Mitchell Stacha
Pranav Konduri
Cora English

CSCE 312-201

Final Project

4/21/2020

Spring 2020

Design a minimal instruction set architecture for an 8-bit load-store processor with appropriate binary encoding. Describe the ISA in detail in the report with all your design assumptions and rationale. *[Ref] Detailed description of your Instruction Set Architecture covering the following points...*

Instruction	Byte 1	Byte 2	Byte 3	Comments
halt	0 0			Stops the instruction from executing any more commands.
nop	1 0			No operation.
irmovl	2 0	F	rB 0	Takes an immediate value (F) and inserts it into a given register.
rmmovl	3 0	rA rB	D	Puts the contents of a given register into the RAM. D designates the offset for rB.
mrmovl	4 0	rB rA	D	Puts the contents of place in the RAM into a given register. D designates the offset for rB.
OPl	5 fn	rA rB		Does an operation, destination targeted at rB.
jXX	6 fn	D		Given a designated jump fn flag, jumps to the designated location.
call	7 0	D		Calls a function at a designated location, pushing the return address into the stack.
ret	8 0			Pops the return address, and then jumps to that address.
push	9 0	rA 0		Pushes a value from the stack.
pop	A 0	rA 0		Pops a value from the stack.

Registers	
%eax	0x1
%ecx	0x2
%edx	0x3
%ebx	0x4
%esp	0x5
%ebp	0x6
%esi	0x7
%edi	0x8

Functions for OPq		Functions for jXX	
0	add	0	Jmp (unconditional)
1	sub	1	Jle
2	and	2	jl
3	or	3	je
4	xor	4	jne
5	mult	5	jge
		6	jg

Using the assembly language code that you designed in the previous step, create a program to perform the matrix addition. *[Ref] Attach your Y86 assembly code.*

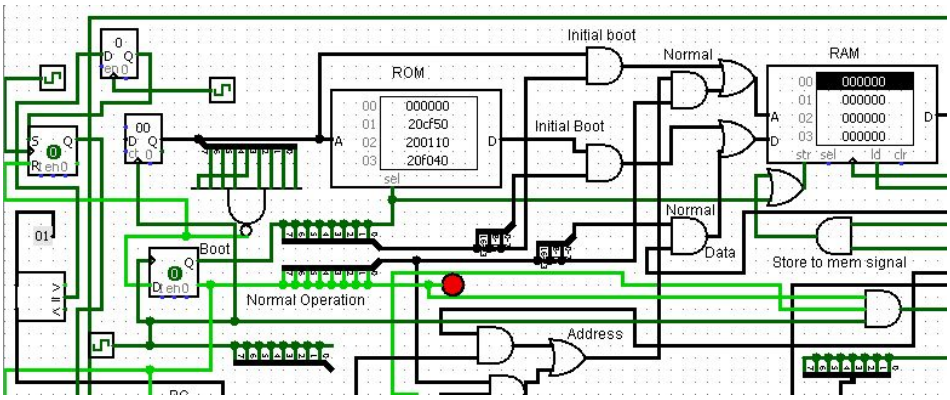
$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \qquad B = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} \qquad A + B = C = \begin{pmatrix} 2 & 2 \\ 4 & 4 \end{pmatrix}$$

```
1  Main:
   # FIRST MATRIX
2  irmovl $ff, %esp
3  irmovl $1, %eax
4  irmovl $0, %ebx
5  rmmovl %eax, 0(%ebx)
6  irmovl $2, %eax
7  rmmovl %eax, 1(%ebx)
8  irmovl $3, %eax
9  rmmovl %eax, 2(%ebx)
10 irmovl $4, %eax
11 rmmovl %eax, 3(%ebx)
   # SECOND MATRIX
12 irmovl $1, %eax
13 irmovl $4, %ecx
14 rmmovl %eax, 0(%ecx)
15 irmovl $0, %eax
16 rmmovl %eax, 1(%ecx)
17 irmovl $1, %eax
18 rmmovl %eax, 2(%ecx)
19 irmovl $0, %eax
20 rmmovl %eax, 3(%ecx)
21 irmovl $0, %eax
22 jmp Test
23 Loop:
24 pushl %eax
25 popl %edx
26 addl %ebx, %edx
27 mrmovl 0(%edx), %esi
28 pushl %eax
29 popl %edx
   # ADDING
30 addl %ecx, %edx
31 mrmovl 0(%edx), %edi
32 addl %esi, %edi
33 rmmovl %edi, 0(%edx)
34 irmovl 1, %edx
35 addl %edx, %eax
36 Test:
37 irmovl 4, %edx
38 subl %eax, %edx
39 jne Loop
40 End:
41 halt
```

