Processor Description: 8-bit single CPU with branch, jump, slt, sgt and function calls. A carry look-ahead adder for 8-bit inputs is also included.

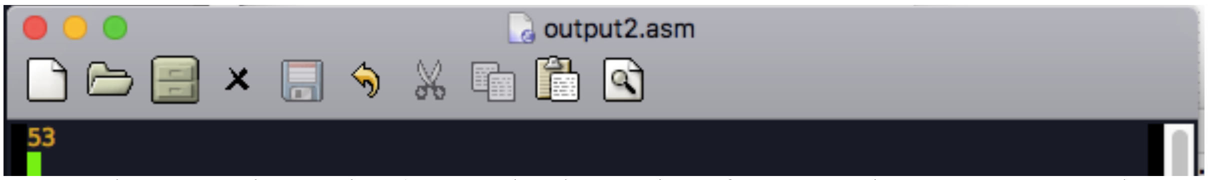
Grade Sheet for successfully implemented features

Feature	Expected Value
Basic: ALU	10
Basic: R and I type	10
Basic: LW and SW	10
Basic: Assembler	20
Basic: Writeup	20
A la Carte	
Support for SLT/SGT	10
Carry Look-Ahead Adder	10
Branch-if-equal (beq)	10
Jump (j) Instructions	10
Support for function calls	20
Total	130

1/ Assembler

Written in python, convert assembly code into hexadecimal machine language, each instruction is 14-bit long (the same as Lab4). The output file can be loaded directly to Logisim's instruction memory.

add $1 $2 $3 -> expected: 00000 001 010 011 = 53



Test on the reverse instruction (I use a simpler version of your test since my processor does not support intermediate number bigger than +3 and less than -4; it does the same job of switching data memory, I write this version so I can prove its correctness using my CPU.

```
#Reverse contents in data memory
addi $3 $0 3
addi $4 $3 1
addi $5 $4 1

lw $1 0 ($0)
lw $2 0 ($5)
sw $1 0 ($5)
sw $2 0 ($0)

lw $1 1 ($0)
lw $2 0 ($4)
sw $1 0 ($4)
sw $2 1 ($0)

lw $1 2 ($0)
lw $2 3 ($0)
sw $1 3 ($3)
sw $2 2 ($0)
```

```
20c3
2119
2161
2840
28a8
3068
3080
2841
28a0
3060
3081
2842
2883
305b
3082
```

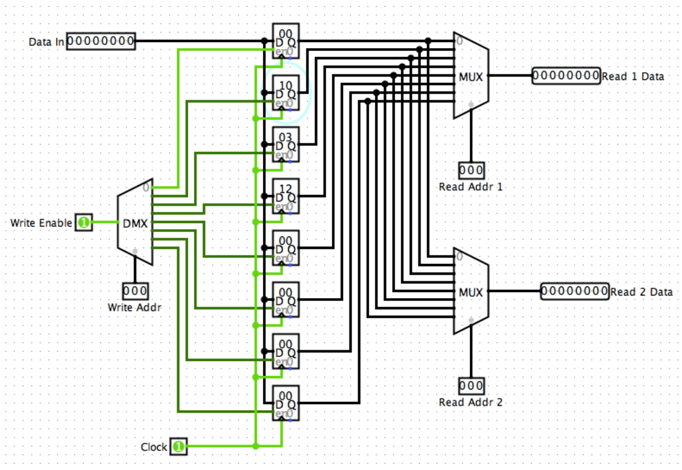
2/ Basic 8-bit processor

- An 8-bit single cycle CPU with an ALU capable of addition, subtraction, and logical operations, supporting R-type, I-type, and load/store word from/to Data Memory.
- The processor uses 14-bit instruction and includes:
  - a register file with 8 registers.
  - a Program Counter (increment by 1 each time)
  - Instruction Memory (8-bit address width)
  - Data Memory (8-bit address width)

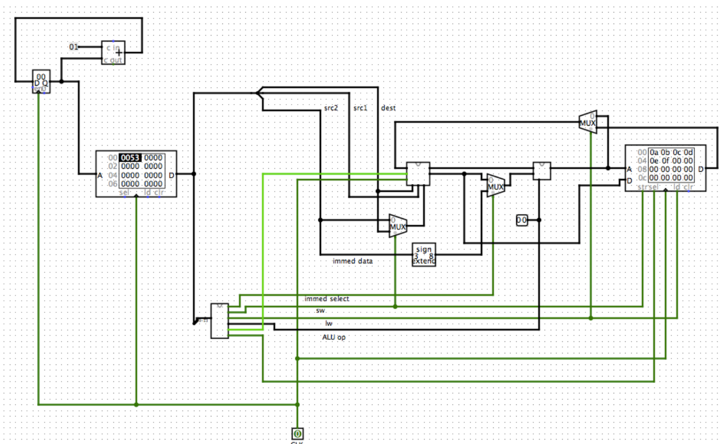
- Test the functionality of this processor using the hexadecimal translation from section 1:

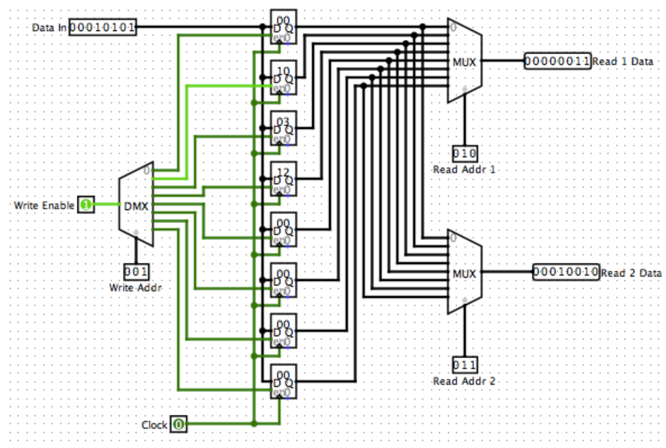
- **Addition:**

Beginning State:

