ECE366 Lab1,2,3: 8-bit Processor Design

In this course you will design a specialized processor. This processor should be optimized for two given programs, **Square** (P1) and **Widest** (P2) (see following for details). You should assume:

- The processor is a single-cycle machine, so realize that there is a limit to what can be done in one cycle.
- The processor features fixed-length instructions of 8 bits wide. In other words, the instruction memory is byte addressable and its data bus is 8-bit wide.
- Data memory is byte also addressable, and *loads* and *stores* can read and write exactly 8 bits.
- Data memory is a single-ported memory (a maximum of one read or one write per cycle, not both).
- Processor can *write* only one register per cycle. The only exception to this rule is that you may have a single 1-bit condition register (e.g., carry out, or shift out, sign result, etc) that can be written at the same time as an 8-bit register. Of course, you can *read* more than one register per cycle.
- All registers should also be 8-bit wide.
- Your optimization goals are:
    - Minimize dynamic instruction count (i.e., the number of cycles executed during the running of a program).
    - Simplify your processor hardware design.

  You are welcome to also optimize for other things (e.g., cycle time, ease of pipelining), but if you do so, we will expect you to discuss that optimization intelligently, and these two goals should still take highest priority. You will be rewarded, in particular, for doing a good job with the first goal.

Above description is valid for the first three labs. So keep in mind to review these limitations frequently and revise your design accordingly, if one is violated.

Programs:

Square: Write a program that calculates the square of an unsigned number. The 8-bit input is found in data memory location 0, and the 16-bit answer is to be written in locations 1 (high byte) and 2 (low byte). You are NOT allowed to have a square or multiply instruction in your ISA.

Widest: Write a program to find the "widest" integer in an array of 32 integers. Note that there may be more than one widest number (of equal width), and in this case, find the *first* widest one in the array. The array begins at data memory location 32. The *width* of an integer is defined to be the distance between the least significant and the most significant "1" in the binary representation. For example, the width of 00110100 is four, the width of 10000000 is one, the width of 00000000 is zero, and the width of 11010001 is eight. Write the widest width in memory location 3, the number itself in location 4, and the address where you found the number in location 5. For example, if the four values above were the inputs, you would write 8 (the width) in location 3, 11010001 (the number) in location 4, and 35 (the address of the number) in location 5.

Lab 3: Single cycle CPU

In this assignment, you will finally design a single-cycle implementation of a processor to execute your 8-bit ISA. Your design will have a program counter (PC) controller, an instruction memory, a register file, an ALU, a data memory, and a lot of "glue logic". The ALU and register file should not be significantly different from the last lab. Your machine code will be stored in the instruction memory. Use a PROM to create this memory. You can use a dedicated PROM memory for each program (machine code), both starting at address zero of their dedicated memory. Or, you may like to put both codes in one single PROM and run both programs from the same code. For data memory you will use a RAM. The input data for both programs are found in data memory and the results of the programs are stored in data memory, as well. Three sample input-data files were posted on Piazza previously. You need also to design a circuitry for:

1- an *Init* signal that initializes PC to the beginning of each program (e.g. zero).
2- Dynamic Instruction Counter: This block should count the number of cycles required for each code to be executed. *Init* signal will set the counter to zero. *Halt* instruction stops the counter from counting up.

Again we will use LogicWorks5 to implement our CPU. You will have two weeks to complete this lab. You will demo your work every Tuesday. In demos, you will load your schematic in the LogicWorks5, modify the contents of the data memory (input data) and run the simulator. The result should be presented both in timing diagrams and data memory. You will have an exam based on your ISA design and implementation on the third week of this lab. You need to be able to describe why each block is designed in the way they are.

Lab3A: Control Design

For this part, you will design the control block of your CPU in LogicWorks. It should consist of the following parts:

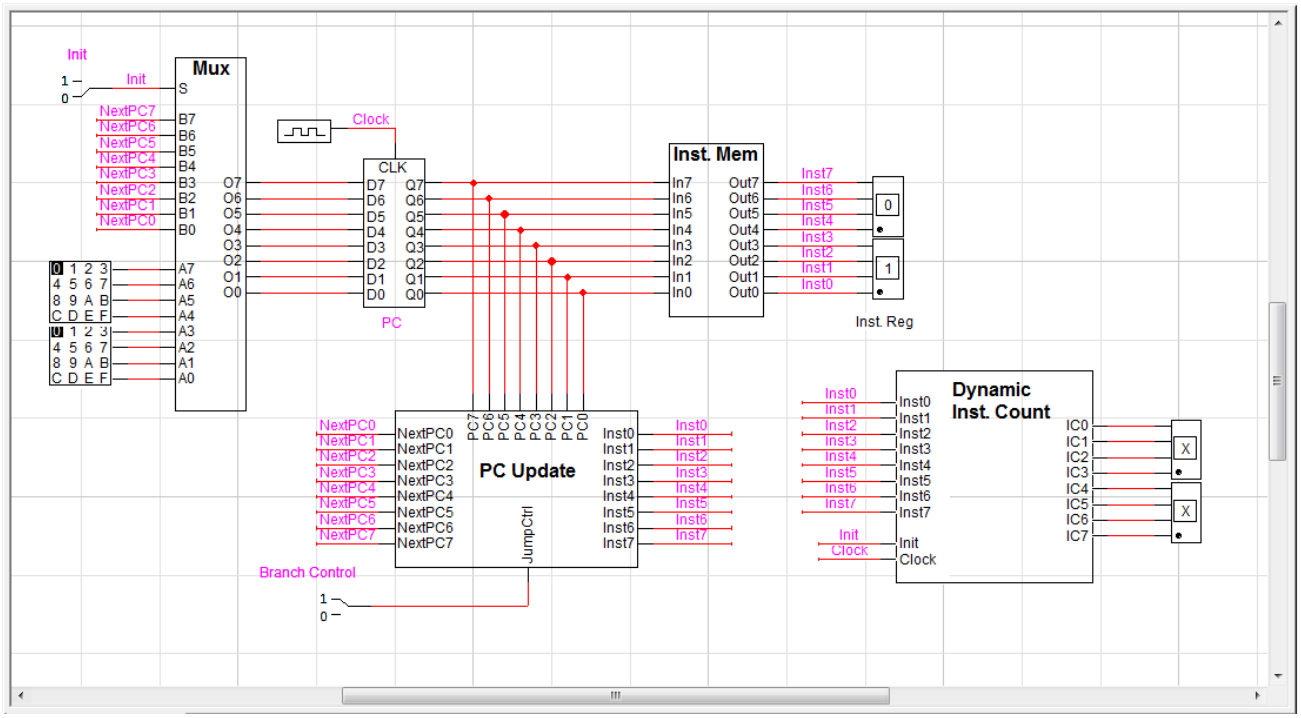
1- Instruction memory: You will use a PROM for instruction memory.
2- Program Counter (PC) register: this register holds the address of the machine code that is now being executed.
3- Init signal: this signal sets PC to the address of the beginning of the code.
4- PC-update: provides the address of the next instruction to be executed.
5- Dynamic instruction counter: counts number of required cycles to perform a task (**P1** or **P2**). Note that *Init* signal should reset the counter (set to zero).

Note that in PC-update you are implementing your branch instructions. If the instruction is not a branch, it simply adds +1 to PC. On the other hand, when the instruction is a branch, then if the branch is taken, PC should be updated by the new value. Otherwise, again PC is updated by PC + 1. Whether the branch is taken or not will be determined by ALU. For now use a binary switch to control the branch.
Also, note that if the destination address of the branch instruction in your ISA is not part of your instruction and is needed to be access from register file or memory, PC-update block should support another input port for that. In that case, use two hex keyboards to feed the value to the port for now.
Finally, note that a Halt instruction will have two roles:

1- Tells the PC-update to apply PC = PC + 0
2- Stops the dynamic instruction counter.

Your high-level schematic will look like the following image:

An Example High-level schematic:



Lab3A demo

You will demo the controller in the first week. Load your machine code in PROM, initialize the PC with beginning address of your code and show how the codes are fetched according to the program flow. Also be prepared for the following questions:

a) Which instructions in your ISA affect the program flow (e.g. Halt, Branch, …)?
b) What will be your dynamic instruction count if everything goes well?
c) What will happen if BranchConrol signal goes high during a non-branch instruction?
d) Will your code work if started from an address different than zero?

Lab3A Extra Credit:

You will earn extra credit for submitting your progress report. This report follows the same format as the 5th part of Lab3-final-report (see below for details).

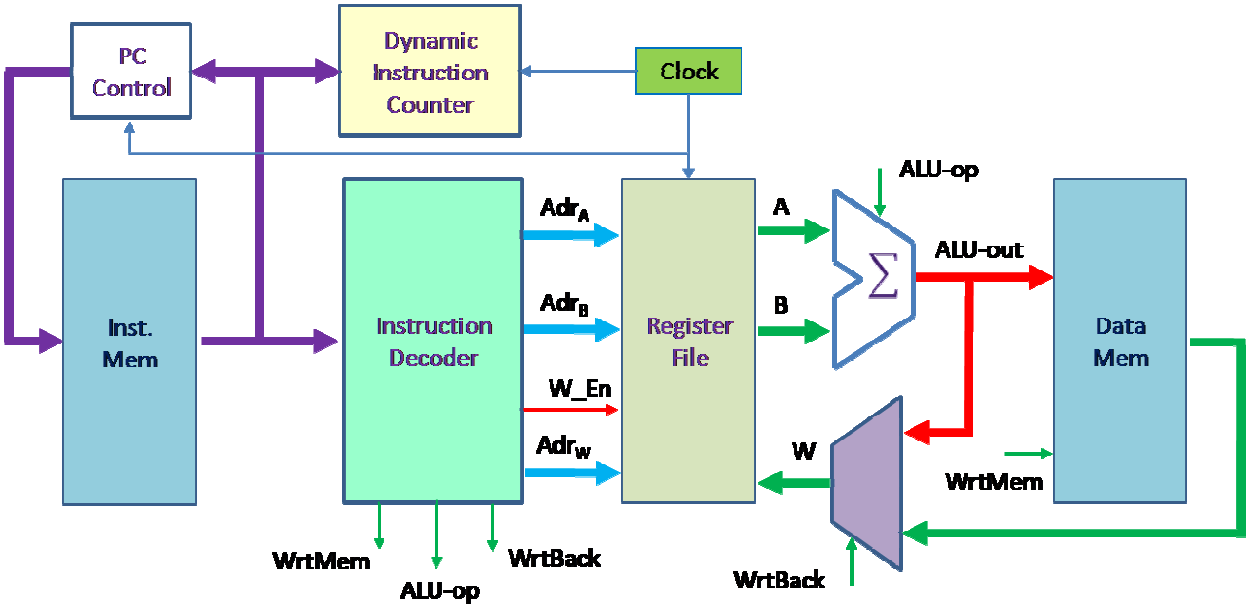
Lab3B: Single Cycle CPU

In this lab assignment, you will design your single-cycle CPU. You will connect the control block to the ALU and register file from the previous lab and also the memories. The high-level conceptual schematic of your final design should look like the following image. Note that, your schematic will contain much more details than the one shown here. For instance:

1- There is an input signal for "PC Control" block, called Init signal. It comes from a binary switch and sets the initial value of PC register, see Lab 3A.
2- There is another input signal for "PC Control" block, called "Branch Control" signal. It is generated by ALU or maybe is a function of some flags coming from ALU.
3- Dynamic Instruction Counter should have an input for Init signal, see Lab 3A.
4- Your ALU has several output flags (like Carry).

5- If your ISA supports immediate values in the instructions, you may need to bring it to the ALU ports. So you will need a MUX to select between the immediate value or (let's say) register B as the second input of ALU. Also, you need a control signal for this MUX to be generated by Instruction Decoder.
6- If the ALU in your ISA works directly with memory, you need another MUX to select whether the data comes from register file or form data memory. The control for this MUX should also be considered.

An Example Conceptual High-level Schematic:



Lab3B demo

You will demo your single cycle CPU running **P1** and **P2**. You need to load your machine codes in instruction memory and the input values in data memory. When you run the simulator, the clock runs. You will control the execution by Init signal. When Halt instruction is executed, the dynamic instruction count will stop counting up. Then you will show us the contents of the data memory, which should contain Square or Width. We will test your design by changing values in data memory and observing the results. Also be prepared for the following questions:

a) Does your dynamic instruction count depend on input values? Why or why not?
b) How did you manage writing back to registers?
c) Describe your data path.
d) Is it a register-register architecture, a register-memory architecture or another type?

Lab3B Extra Credit:

You will receive extra credit for evaluating your hardware complexity. For example:

1- How many gates did you used in your CPU (hardware cost)?
2- What is the longest instruction and how much delay it requires?
3- How you can improve the delay? What about hardware cost?