Design and verify a minimal processor that can execute the ISA in activity # 1. Only design the bare minimum circuits which are absolutely essential to execute the program that you created in assembly, and design the peripheral input/output circuit(s) for this designed processor, which can interface with the RAM/ROM modules. *[Ref] For each functional block of the processor.*

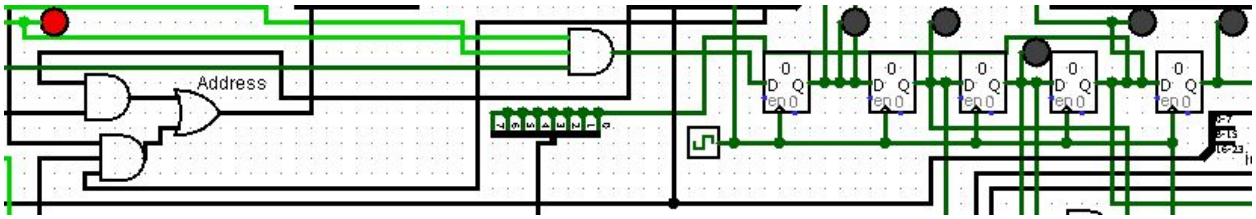
a. Explain its operation. (You may want to insert screenshots from Logisim to help you in your explanation)
b. If you split the work of creating these various sub-blocks, clearly demarcate which parts were contributed by each of the members.

ROM to RAM



Pictured here is the circuit that copies the contents of the ROM to the RAM. It works by using a counter that counts to 0xFF (then halts) which copies the value at that address in the ROM to that address in the RAM. Once this counter reaches 0xFF, the D flip flop is triggered which starts the normal operation of the circuit. There are various gates and wires in place around the RAM which coordinate the reading and writing to the RAM while the circuit boots up and while the circuit is operating normally.

