CE 12 Lab Report 2

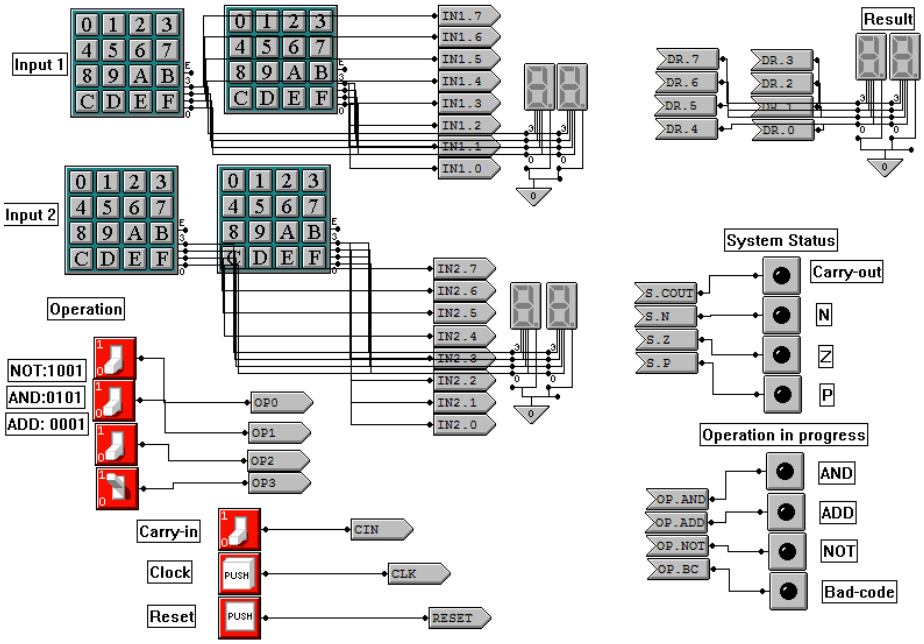
Arithmetic Logic Units

Brandon Gomez

Lab Sec: 2

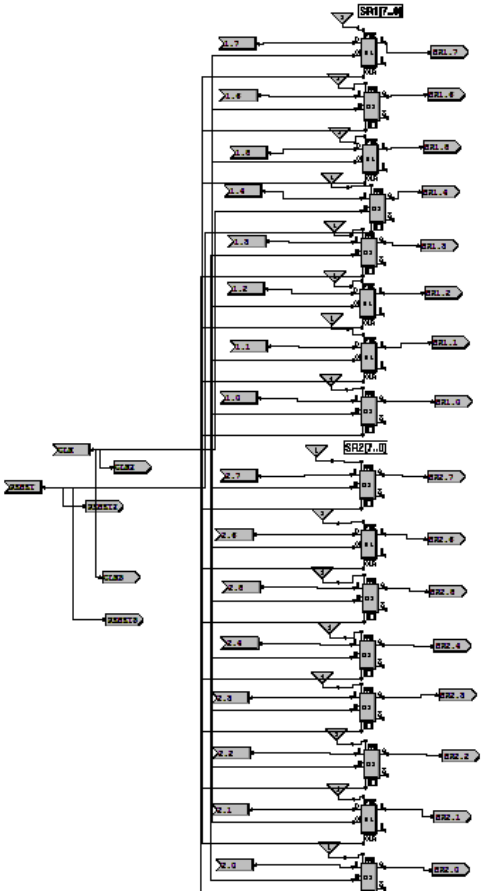
10.21.14

DESCRIPTION:
The purpose of this lab is to get familiar to how an 8-bit ALU works and how flips flops work. The ALU must be able to do three operations and those were NOT, AND, and ADD. The LC-3 ALU is normally 16-bits but for our lab, we are just doing an 8-bit one. The Lab can be divided into many parts.



MATERIALS:
Two 8-bit inputs, One 8-bit output , Two keypads for 8-bit input , Three 7-segment displays (2 for input, 1 for output), A bunch of switches for the opcode, A bunch of LEDs, One button for clock, One button for reset, One switch for carry-in, D Flip Flops for the input and output, power for the flip flops, AND OR NOT logic gates, 8 Muxs for the operation selector.

Part A: SR1 and SR2, Clock, Reset
The input of the 8-bit keypads is captured in D flip flops with triggered edges and presets. The D flip flops save the bits until you reset the ALU. The clock is connect to the D flip flops and the mux. When the clock starts, the d flip flop can be written on. I used a edged triggered d flip flop so you have to press the clock twice in order to get the appropriate operation. The Reset is connected to every D flip flop so the flip flop can know when to store a bit. If you press reset, the D flip flop will go back to 0.

