# LAB4: Multi Cycle CPU

**EE 312 Computer Architecture**

**Professor:** Minsso Rhu

20150912 Maro Han

20150146 Sanghyun Kim

**Introduction:**

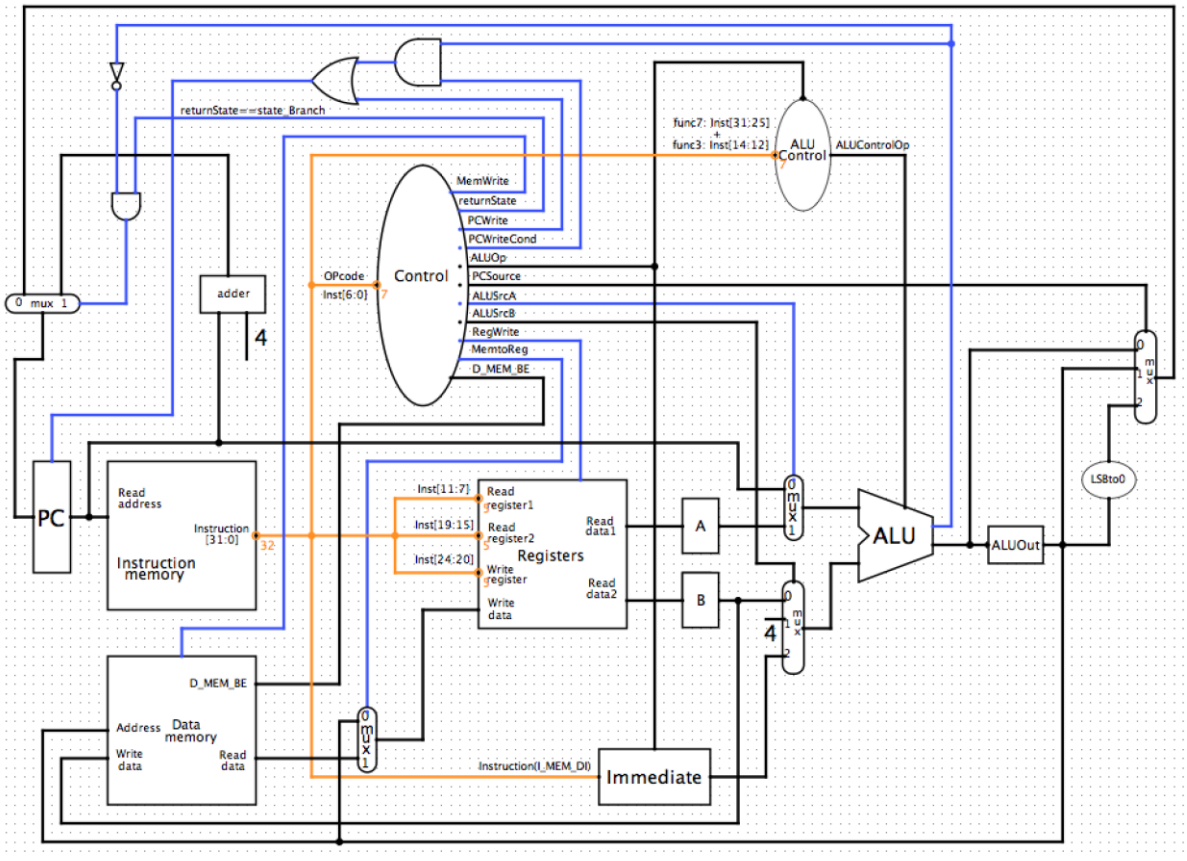
In our fourth lab, we were supposed to implement the multi cycle CPU for the RISC-V ISA. We were asked to implement the instructions:

JAL, JALR, BEQ, BNE, BLT, BGE, BLTU, BGEU, LW, SW, ADDI, SLTI, SLTIU, XORI, ORI, ANDI, SLLI, SRLI, SRAI, ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR, AND

