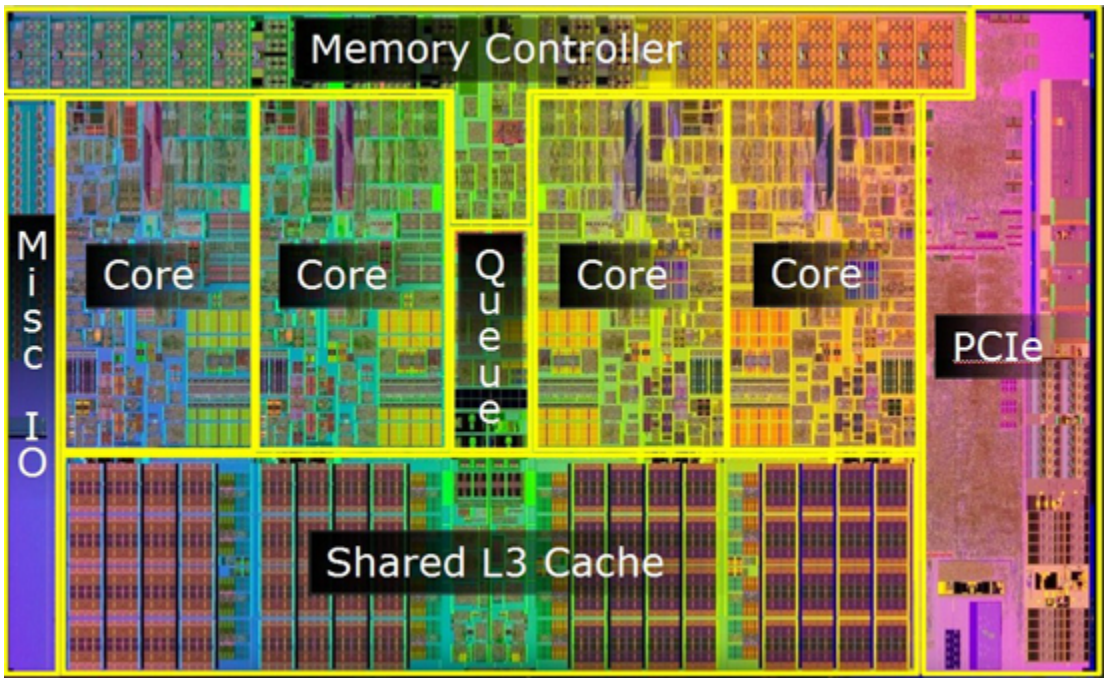
EC413 Computer Organization

Instruction Set Implementation

Petar Ojdrovic



Fall 2015

Purpose

This machine was built to demonstrate the functionality of a simple CPU, and to illustrate the interaction of all the modules within a functioning CPU. It was implemented using a non-pipelined multicycle design. The processor was created using a modular methodology to show that a relatively complex machine can be broken up into much smaller components and modules which are more easily developed.

By using a multi-cycle design, I was able to strike a good balance between the speed of a pipelined design and the ease of a single-cycle design. For the scope of this class, creating a pipelined CPU would have been rather complex, and as such, it was not chosen. A single-cycle CPU, however, is rather simple do design. It has a major flaw in the fact that you must clock it slow enough so that each instruction has time to complete within one clock cycle. This is very large overhead, especially when an instruction is able to execute in fewer clock cycles than another different instruction. Therefore, the execution time will on average be quite a bit faster than a single cycle CPU.

To make design and synthesis easier, the Verilog HDL was used, and a modular design methodology was implemented. This allows complex modules to be design individually, and contained within themselves. This allows changes and upgrades to be done without affecting the rest of the design in a profound way.

Instruction Set Definition

In this design, a custom instruction set was used. The functionality of the instructions is based on the MIPS ISA, however, the encoding of the instructions is different. Unlike the MIPS instruction set, our instruction set does not have a function field for the R-Type instructions. All of the execution requirements are stored within a 6-bit opcode. This has its pros and cons. First, it allows for a simpler design as the controller does not have to be as robust or as detailed. However, it reduces the number of possible instructions to 64, which is not enough for a full instruction set and would probably not be used in a real CPU.