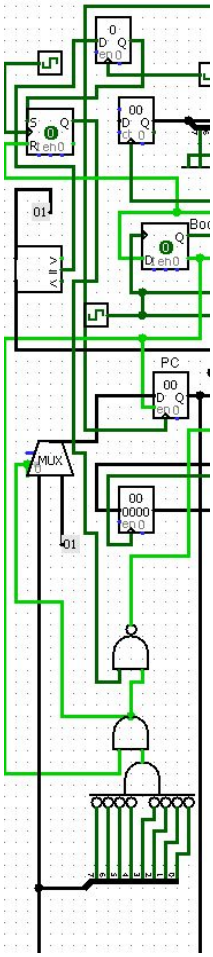
Syncing of the Components

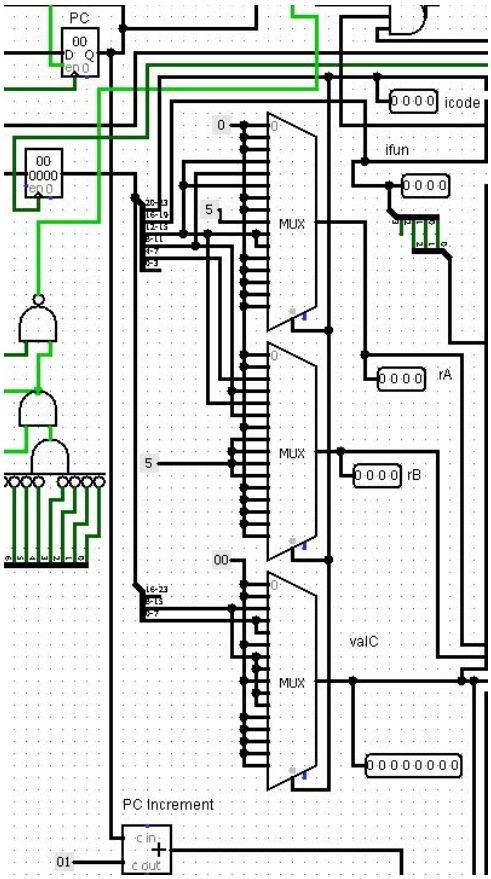


Due to the non-sequential aspects built into the registers and the RAM in logisim, all of the components could not be synced to one central clock. Instead, their operation had to be offset to ensure that they worked in a coordinated fashion. This was done by using a central clock signal that would be passed between sections of the circuit by using smaller registers (that were synced to a clock going much faster than the central clock).

Halting the Program

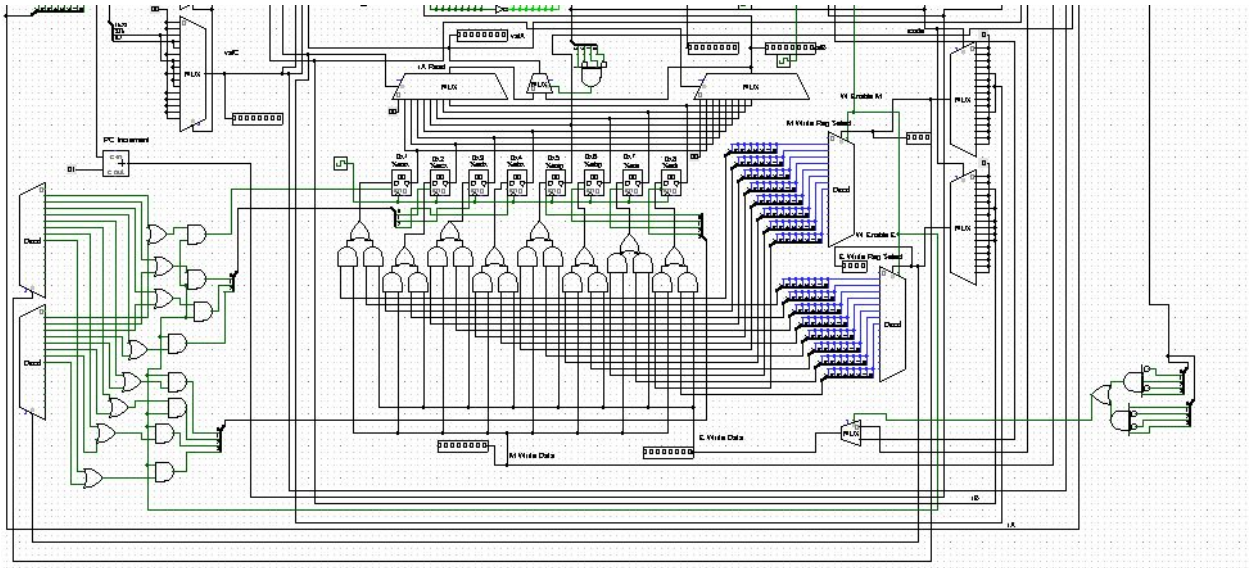
The circuit is set up to start at PC = 0x01. The default value that the PC starts at is 0x00 which in a valid program should correspond with the instruction 0x000000. The part of the circuit that is pictured here is designed to "escape" 0x000000 only once and operate the program that is started at address 0x01. Once the instruction 0x000000 (halt instruction) is encountered again, the central clock signal's interface with the synchronization registers is severed, which halts the program.





