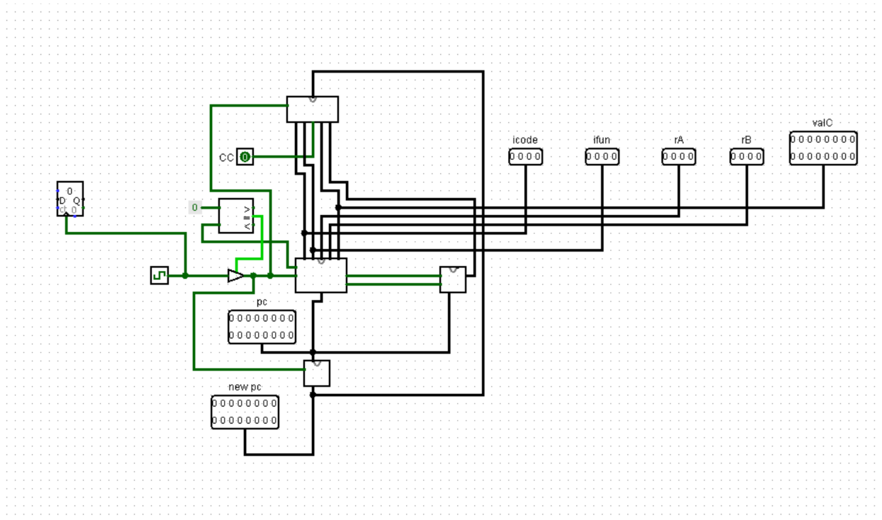
Lab 6 (Final Project)

Final Group Report

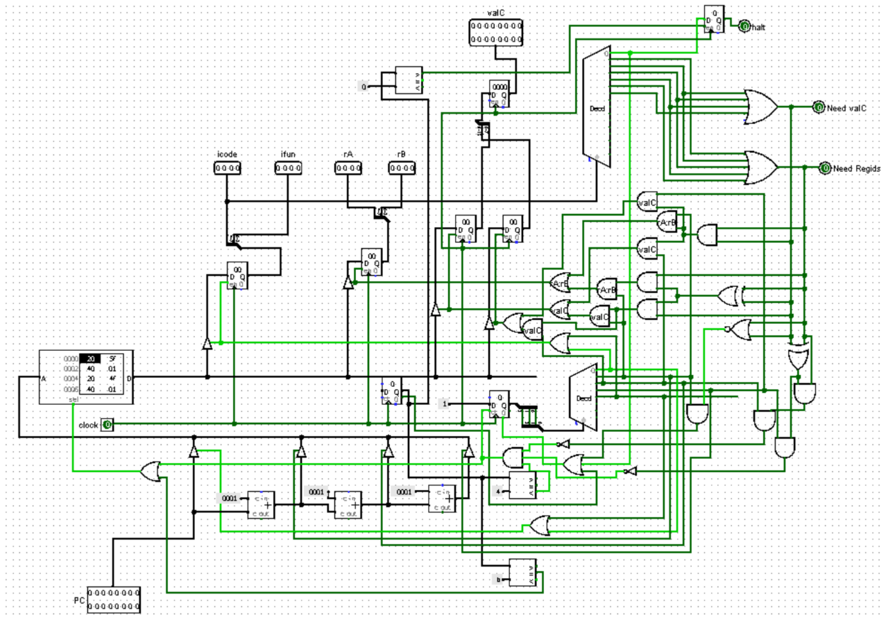
Kyle Diano, Muneeb Hashmi, Patricio Bunt, Nick Martinez, Trisha Narwekar