Lab3C: Exam

In the third week of this lab, you will demo your single cycle CPU again. But this time, you will assume that you are in an interview for a job at Intel. You should do your best to sell your design and show us that you comprehend the material. You need to start from the tasks (**P1** and **P2**), explain your design steps, justify your ISA design and present your implementation in LogicWorks. You only have 15 minutes to represent yourself and you are competing with other groups. Your presentation may include the following parts:

- Your algorithms for **P1** and **P2** in the form of a flowchart or pseudo-code
- The assembly code for **P1** and **P2** and a list of all instructions that are used by those programs.
- Your 8-bit ISA description, including name, syntax, operation, encoding, tables and etc.
- Description of your Register File design
- Description of your ALU design
- Description of your Memory Access design
- Description of your PC-Controller design
- Description of your Instruction Decoder design and all control signals
- Demo the Final CPU working on input data
- Self-evaluation on your design and suggestions for its improvement

All of the students should be in the lab, for the whole lab time (not only your timeslot). While presenting your work, see your classmates as your employers' interview committee. Everyone evaluates the presented work and gives his opinion on the interviewee. We will forward you the results and comments of interview.

Lab3 submission:

As the final report, you will submit a *zip file* containing two files:

- Your report in the following format.
- Source files of your CPU schematic design (cct and clf files).

Final Report:

Your final report should include every detail of your work. It should be such detailed that any other student in your class can design your processor by follow your report. It should follow the format described below:

1) ISA overview:

a) **Introduction.** The name of the architecture, overall philosophy, specific goals strived for.

b) **Instruction description.** Which instructions are supported, the format of encoding for each, and an example with machine code. Use well-organized table(s) to present your instruction design. The description should be detailed enough that someone could write an assembler (a program that creates machine code from assembly code) for it.

c) **Register design.** How many registers are supported? Are they general-purpose or specialized? Is there anything special about your registers?

d) **Memory and addressing modes.** How large is your data memory? How many bits are needed for memory addresses? How are memory addresses constructed / calculated in your instructions (for load/store)? Give examples.

e) **Control flow (branches).** What types of branches are supported? How are the target addresses

calculated? What is the maximum branch distance supported in your ISA? Give examples.

f) **P1 and P2 programs (assembly language); well commented**. State any assumptions you make. You cannot assume anything about the values in registers or data memory, other than those specifically given, when the program starts. This means, for example, that if you need zeroes in registers or memory, you need to put them there.

2) ALU design:

a) **ALU operations:** the list of all ALU operations, accompanied by the instructions they are relevant to. Describe your ALUop table and encoding details/challenges.

b) **Schematics:** hierarchically organized schematics of all components in your ALU.

3) Register File design:

a) **Registers types:** a description of how many registers of which types.

b) **Schematics:** hierarchically organized schematics of all components in register file and its high level integration with ALU.

4) Instruction Decoder design:

a) **Instruction Decoder description:** including a description for each control signal and the associated truth table. Describe how you reduce number of gates required in your design.

b) **Schematics:** hierarchically organized schematics of all components in your instruction decoder and its high level integration with ALU and register file (different MUXs that you used …).

5) PC Control design:

a) **PC Control description:** including a description on how you have implemented each branch instruction and what are the requirements (e.g. lookup tables) for that.

b) **Schematics:** hierarchically organized schematics of all components in your PC control and Dynamic Instruction Counter. Also, include schematics for integration with Instruction Memory.

6) CPU wrap up:

a) **Glue Logic description:** including a description for every logic that you used to connect the basic components together and their functionality/truth table.

b) **Schematics:** hierarchically organized schematics of all glue logic components in your CPU. Also, present your highest level of schematics here.

c) **Timing diagrams with clear annotations**. It should demonstrate correct operation of CPU on the input set as well as other important data. Once again, the timing diagrams should be annotated heavily. The inputs for data memory were posted on Piazza.

i) It should at a minimum show execution of the program through at least the first few iterations of each loop structure that you have, the cycle counter, the PC. It need not show the whole execution of the program (particularly if it takes over 100 cycles or so! – again, cut out some of the loop iterations. It, once again, should be heavily annotated so we can figure out what is going on.

ii) You MUST point out any particular instructions or overall execution that is not functioning. Give

your explanation of what you think is going wrong.

7) Other details that you need to note.

8) Answers to the Questions

In these questions we are not looking for you to convince us that your design is wonderful, but rather looking at how effectively you critique your own design.

a) Have you made any changes to your ISA from lab 1? What were they? Why did you make them? (we hope you didn't, but if you did, this is the place).
b) What are your dynamic instruction counts for program 1? program 2? Use the hardware cycle counter on the given input.
c) What could have been done differently to better optimize for dynamic instruction count? Give examples.
d) How successful were you at optimizing for ease of design and what was particularly difficult to design? Give examples.
e) What could you have done differently to better optimize for ease of design?
f) How easy/difficult would it be to extend your design to a multi-cycle implementation? a pipe-lined implementation? Give examples.
g) What might you have done differently if the priority was ease of assembly programming? Give examples.
h) What instruction takes the longest on your machine? This instruction would determine the cycle time of the processor. Use rough estimates. (e.g. assume each device introduces a constant delay).
i) What might you have done differently if the priority was short cycle time? Give examples.
j) State any known problems/bugs with the design in executing the programs. This will facilitate grading if we know problems ahead of time, and can allow you to receive more partial credit if something isn't working.
k) Reflect on this lab(1-3) experience.
    - What did you learn from this project?
    - What was the best thing about it?
    - What was the worst thing about it?
    - What advice would you give to someone taking this lab next semester?
    - What would you do as the professor / TA to make this lab a better experience?
    - How would you describe (in 3 sentences) the value of this lab experience in a job interview?

ECE 366 Lab 1-3 Final Report
CPU Design
2013 Fall

Kaushal Patel 679868139
Kai Zhao 670720413

Due Date: 2013 Nov 10
Lab Section: T8
TA: Shafagh Kamkar

1. ISA Overview

a) Introduction. The name of the architecture, overall philosophy, specific goals strived for.

The architecture name is "SW-8" (Square Width, 8-bit). This architecture is designed specifically for finding the square of a number and for finding the largest width of an array of numbers in the minimum number of instruction counts. This architecture strives to saves instruction counts by avoiding unnecessary reads, writes, comparisons, increments, branches, and intermediate steps.

b) Instruction description. Which instructions are supported, the format of encoding for each, and an example with machine code. Use well-organized table(s) to present your instruction design. The description should be detailed enough that someone could write an assembler (a program that creates machine code from assembly code) for it.

Machine Code	Instruction	Syntax	Operation	Explanation
000xxxxx	Load0	Load0 Rx	Rx = Mem(0)	Loads register x with data from Mem(0)
001xxxxx	Load32	Load32 Rx	Rx = Mem(x+32)	Loads register x with data in Mem(x+32)
010xxxxx	InitR1	InitR1	Rx = 0	Initialize R1 to 0
01100001	AndAddShiftR1	AndAddShiftR1	R1 = shift((rightmost bit of R2 AND R0) + R1)	The rightmost bit of R2 will be ANDed with R0, the product will be added to R1, the sum will be shifted to update the flag, and the shifted result will be stored to R1.
10000010	SLR2	SLR2	R2 = flag*2^7 + R2 / 2	Shift logic right of R2 and use carry flag as the leftmost bit of R2
10100001	StoreR1Mem1	StoreR1Mem1	Mem(1) = R1	Stores R1 to Mem(1)
10100010	SLR2StoreMem2	SLR2StoreMem2	Mem(2) = flag*2^7 + R2 / 2	Use the carry flag as the right most bit of a shift logic right of R2, and store to Mem(2)
10100011	WidestWidth	WidestWidth	Mem(3) = Widest(R0, R1, R2, R3, ..., R31)	Writes the widest width of all registers to Mem(3)
10100100	WidestNumber	WidestNumber	Mem(4) = Num(Widest(R0, R1, R2, R3, ..., R31))	Writes the number of the widest width of all registers to Mem(4)
10100101	WidestAddress	WidestAddress	Mem(5) = registerAddress(Widest(R0, R1, R2, R3, ..., R31)) + 32	Writes the register number of the widest width + 32 of all registers to Mem(5). +32 because there is an offset of 32 between memory and the registers
11111111	Halt	Halt		Halt

c) **Register design: How many registers are supported? Are they general-purpose or specialized? Is there anything special about your registers?**
32 registers are supported. They are all general-purpose registers. The registers data is a one to one mapping memory data with an offset. Therefore, it will be able to detect the location of the widest based on which register the widest was in.

d) **Memory and addressing modes: How large is your data memory? How many bits are needed for memory addresses? How are memory addresses constructed / calculated in your instructions (for load/store)? Give examples.**
The data memory is 64 bytes (0 through 63). 6 bits are needed for the memory address since its 0 to 63 bytes long. Memory addresses are not constructed because it is handled by the ISA. An adder will deal with the register offset in program widest. For example, knowing the offset of 32, if the program found the widest in register 5, then it know that Mem(5+32) contains the widest, so 37 will be written into Mem(5) for program 2.

e) **Control flow (branches): What types of branches are supported? How are the target addresses calculated? What is the maximum branch distance supported in your ISA? Give examples.**
No branching was supported. The target address is not applicable. The maximum distance supported in the ISA is 0. The ISA supports no examples of branching.

f) **P1 and P2 programs (assembly language); well commented. State any assumptions you make. You cannot assume anything about the values in registers or data memory, other than those specifically given, when the program starts. This means, for example, that if you need zeroes in registers or memory, you need to put them there.**
P1: Square

00000000	Load0 R00	# Loads the value to be squared into R0 (X)
01000001	InitR1	# Sets R1 (Z_H) to zeros
00000010	Load0 R02	# Loads the value to be squared into R2 (Z_L)
01100001	AndAddShiftR1	# AND the right most bit of R2 with R0, then add with r1, then shift, update flag, and finally store to R1
10000010	SLR2	# Use the flag to shift the bits of R2
01100001	AndAddShiftR1	# AND the right most bit of R2 with R0, then add with r1, then shift, update flag, and finally store to R1
10000010	SLR2	# Use the flag to shift the bits of R2
01100001	AndAddShiftR1	# AND the right most bit of R2 with R0, then add with r1, then shift, update flag, and finally store to R1
10000010	SLR2	# Use the flag to shift the bits of R2
01100001	AndAddShiftR1	# AND the right most bit of R2 with R0, then add with r1, then shift, update flag, and finally store to R1
10000010	SLR2	# Use the flag to shift the bits of R2
01100001	AndAddShiftR1	# AND the right most bit of R2 with R0, then add with r1, then shift, update flag, and finally store to R1
10000010	SLR2	# Use the flag to shift the bits of R2

01100001	AndAddShiftR1	# AND the right most bit of R2 with R0, then add with r1, then shift, update flag, and finally store to R1
10000010	SLR2	# Use the flag to shift the bits of R2
01100001	AndAddShiftR1	# AND the right most bit of R2 with R0, then add with r1, then shift, update flag, and finally store to R1
10000010	SLR2	# Use the flag to shift the bits of R2
01100001	AndAddShiftR1	# AND the right most bit of R2 with R0, then add with r1, then shift, update flag, and finally store to R1
10100001	StoreR1Mem1	# Stores R1 to Mem(1)
10100010	SLR2StoreMem2	# Use the carry flag as the right most bit of a shift logic right of R2, and store to Mem(2)

P2: Widest

00100000	Load32 R00	# Loads R00 with data from Mem(00+32)
00100001	Load32 R01	# Loads R01 with data from Mem(01+32)
00100010	Load32 R02	# Loads R02 with data from Mem(02+32)
00100011	Load32 R03	# Loads R03 with data from Mem(03+32)
00100100	Load32 R04	# Loads R04 with data from Mem(04+32)
00100101	Load32 R05	# Loads R05 with data from Mem(05+32)
00100110	Load32 R06	# Loads R06 with data from Mem(06+32)
00100111	Load32 R07	# Loads R07 with data from Mem(07+32)
00101000	Load32 R08	# Loads R08 with data from Mem(08+32)
00101001	Load32 R09	# Loads R09 with data from Mem(09+32)
00101010	Load32 R10	# Loads R10 with data from Mem(10+32)
00101011	Load32 R11	# Loads R11 with data from Mem(11+32)
00101100	Load32 R12	# Loads R12 with data from Mem(12+32)
00101101	Load32 R13	# Loads R13 with data from Mem(13+32)
00101110	Load32 R14	# Loads R14 with data from Mem(14+32)
00101111	Load32 R15	# Loads R15 with data from Mem(15+32)
00110000	Load32 R16	# Loads R16 with data from Mem(16+32)
00110001	Load32 R17	# Loads R17 with data from Mem(17+32)
00110010	Load32 R18	# Loads R18 with data from Mem(18+32)
00110011	Load32 R19	# Loads R19 with data from Mem(19+32)
00110100	Load32 R20	# Loads R20 with data from Mem(20+32)
00110101	Load32 R21	# Loads R21 with data from Mem(21+32)
00110110	Load32 R22	# Loads R22 with data from Mem(22+32)
00110111	Load32 R23	# Loads R23 with data from Mem(23+32)
00111000	Load32 R24	# Loads R24 with data from Mem(24+32)
00111001	Load32 R25	# Loads R25 with data from Mem(25+32)
00111010	Load32 R26	# Loads R26 with data from Mem(26+32)
00111011	Load32 R27	# Loads R27 with data from Mem(27+32)
00111100	Load32 R28	# Loads R28 with data from Mem(28+32)
00111101	Load32 R29	# Loads R29 with data from Mem(29+32)
00111110	Load32 R30	# Loads R30 with data from Mem(30+32)
00111111	Load32 R31	# Loads R31 with data from Mem(31+32)

10100011	WidestWidth	# Writes the widest width of all registers in memory location 3
10100100	WidestNumber	# Writes the number of the widest width of all registers in memory location 4
10100101	WidestAddress	# Writes the register number of the widest width + 32 of all registers in memory location 5

2. ALU Design

a) ALU operations: the list of all ALU operations, accompanied by the instructions they are relevant to. Describe your ALUop table and encoding details/challenges.