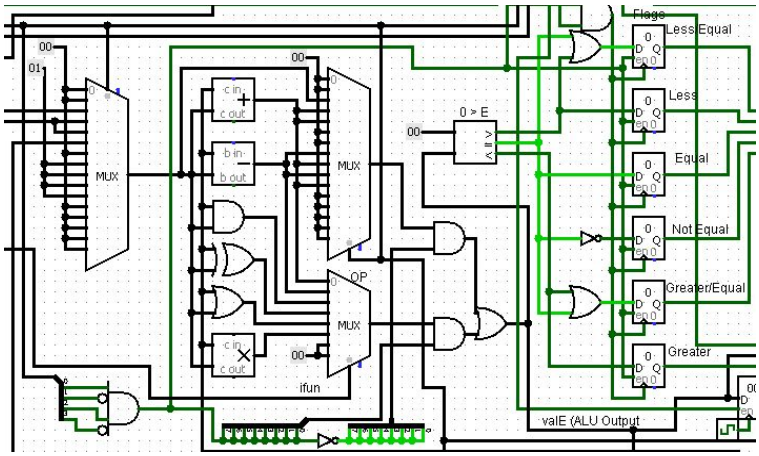
Fetch Stage

Here is the fetch stage of the processor. It first reads the current instruction from memory at the address designated by the PC register. This instruction is then read into the instruction register since it will be used throughout the cycle. The correct values are read into icode, ifun directly while rA, rB, and valC are processed by using multiplexers that are using icode to determine their proper value. The default PC increment, which will be used later on unless icode is a call, return, or jump instruction (which may still use the next PC) by adding 1 to the current instruction since our instructions have a standardized length of 3 bytes (while the addresses are 8 bits and have data of 24 bits).



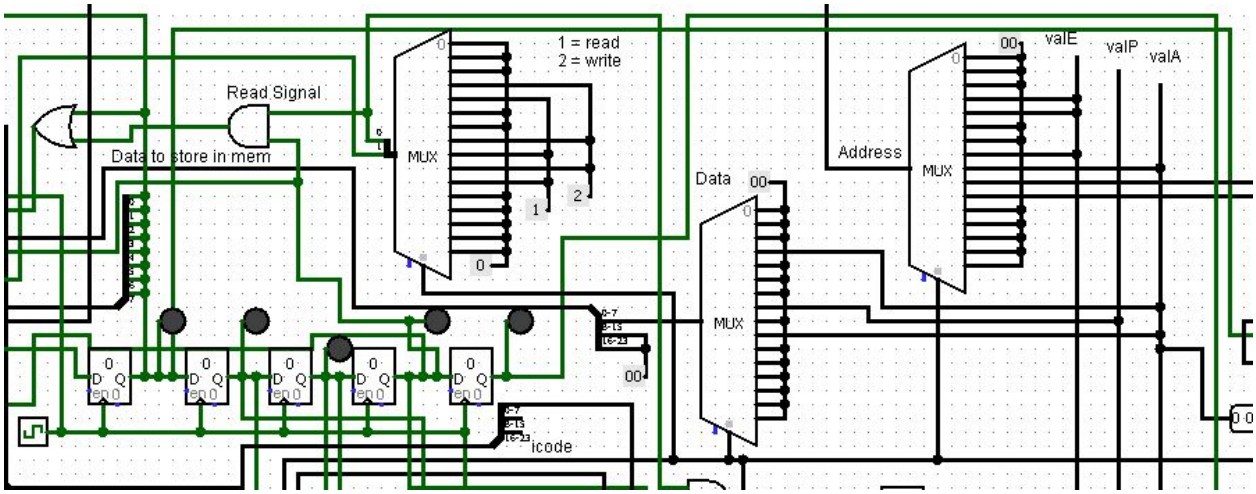
Decode Stage

Here is the register file for our program. It isn't very notable with the exception of the fact that the write signals are coordinated to come in whenever the write back stage is encountered and a write back needs to occur.