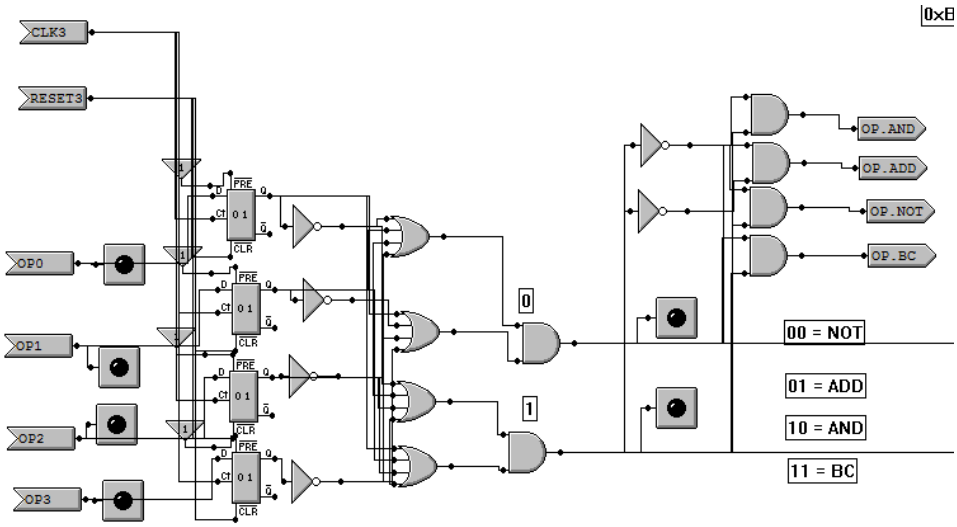


Part B: Opcodes

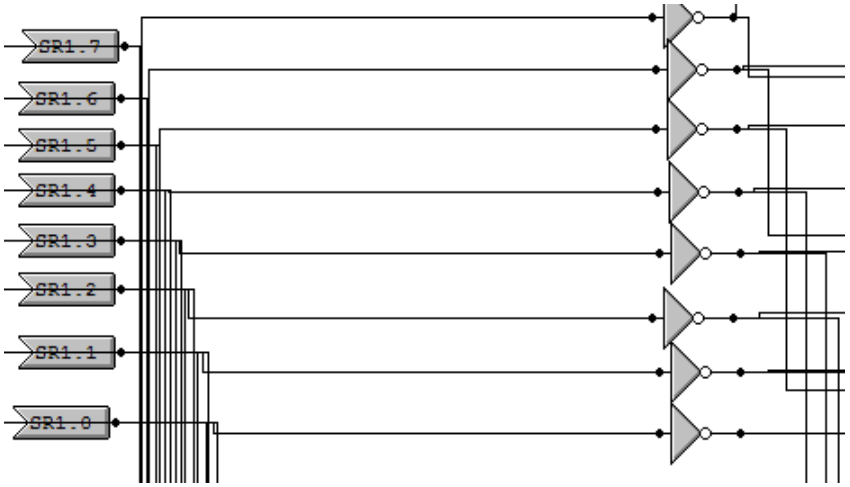
In order for the user to be able to use NOT, AND, and ADD, I used a IR[3..0] so the user can pick their operation. If the user input a wrong operation, the LED output will display a BC. The operations in progress will display on LEDs on the first page. In order to do this I had to get the product of sums of the operations from the truth table:

ADD 0001 10
NOT 1001 00
AND 0101 01
BC else 11

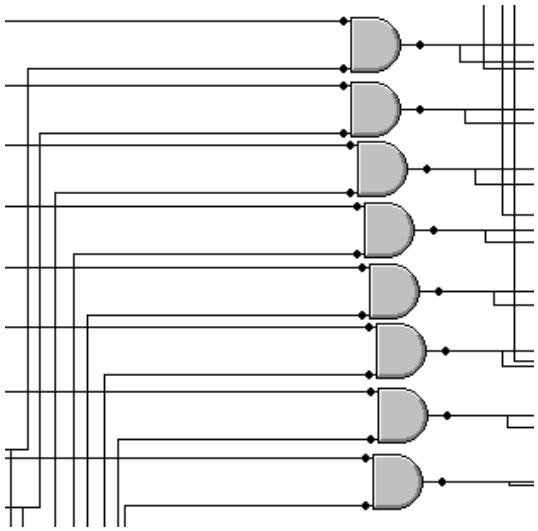
I got A'BCD' + AB'CD' and A'BCD' + ABCD' and implemented it in the form of logic gates using ORs and ANDs:



The NOT operand is takes data from SR1 and inverts it. This is done by taking SR1 and use the NOT gate.