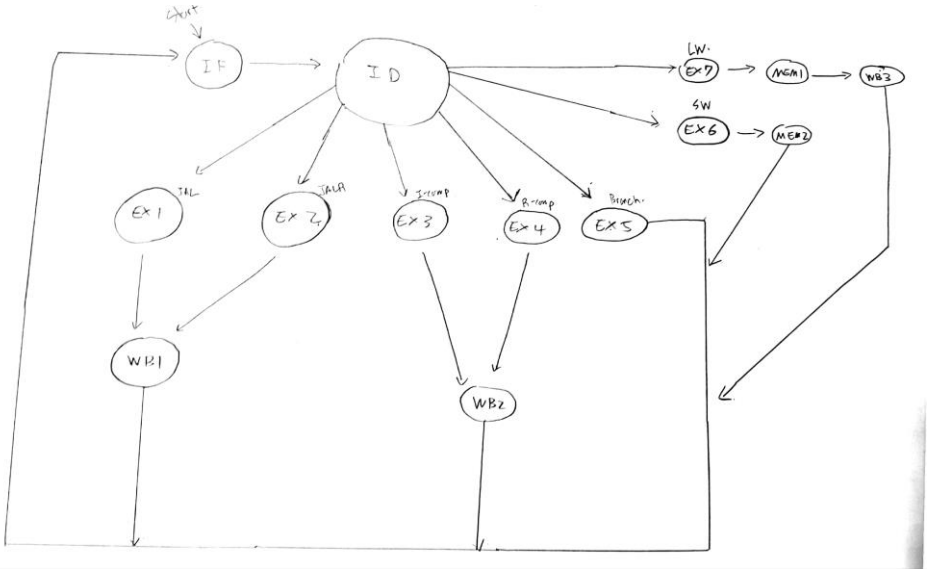
I used the data path and control introduced to us in our lecture slides and appendix D of our textbook. However, there were changes made due to the difference in instructions such as JAL, JALR and branch instructions.

**Design:**

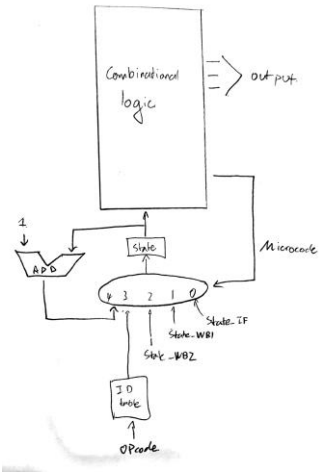
The data path of our machine is drawn below. I will elaborate on the specifics of this extensively in the implementation section of the report.



We implemented the control path of our design first by drawing the FSM of our CPU as seen below:




Then we proceeded to transform control into a table that outputs a value by referencing the current state and microcode. A drawn diagram of the control logic and table for ID table is shown in the figures below:




| ID table | Opcode | Output |
|---|---|---|
| | 1100111 | JALR(0011) |
| | 1100011 | Branch(0100) |
| | 0000011 | Load(0101) |
| | 0100011 | Store(1000) |
| | 0010011 | I-comp(1010) |
| | 0110011 | R-comp(1011) |
| | 1101111 | JAL(0010) |

The combinational logic block follows the following table:

| | PCSource | ALUOp | ALUSrcA | ALUSrcB | RegWrite | PCWritecond | PCWrite | D_MEM_BE | MemWrite | MemtoReg | Microcontrol |
|---|---|---|---|---|---|---|---|---|---|---|---|
| IF(0000) | xx | xxx | x | xx | 0 | 0 | 0 | 0000 | 0 | x | 100 |
| ID(0001) | xx | 100 | 0 or 1 | 10 | 0 | 0 | 0 | 0000 | 0 | x | 011 |
| JAL(0010) | 01 | 100 | 0 | 01 | 0 | 0 | 1 | 0000 | 0 | x | 001 |
| JALR(0100) | 10 | 100 | 0 | 01 | 0 | 0 | 1 | 0000 | 0 | x | 001 |
| Branch(0110) | 01 | 011 | 1 | 00 | 0 | 1 | 1 | 0000 | 0 | x | 000 |
| LW(0111) | xx | 110 | 1 | 10 | 0 | 0 | 0 | 0000 | 0 | x | 100 |
| SW(1010) | xx | 010 | 1 | 10 | 0 | 0 | 0 | 0000 | 0 | x | 100 |
| I-comp(1100) | xx | 001 | 1 | 10 | 0 | 0 | 0 | 0000 | 0 | x | 010 |
| R-comp(1101) | xx | 000 | 1 | 00 | 0 | 0 | 0 | 0000 | 0 | x | 010 |
| Mem1(1000) | xx | 100 | 0 | 01 | 0 | 0 | 0 | 1111 | 1 | x | 100 |
| Mem2(1011) | 00 | 100 | 0 | 01 | 0 | 0 | 1 | 1111 | 0 | x | 000 |
| WB1(0011) | xx | xxx | x | xx | 1 | 0 | 0 | 0000 | 0 | 0 | 000 |
| WB2(0100) | 00 | 100 | 0 | 01 | 1 | 0 | 1 | 0000 | 0 | 0 | 000 |
| WB3(1001) | 01 | xxx | x | xx | 1 | 0 | 1 | 1111 | 1 | 1 | 000 |

*In the ID state, ALUSrcA is 1 if the OPcode indicates that the instruction is JALR and is 0 if not (i.e. ALUSrcA = (OPcode == JALR) ? 1:0).