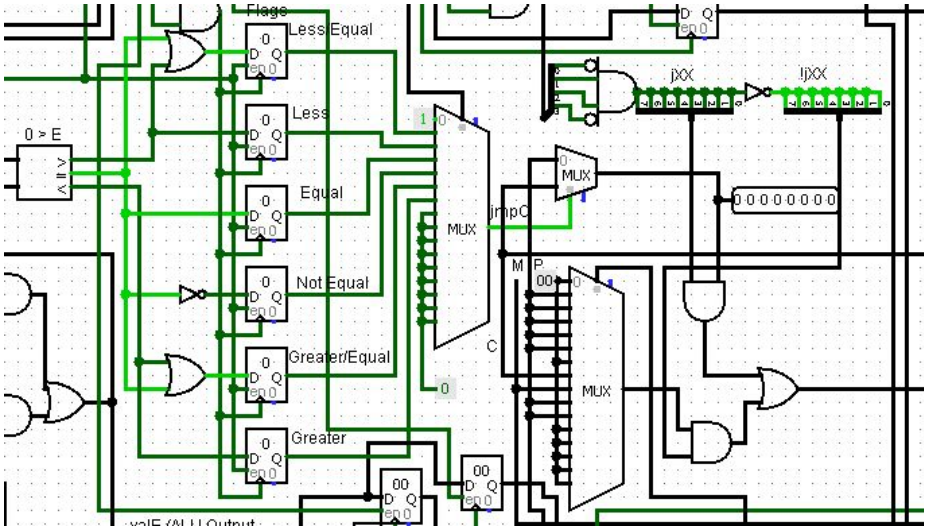
Execute Stage

Here is the execute stage, since valB is always used in the execute stage, it is already fed into all of the different arithmetic units while the multiplexer all the way to the left determines what the other value that gets passed into the arithmetic units. The top multiplexor after the arithmetic units determines what the output is when the instruction is not a mathematical operation. The multiplexor below it designates the output when the instruction is a mathematical operation. The gates to the left help select which output to use and the registers attached to the comparator store all of the potential jump conditions which would be used later in the program and they are refreshed whenever a mathematical operation is performed



Memory Stage

The multiplexers here coordinate the addresses, data, and read/write signals for the RAM. There isn't much that is notable here. Once complete, any data that is read from the RAM is in transferred to a special data register where it can be utilized in in the write back stage



Write Back Stage

The part of the write back stage that writes to the registers was already shown in the decode stage and is pretty self-explanatory. Shown here is the part of the circuit that designates what the next PC value should be. The leftmost multiplexer feeds in the register that holds the jump conditions and selects the output based on ifun (0 is always on in this mux because it is the unconditional jump). If this mux has an output of 1 and the current instruction is a jump instruction, the next PC will be valC. If not, the other large mux decides where the next PC will be (valC if the instruction is call, the value retrieved from memory if the current instruction is return, and the PC increment for all other instructions).

Extra Design Tasks

c. Extend your ISA to support “call” and “return” instructions.
d. Write a matrix multiplication program that is called from the main method and after the computation is done return to the main method.

Multiplication Program: This program calls the multiplication program for 2 matrices (in ecx and ebx, populated at the beginning of MULT), multiplies them, and returns the values in eax.

```
1  Main:
2  call MULT
3  halt
4  MULT:
#See ADD for matrix populating
5  //Just somewhere far away
6  irmovl $0x64, %esp
7  irmovl $0, %eax
8  irmovl $0, %edx
9  CheckA:
10 irmovl $4, %esi
11 subl %eax, %esi
12 jne LoopA
13 jmp Final
14 LoopA:
15 irmovl $0, %edx
16 CheckB:
17 irmovl $2, %esi
18 subl %edx, %esi
19 jne LoopB
20 irmovl $2, %esi
21 addl %esi, %eax
22 jmp CheckA
23 LoopB:
24 pushl %eax
25 popl %esi
26 pushl %edx
27 popl %edi
28 addl %ebx, %esi
29 addl %ecx, %edi
30 mrmovl (%esi), %esi
31 mrmovl (%edi), %edi
32 mull %esi, %edi
33 pushl %edi
34 irmovl $1, %esi
35 irmovl $2, %edi
36 addl %eax, %esi
37 addl %edx, %edi
38 addl %ebx, %esi
39 addl %ecx, %edi
40 mrmovl (%esi), %esi
41 mrmovl (%edi), %edi
42 mull %esi, %edi
43 popl %esi
44 addl %esi, %edi
45 pushl %edi
46 irmovl $1, %esi
47 addl %esi, %edx
48 jmp CheckB
49 Final:
50 irmovl $8, %eax
51 popl %edx
52 rmmovl %edx, 3(%eax)
53 popl %edx
54 rmmovl %edx, 2(%eax)
55 popl %edx
56 rmmovl %edx, 1(%eax)
57 popl %edx
58 rmmovl %edx, 0(%eax)
59 ret
```