LAB5: Pipeline CPU Lab

EE 312 Computer Architecture

Professor: Minsso Rhu

20150912 Maro Han

20150146 Sanghyun Kim

Introduction:

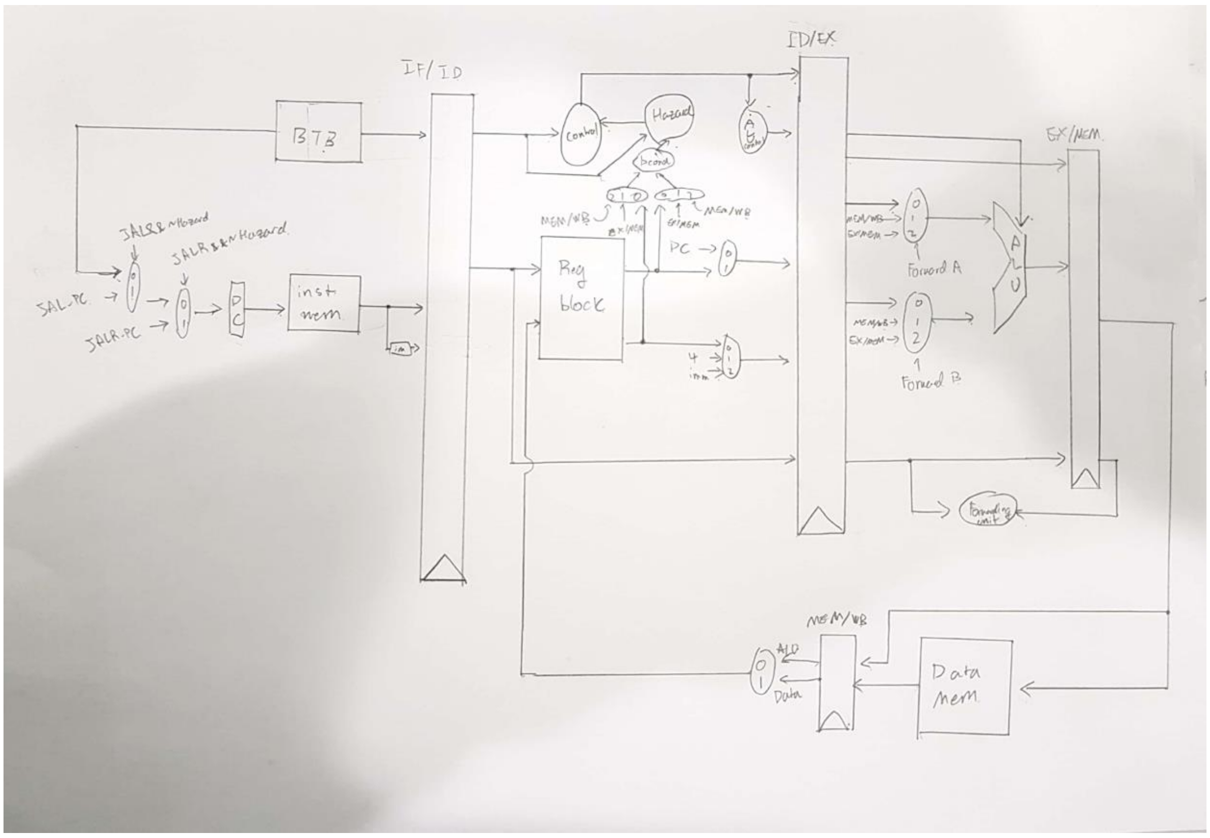
In our fifth lab, we were supposed to implement a pipeline CPU for the RISC-V ISA. We were asked to implement the instructions:

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

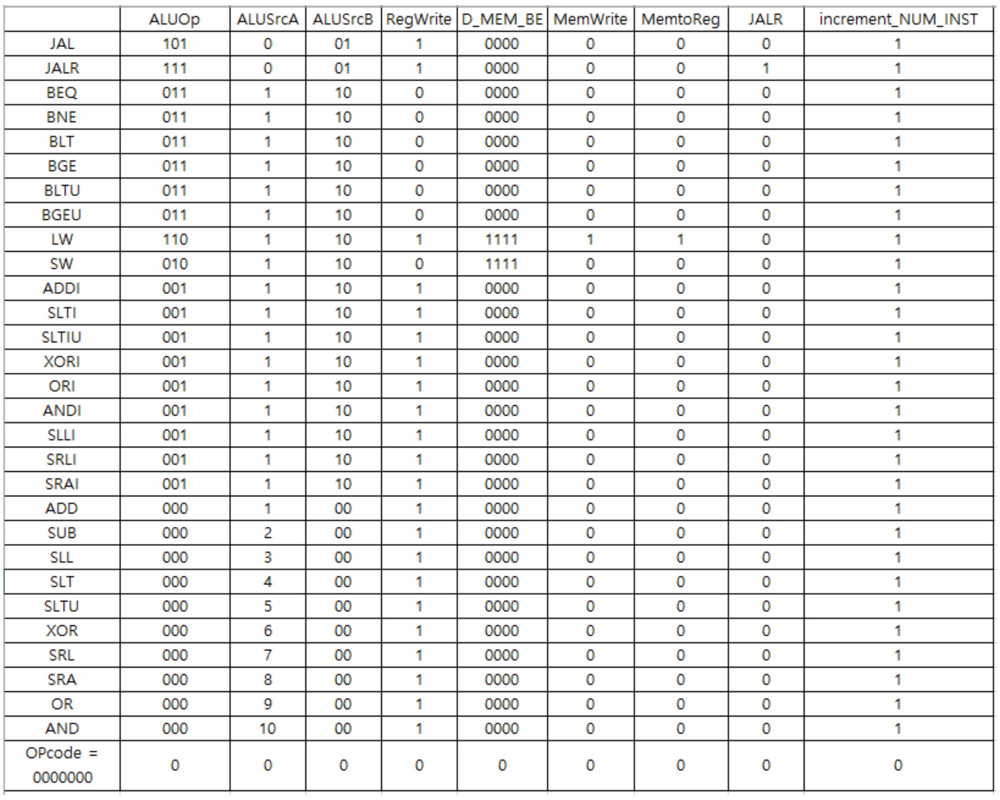
We took ideas from the pipeline implementation for MIPS ISA in our book to design our pipeline CPU.

Design:

The drawing below is a very simplified version of our datapath. We focused more on showing the data modules in our data path instead of showing all the connections because that would just make the drawing too complicated.



The logic table for control unit (which is combinational) is shown below:



	ALUOp	ALUSrcA	ALUSrcB	RegWrite	D_MEM_BE	MemWrite	MemtoReg	JALR	increment_NUM_INST
JAL	101	0	01	1	0000	0	0	0	1
JALR	111	0	01	1	0000	0	0	1	1
BEQ	011	1	10	0	0000	0	0	0	1
BNE	011	1	10	0	0000	0	0	0	1
BLT	011	1	10	0	0000	0	0	0	1
BGE	011	1	10	0	0000	0	0	0	1
BLTU	011	1	10	0	0000	0	0	0	1
BGEU	011	1	10	0	0000	0	0	0	1
LW	110	1	10	1	1111	1	1	0	1
SW	010	1	10	0	1111	0	0	0	1
ADDI	001	1	10	1	0000	0	0	0	1
SLTI	001	1	10	1	0000	0	0	0	1
SLTIU	001	1	10	1	0000	0	0	0	1
XORI	001	1	10	1	0000	0	0	0	1
ORI	001	1	10	1	0000	0	0	0	1
ANDI	001	1	10	1	0000	0	0	0	1
SLLI	001	1	10	1	0000	0	0	0	1
SRLI	001	1	10	1	0000	0	0	0	1
SRAI	001	1	10	1	0000	0	0	0	1
ADD	000	1	00	1	0000	0	0	0	1
SUB	000	2	00	1	0000	0	0	0	1
SLL	000	3	00	1	0000	0	0	0	1
SLT	000	4	00	1	0000	0	0	0	1
SLTU	000	5	00	1	0000	0	0	0	1
XOR	000	6	00	1	0000	0	0	0	1
SRL	000	7	00	1	0000	0	0	0	1
SRA	000	8	00	1	0000	0	0	0	1
OR	000	9	00	1	0000	0	0	0	1
AND	000	10	00	1	0000	0	0	0	1
OPcode = 0000000	0	0	0	0	0	0	0	0	0

The machine we implemented is a pipeline CPU. Most of the parts implemented in our machine is very similar to how the book did it. For example, we make control signals travel through the pipeline with extended pipeline registers. The different sections include Hazard unit, Forwarding unit, and BTB with the 2-bit saturation counter. Hence, we will focus on explaining these modules.

Hazard Unit:

The hazard unit either stall the pipeline or flushes the pipeline in certain conditions. The conditions are as follows:

1. For instruction **JALR**, flush once.
2. If **load instruction** directly precedes either **I-type, R-type or Store instruction** and uses address of rd of load instruction as rs1 stall and flush the pipeline once.
3. If **load instruction** directly precedes a **R-type** instruction and uses address of rd of load instruction as rs1 stall and flush the pipeline once.
4. If an **instruction that writes on the register (RegWrite == 1)** uses the address of rs1 or rs2 of the branch instruction as the address of its destination register directly precedes a **branch instruction** stall and flush the pipeline once.
5. If a **load instruction** that uses the address of either rs1 or rs2 of the branch instruction as the address of its destination register **directly precedes a branch instruction**, stall and flush the pipeline twice
6. If a **load instruction** that uses the address of either rs1 or rs2 of the branch instruction

as the address of its destination register **precedes a branch instruction after one instruction** (there is an instruction between load and branch instruction_, stall and flush the pipeline once.

