**Annie Brey, Trey Kneads, Shreyas Patil, Zabeeh Babar**

**CDA 4205**

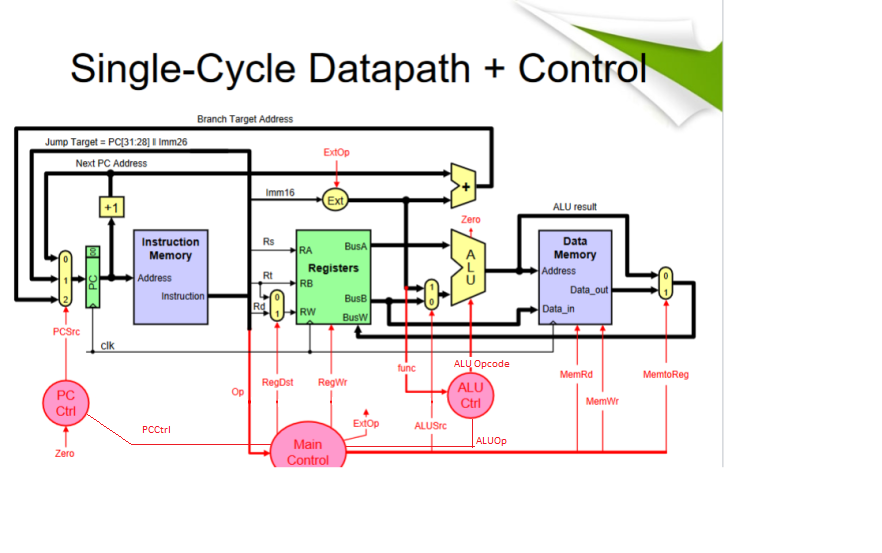
**Single Cycle CPU Report**

**5/1/2019**

We also made a video describing how our design works:

[**https://youtu.be/jU8gVB3Wnws**](https://youtu.be/jU8gVB3Wnws)

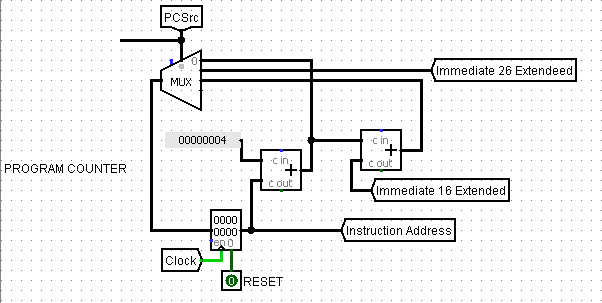
**DESIGN AND IMPLEMENTATION**



The picture above describes our datapath of the single-cycle CPU we have built. We have made a few changes to the original datapath listed in our powerpoints, these mainly include the PCCtrl and ALUOp that are sent out from our Main Control. Also, rather than having our Instruction opcode going to PCCtrl, ALUCtrl, and Main Control, we instead have it going to Main Control which sends other control signal to the ALU and PC Control.

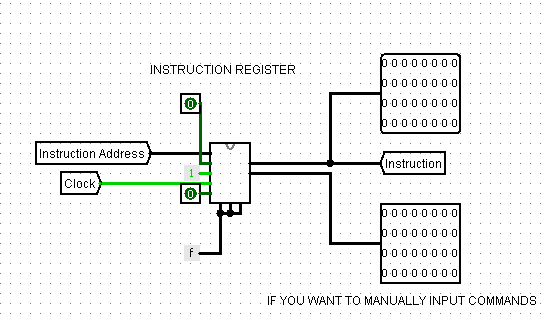
Now we get into the individual pieces of our CPU

**PROGRAM COUNTER**



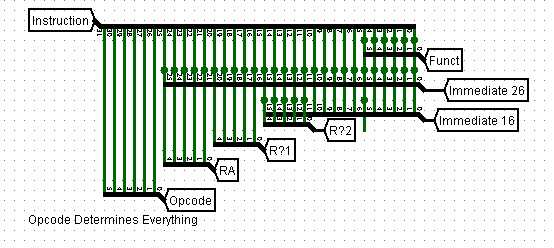
Here is our Program Counter, every clock cycle it automatically adds by 4 to the instruction address it is currently pointing to. Whenever a command is given that changes the PCSrc in any way, it will either select the multiplexer options for branch or jump.