ALU operations

1. **Find Leftmost**: compute location of leftmost bit given 1 8-bit input
   Relevant to**:** finding width (ALU #5)
   Details**:** This uses the if-else-if structure. If the leftmost bit is 1, then return 0. Else if the left most bit is 0 and the 2nd from leftmost bit is 1, then return 2. Else if the left most bit is 0 and the 2nd from leftmost bit is 0 and the 3rd from leftmost bit is 1, then return 3. … If all bits are 0, then return 0.
   Challenges: The orientation of this ALU was originally wrong. Instead of finding the leftmost bit from the left, it originally found the leftmost bit from the right. This challenge was fixed by having it consistently wrong and rewiring the inputs of Width (ALU #5), which uses Find Leftmost (ALU #1).
2. **Find Rightmost**: compute location of rightmost bit given 1 8-bit input
   Relevant to: finding width (ALU #5)
3. **Subtractor**: subtracts 1 8-bit input from another
   Relevant to: comparator (ALU #4) by subtracting and checking if all bits are 0 and width (ALU #5) by subtracting leftmost and rightmost from 10
   Details: 2 4-bit subtractors were combined sequentially to make this 8-bit subtractor
4. **Comparator**: checks if 2 8-bit inputs are equal
   Relevant to: comparator (ALU #4) by subtracting and checking if all bits are 0 and width (ALU #5) by subtracting leftmost and rightmost from 10
   Details: this was originally built with 8 xnor gates. However, to help reduce the number of gates, this was revised to use the subtractor instead.
5. **Width**: takes 1 8-bit input and return the width
   Relevant to: widest program, need to find the width of elements of an array to compare
   Details: this ALU is 0x0a minus leftmost minus rightmost. The comparator is then used to check if the difference is equal to 0x0a, and if so, then the width is set to 0.
6. **Max**: takes 2 8-bit inputs and return the larger
   Relevant to: widest program, need to find widest (max) width
   Details: this uses the subtractor and checks the sign of the difference
7. **Sum**: takes 2 8-bit inputs and return the sum
   Relevant to: And Add Shift (ALU #11)
   Details: used the built-in 8-bit adder found in the libraries
8. **Shift Logic Right**: takes 1 8-bit input and return every bit shifted to the right
   Relevant to: And Add Shift (ALU #11) and square program (to shift logic right of Z_L)
   Details: no gates were used. This just directly connected 1 pin to another

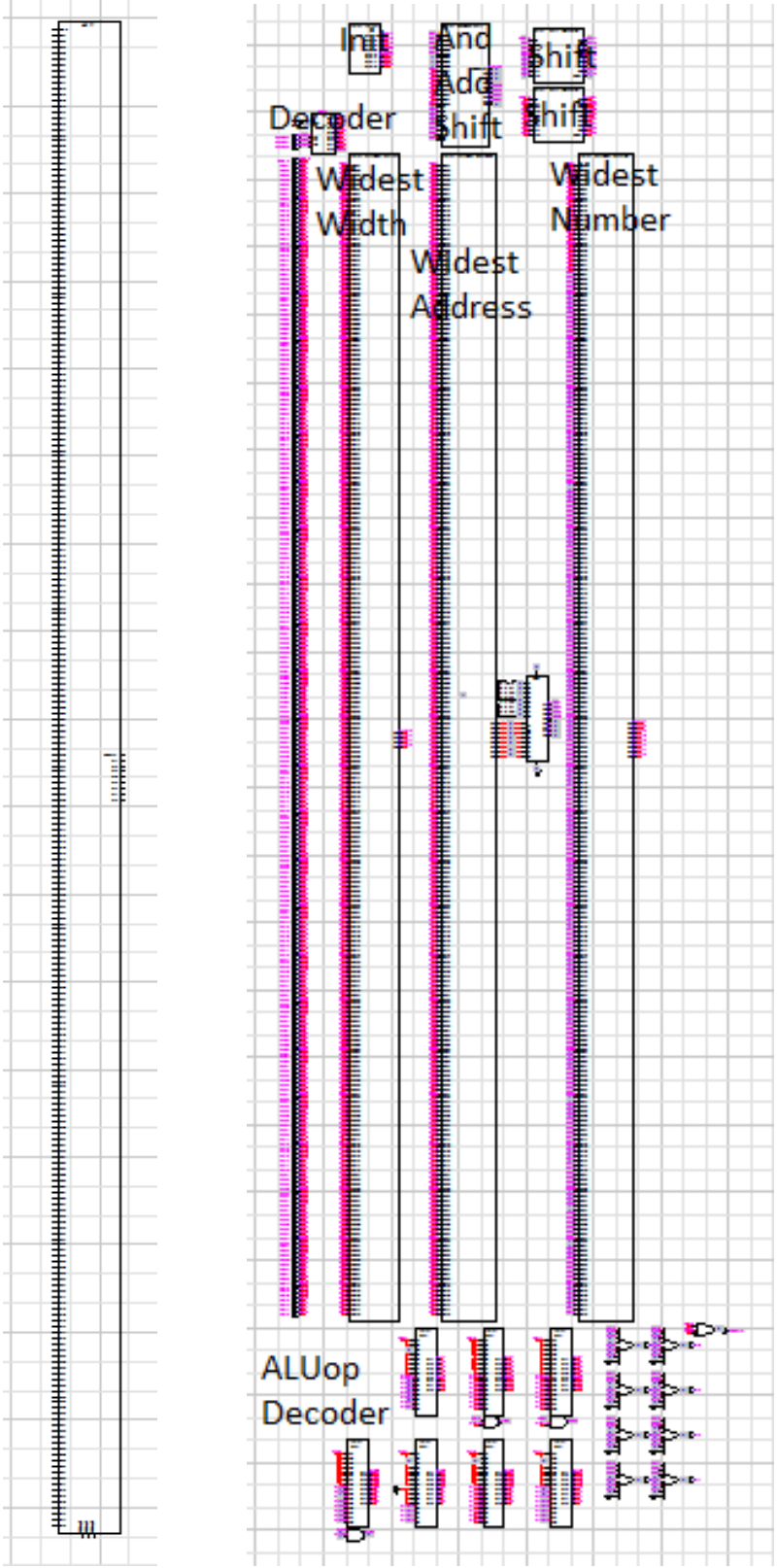
9. **And**: takes 2 8-bit inputs and return an 8-bit output of element-by-element AND operation
   Relevant to: And Add Shift (ALU #11)
   Details: 8 and gates were used
10. **Init**: takes no input and return an output of 0, used for initializing the registers
   Relevant to: square program, initialize Z_H to 0 because we cannot assume that the registers are initialized to 0
   Details: all of the outputs were tied to ground
11. **And Add Shift**: takes 3 inputs, ANDs the 1st input with the lowest bit of 2nd input, then adds to the 3rd input, and finally shifts right
   Relevant to: square program, , ANDs the lowest bit of Z_L with X, then adds to the Z_H, and finally shifts right to find the new Z_H
   Details: uses and (ALU #9), add (ALU #7), and shift (ALU #8)

Instruction	**ALUop**
Load0	xxx
Load32	xxx
InitR1	001
AndAddShiftR1	010
SLR2	011
StoreR1Mem1	xxx
SLR2StoreMem2	100
WidestWidth	101
WidestNumber	110
WidestAddress	111
Halt	xxx

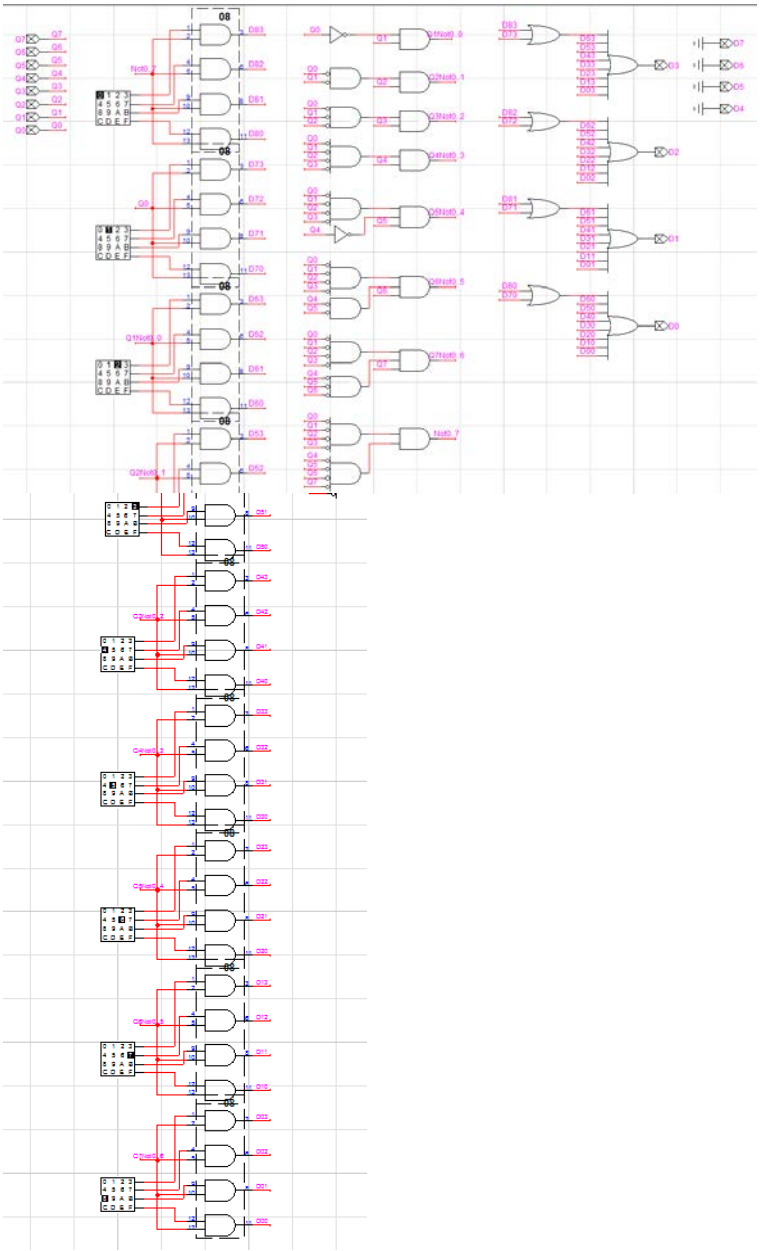
The ALUops that are xxx are don't cares because those instructions does not go through the ALU.

b) Schematics: hierarchically organized schematics of all components in your ALU.

