# EE312 Lab #3 Report

20180610 정의준

## 1. Introduction

In Lab 3, we implement a single-cycle CPU. We have to implement a module that can decode and run all the given instructions. We first have to make a decoder that can handle various types of instructions. Then, we design the data path and connect it with the register controller and the memory controllers. We also design the control path that manipulates the circuit and the modules with respect to the given instruction. By using the three testbench files, we can check whether the CPU performs as we have desired.

## 2. Design

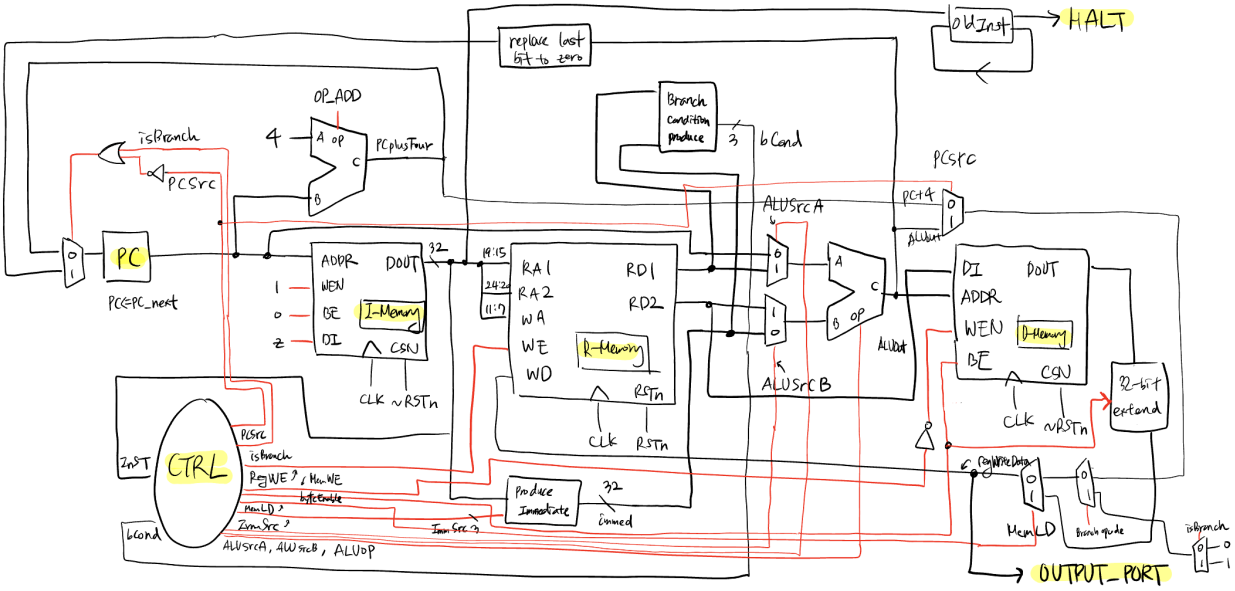



**Figure 1 Overall design of single-cycle CPU**

The design of the CPU is given as the above. The data path is denoted in black, and the control path is drawn in red. The data flows through I-Memory, R-Memory, ALU, D-Memory, back to the R-Memory. Control signals are produced by the instruction and the branch-condition (bCond). The control signals control every 2-to-1 muxes in the circuit and let the CPU to make the desired output for every instruction. For instance, ALUSrcA controls which data to use as a source of the operand of an ALU at the center. "Produce Immediate" box at the bottom decodes the 32-bit instruction and recovers the immediate value into a 32-bit number.

## 3. Implementation

For the control path, the followings are the brief explanations about each control signals. PCSrc is 1 for almost all cases, except for JAL and JALR instructions, which means "the instruction is not JUMP". This control

signal is used for determining the next PC. Also, it determines which value to store to WD because JUMP instructions store PC+4 in the destination register.

isBranch is 1 when the branch condition is satisfied. Thus, PC becomes the ALU output when PCSrc or isBranch is 1, else PC+4.

RegWE, MemWE, and byteEnable directly controls R-Memory and D-Memory. WE stands for "Write enable." byteEnable is for controlling the number of bits to read or write. Depending on the instruction, it becomes 0001 (SB, LB), 0011 (SH, LH), or 1111(SW, LW).

MemLD determines whether the value to WD should come from the ALU or the D-memory output. If it is 1, it means the memory should be "loaded" and puts the D-memory output into WD wire.

ImmSrc is a 3-bit length signal that determines which bits in the instruction to use as an immediate. This is important since the position of the immediate is different for various instructions. The code goes as below.

```
case (Immsrc)
    3'b000: immed = {{20{I_MEM_DI[31]}}, I_MEM_DI[31:20]}; // I-type, Load, JALR
    3'b001: immed = {{20{I_MEM_DI[31]}}, I_MEM_DI[31:25], I_MEM_DI[11:7]}; // Store
    3'b010: immed = {{19{I_MEM_DI[31]}}, I_MEM_DI[31], I_MEM_DI[7], I_MEM_DI[30:25], I_MEM_DI[11:8], 1'b0}; // Branch
    3'b011: immed = {{12{I_MEM_DI[31]}}, I_MEM_DI[31], I_MEM_DI[19:12], I_MEM_DI[20], I_MEM_DI[30:21], 1'b0}; // JAL
    3'b100: immed = {I_MEM_DI[31:12], {12{1'b0}}}; // LUI, AUIPC
    default: immed = {{20{I_MEM_DI[31]}}, I_MEM_DI[31:20]}; // Default..
endcase
```

ALUsrcA, ALUsrcB, and ALUOP controls the ALU at the center. ALUsrcA and ALUsrcB controls the operand of the ALU as the code below. ALUOP controls the operation of the ALU. I used the ALU from the Lab 1 and modified and added some operations.

```
ALUoperandA = (ALUsrcA) ? RF_RD1 : PC;
ALUoperandB = (ALUsrcB) ? RF_RD2 : immed;
```

Data path of this CPU is quite obvious; there is no huge difference between the MIPS CPU from the lecture note. I will skip the explanations for this. The CPU uses two ALUs for calculating all general operations and the PC+4.