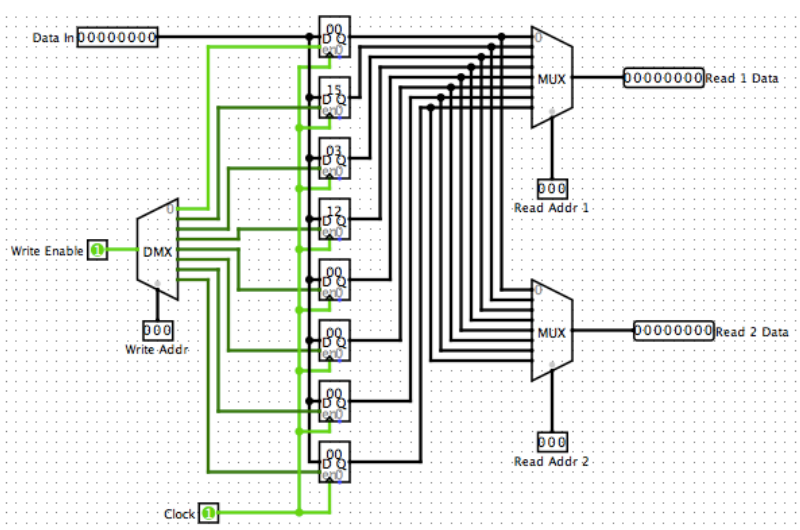


Instruction is fetched:



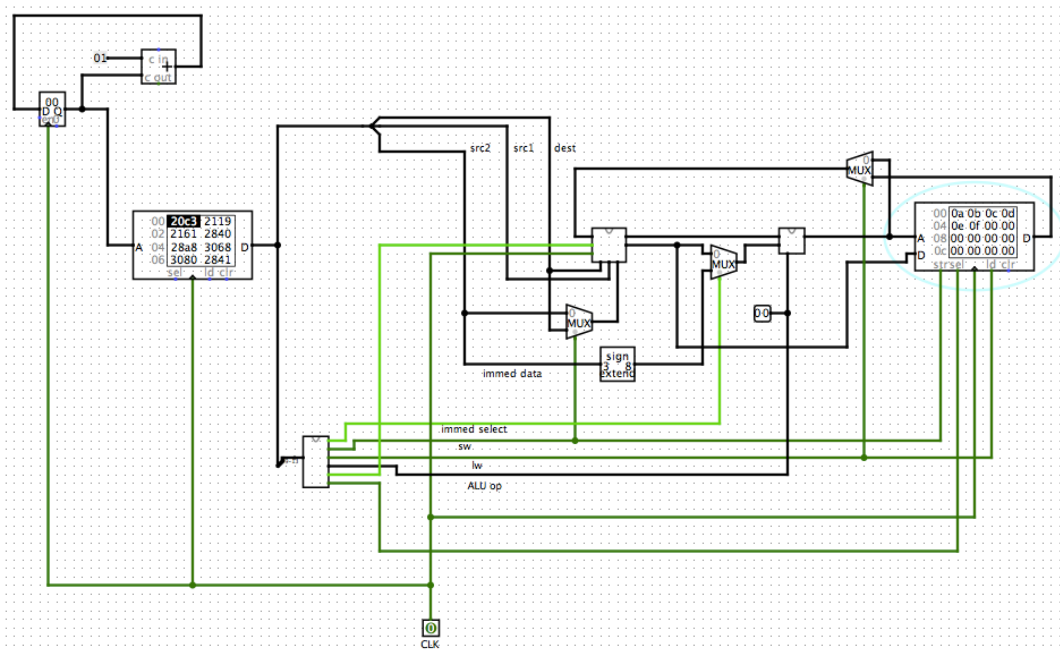


Final State:

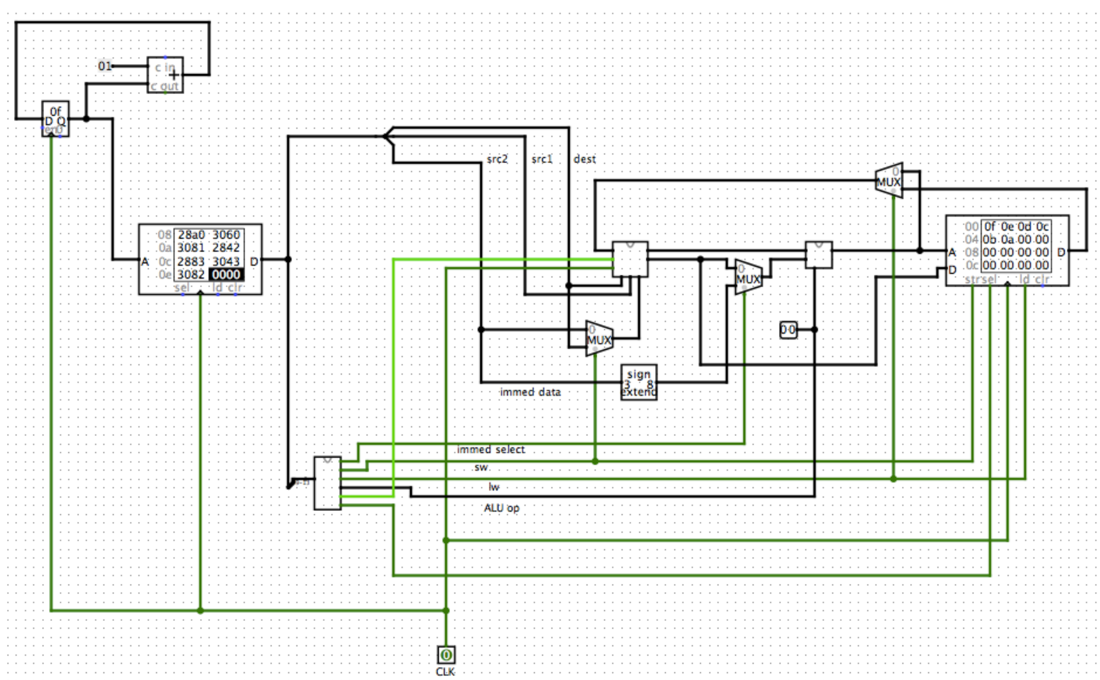


Reverse:

- Beginning State (Data memory: 0a, 0b, 0c, 0d, 0e, 0f)