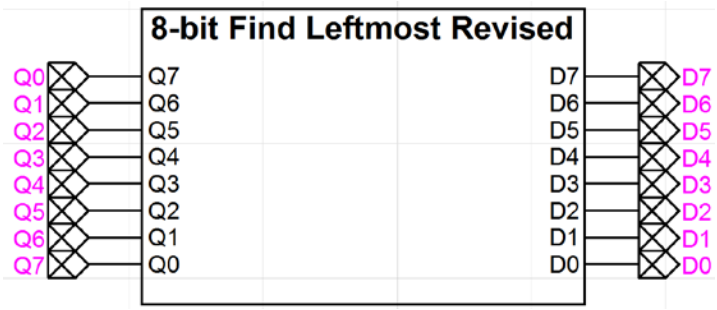
ALU High Level Schematic



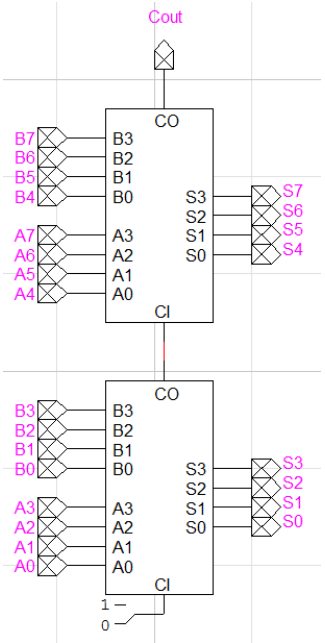
1. **Find Leftmost**



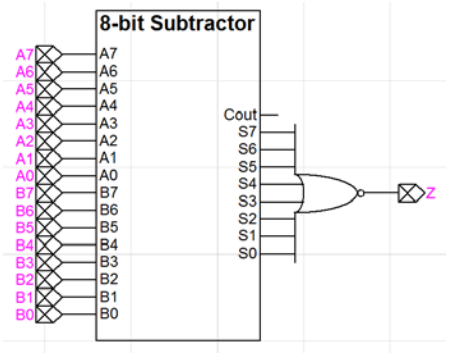
2. **Find Rightmost**



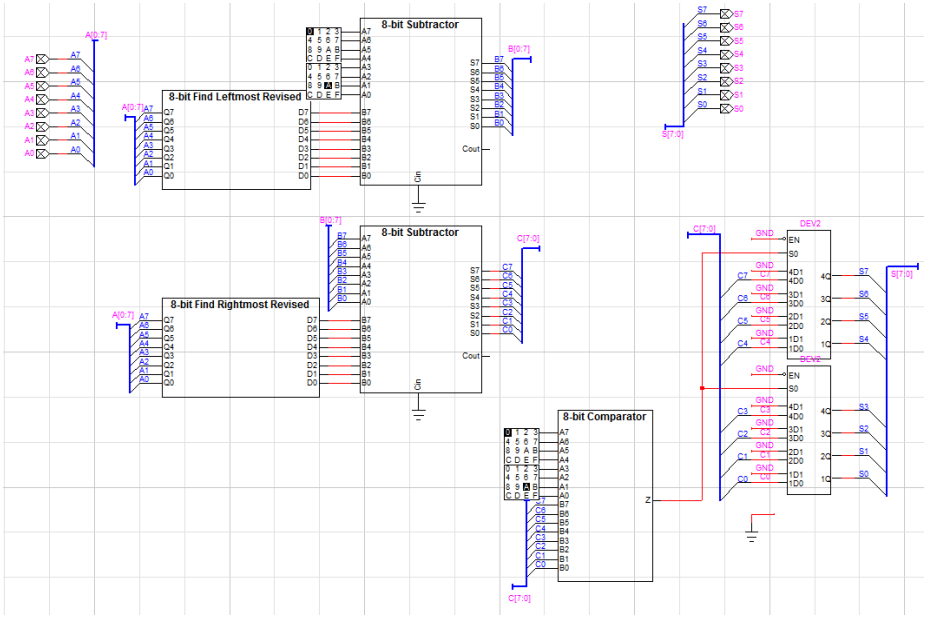
3. Subtractor



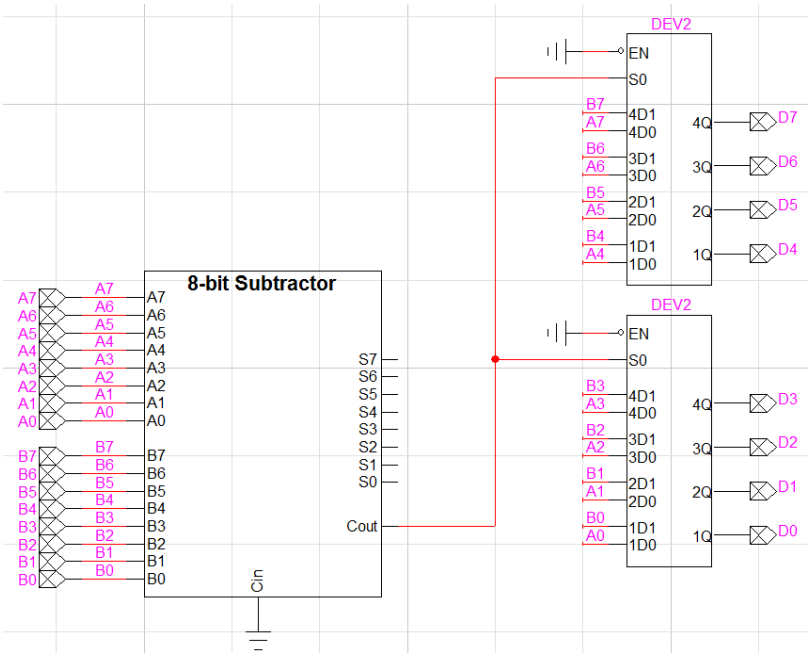
4. Comparator



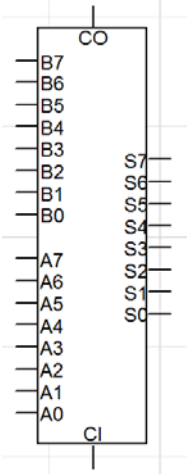
5. Width



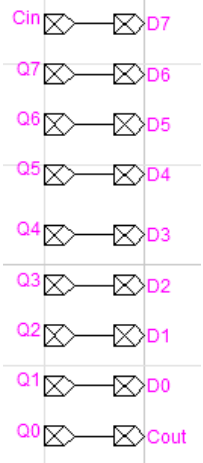
6. Max



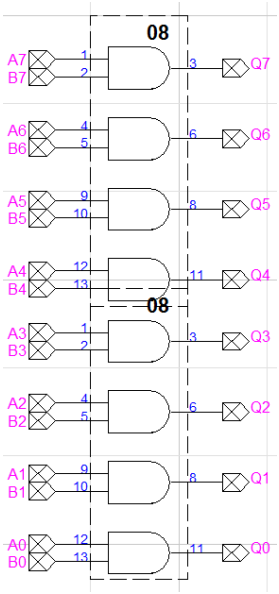
7. Sum



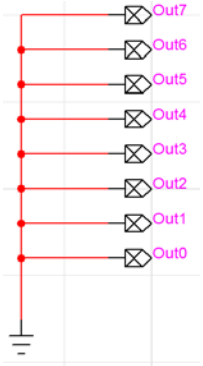
8. Shift Logic Right



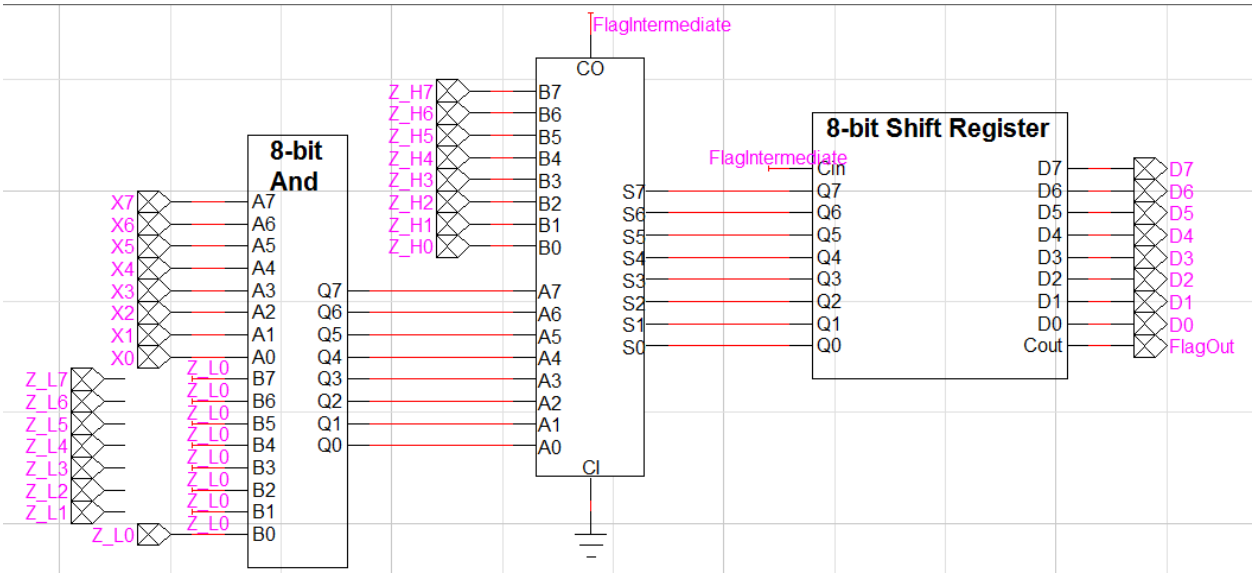
9. And



10. Init



11. And Add Shift



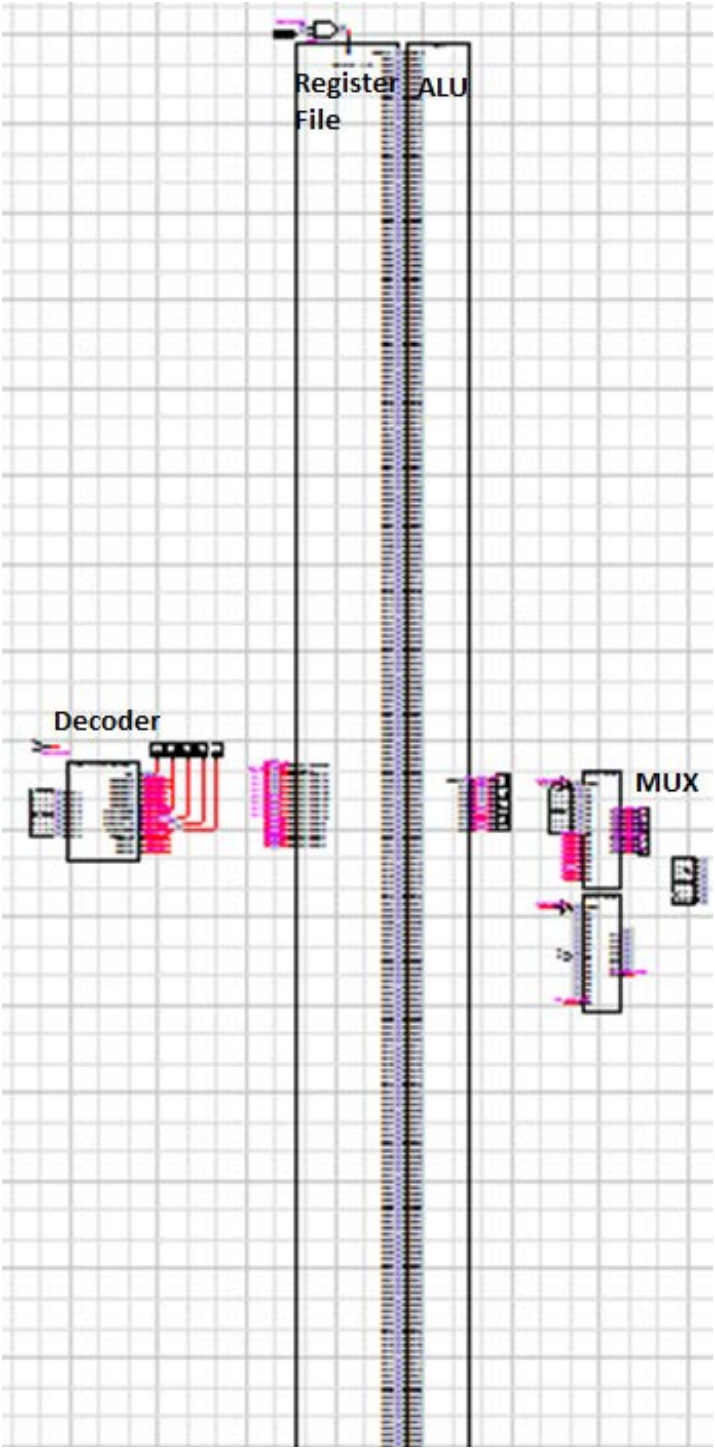
3. Register File Design

a) Registers types: a description of how many registers of which types.

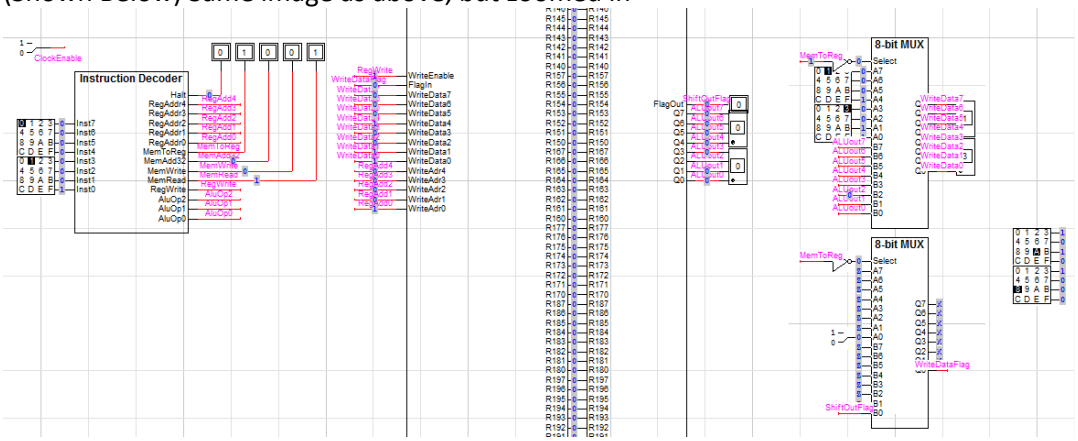
32 general purpose registers

b) Schematics: hierarchically organized schematics of all components in register file and its high level integration with ALU.