The instructions implemented in this design are I, R, and J-type instructions. I also implemented various memory instructions. The following page outlines how the instructions are encoded, and what the opcodes for each instruction are.

Complete List of Instructions

Op3	Op2	Op1	Op0	Op1	Op0	Output	Function Name
0	0	0	0	0	0		NOOP
0	1	0	0	0	0	R1 = R2	MOV
0	1	0	0	0	1	R1 = ~R2	NOT
0	1	0	0	1	0	R1 = R2 + R3	ADD
0	1	0	0	1	1	R1 = R2 - R3	SUB
0	1	0	1	0	0	R1 = R2 | R3	OR
0	1	0	1	0	1	R1 = R2 & R3	AND
0	1	0	1	1	1	R1 = 1 if R2 < R3, else 0	SLT
0	0	0	0	0	1	PC ← ZE(Limm)	J
1	0	0	0	0	0	IF (R1 == R2) THEN PC ← PC + SE(Imm) ELSE PC ← PC + 1	BEQ
1	0	0	0	0	1	IF (R1 != R2) THEN PC ← PC + SE(Imm) ELSE PC ← PC + 1	BNE
1	1	0	0	1	0	R1 = R2 + SE(Imm)	ADDI
1	1	0	0	1	1	R1 = R2 – SE(Imm)	SUBI
1	1	0	1	0	0	R1 = R2 | ZE(Imm)	ORI
1	1	0	1	0	1	R1 = R2 & ZE(Imm)	ANDI
1	1	0	1	1	1	R1 = 1 if R2 < SE(Imm), else 0	SLTI
1	1	1	0	0	1	R1[15:0] ← ZE(Imm)	LI
1	1	1	0	1	1	R1 ← M[ZE(Imm)]	LWI
1	1	1	1	0	0	M[ZE(Imm)] ← R1	SWI

The encoding of instruction looks as follows.

Type	31	format (bits)			0
R	opcode(6)	r1(5)	r2(5)	r3(5)	
I	opcode(6)	r1(5)	r2(5)	imm(16)	
J	opcode(6)	imm(26)			

It is important to note that this is not the MIPS encoding system. R-type instructions do not have an offset/shift field or a function field. Also, the registers referenced, and the order in which registers are looked at is different from the MIPS instruction set.

