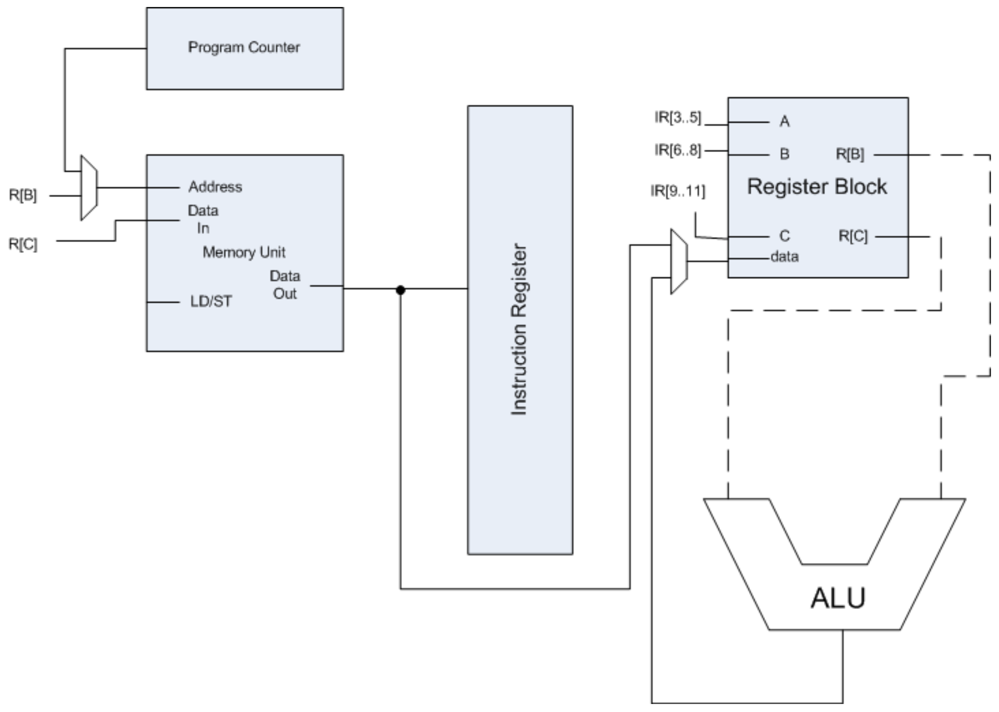
ECS 154A 12-bit CPU

The purpose of this lab is to build a simple processor without a Control Unit that is 12 bits wide and can do various register transfer and ALU operations over a common bus.

Here is a rough outline of the overall CPU design.



The CPU consists of 4 components:

1. **ALU**
   You have already built a 4-bit ALU in Assignment 1, you will need to expand the ALU into 12-bits. If your ALU was not working correctly, please repair it for this assignment. In the figure above all ALU operations would take in the contents of registers B and C, perform the operation, and then output it back to the main CPU, at which point the result is written to register A.
2. **Register Bank**
   You will build eight 12-bit parallel-load registers from D flip flops. You are allowed to use the built in D flip-flops here. All registers should be initialized to 0.
   Just for reminding, each of the registers has a common clock and distinct input and output bits. Each register also has a write enable input.
3. **Memory**
   The memory unit is available in LogiSim titled RAM under the Memory menu. The memory unit should be specified as a 12-bit address, 12-bit data length, and separate load and store data ports (You can specify these specifications on the left bar of Logisim).
   The memory takes a clock cycle to store data, and does so on the rising edge. In order to accommodate this, and avoid interpreting data as an instruction you will want to use an Instruction Register.
4. **Program Counter (PC)**

The program counter points to the next instruction in memory. It should increment at the end of an instruction cycle in order for the program to move on to the next instruction.

A hint: Multiplexor is your friend. You might want to use multiplexors to select between the registers to output from your register bank. You will use multiplexors to select the input address to memory, too.

Operation Description

Once we build the CPU, we need to program it. As we all know, in computers, higher level languages eventually get converted to assembly instructions and finally to binary machine code. This is exactly what we will have to do for our CPU, too. Our instructions are 12 bits each and divided in the following manner.

Operation (bits 9 – 11)	The operation the CPU is to perform, opcodes are given in the table below.
Register A (bits 6 – 8)	The destination register for arithmetic operations and the Store operation.
Register B (bits 3 – 5)	An operand, output from the Register bank
Register C (bits 0 – 2)	An operand, output from the Register bank

We define 8 instructions for the CPU:

Operation	OpCode	Additional Instruction
NOT	000	R[A] = NOT R[B]
OR	001	R[A] = R[B] OR R[C]
AND	010	R[A] = R[B] AND R[C]
ADD	011	R[A] = R[B] + R[C]
SUB	100	R[A] = R[B] – R[C]
Store	101	M[R[B]] = R[C]
Load	110	R[A] = M[R[B]]
Move	111	R[A] = R[B]

Testing your circuit

For this lab, I will provide a small sample program to test your CPU (shown below), however I **highly recommend** you create additional instructions to test your circuit. When creating an instruction you will need to input it as hex. This can be done by first creating the binary representation, and then simply converting that representation into hex. For example if I want to perform R[1] = R[4] + R[0] the instruction would be: (OP) 011 (Reg A) 001 (Reg B) 100 (Reg C) 000 = 011001100000, where the MSB is on the left and the LSB is on the right. This translates to 660h in hex, which is the value you would input to memory. Once you have input the instructions into memory you can save the image of the program in order to reinitialize on demand. The memory should change during your program execution if you are performing any Store commands. Be sure to include the memory image of this program in your submission as it is the bare minimum needed to get credit**. Full credit will be given to groups whose CPUs can run an additional program provided during interactive grading.**

Initialize M[FFFh] = 004h, all subsequent instructions should being at 000h, which is the initial value of your PC.

Operation	**Binary**
NOT R0,R1	000 000 001 101
LD R1,R0	110 001 000 111
ADD R4,R0,R1	011 100 000 001
AND R5,R0,R4	010 101 000 100
LD R1,R4	110 001 100 000
NOT R4,R1	000 100 001 000
ST R0,R5	101 000 000 101
ADD R6,R0,R0	011 110 000 000
ST R6,R1	101 000 110 001
ST R4,R6	101 000 100 110

At the end of this program If would check the store locations. This should end with the following memory contents:

M[FFFh] = 003h

M[FFEh] = 144h

M[EBBh] = FFEh

Please send e-mails to yfhong@ucdavis.edu if you find any errors within this text. Thank you.