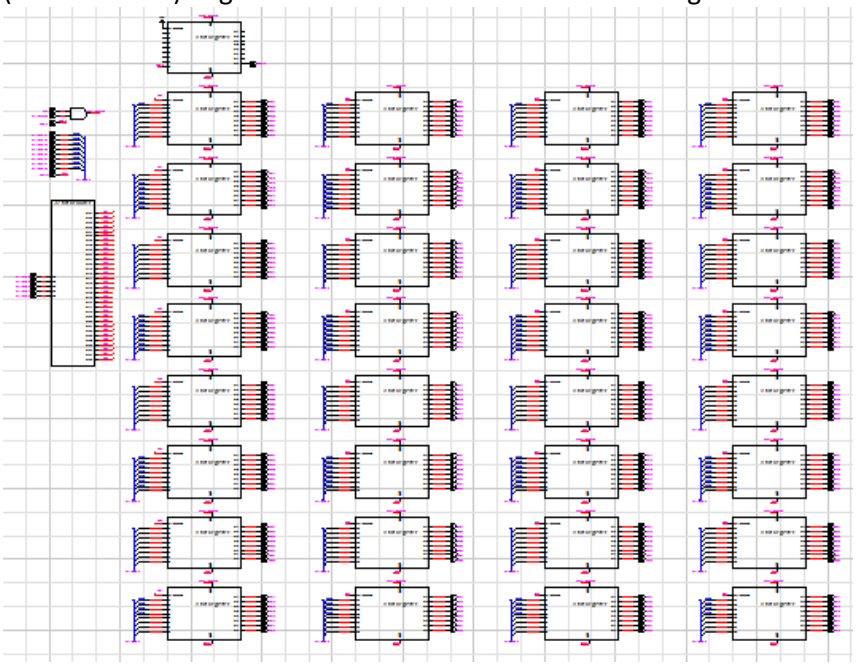
(Shown Below) High level schematic zoomed out with integration with Decoder and ALU



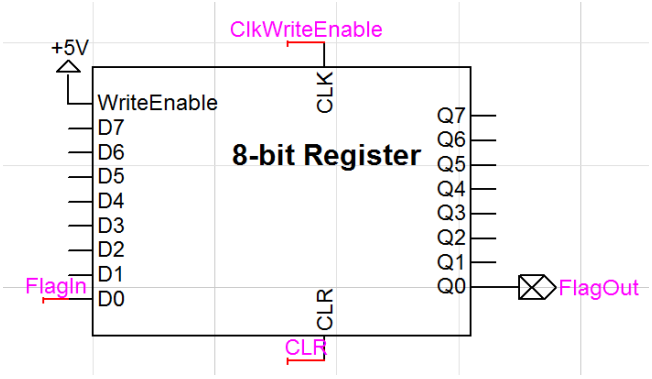
(Shown Below) Same image as above, but zoomed in



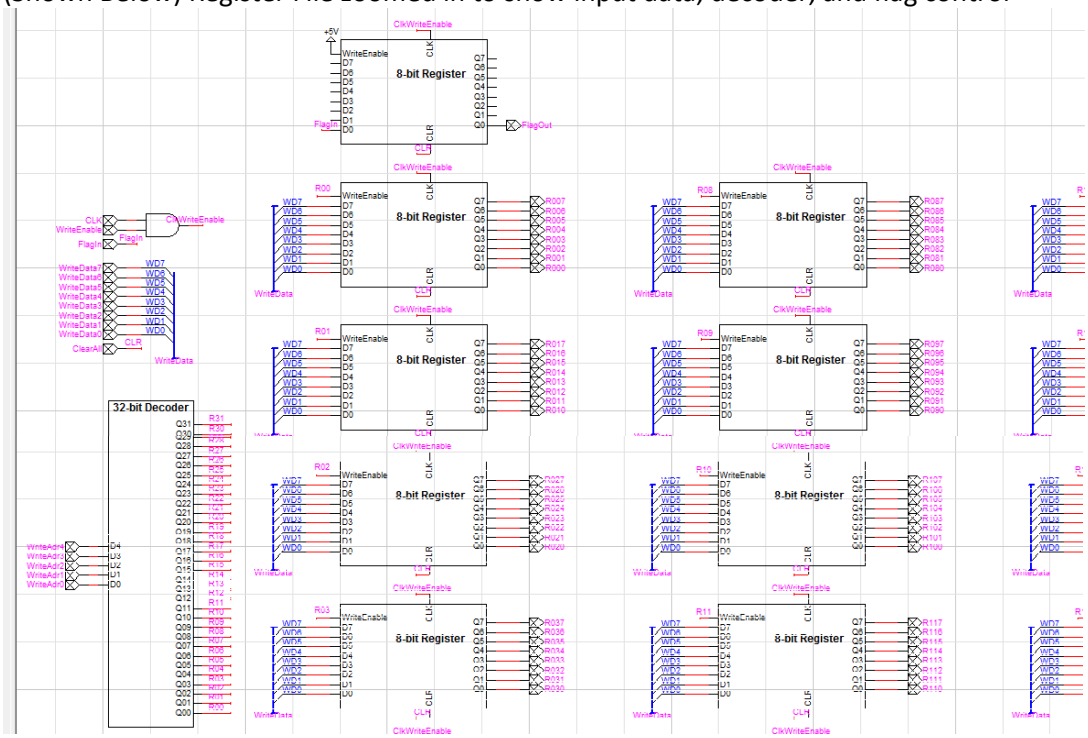
(Shown Below) Register File zoomed out to show all 32 registers



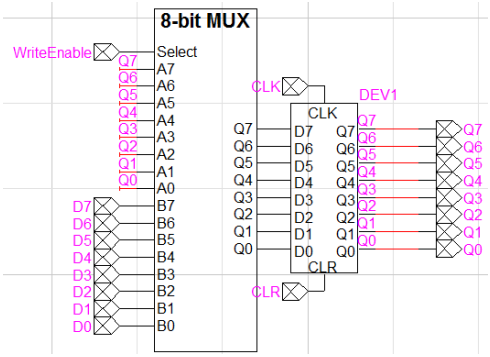
(Shown Below) Flag In and Flag Out



(Shown Below) Register File zoomed in to show input data, decoder, and flag control



(Shown Below) Low level schematic of each register



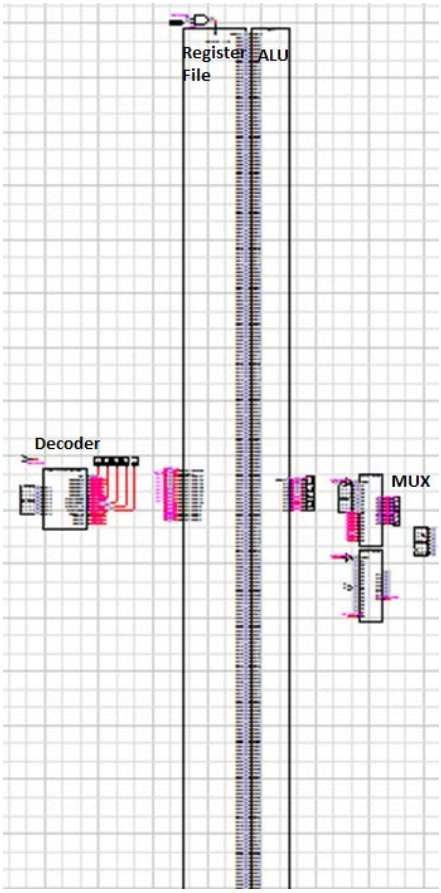
4. Instruction Decoder Design

a) Instruction Decoder description: including a description for each control signal and the associated truth table. Describe how you reduce number of gates required in your design.

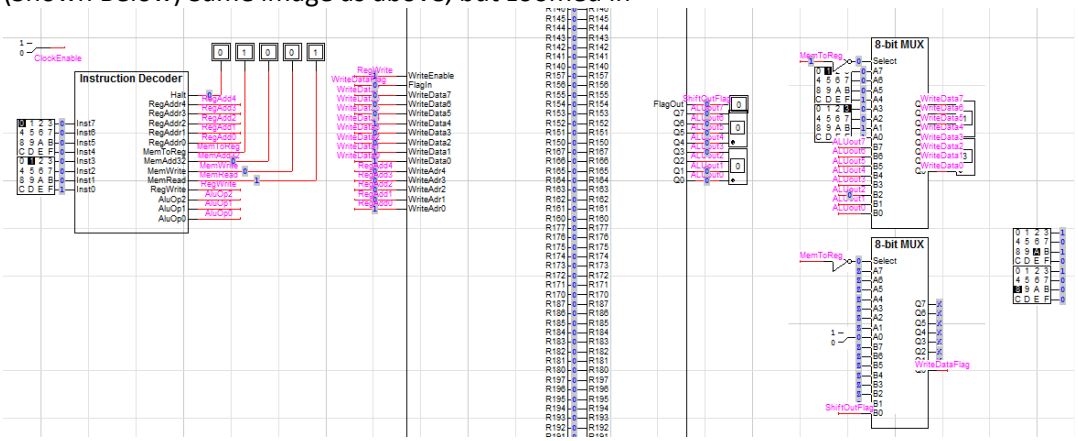
Instruction	Machine Code	ALU op	Halt	Mem Add32	Reg Write	MemRead	Mem Write	Mem ToReg	Reg Address
Load0	000xxxxx	xxx	0	0	1	1	0	1	xxxxx
Load32	001xxxxx	xxx	0	1	1	1	0	1	xxxxx
InitR1	010xxxxx	001	0	x	1	0	0	0	xxxxx
AndAddShiftR1	01100001	010	0	x	1	0	0	0	00001
SLR2	10000010	011	0	x	1	0	0	0	00010
StoreR1Mem1	10100001	xxx	0	x	0	0	1	0	Don't Care
SLR2StoreMem2	10100010	100	0	x	0	0	1	0	Don't Care
WidestWidth	10100011	101	0	x	0	0	1	0	Don't Care
WidestNumber	10100100	110	0	x	0	0	1	0	Don't Care
WidestAddress	10100101	111	0	x	0	0	1	0	Don't Care
Halt	11111111	xxx	1	x	0	0	0	0	Don't Care

b) Schematics: hierarchically organized schematics of all components in your instruction decoder and its high level integration with ALU and register file (different MUXs that you used …).

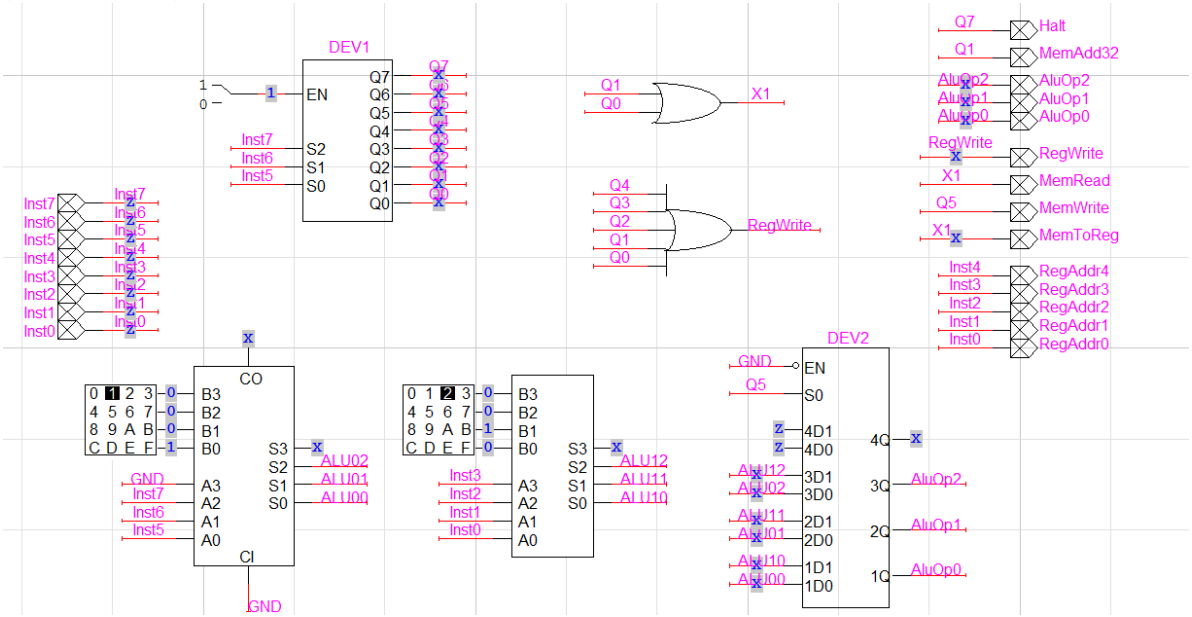
(Shown Below) High level schematic zoomed out with integration with Decoder and ALU



