LAB3: Single Cycle CPU

EE 312 Computer Architecture

Professor: Minsso Rhu

20150912 Maro Han

20150146 Sanghyun Kim

Introduction:

In our third lab, we were supposed to implement the single cycle CPU for the RISCV ISA. Our module should run the following instructions:

a. LUI, AUIPC

b. JAL

c. JALR

d. BEQ, BNE, BLT, BGE, BLTU, BGEU

e. LB, LH, LW, LBU, LHU,

f. SB, SH, SW

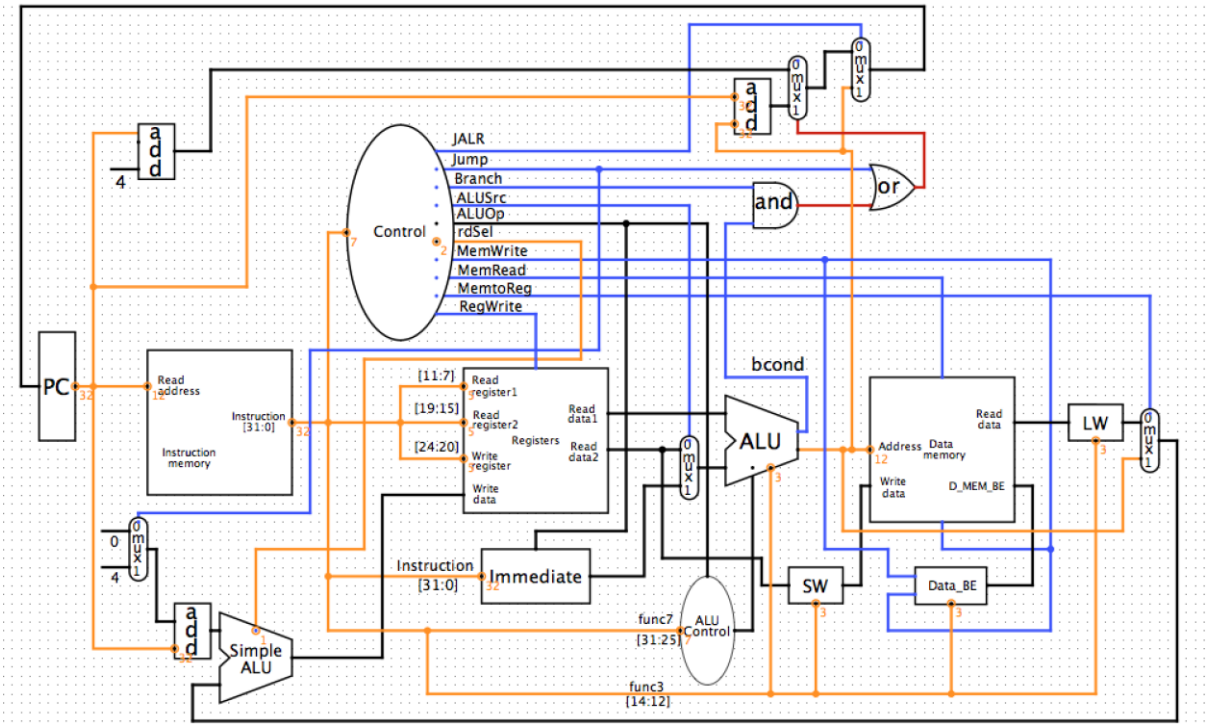
g. ADDI, SLTI, SLTIU, XORI, ORI, ANDI, SLLI, SRLI, SRAI

h. ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR, AND

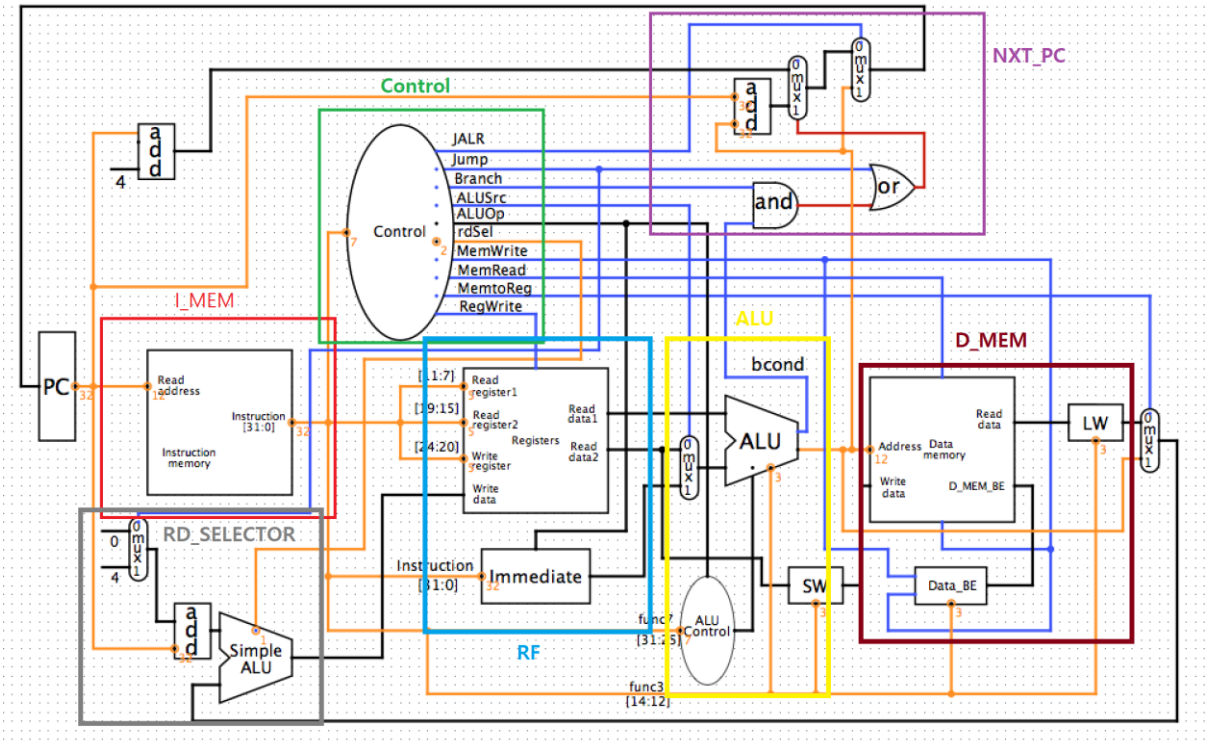
Our design was made referencing the MIPS ISA architecture in our textbook. Adjusting necessary elements to fit the RISCV architecture. As you will see, our core module we were supposed to implement gets the instruction from I_MEM and sends it either to update the PC, update the register, or update D_MEM. I will explain this further in the next few sections.

Our report is four pages long, because we included figures. Please understand.

Design:



The above diagram explains our module just like how our textbook explains the MIPS architecture. I will explain our design by dividing sections as seen below.



Implementation:

The control module of this CPU has a table corresponding to the Opcode of each instruction. The control module receives the Opcode and outputs corresponding control signals. The table below illustrates what signals are transmitted where. Most of these signals are for multiplexers. ALUOp is a control signal sent to ALU, ALU_control and immediate block to control the flow of the ALU.

	Jump	Branch	MemRead	MemtoReg	ALUOp	MemWrite	ALUSrc	RegWrite	JALR	rdSel
LUI	0	0	0	0	100	0	1	1	0	01
AUIPC	0	0	0	0	100	0	1	1	0	11
JAL	1	0	0	0	101	0	1	1	0	10
JALR	1	0	0	0	111	0	1	1	1	10
BEQ	0	1	0	1	011	0	0	0	0	01
BNE	0	1	0	1	011	0	0	0	0	01
BLT	0	1	0	1	011	0	0	0	0	01
BGE	0	1	0	1	011	0	0	0	0	01
BLTU	0	1	0	1	011	0	0	0	0	01
BGEU	0	1	0	1	011	0	0	0	0	01
LB	0	0	1	0	110	0	1	1	0	01
LH	0	0	1	0	110	0	1	1	0	01
LW	0	0	1	0	110	0	1	1	0	01
LBU	0	0	1	0	110	0	1	1	0	01
LHU	0	0	1	0	110	0	1	1	0	01
SB	0	0	0	1	010	1	1	0	0	01
SB	0	0	0	1	010	1	1	0	0	01
SB	0	0	0	1	010	1	1	0	0	01
ADDI	0	0	0	1	001	0	1	1	0	01