The AND operand compares bits from SR1 and SR2 and ANDs them. If both bits have a 1 then the result will be a 1, else the result will be a zero for that bit. This is done by taking the corresponding bits from SR1 and SR2 and use a AND gate.



The ADD operand takes the first bits of SR1 and SR2 and compares them with the CIN and then decides if the sum will be a 1 or a 0 and if the COUT will be a 1 or a zero. Here is the truth table for A, B, CIN, SUM, aand COUT:

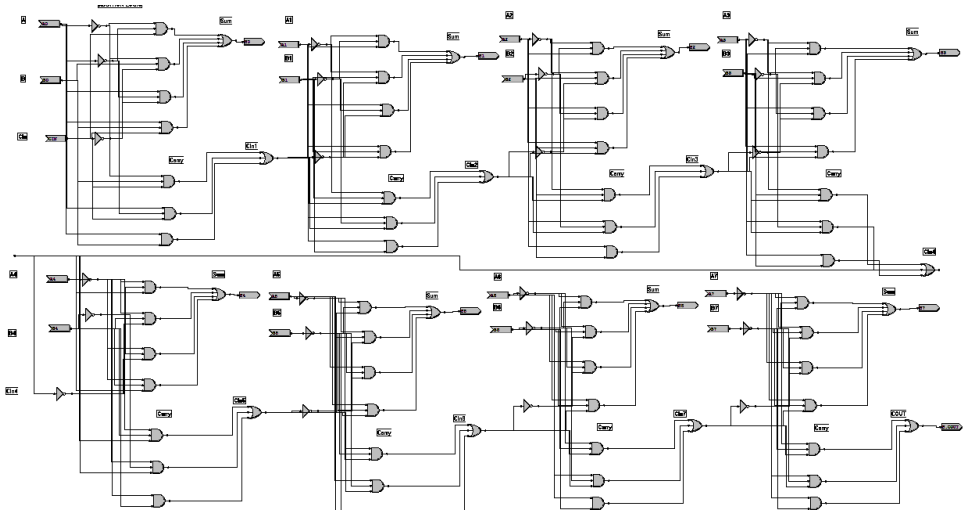
A	B	CIN	S	COUT
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

In addition, with this I was able to use sum of products to get:

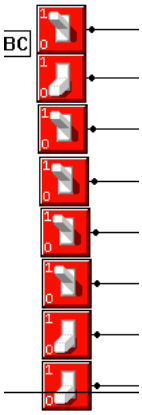
S = A’B’CIN + A’BCIN’ + AB’CIN’ + ABCIN

COUT = A’BCIN + AB’CIN + ABCIN’ + ABCIN

And with that I was able to implement it into logic form and get a sum and carry out for one bit, so I had to do this eight times in order to get the sum and carry out for eight bits.