(Shown Below) Same image as above, but zoomed in



(Shown Below) Low level schematic of the decoder



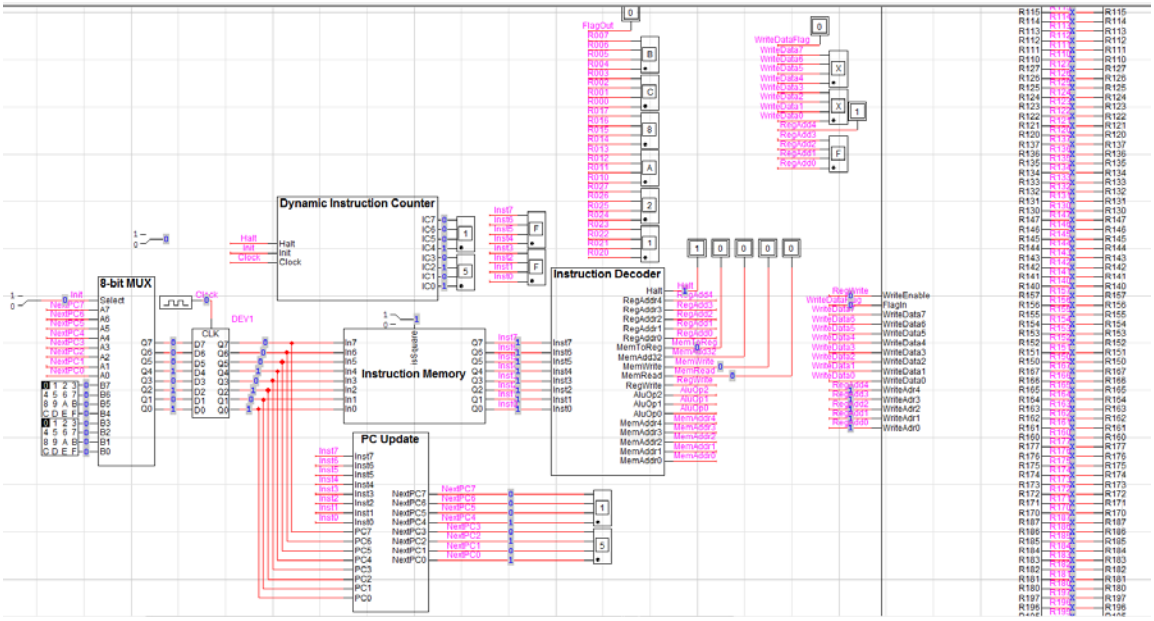
5. PC Control Design

a) PC Control description: including a description on how you have implemented each branch instruction and what are the requirements (e.g. lookup tables) for that.

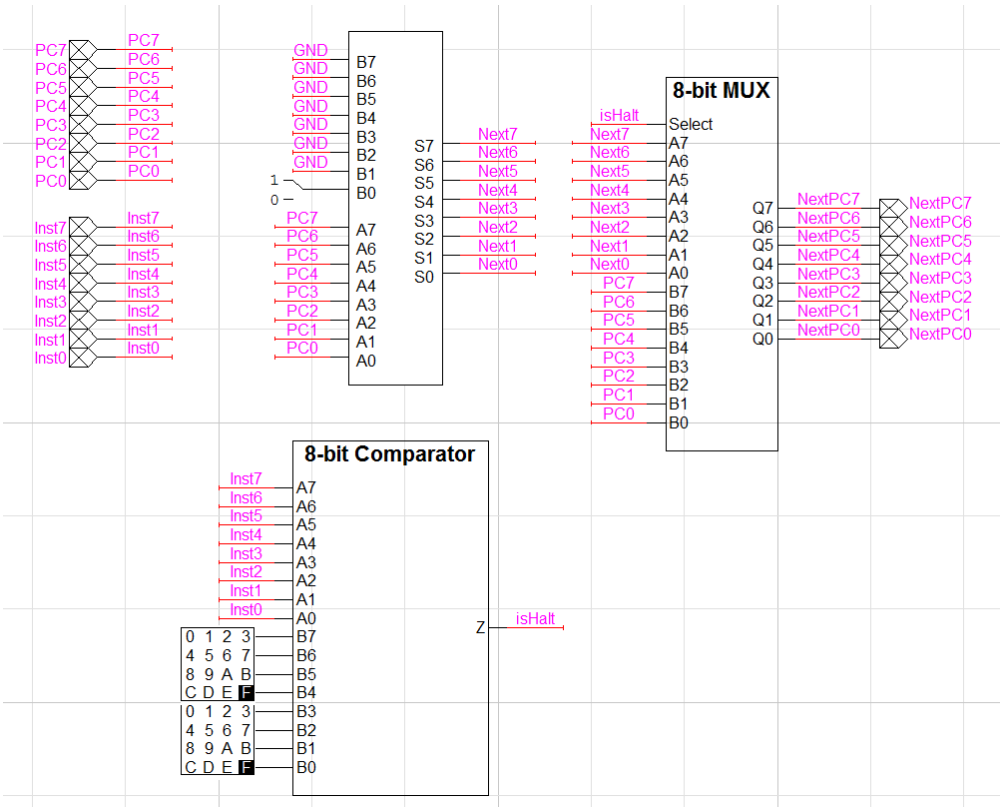
There was no branching instruction. The PC control just needs a comparator and a mux to determine whether to increment based on whether the instructor was halt.

b) Schematics: hierarchically organized schematics of all components in your PC control and Dynamic Instruction Counter. Also, include schematics for integration with Instruction Memory.

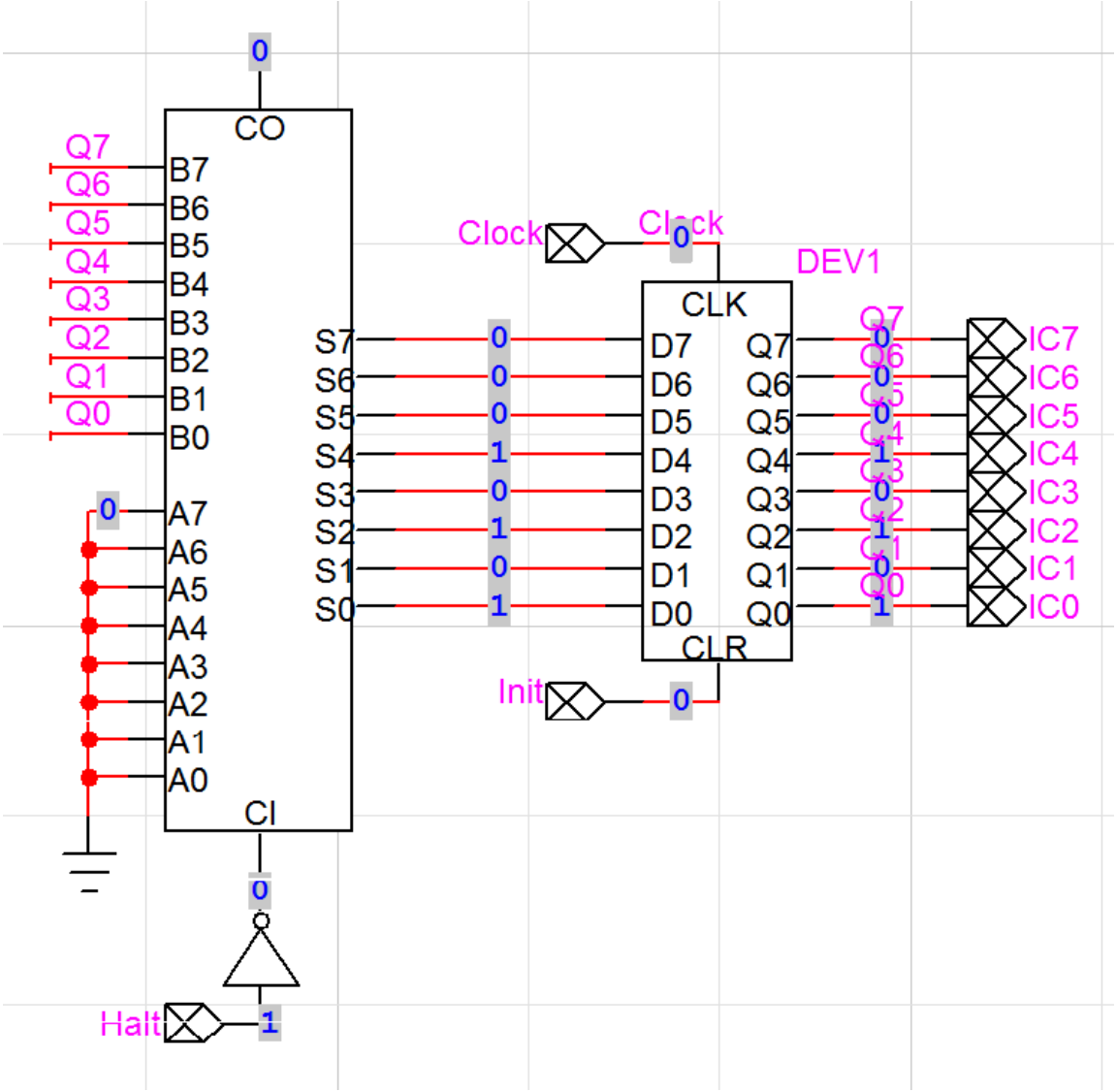
(Shown below) High level schematic of the PC control and Dynamic Instruction Counter



(Shown below) Low level schematic of the PC control



(Shown below) Low level schematic of the Dynamic Instruction Counter



6. CPU wrap up:

a) Glue Logic description: including a description for every logic that you used to connect the basic components together and their functionality/truth table.

Clock Inverter – Inverts the clock so that the register file will update at the falling edge of the clock to avoid any issues.

MemWrite – Inverts the WriteEnable input to the data memory due to the WriteEnable being activated when the input is low.

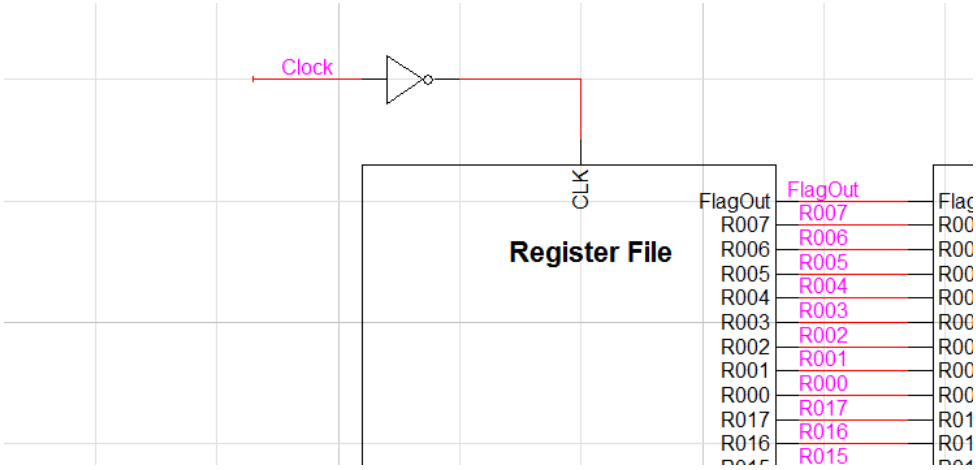
WriteMemDataMux – Decides what value will be sent into the data input port of the data memory

Inst2	Inst1	Inst0	Output
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

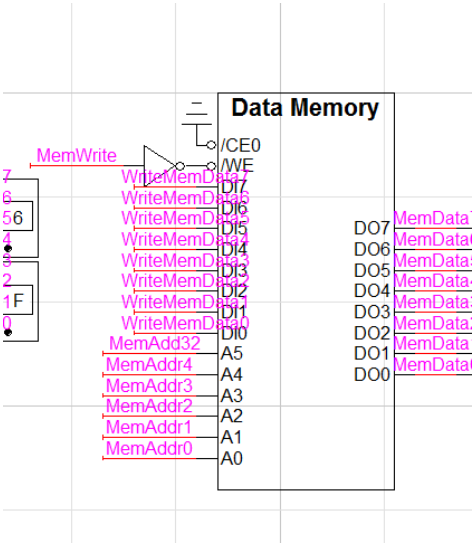
So, if the 3 LSBs of the instruction are 001, then the register 1 value will be sent to the data memory or the ALU output will be sent to the memory.

b) Schematics: hierarchically organized schematics of all glue logic components in your CPU. Also, present your highest level of schematics here.