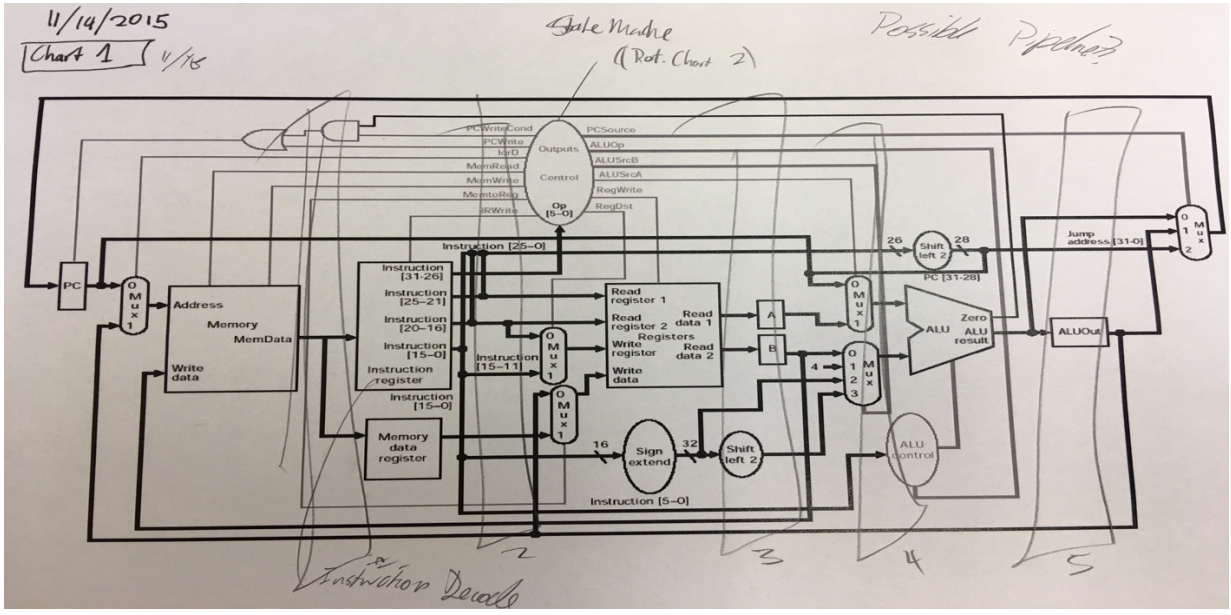
Architecture

The most challenging aspect of this project was implementing an architecture which would allow for the execution of the necessary instructions, and in the way in which the instructions are encoded. The three different instruction types outline a template which all instructions need to follow. This allows us, the designers, to take advantage of control signals which can be shared by many of the instructions.

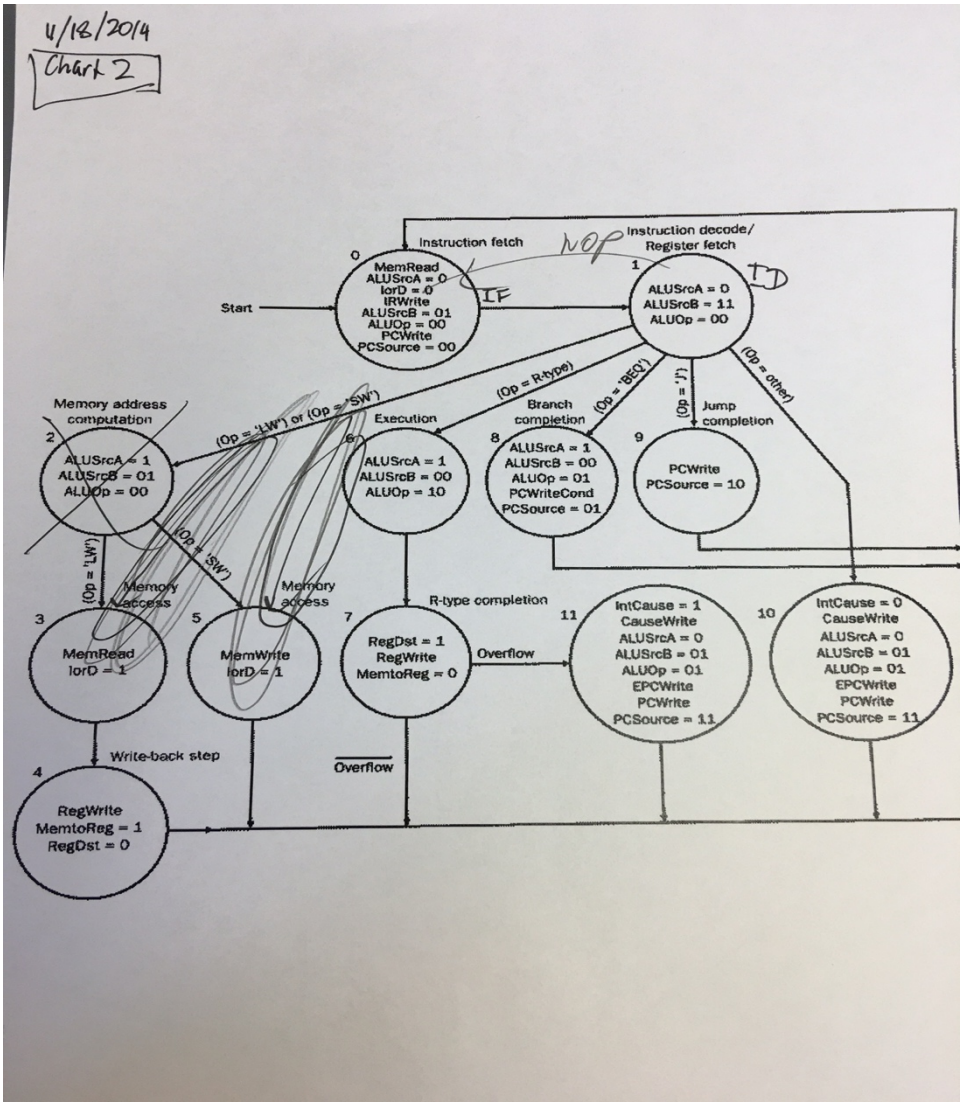
When I started designing the CPU, I used the MIPS datapath studied in class as a template, and built off that. My first iteration/design essentially replicated the given datapath. However, this design had its substantial flaws. First, there was only one memory module. This complicated the control quite a bit, as you'd have to tell the processor exactly at what addresses to read from and write from. This got problematic when then instructions and data were in the same place. I would often run into issues in which the write back would delete and overwrite instructions. Unbeknownst to me (as I was rather sick for the majority of the past month and not able to come to discussions and office hours), the TA's provided us with separate memory modules for Instructions and Data.