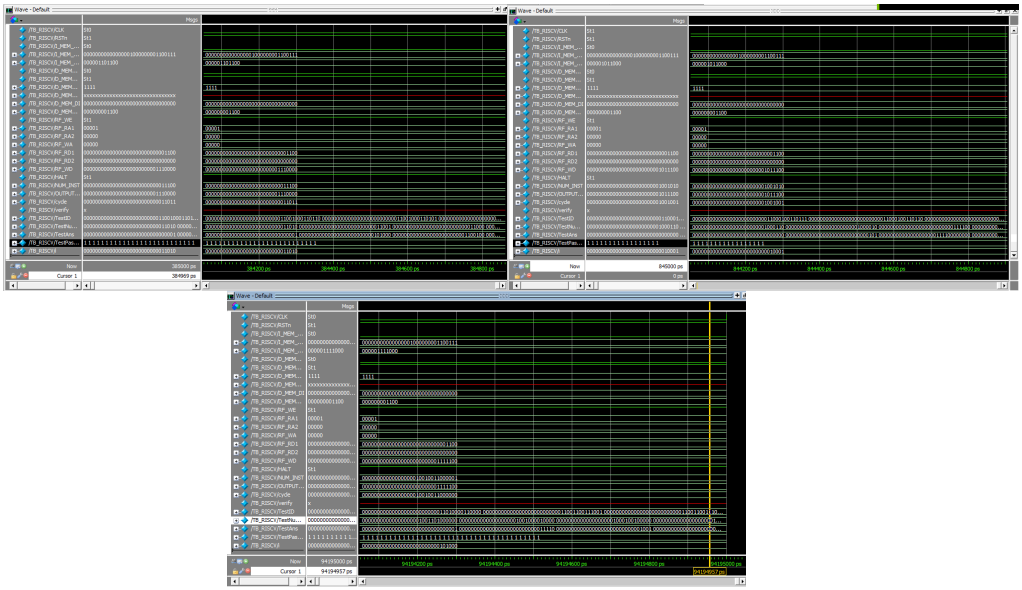
Separate module is used to calculate branch condition (bCond) which is a 3-bit flag. First bit represents whether the two operands are equal or not, second bit represents whether A is bigger or B is bigger in signed number, and the third bit represents the same thing in unsigned number. This bCond is put into the control signal and produces the isBranch signal.

4. Evaluation

```
# Test #   26 has been passed
# Finish:         27 cycle
# Success.
# ** Note: $finish    : C:/Users/modelsim/Desktop/final/TB_RISCV_inst.v(179)
#    Time: 385 ns  Iteration: 1  Instance: /TB_RISCV
```

```
# Test #   17 has been passed
# Finish:         73 cycle
# Success.
# ** Note: $finish    : C:/Users/modelsim/Desktop/final/TB_RISCV_forloop.v(167)
#    Time: 845 ns  Iteration: 1  Instance: /TB_RISCV
```

```
# Test #   40 has been passed
# Finish:       9408 cycle
# Success.
# ** Note: $finish    : C:/Users/modelsim/Desktop/final/TB_RISCV_sort.v(193)
#    Time: 94195 ns  Iteration: 1  Instance: /TB_RISCV
```



By running the testbench simulations, I was able to obtain the above simulation results and waves. My CPU has passed all 3/3 tests in the testbench folder.

However, I was able to find some parts of my CPU that have some rooms for the improvement. I will talk about this in the discussion section.

## 5. Discussion

Because the single-cycle CPU does not require complicated and carefully-designed structure, my CPU was able to succeed all the testbench tests. However, to implement multi-cycle CPU with this circuit, some points of the circuits should be changed to improve the visibility and the data hazards.

First, the module that checks branch condition should put its output directly to the muxes, not to the controller. This is to reduce the penalty for the wrong branch prediction. I will fix the module for the next Labs. Also, "Branch Opcode" and WD part at the bottom right should be fixed to some other control signal to reduce the number of bits carried through the pipeline. I think there will be more problems to my CPU design when I naively upgrade it into multi-cycle CPU, so such future problems will be fixed in the later Labs.

Most difficult part was the debugging part for TB_inst.v, since there were no assembly language translation to the hex instructions. Therefore, I had to translate all of them by hand. By running the testbech file several times, I was able to fix the wrong parts.

## 6. Conclusion

Designing a single circuit that can work with several instruction types was quite challenging but interesting. I always had to think about the compatibility and the simplicity when I gradually added the compatibility for the new instructions. Also, I had to think about how my CPU should be to implement pipelining in the future, and tried to keep the circuit and the signals as simple and intuitive as I could. Still, my CPU's design has many things to improve for the next Labs.