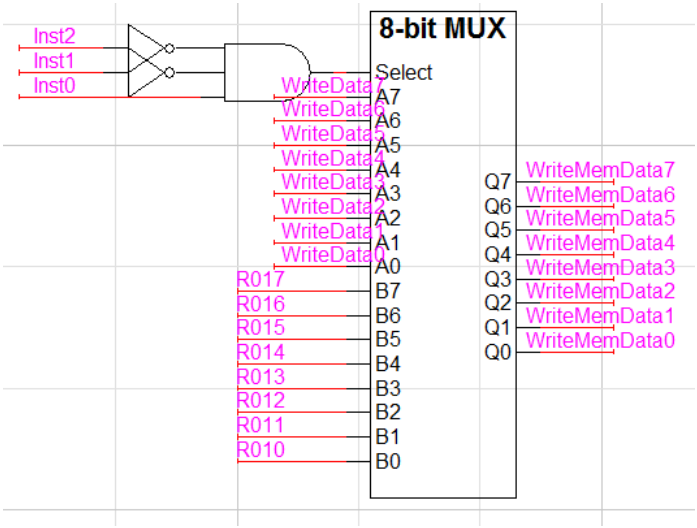
Clock Inverter



MemWrite



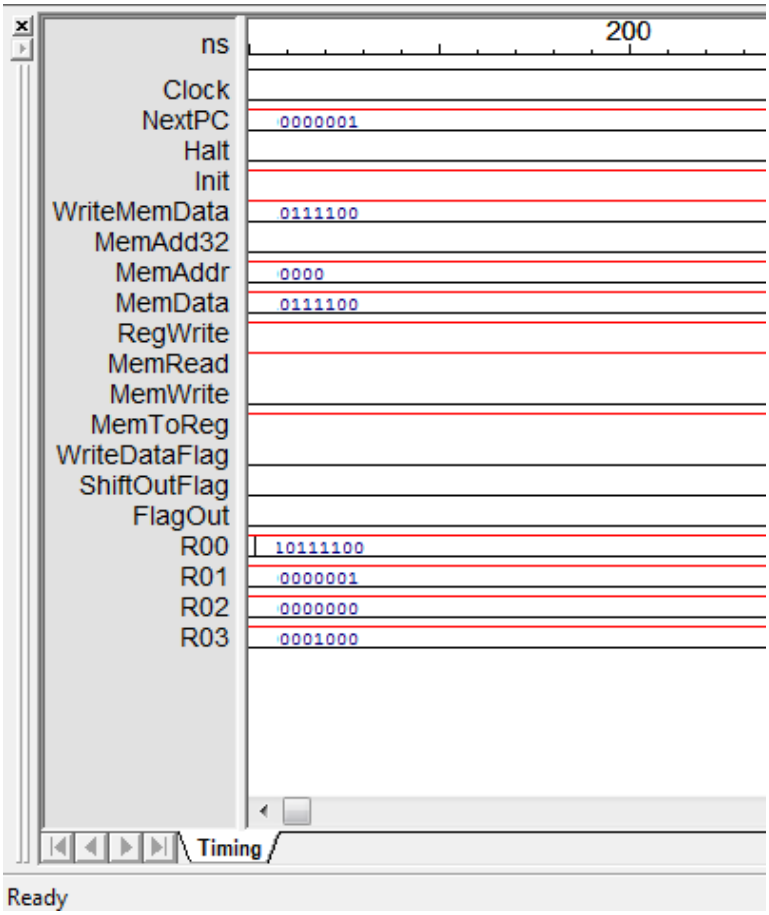
WriteMemDataMux



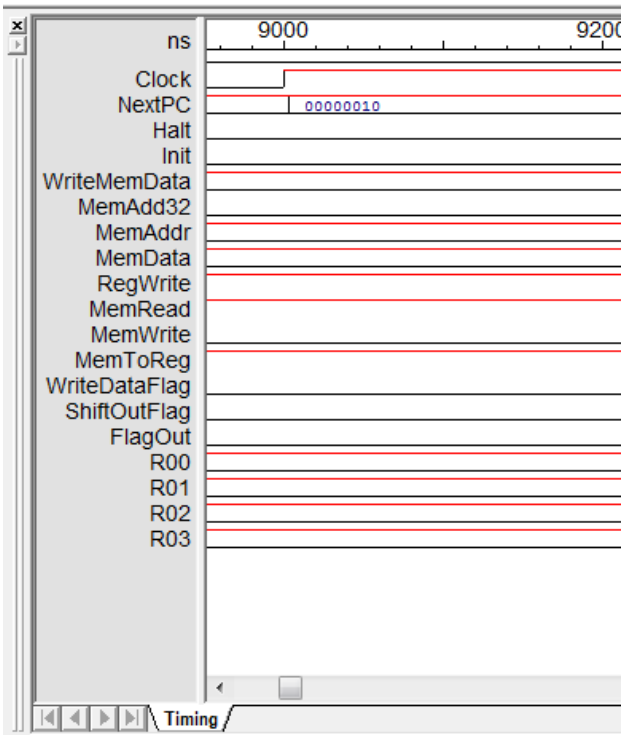
c) Timing diagrams with clear annotations. It should demonstrate correct operation of CPU on the input set as well as other important data. Once again, the timing diagrams should be annotated heavily. The inputs for data memory were posted on Piazza.