**INSTRUCTION REGISTER**



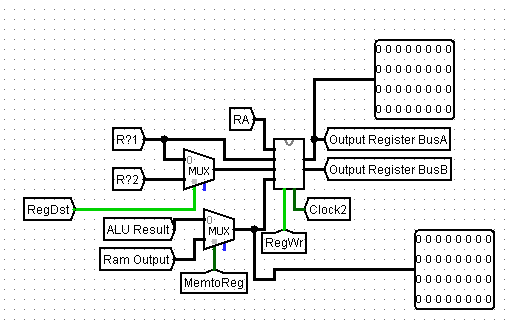
Our instruction register takes the instruction address that our program counter is currently looks at and outputs the instruction located in its memory at that address. Due to the setup of our instruction memory, we have to manually input commands 1 command every clock cycle, which takes some setup to begin. The constant F we have there makes it so it loads “one whole word” every time.

**BIT SELECTION**



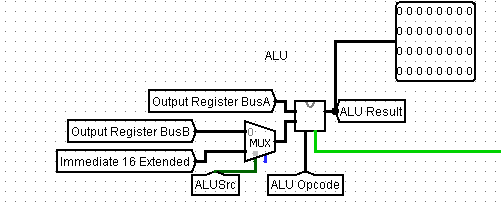
I am just including this picture because it includes a large portion of the individual bit separation for every command. Most of these bits select importing things like Opcode, Registers, Immediates, and function code, which usually go into a multiplexor waiting to be selected.

**REGISTERS**



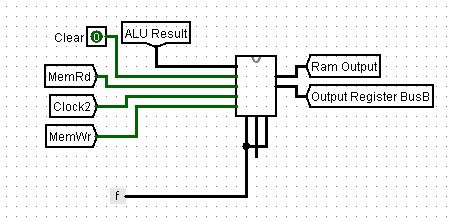
Here are our registers, the registers being selected are based off what time of command is being run. RegDst selects the register that is going to be written to, MemtoReg selects if the value in memory is going to write back to register or the value in ALU will. RegWr just determines if there is going to be a value that is writing to register. It runs on a 1 clock cycle delay due to timing issues with our current CPU. There are two 32bit output blocks, one shows the output of register being selected and output into BusA, the other selects the output of the multiplexer preparing to be written to the register.

**ALU**



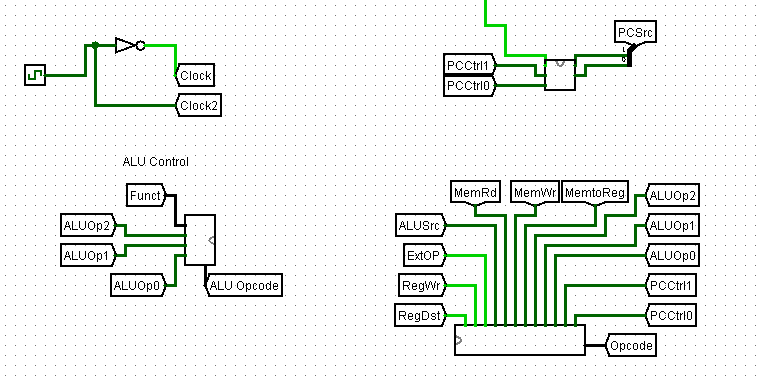
Here is our current ALU with an additional 32bit output to view it’s current output. It takes in values that the register select, and it might also take an immediate from the command itself, which is selected by a multiplexor. The actual command that is run inside the ALU is determined by the ALU Opcode and what instruction is currently being run.

