time: 414885 ns  Iteration: 1  Instance: /TB_RISCV
```

My uArch has passed all 3/3 testbench files. Also, I was able to obtain 113, 310, and 41478 cycles for inst, forloop, and sort TB respectively. I included the waveforms with the report in the zip file.

5. Discussion

The toughest part of the Lab was indeed debugging. While making the FSM for the control module, I accidentally wrote a bit of a control signal at one state wrong. Then, the whole code did not work well as I had expected and I was not able to pass through the testbenches. I tried to see all the bits in the uArch to see which one's wrong, and I was barely able to find that the problem was at the control module.

The biggest challenge was that while my code running on iVerilog of macOS did not have any problem, the code failed the testbench in Windows with Modelsim. I was able to find out that the PC_next pipeline register was the problem and changed its clock to negedge. I was very nervous because I had to fix the problem in 4 hours before the submission due.

Also, I wish I had understood how the microprogram-based control unit works, which is shown in the lecture material. I wish there were more explanations about what they are, and how we should implement them.

6. Conclusion

By using multiple pipeline registers, I was able to implement multi-cycle CPU while maintaining the overall structure of my single-cycle CPU. Also, I made small parts of the logic circuit such as branch resolver or immediate producer into separate module, and thus the code has become simpler and more organized.