Square Program

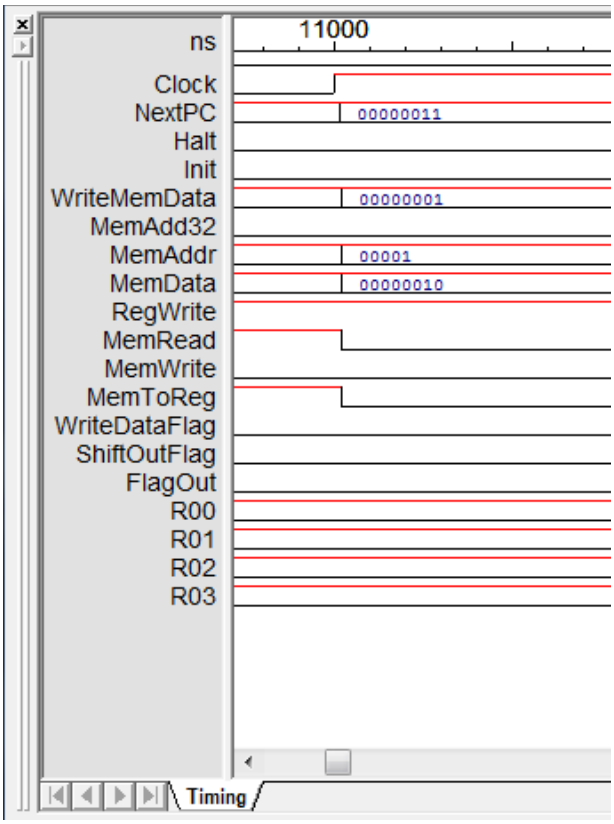
Clock 1



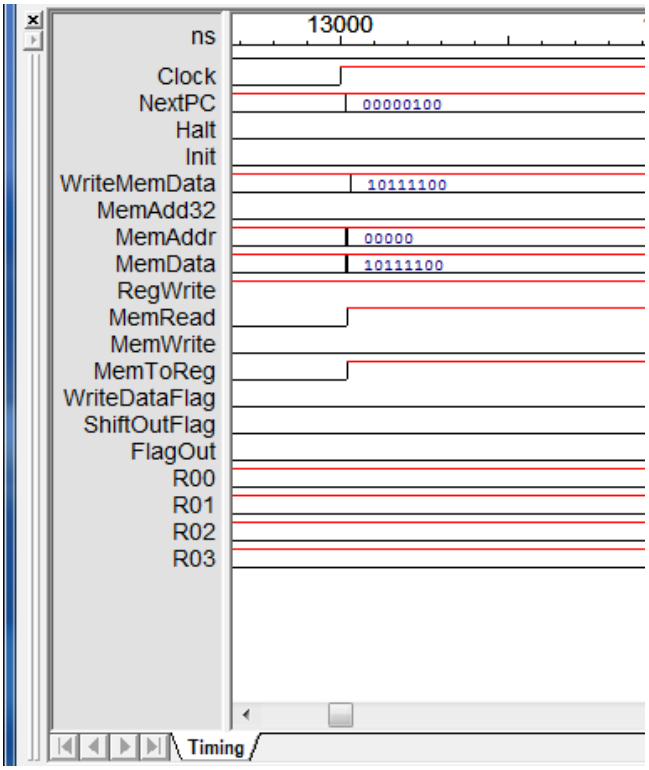
Clock 2



Clock 3



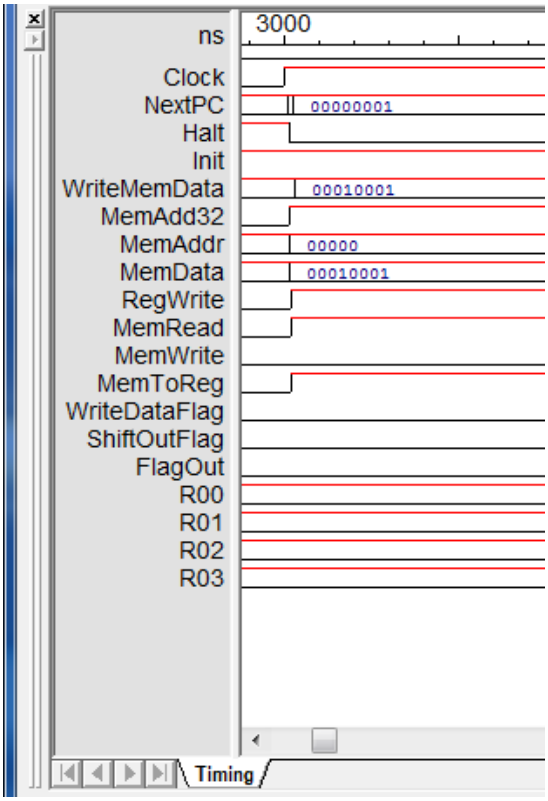
Clock 4



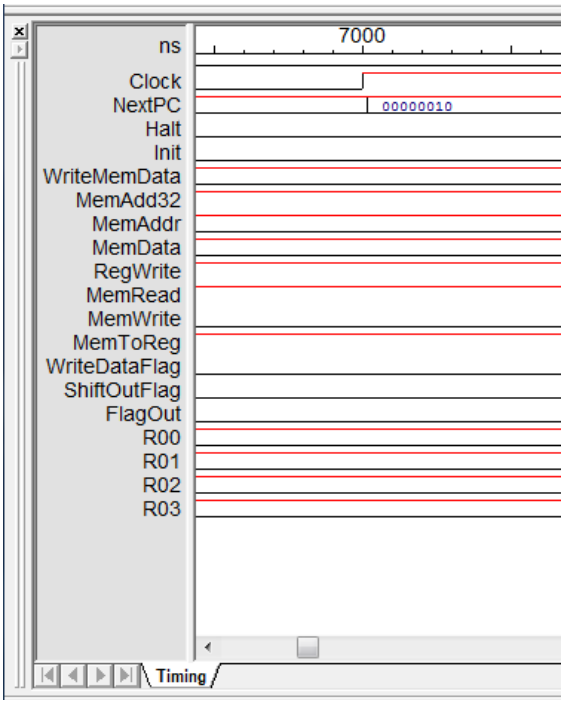
Clock 5

Widest Program

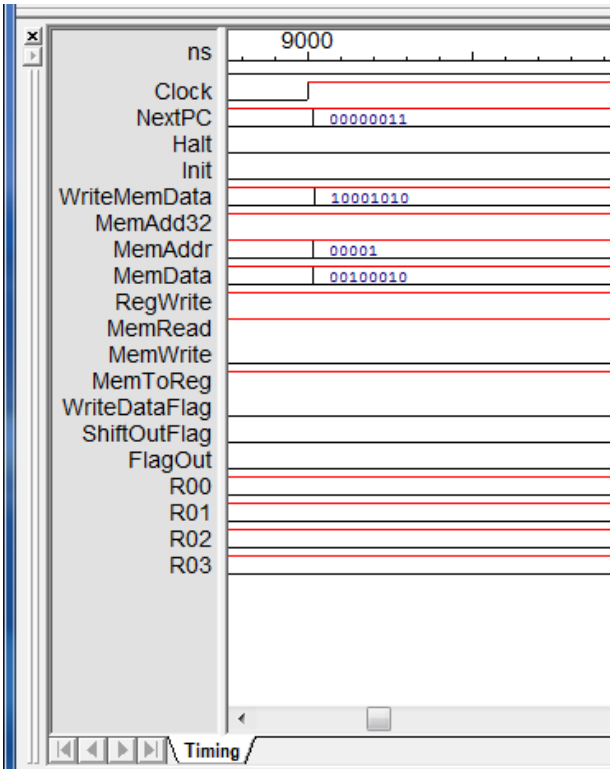
Clock 1



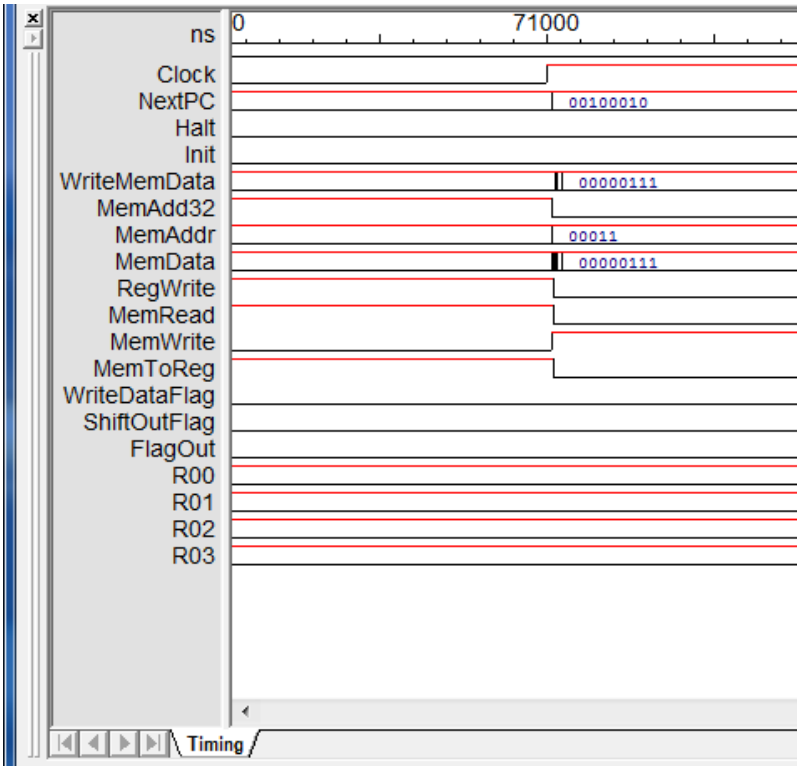
Clock 2



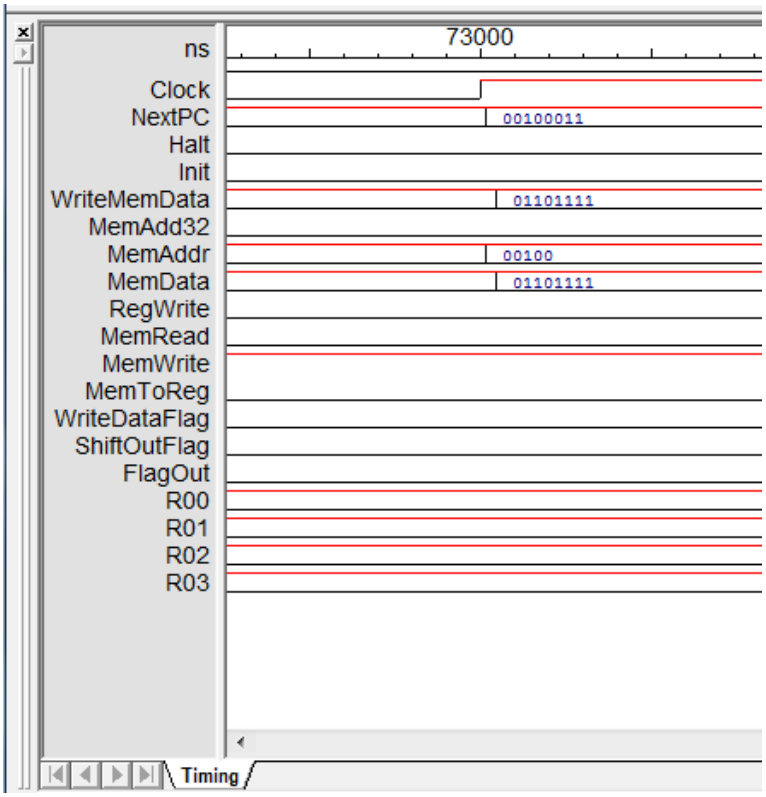
Clock 3



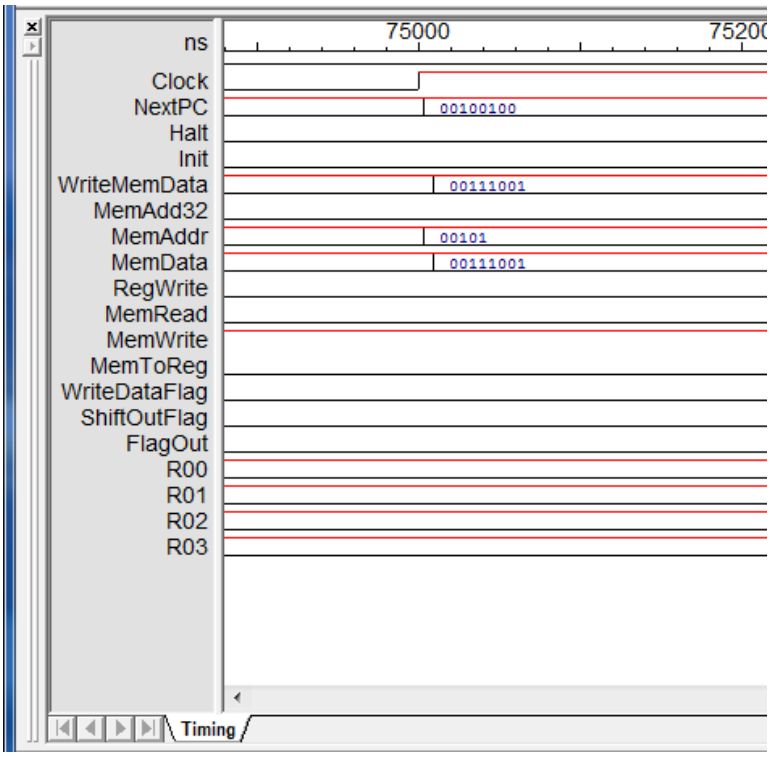
Clock 34



Clock 35



Clock 36



7. Other details that you need to note.

Our CPU seems to work.

Our CPU, which was optimized for dynamic instruction count and not hardware, will tend to operate slower than other group's CPU. We did not bother to optimize for hardware at all because it was not a priority and copying and pasting hardware in LogicWorks5 is simple. However, if we did have to optimize for hardware, then we will add another instruction to find the register address of the widest width. Then do the rest of the computation from that register as opposed to computing the width again for all 3 solutions in program 2.

In other words,

10100011	WidestWidth	WidestWidth	Mem(3) = *Widest(R0, R1, R2, R3, ..., R31)*	Writes the widest width of all registers to Mem(3)
10100100	WidestNumber	WidestNumber	Mem(4) = Num(*Widest(R0, R1, R2, R3, ..., R31)*)	Writes the number of the widest width of all registers to Mem(4)
10100101	WidestAddress	WidestAddress	Mem(5) = registerAddress(*Widest(R0, R1, R2, R3, ..., R31)*) + 32	Writes the register number of the widest width + 32 of all registers to Mem(5). +32 because there is an offset of 32 between memory and the registers

will be changed to

10100000	WidestRegister	WidestRegister	R32 = registerAddress(Widest(R0, R1, R2, R3, ..., R31))	Writes the register number of the widest width into R32
10100011	WidestWidth	WidestWidth	Mem(3) = Width(R(R32))	Writes the width register that R32 points to in Mem(3)
10100100	WidestNumber	WidestNumber	Mem(4) = R(R32)	Writes the number register that R32 points to in Mem(4)
10100101	WidestAddress	WidestAddress	Mem(5) = R32+32	Writes R32 + 32 to Mem(5). +32 converts from register address to memory address

.

Our software and hardware design grows linearly with the array size and data word size. If another word is added in the array, then our design will require just another register. If the word length is increased by 1, then the leftmost and rightmost device will just have to check another bit. The other designs such as "waterfall" will require a higher complexity to accommodate for another bit and PROM will require double the memory.

8. Answers to Questions

a) Have you made any changes to your ISA from lab 1? What were they? Why did you make them? (we hope you didn't, but if you did, this is the place).

Yes, but only minor changes. The 3 bit op-codes were rearranged between the instructions to make the ISA look cleaner. The last 3 bits of the instructions that stores the results to memory were changed so that those 3 bits can feed directly to the memory address.

b) What are your dynamic instruction counts for program 1? program 2? Use the hardware cycle counter on the given input.

The dynamic instruction counter for program 1 is 21 (3 to initialize, 15 for 7.5 cycles of AndAddShift, 2 to store the results to memory, and 1 to halt).

The dynamic instruction counter for program 2 is 36 (32 to load from memory to register, 3 to store the results to memory, and 1 to halt).

c) What could have been done differently to better optimize for dynamic instruction count? Give examples .

Doing more multiplication cycles per an instruction could possibly optimize for dynamic instruction count. For example, AndAddShift could be changed to AndAddShiftAndAddShift to reduce the number of multiplication cycles required by half.

d) How successful were you at optimizing for ease of design and what was particularly difficult to design? Give examples.

We were somewhat successful in optimizing for ease of design. For example, finding the rightmost bit uses the same device as finding the leftmost bit with the inputs reversed. Furthermore, finding the largest width (in the 3 instructions that computes and stores the solution in program 2) is just simply copying and pasting in the ALU. The 32 registers were particularly tedious to design because they each require a different wire names.

e) What could you have done differently to better optimize for ease of design?

We could have used 4 registers along with some looping in the ISA to make the Register file design easier, but it will make the PC counter design more difficult. After viewing presentations from other groups, using PROM to store the width of each value will make designing the ALU easier by avoiding devices used to find the leftmost, find the rightmost, and find the width.

f) How easy/difficult would it be to extend your design to a multi-cycle implementation? a pipe-lined implementation? Give examples.

Extending the design to a multi-cycle implementation will require breaking the ALU into multiple steps including 1) find the width of each register, 2) finding the largest width, 3) finding the register with the largest width, 4) extracting data from the register with the largest width. However, the multi-cycle implementation will require more hardware and registers to store the intermediate results.

Extending the design to a pipelined implementation will require the ALU to wait until data is finished writing to all registers. However, a pipelined implementation is ineffective because the ALU has the longest delay.

g) What might you have done differently if the priority was ease of assembly programming? Give examples.

Depending on the programmer's experience loops in the program will make the assembly programming easier by avoiding having to copy and paste the algorithm for each program. Therefore, a register will be added to point to the address to load from memory to register. This register will allow users to load data from memory to the corresponding register, and then increment the point. This register will make writing the assembly code easier because the user will not have to type the register address manually.

h) What instruction takes the longest on your machine? This instruction would determine the cycle time of the processor. Use rough estimates. (e.g. assume each device introduces a constant delay).

The longest instruction is widest address because it has to 1) find and compare the width of all registers to find the widest one, 2) check the widest width against every register until the width matches, 3) extra the corresponding address from the register.

Instruction fetch: 3n s
Instruction decoder: 3n s
Register read: 6n s
ALU:
    Find widest width:
        Find width of each register: 15n s
        Compare each width: 10n s
    Find register with widest width: 20n s
    Get the address of the register with widest width: 3n s
    Get the widest width address from the ALU: 2n s
    Add 32: 1 n s
Data memory write: 2n s
Total delay required: 65n s

i) What might you have done differently if the priority was short cycle time? Give examples.

Comparing all 32 registers requires a tournament format of at least 5 rounds. Comparing 2 registers at a time will only require 1 round. Therefore, if the priority was a shorter cycle time, then program 2 will loop through the data, compare the width of current widest and the next data in the loop, and extract data (width, number, and address) individually. Meanwhile, program 1 will split AndAddShift to And, Add, and Shift.

j) State any known problems/bugs with the design in executing the programs. This will facilitate grading if we know problems ahead of time, and can allow you to receive more partial credit if something isn't working.

There are no known problems with the design or with executing the programs.

k) Reflect on this lab(1-3) experience.

What did you learn from this project? Lab 1-3 taught us how to design the hardware and software so that combining the two will be simple.

What was the best thing about it?

The best thing was having no jumping/branching lab 3 and, which made the simulator, PC updater, and control signals very easy to design and implement.

What was the worst thing about it?

The worst thing was the register file and ALU in lab 2, which required supporting 32 registers and naming each data line individually within the ALU.

What advice would you give to someone taking this lab next semester?

Advice we can give to someone taking this lab is to plan ahead all the steps that are needed to implement something, because changes are more difficult to implement the farther you get into the project and files will be out of sync. Also, try to balance the burden between hardware and software, so that both can be done without too many difficulties.

What would you do as the professor / TA to make this lab a better experience?

We would not have done much different. Professor Noohi demonstrated how to use Logic Works and demonstrated how to build the register file, which is exactly what we had trouble with. Importing the data into memory could have been clarified a bit better in class, only because there was a discussion on Piazza about it. TA Shafagh clarified several of the components that were expected in the CPU so we knew what we had to do to finish building the processor.

How would you describe (in 3 sentences) the value of this lab experience in a job interview?

Creating a good design is great, but the good designs always leads to a lot of difficulties. The best learning experience comes from encountering those difficulties, looking into them, and implementing a fix or an alternative design. This lab taught us how to combine software and hardware, essential for understanding computer organization and computer architecture.