**MEMORY**



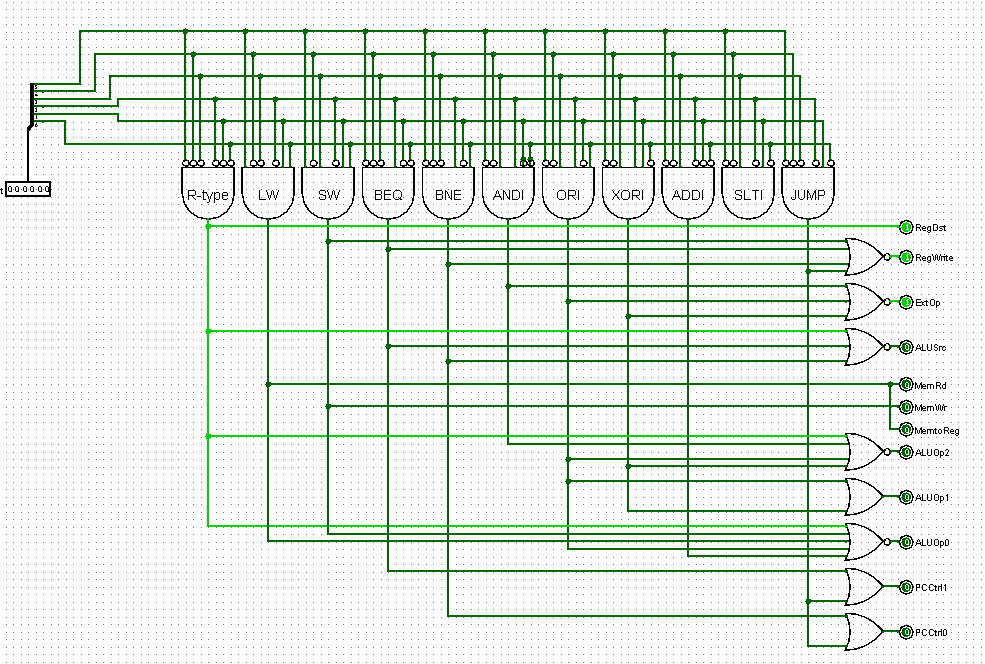
Our memory is nothing special, it is the exact same memory we used for our Instruction Register, and it is hard coded similarly to only handle full 32bit numbers. Inside of it is extremely complicated however, and consists of banks, modules, and chips.

**CONTROL UNITS**



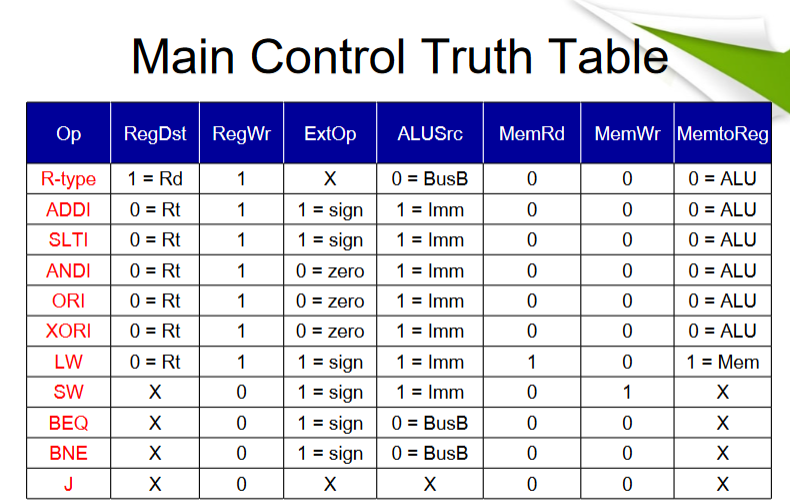
Here, I’m going to combine the control units into one picture/diagram. The Main Control is the bottom right and takes in the Opcode which determines pretty much everything else in our CPU. Right now the bits you see selected are based on the fact that the command currently running is a 32bit 0, so it thinks it is an R type instruction and selects bits accordingly. The Main Control sends its signals through the CPU. The ALU Control on the left, as well as the PC Control above take in these signals and determine the course of action the ALU and PC will take. The clock on the top left is just how we spaced out the instructions so they didn’t interfere with the rest of the program when running.