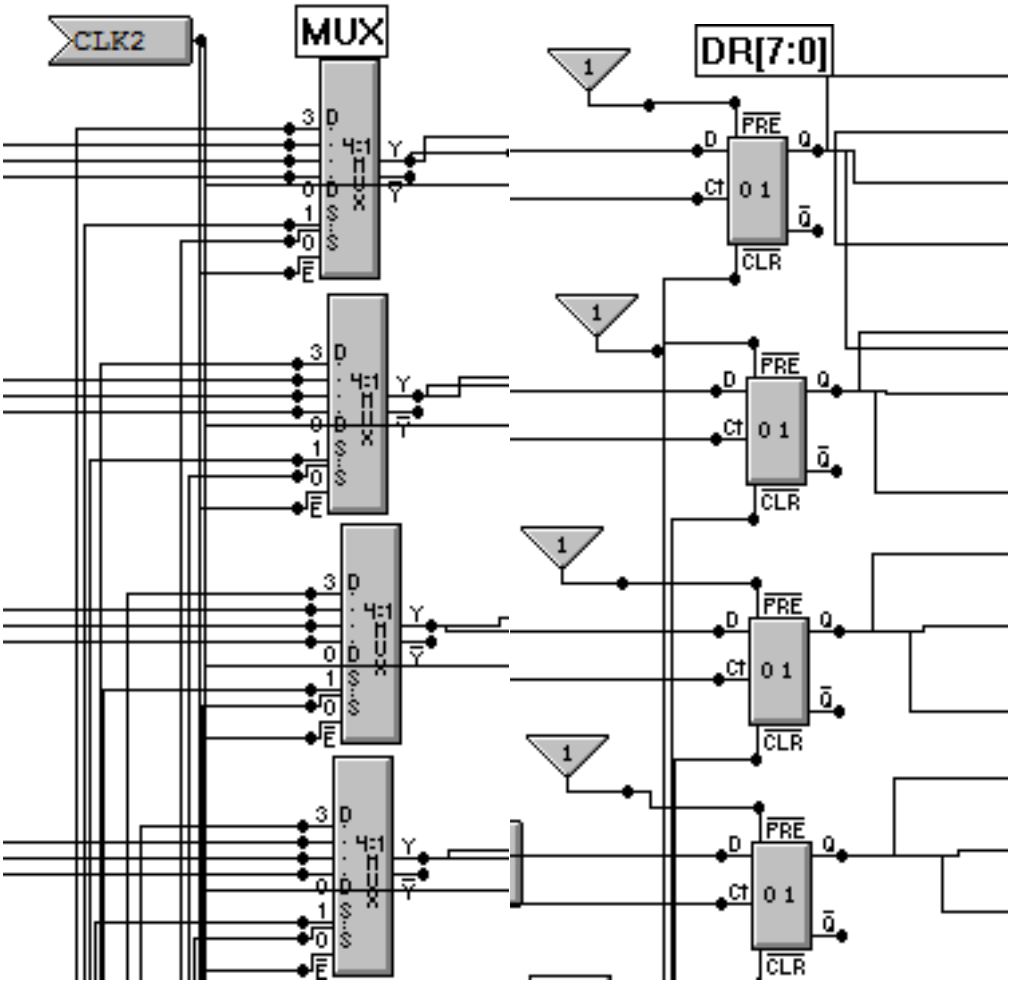


The BC operand is for when the user inputs operation that is not NOT, AND or ADD, then the program displays BC on the output leds. Having the code 10111100 and switches performs this.



Part C: MUX and results

The results of the operations are connected to 8 muxs and the muxs takes the opcodes from the user in order to know with of the three operations and bad code to go to D flip flops. Those flip flops then store the bits and display them on LEDs. With the results, I was able to connect logic gates in order to find if the result is a ZERO, POSITIVE, NEGATIVE and if the result has a carry output. This is for to know is the operation made the two numbers wrap around.



RESLUTS:

NOT: When the user picks a number from SR1, it is converted to binary and those bits are then inverted. For example, if the user selects the number 4, it will be converted to 00000100 and then the bits are inverted to become 11111011. If the user has selected the NOT opcode, then the inverted number will display to the output LEDs.

AND: When the user picks a number from SR1 and SR2, it is converted to binary. It then compares the bits of each corresponding location and ANDs them. If the user selected the number 4 and 5 then is turned to binary, which is 00000100 and 00000101. These are then ANDed and the results will be 00000100 and if the user has selected the AND operand, then the result on the LED will be 4.

ADD: When the user picks a number from SR1 and SR2, it is converted to binary. It then compares the bits of each corresponding location and ADDs them. If the user selected the number 4 and 5 then is turned to binary, which is 00000100 and 00000101. These are then added to be the binary number 00001001, and the number displayed will be 9.

BC: If the user selects any operation that is not NOT, AND, or ADD, then BC will display on the LED output.

NEGATIVE, ZERO, POSITIVE, CARRY:
If the user's output is negative then the corresponding LED will light and same with the positive numbers. If the user input all 0s then the ZERO LED will light up. If the user's two numbers are added and it overflows (FF + FF) then the carry out LED will light up. If the user inputs a wrong opcode then the bad code LED will light up.

CONCLUSION:
In conclusion, the ALU is a complicated piece of machinery, but if I dissect it and put it into smaller parts, it is just small problems that I already know how to do. Sequential logic can be useful because it can store information for you and can be an easy reset with an easy use of one button. The SUB instruction is not in this LC-3 because it is quite simple to implement it. We just need to get the complement of the number and add it then add 1 to the result. This should get us the SUB instruction. NAND is not one of the operations because it is simple to implement it. In order to get a NAND instruction in to this ALU we will have to expand our opcode and expand our Muxs since they only allow 4 operation. We will then do the same thing as the AND operation but have a NAND gates instead. Or we can just invert our current AND results.