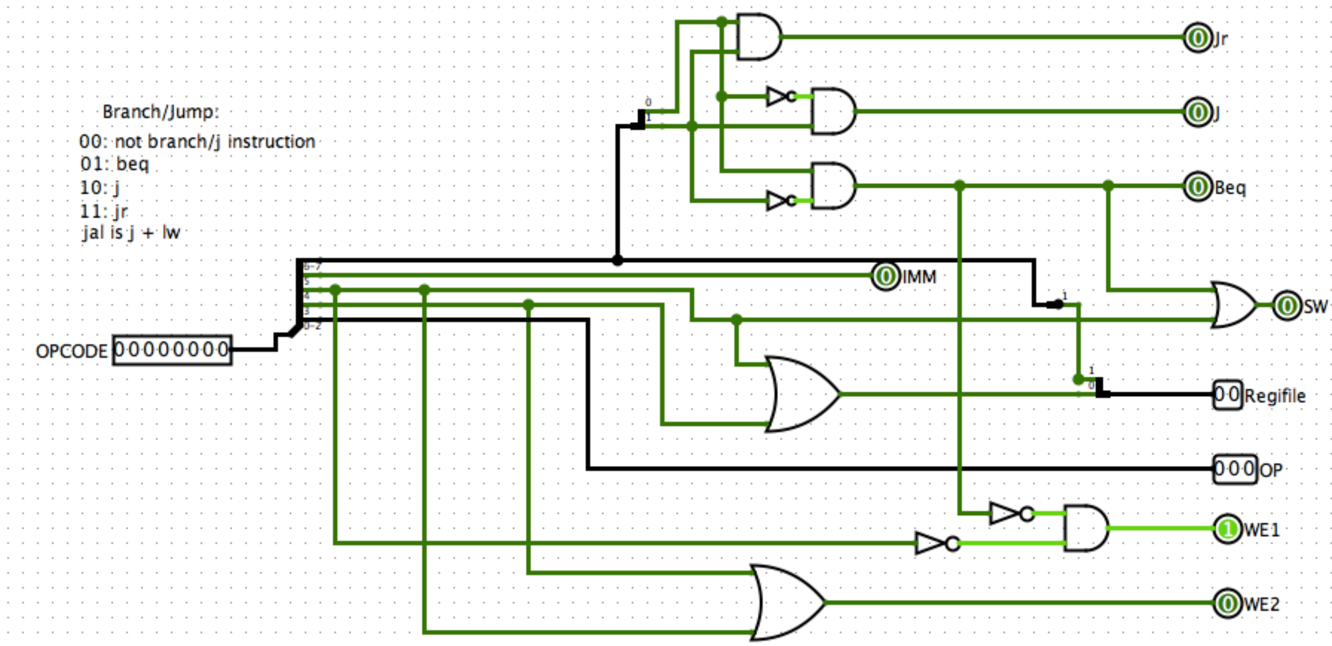


- Final State:



Added Features:

- Here I use a 16-bit instruction, with 8-bit opcode (2 for jump/branch, 1 each for sw/lw/imm, 3 for alu operation).

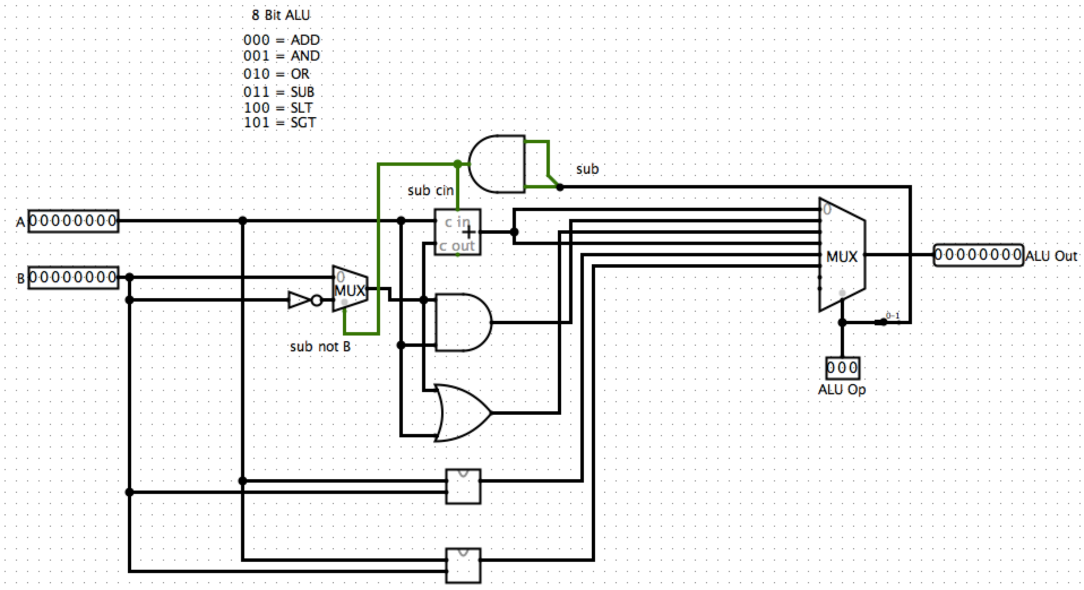


Instruction opcode:

Instruction	Branch (2-bit)	Intermediate (1-bit)	sw (1-bit)	lw (1-bit)	ALU-op (3-bit)
R-type: add	00	0	0	0	000
R-type: and	00	0	0	0	001
R-type: or	00	0	0	0	010
R-type: sub	00	0	0	0	011
R-type: slt	00	0	0	0	100
R-type: sgt	00	0	0	0	101
I-type: addi	00	1	0	0	000
I-type: lw	00	1	0	1	000
I-type: sw	00	1	1	0	000
R-type: beq	01	0	0	0	011
Jump: j	10	0	0	0	000
Jump-and-link: jal	10	0	0	1	000
Jump-return: jr	11	0	0	0	000