**MAIN CONTROL**

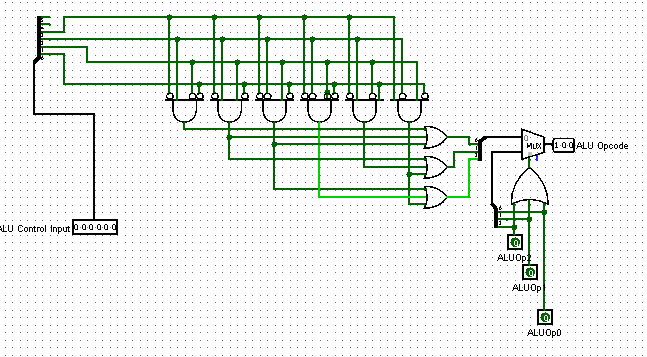


This is the main control, depending on the 6 bit opcode it will select it will select the correct control bits.

This is what we based all of the control bits for the Main Control.



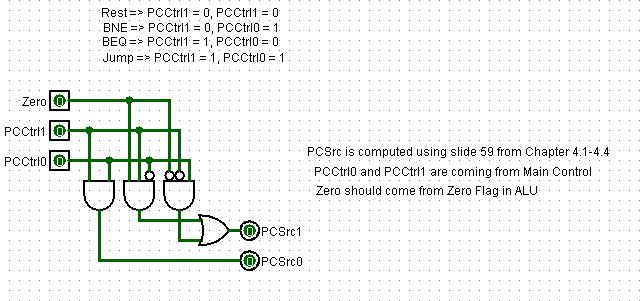
**ALU CONTROL**



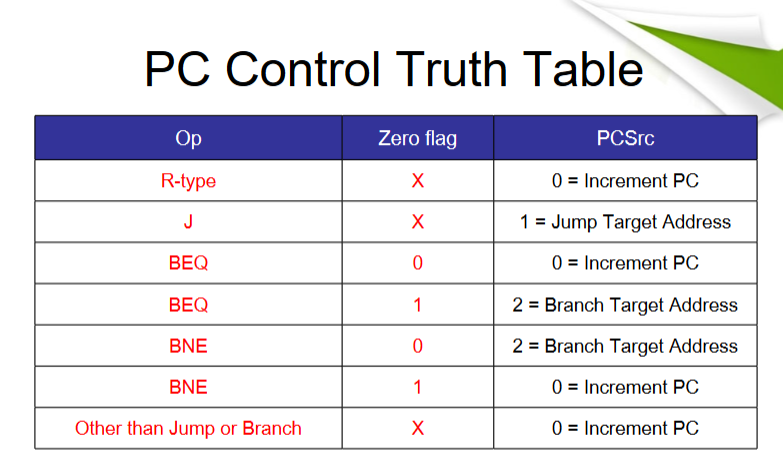
This is the ALU control, this will select the ALU opcode depending on what 6 bits are input.

This is the table we based our function codes and opcodes on for the ALU control:

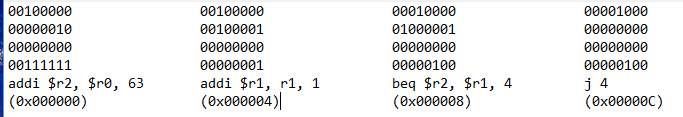
**PROGRAM COUNTER CONTROL**

****

Depending on the PCCtrl1 and PCCtrl0 and Zero input, the PC Control will select the correct PCSrc1 and PCSrc0. These outputs are for if we have a jump instruction or if we need to go to the next instruction, 0 is to increment PC and 1 is for Jumping to an address.

This is the table we based our PCSrc on for the PC Control:

**SIMULATION AND TESTING**



Here is a program I created for testing purposes. It sets register $2 = 63, and increments register 1 by jumping back to that instruction until register 2 = register 1. Once Register 2 and Register 1 are equal, the program skips the jump and continues out of the loop.

The actual running of this simulation was done in our youtube video, and the correct response occurred for the program. It jumped between instruction address 4, 8, until register 1 was equal to register 2, and then it continued on as planned.