After I implemented the two separate memory modules, the design got substantially easier, as I no longer had to worry about instructions and critical pieces of memory being overwritten by the outputs of other instructions.

At this phase in the design process, I was considering the implementation of a pipelined design, and the following image reflects that.

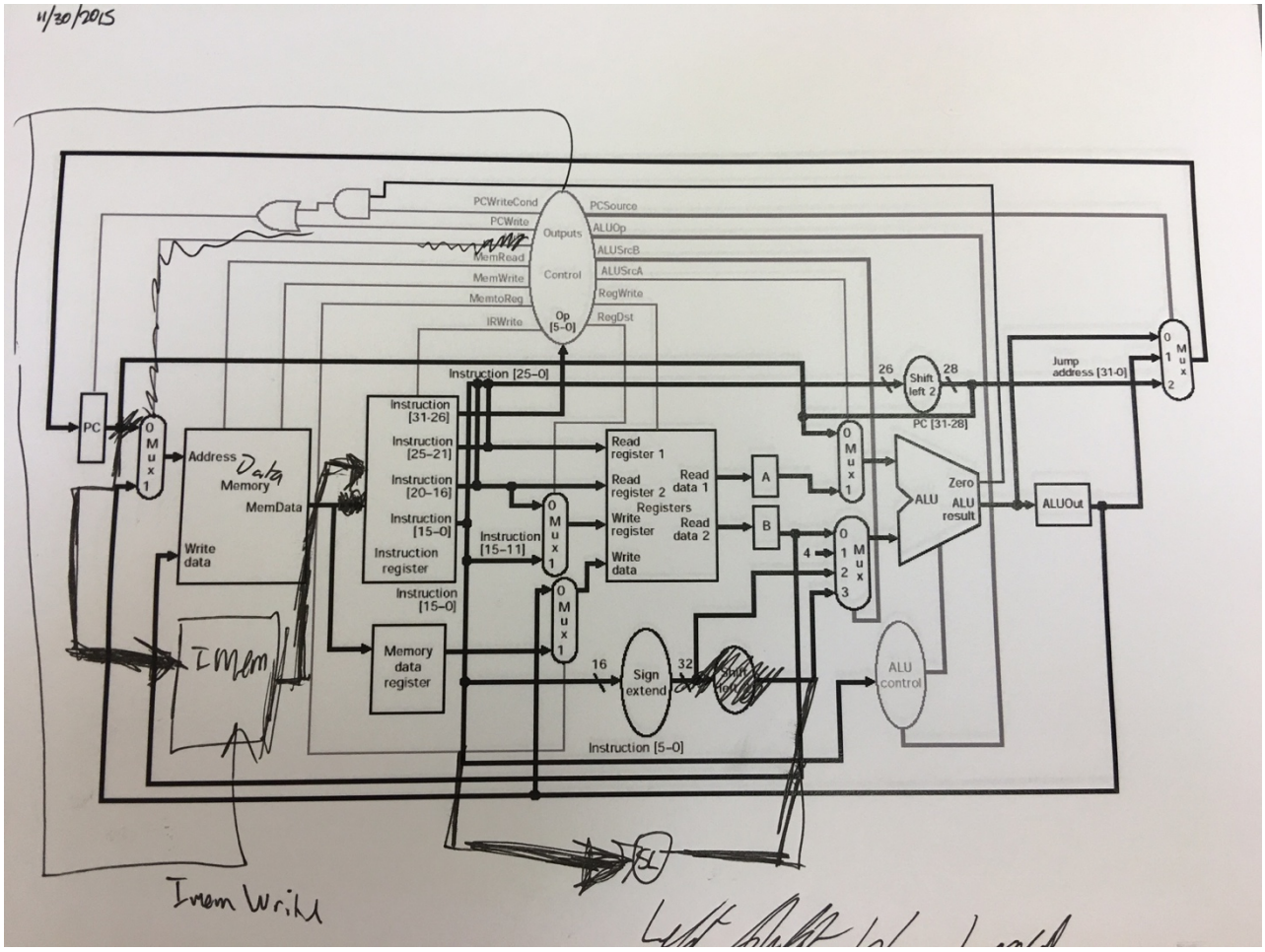


The state machine at this phase in the design process look like this:



This type of design, however, proved to be rather impossible to execute in a timely manner, and would make it difficult to use the provided instruction set which was different from the studied MIPS instruction set.