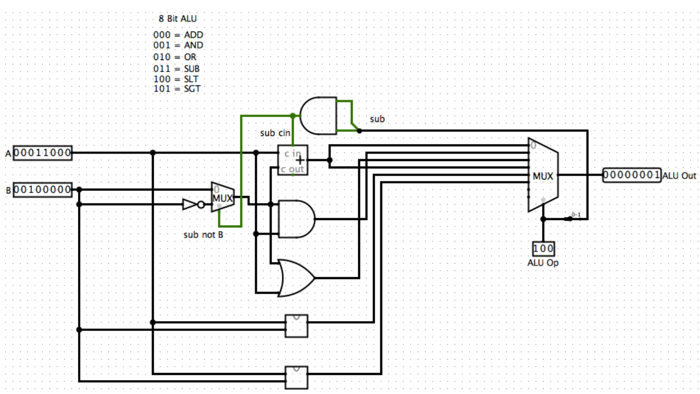
3/ SLT/SGT

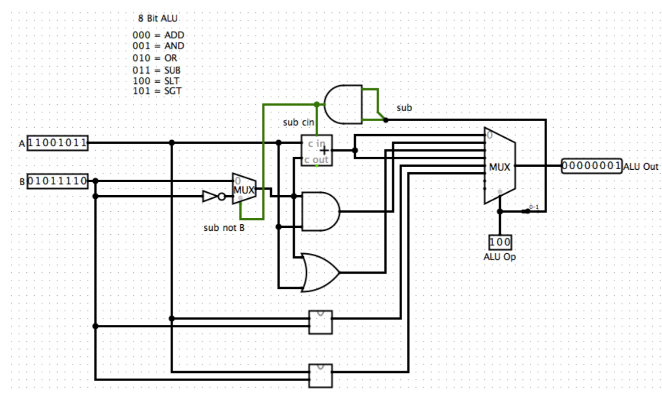
- ALU operation, for two's complement



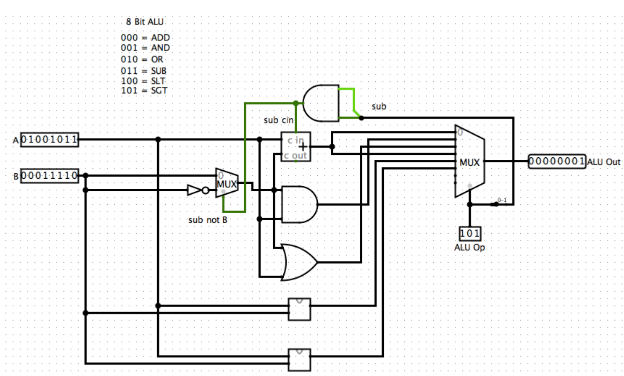
slt $dest, $reg1, $reg2 will set $dest to 1 if reg1 < reg2
sgt $dest, $reg1, $reg2 will set $dest to 1 if reg1 > reg2

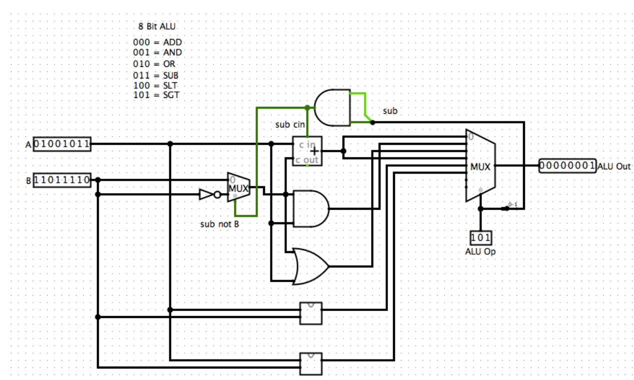
- ALU test (slt)





ALU test (sgt)





$\$1 = 2$; $\$2 = -2$; $\$5 = \$6 = 1$

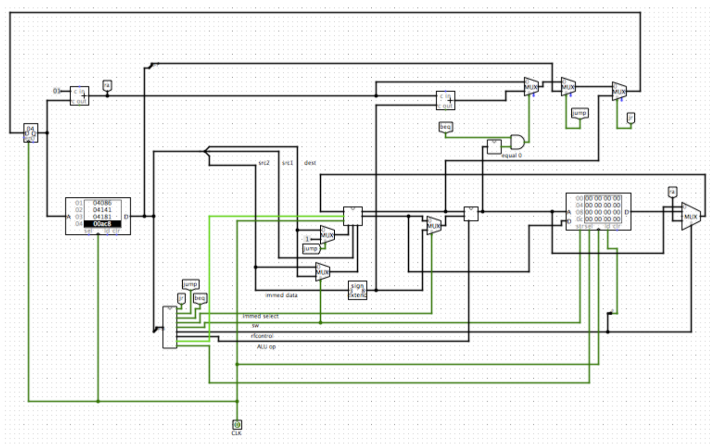
True case: sgt \$3 \$1 \$0

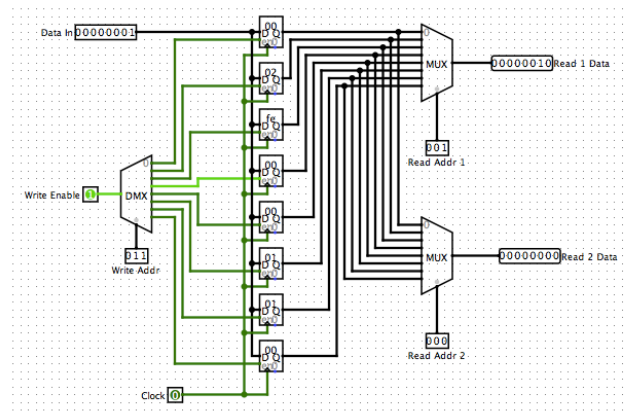
slt \$4 \$2 \$0

False case: sgt \$5 \$0 \$1

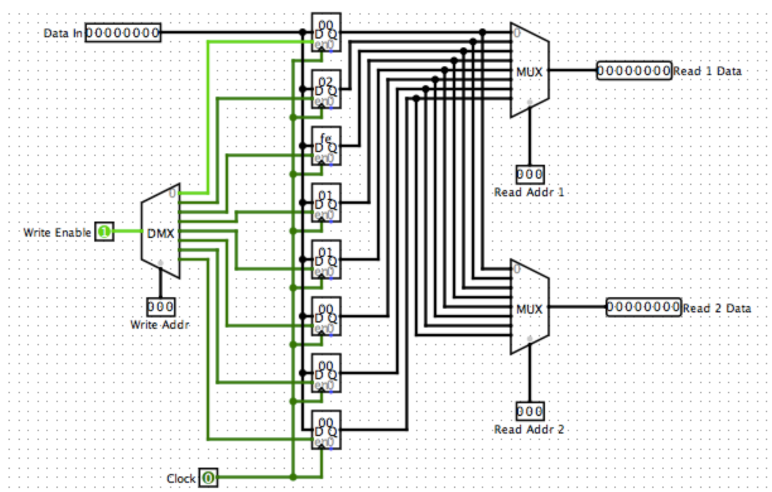
slt \$6 \$0 \$2

Before:

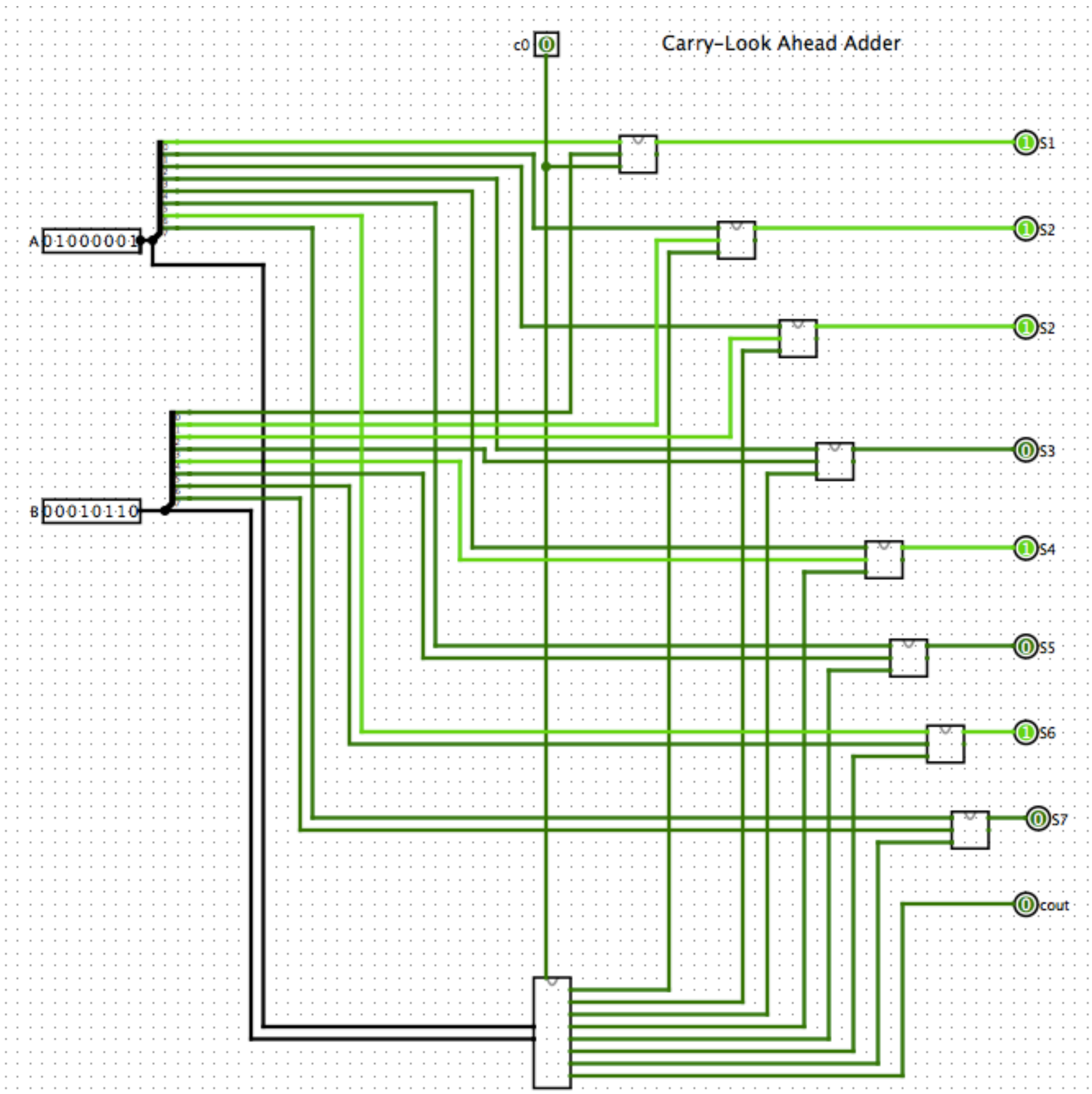




After (value at \$3, \$4 change to 1; \$5, \$6 still 0)



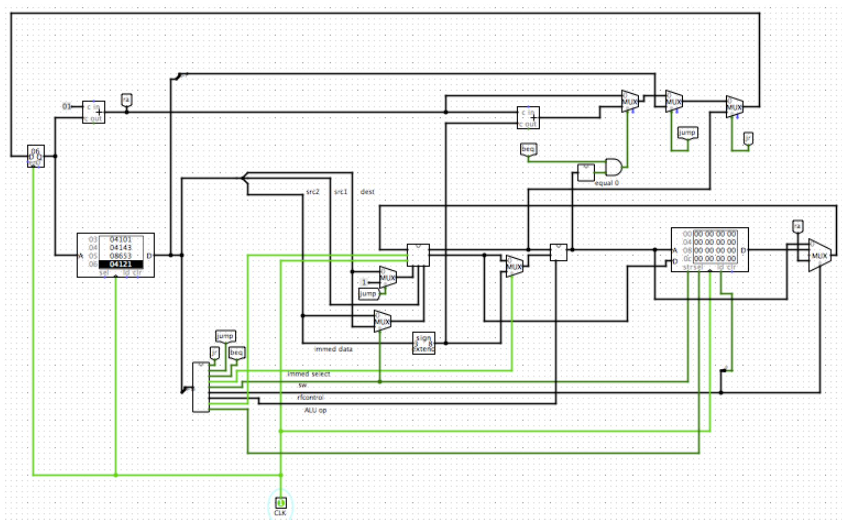
4/ Carry Look-ahead Adder:



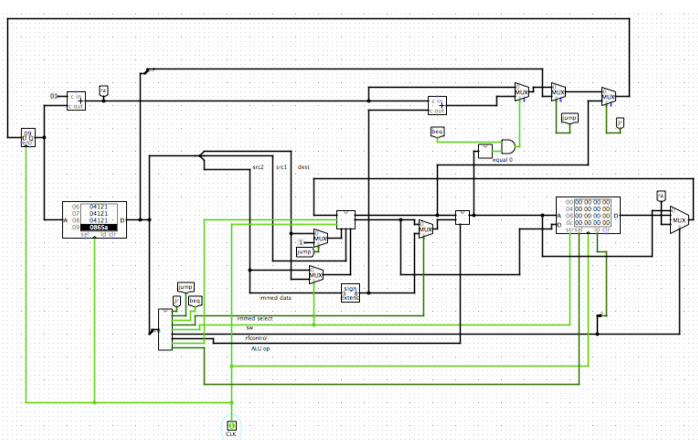
5/ Branch-if-equal (beq)