**Implementation:**

IF stage:

In the IF stage, PC address is put in the I_MEM unit to fetch the instruction. Because instruction unit has synchronous read and write, it acts as if it has an instruction register. This is why unlike the figure in the lecture note, we do not have a separate instruction register. PC is also incremented by 4 and is input as the second option for the PCWritecond multiplexer for branch instructions.

ID stage:

The output instruction is fed into the register unit and the immediate unit. The 7 lowest bits of the instruction is fed into the control unit and bits corresponding to the func3 and func7 is fed into the ALU control unit. Control utilizes the OPcode in the ID state to determine the next state. The immediate unit produces different types immediates by reading the instruction and OPcode. The ALUOp unit decides what computations the ALU will do by reading the func3 and func7 of the instruction and the ALUOp signal coming from control. In the ID stage we compute PC + immediate (jump address) and stores it in ALUout so jump and branch instructions can save one cycle by being able to calculate other values in later stages.

EX stage:

In the EX stage we carry out appropriate ALU operation depending on the instruction. ALUout register allows us to store ALU instructions and use the timing difference and prevent the use of another ALU unit in our implementation of the multi cycle CPU.

In case of branch instructions, bcond is 1 if the condition is met. Then the previously calculated jump address is latched onto the PC. If not, the PC is incremented to PC+4.

MEM stage:

This stage, we access the D_MEM. In case of the load instruction, we read the data in the address specified by ALUout. Since D_MEM also has synchronous read and write, we didn't include a separate D_MEM_DOUT register. In case we are executing the store instruction, we store the value of register_out 2 into the address specified by ALUout.

WB stage:

In this stage we write into the register file. For I and R type computations we are executing a computation, we store the result of ALUout into the register block. For jump instructions we store PC+4 into the register block. For load instructions, we store the loaded value from the MEM stage into the register block. The write address is specified by bits 20~24 of the instruction.

Overall:

Overall, jump instructions take 4 cycles (IF, ID, EX, WB) since it has to write PC+4 back to the register. Branch instructions take 3 cycles. I-type computations (ex: ADDI) and R-type

computations (ex: ADD) take 4 cycles (IF, ID, EX, WB) and store instructions take 4 cycles (IF, ID, EX, MEM). Load instruction takes 5 cycles (IF, ID, EX, MEM, WB).

Extra modules we developed in order to implement the datapath include ALU.v, ALU_Control.v, Control.v, Immediate.v, Multiplexer.v, Multiplexer3to1.

**Evaluation:**

I am quite satisfied with the result. Our CPU runs the instructions in a reasonable timeframe. However, I think the ALUcontrol logic could be improved so the size of ALUControlOp signal can be reduced.

We pass all the tests in all three testbenches. In TB_inst the number of clock cycles is 90, but I believe this is due to the fact that 6 cycles are spent while our logic is determining the terminal condition. I tested the cycle number by writing $display(“cycle: %d”,cycle) after

```
TestPassed[i] <= 1'b1;

$display("Test #%s has been passed", TestID[i]);
```

In the TB_RISCV_inst code and was able to confirm that right after test 21, 84 cycle has been passed. Hence, we believe our implementation utilizes the correct number of clock cycles (at least for TB_RISCV_inst).

**Discussion:**

Overall, I think this lab was well designed and allowed me to acquire in-depth knowledge on the multi cycle CPU. There were no particular struggle that we experienced in this lab, and we felt that the assignment PDF aptly explained all the necessary details for the assignment.

**Concluding:**

For me personally, I felt like I was putting together a jigsaw puzzle doing the assignment (although the difficulty was more like putting together a puzzle blindfolded). From the initial process of carefully observing the overall picture, to putting together and connecting pieces of my CPU. When all the pieces came together, it rewarded me enough to forget about all the painful times I had to spend debugging.

This rewarding feeling is why I’m looking forward to the next assignment.