7. If **BTB made a wrong prediction**, flush the pipeline once.

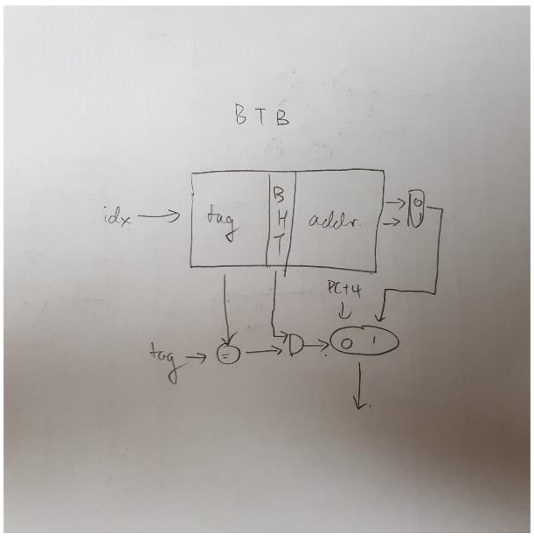
Forwarding Unit:

The forwarding unit forwards some result inside the pipeline to the next instruction before the previous instruction reaches the end of the pipeline. This is done to reduce the amounts of stall we have to do. Forwarding is done on three fronts. ALU input, b condition input and register rs2 in case of store instruction. The cases in which forwarding is done is shown below:

1. Address of rd in EX_MEM stage is equal to the address of rs1 or rs2 in ID_EX stage. We forward to the ALU front.
2. Address of rd in EX_MEM stage is equal to the address of rs1 or rs2 in IF_ID stage. We forward to the branch condition front.
3. Address of rd in EX_MEM stage is equal to the address of rs2 in ID_EX stage. We forward to the register 2 (for store) front.
4. Address of rd in MEM_WB stage is equal to the address of rs1 or rs2 in ID_EX stage. We forward to the ALU front.
5. Address of rd in MEM_WB stage is equal to the address of rs1 or rs2 in IF_ID stage. We forward to the branch condition front.
6. Address of rd in MEM_WB stage is equal to the address of rs2 in ID_EX stage. We forward to the register 2 (for store) front.
7. Our implementation of the BTB with 2-bit saturation counter is as follows

BTB with 2-bit saturation counter:

A simplified diagram of the BTB table we implemented might look like this:



Our implementation of BTB includes a 2-bit saturation counter. The BTB table stores the tag and branch address when it first encounters a branch instruction.

The initial value of the 2-bit saturation counter is set to 10 (taken), because branch instructions are taken more often statistically speaking. This is all done in the IF stage.

In the ID stage, we do the branch condition calculation to see if our prediction was right or not. If it is, we increment the 2-bit saturation counter for that index and if it isn't we decrement the 2-bit saturation counter for that index.

If we predicted that branch would not be taken and it turns out that we had to take the branch, we increment the 2-bit saturation counter and find the branch address of that index in our BTB. Then we make that our NXT_PC.

If we predicted that branch would be taken and it shouldn't, we decrement the 2-bit saturation counter and make the PC (of the instruction that shouldn't take the branch) +4 our NXT_PC.

Of course as written above, when branch prediction is wrong we flush the pipeline once.

Implementation:

The machine we implemented is a pipeline CPU consisting of 5 stages. We implemented forwarding logic, hazard control and branch prediction with BTB and 2-bit saturation counter.

All information regarding NXT_PC (except for JALR) is calculated in the IF stage.

Calculating branch condition (whether or not to branch), control signal generation and hazard control is done in the ID stage. JALR's jump address is also calculated during the ID stage.

Using the ALU and forwarding is done in the EX stage.

Anything regarding the data memory (LW and SW) is done in the MEM stage.

Writing back to the register is done during the WB stage.

Evaluation:

I am quite satisfied with the result. Our CPU runs the instructions in a reasonable timeframe. However, I think the logic to HALT takes more cycles than it needs to be. At this stage, we couldn't really think of a better way of implementing it and also didn't think coming up with a very efficient algorithm for turning on the HALT signal was the big part of this lab.

We pass all the tests in all three test benches.

Discussion:

I liked this lab because we got to learn more in depth about how to program our CPU using different timings (ex: negedge CLK) and seeing how different modules work in various ways depending on the way we timed the modules.

Concluding:

Personally, I felt very proud of us when we finished the lab. At first it almost seemed impossible but segmenting the problem and solving one by one resulted in us finding the final solution.