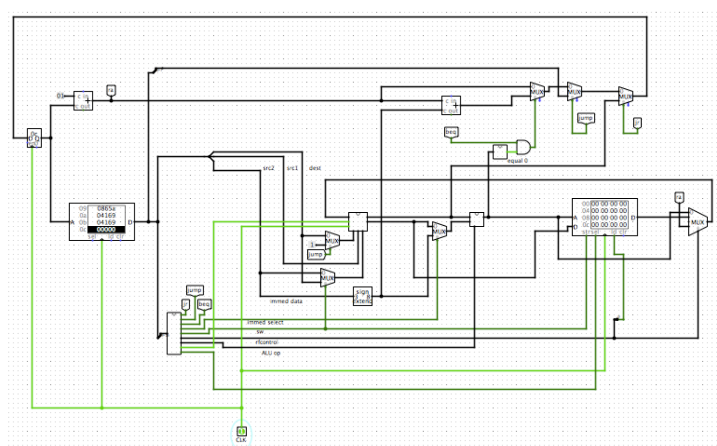
- Branch if reg1= reg2. The opcode is 01000011 (01 to specify beq and also ALU subtract op 011). Since my program counter only increments by 1 every clock cycle to move to the next instruction => just have to add the beq offset to pc. WE is disabled during beq instruction.
- John's test (I modified it to accommodate only the immediate value < 3)

The first branch (instruction #5) is not taken:



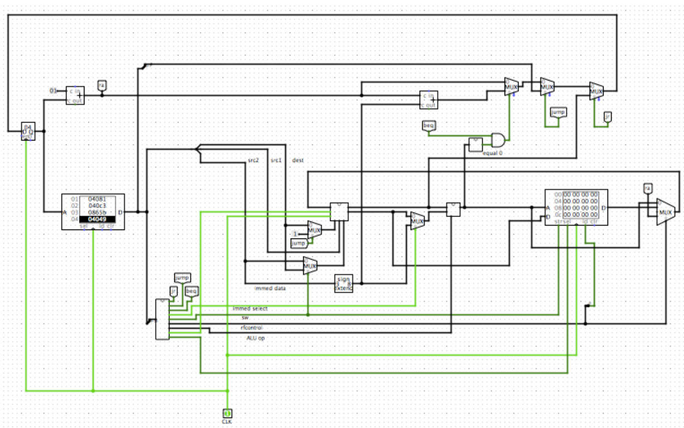
The second branch (instruction #9) is taken:

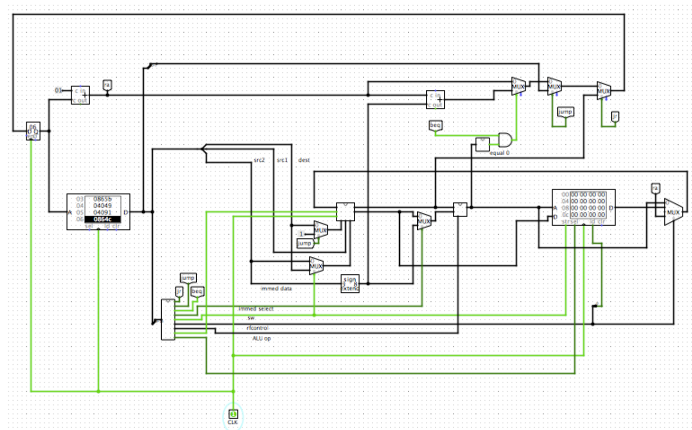




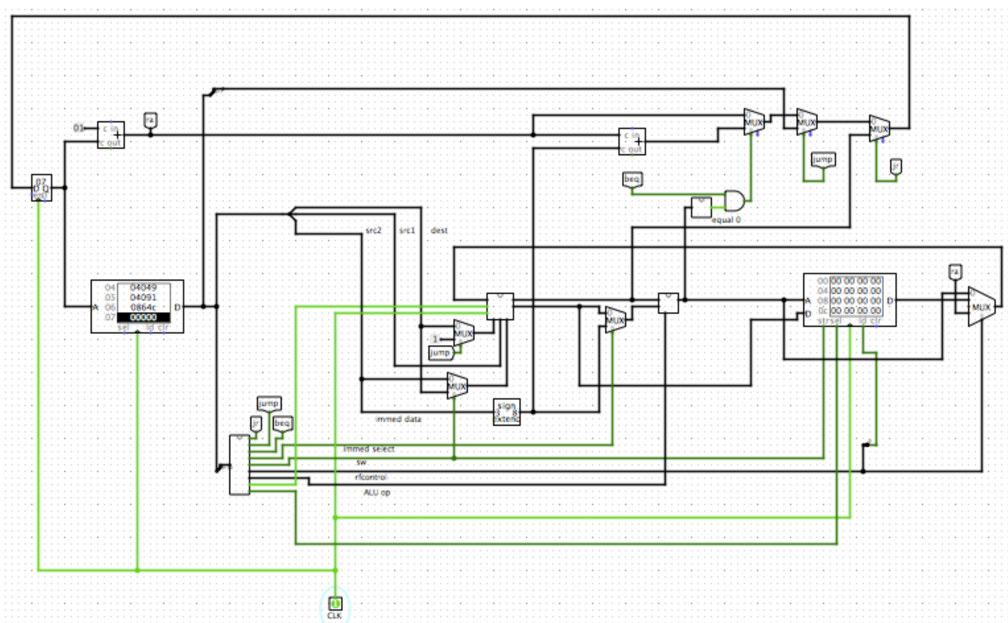
- Back Branch:

First time: first branch is not take, second branch is taken



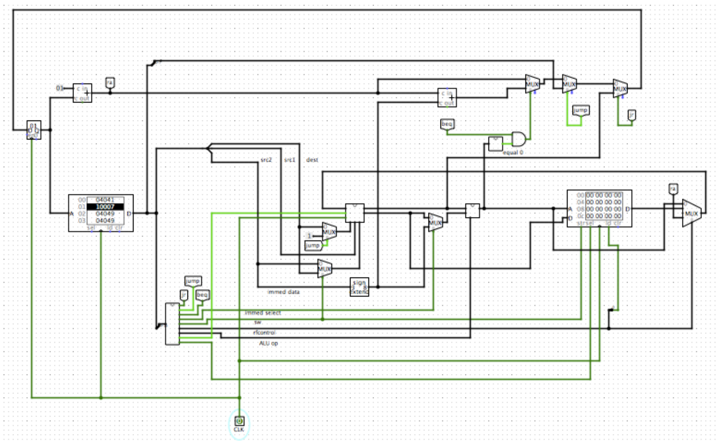
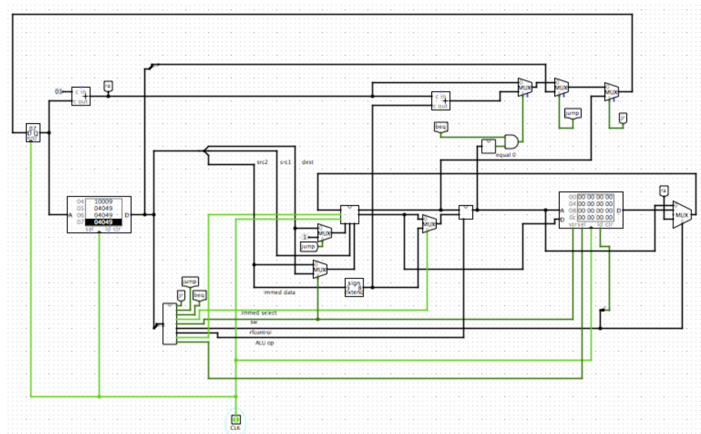


First branch will be taken on the 3rd loop

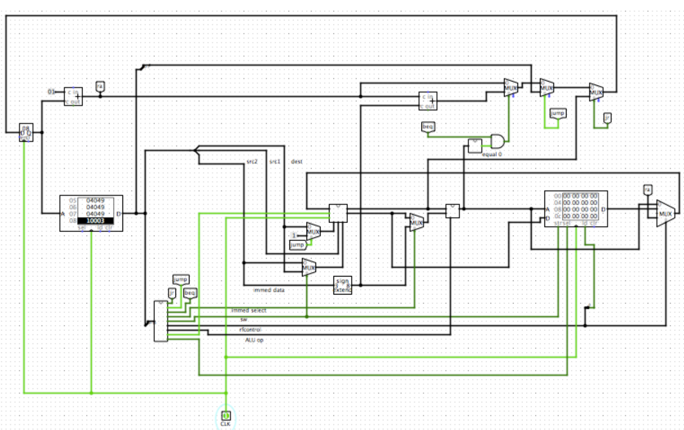
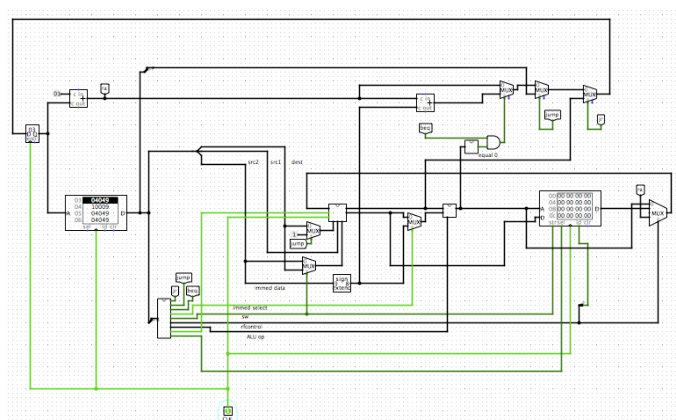


6/ Jump instruction:

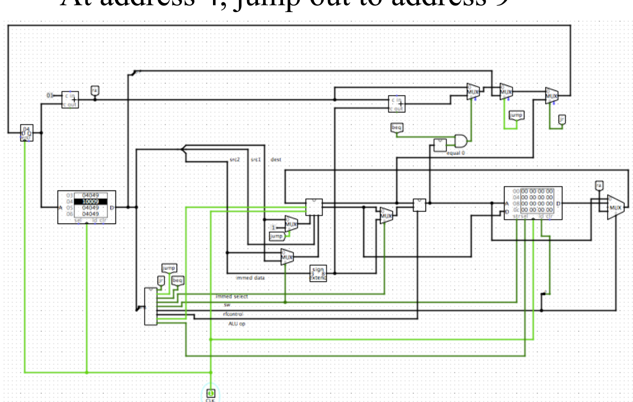
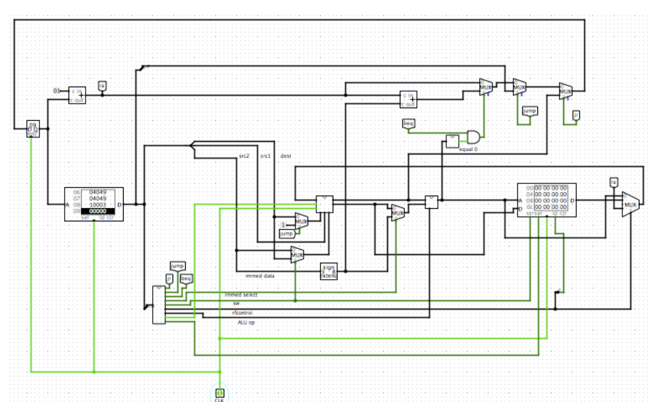
- With 16-bit instruction and 8-bit opcode, the jump instruction is left with 9 bits to specify the address in instruction memory it wants the program to jump to. However, my instruction memory uses 8-bit address, thus, I decide that during jump instruction, my processor will only take into account bit 0-7 in the instruction when deciding which address to jump to. Since I have a much simpler cpu, I don't need to attach the region of the PC to my jump address.
- Test: first jump to address 7

Then at address 8 jump to address 3:

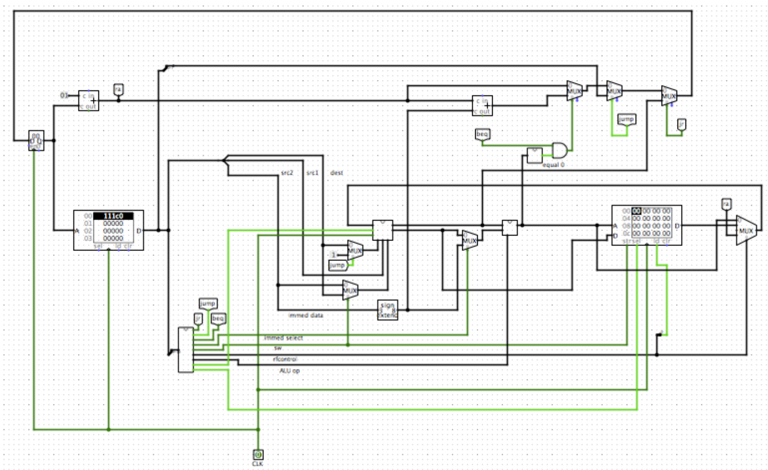
At address 4, jump out to address 9

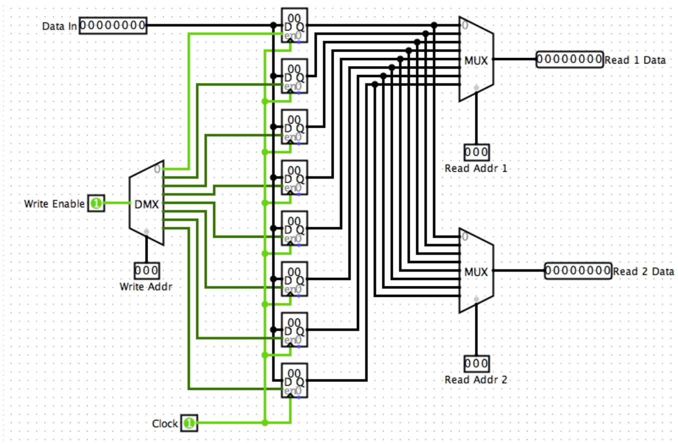



7/ Function call (JAL/JR)

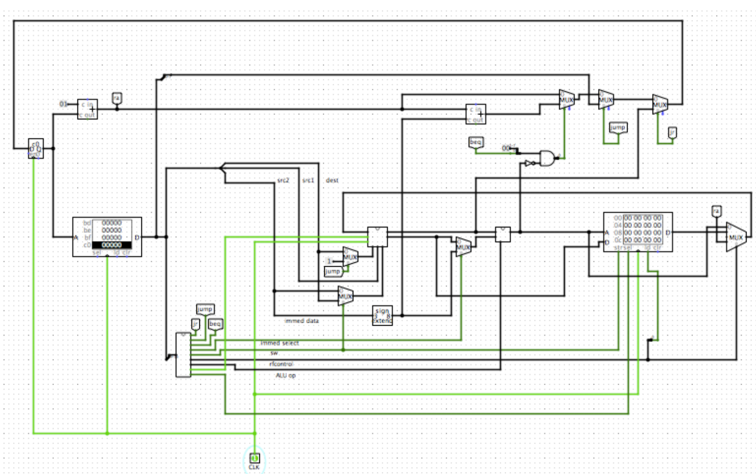
- I only have 8 registers in my processor, thus, for function call I assume $ra is always at register 1. Here I manually tested a very simple case where I jal to a random address (c0 in this case) and then jr back (at address c1).

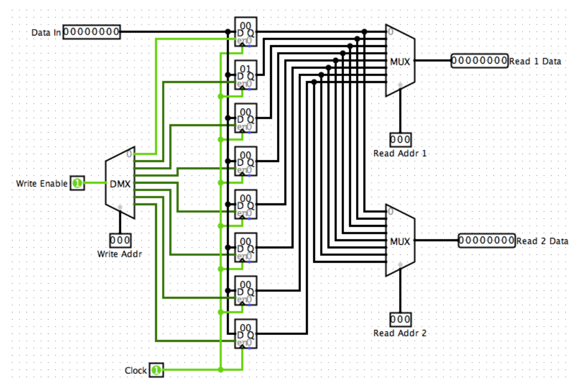
Beginning State (jal)



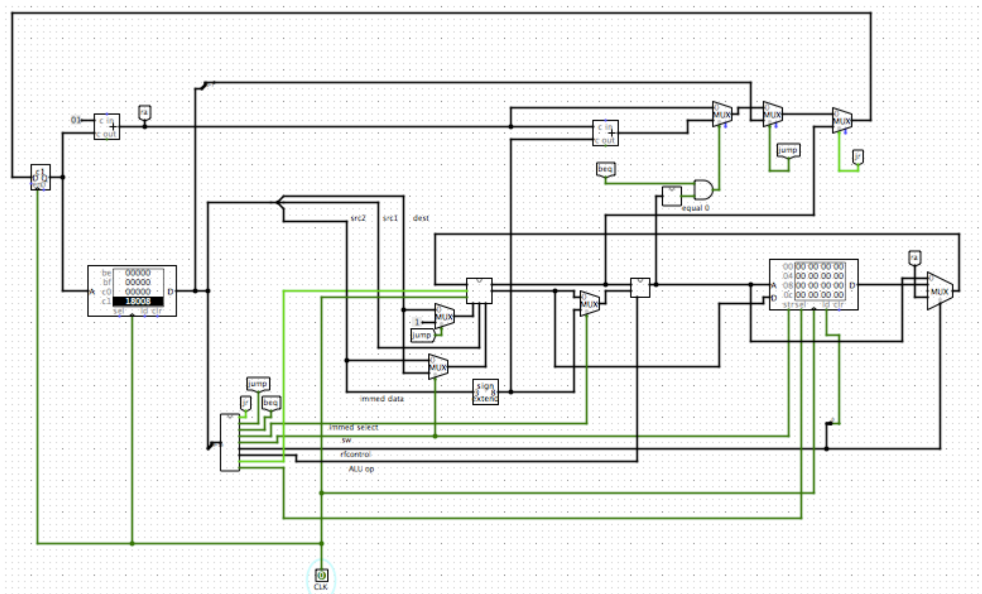


After jal (PC jumps to new address, the return address (address 01 is stored in $1)





Call jr $ra (at address c1)
Beginning State:



Final State -> return to instruction at address 1