	Jump	Branch	MemRead	MemtoReg	ALUOp	MemWrite	ALUSrc	RegWrite	JALR	rdSel
SLTI	0	0	0	1	001	0	1	1	0	01
SLTIU	0	0	0	1	001	0	1	1	0	01
XORI	0	0	0	1	001	0	1	1	0	01
ORI	0	0	0	1	001	0	1	1	0	01
ANDI	0	0	0	1	001	0	1	1	0	01
SLLI	0	0	0	1	001	0	1	1	0	01
SRLI	0	0	0	1	001	0	1	1	0	01
SRAI	0	0	0	1	001	0	1	1	0	01
ADD	0	0	0	1	000	0	0	1	0	01
SUB	0	0	0	1	000	0	0	1	0	01
SLLI	0	0	0	1	000	0	0	1	0	01
SLT	0	0	0	1	000	0	0	1	0	01
SLTU	0	0	0	1	000	0	0	1	0	01
XOR	0	0	0	1	000	0	0	1	0	01
SRL	0	0	0	1	000	0	0	1	0	01
SRA	0	0	0	1	000	0	0	1	0	01
OR	0	0	0	1	000	0	0	1	0	01
AND	0	0	0	1	000	0	0	1	0	01

The I_MEM section is devoted to getting the instruction. We insert the 12 least significant bits of NXT PC value into I_MEM (by using I_MEM_ADDR).

The RF section is concerned about the register data block. We insert parts of the instruction we got to the RF block to specify the address of register 1, register 2 and register destination. The RD value is also input from a module we made named simple_alu, and I will explain this section when talking about the RD section. Data corresponding to the address of rs1 and rs2 is output through the right side of the R_MEM block in the diagram above. The register block is able to be written on only when RegWrite signal from the control block is 1.

The IMM section is a section that allows us to create the appropriate immediate for each type of instruction. Since ALUOp tells the immediate module what type of instruction it is, immediate can obtain the whole instruction and output the appropriate immediate value.

The ALU section is about the ALU (obviously). ALU control receives func7 and ALUOp to send a coded signal to the ALU which contains the information about what type the instruction is and what it's func7 is. Then the ALU uses this coded signal and func3 to determine what operations should be done in that cycle. The ALU sends out a Bcond value as well as the computated value. The computation value is simply what the ALU operated. The Bcond value is only 1 when the branch condition is met.

The D_MEM section of the code has two functions. To load and store. To load, the D_MEM_BE signal must not be zero. The D_MEM_BE signal is computated from the Data_BE module we made. This model takes memwrite, memread and func 3 signal to

determine whether to read, to write and how many bits the memory should read or write. The SW block translates rs2 into required format depending on func3, and puts in the value inside D_MEM block for it to be stored. The LW signal sign extends the signal that comes out of MEM_READ during load and then passes it on.

The RD_SELECTOR section decides what value should rd take. This depends on the instruction. The simple ALU simply adds two input values or selects one of them. For example, for signals like JAL pc+4 is the value we want for rd, so the simple_alu selects the value above to pass as rd. The multiplexer that receives memtoreg is implemented in the same way and the same logic as it is in the MIPS architecture.

The NXT_PC section selects where the next pc value will point to. The logic itself is same with the MIPS architecture except for the added multiplexer on the top. This exists because of the JALR instruction and will output rs`+immediate if we are executing the JALR instruction.

Evaluation:

I think we successfully implemented the core path. However, as you can see we implemented a lot of extra modules for our architecture. I had doubts on whether this positively affected the overall architecture.

When using our source code, you must include the modules ALU.v, ALU_Control.v, Control.v, Data_BE.v, Immediate.v, LW_byteset.v, Mem_Model.v, Multiplexer.v, Simple_ALU.v, SW_byteset.v, in addition to the files given by the TA.

In terms of feedback, I think the overall assignment was challenging but very fun. I think better explanation of the interaction between the core module and the data memory blocks could have made the assignment better. For example, in RISCV_TOP D_MEM_DI is D_MEM_DOUT in testbench and D_MEM_DOUT is D_MEM_DI in testbench. This confused us for a while and we had to waste a lot of time recognizing this issue.

Conclusion

Because this was the first assignment that required us to implement what you might be able to call an actual computer architecture design, I learned a lot about how I should approach it. From allocating my work time and how to debug … etc.

More importantly, through this assignment I learned a lot about how to implement architectures and how we can divide and conquer the task to finally accomplish the overall task.

Finally, we realize we were a day late in submitting the assignment. So we wish to use our one of our late passes for the semester. Thank you.