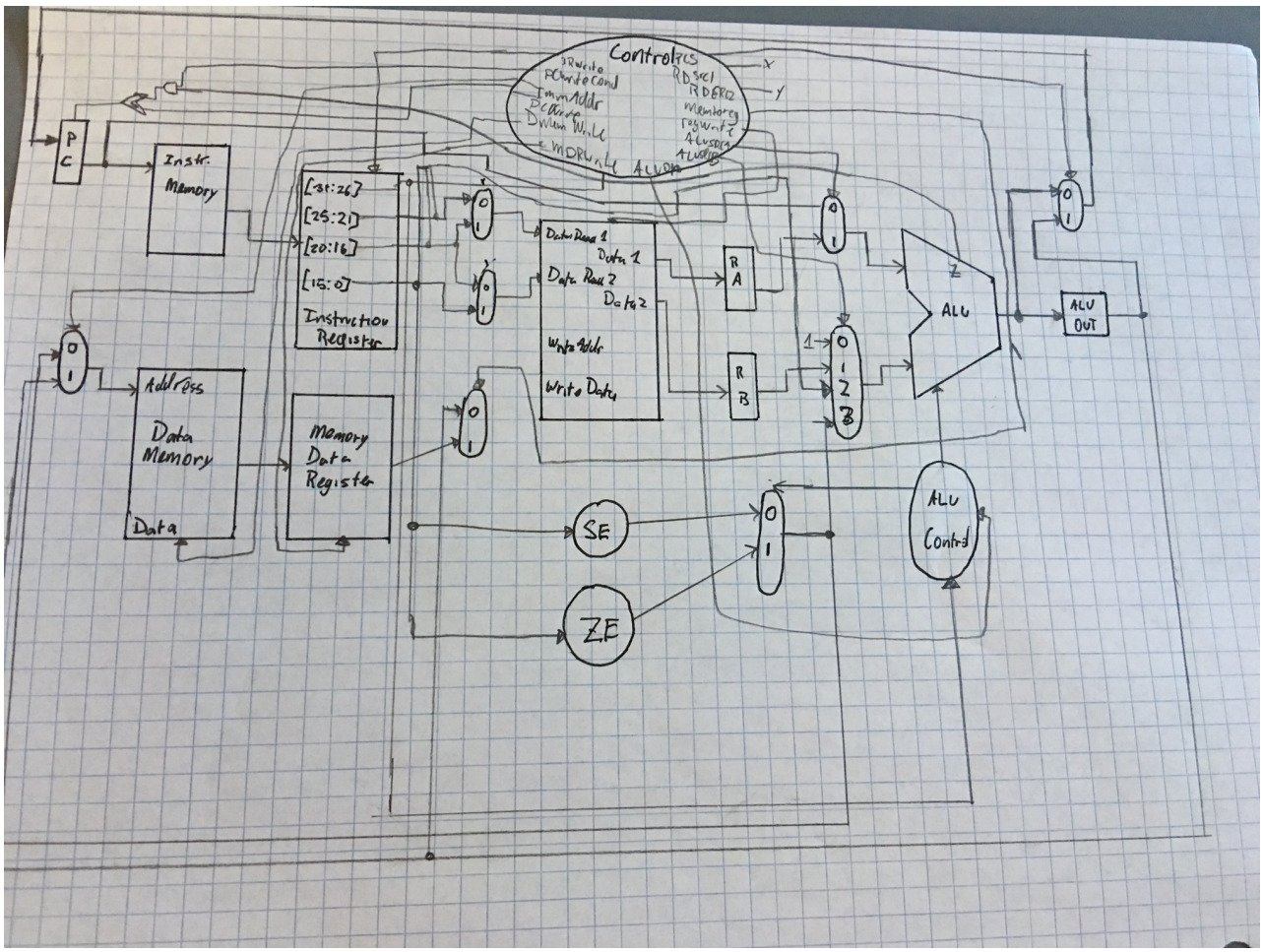
After a few design revisions, I came up with the following base design.



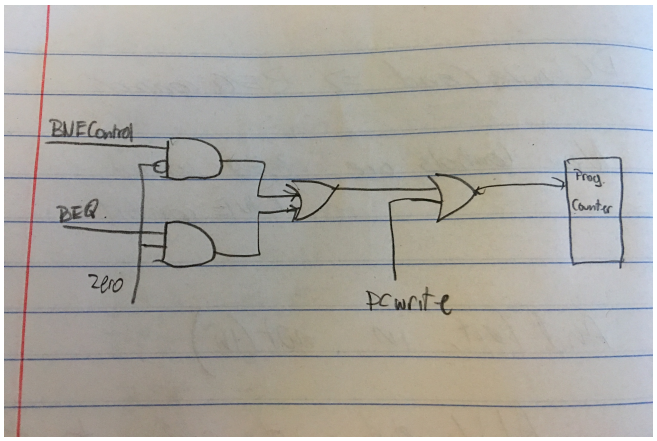
It is important to note that in this revision there are two memory modules – one for instruction memory and the other for data memory.

I soon ran into problems when implementing instructions which were depended on the immediate field. It became apparent that with the desired datapath and instruction set, a single sign-extend unit would not be enough to support all the required immediate instructions. However, It did become apparent that the shift operations would not be needed for the instructions which I wanted to implement.

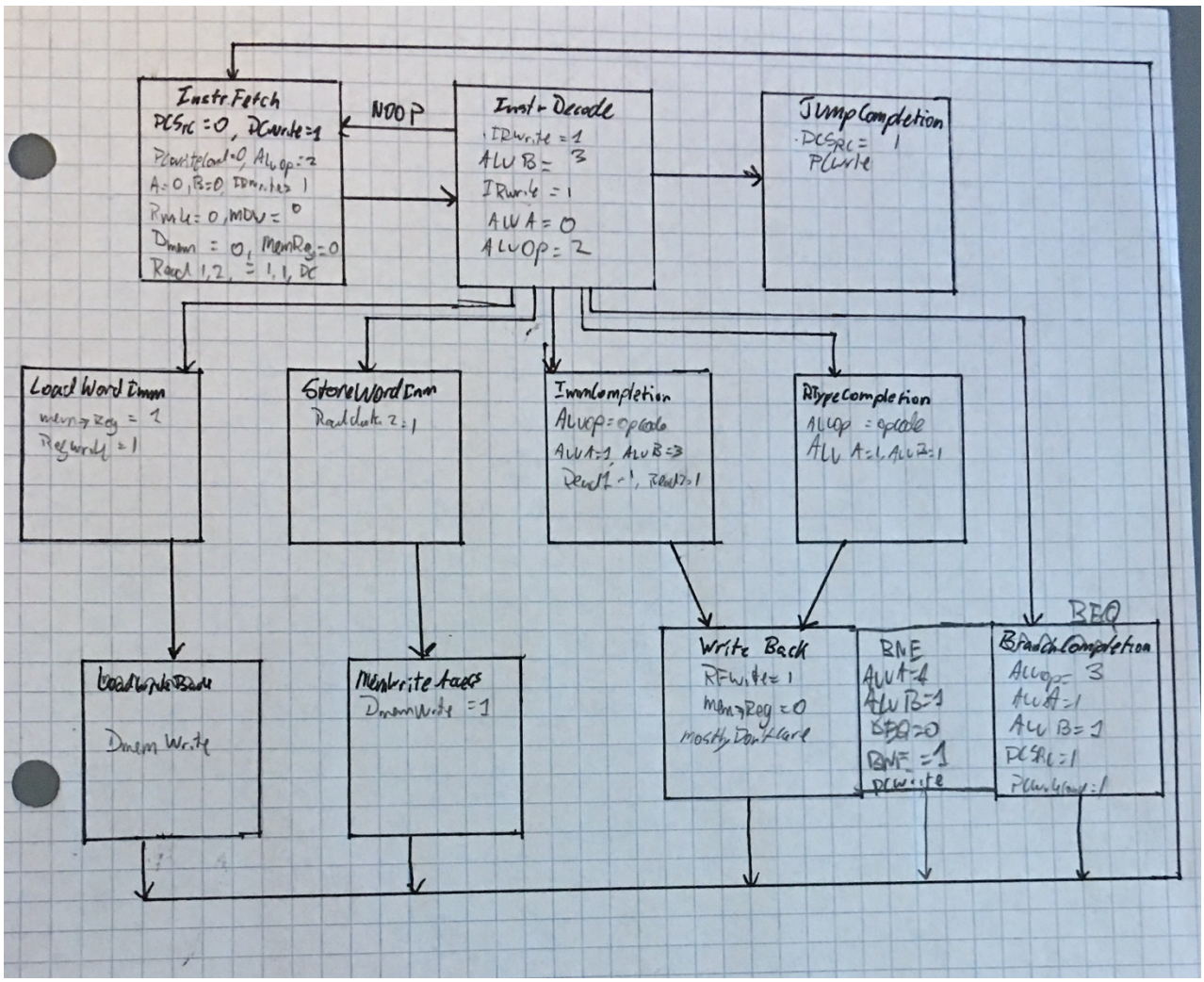
The final design looked like this:



The branch logic may be a bit obscure in this one so I've 'zoomed in' on it in the following image:



The final iteration of the controller state-machine looked like this:



Testing and Debugging:

In order to try and simplify the design, I built the CPU such that the instructions would be stored in the IMEM module. This would make rather simple to design a test bench, as the test bench would only have to cycle the clock and do nothing else.

Overall, testing was a fairly straightforward process. In fact, the most common error I encountered were typos in variable names when instantiating modules. This caused a great deal of aggravation as it would seem as if a much larger problem was lurking in the design, and not simply a typo.