Fetch & PC

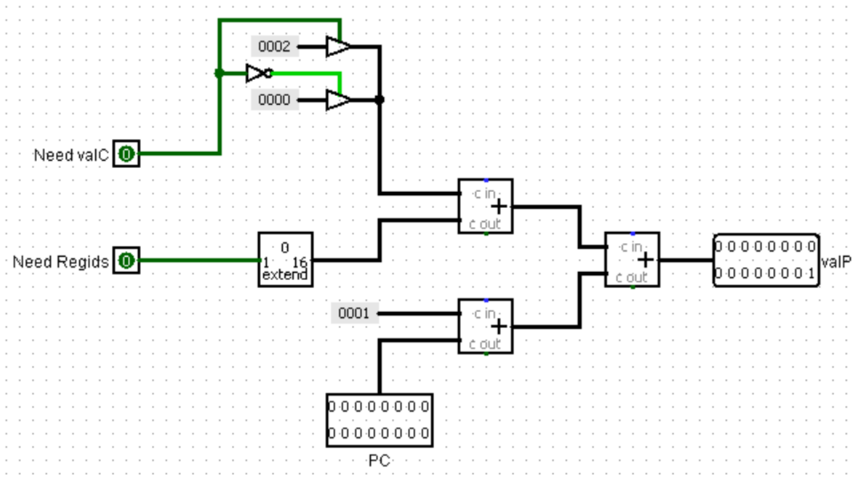
By Kyle Diano



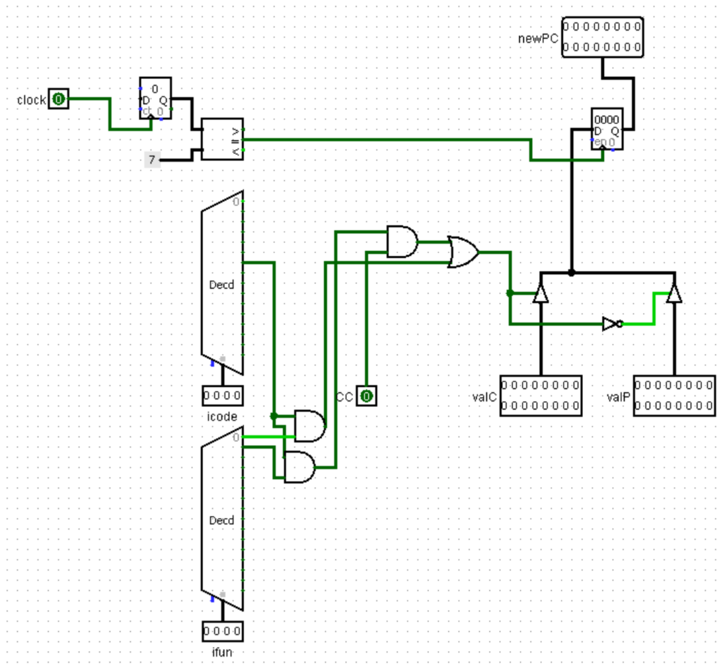
This is a general view of the fetch implemented with PC, PC increment, and New PC. This shows how the clock is going to work for the overall processor. Once halt is reached in the program instructions, the output of the clock for the whole processor is shut off.



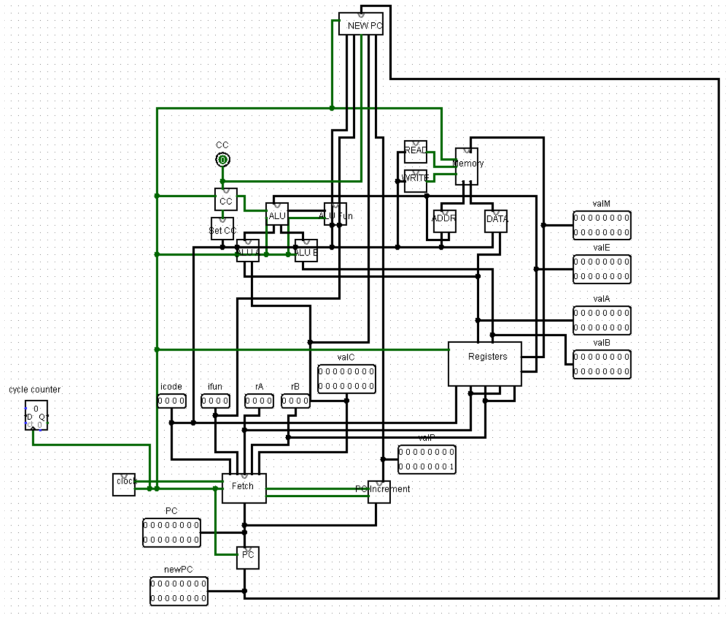
This is the complete circuit of the fetch in our processor. Since fetch is going to always be the first thing to run, it begins on the first clock cycle. The way I decided to implement fetch was to utilize a counter and a decoder to allow me to use specific cycles to activate different things. After the first cycle, icode and ifun are loaded, which allows the processor to determine whether we need registers or valc. This then outputs to PC increment, but also determines how the circuit deals with the clock cycles for that specific instruction. For example, if the instruction is something like "irmov", the fetch needs to get icode:ifun, ra:rb, and valc, which would take a total of 5 cycles. If it was an instruction that just needed icode:ifun, and ra:rb such as "rrmov", then it would only grab those two things from the rom in 2 cycles. If it was an instruction that just needed icode:ifun and valc, such as "jmp", it would only grab those two things from the rom in 4 cycles. If it was an instruction that just needed icode:ifun, such as "halt" it would only get that from the rom. After accounting for these specific cases, the decoder combined with this logic determines which controlled buffer to activate of the 4 calculated values from the PC. There's the PC value, the PC + 1 value, the PC + 2 value, and the PC + 3 value. It is organized this way since the rom stores all the program instructions 1 byte at a time, and the max amount of bytes that one instruction would take would be 4 bytes. Therefore, using this logic allows the fetch to access the correct indexes for what it needs depending on the instruction. From these indexes, the data for a specific value is being run into separate 8-bit registers. Using the decoder and logic decides what controlled buffer is activated, which would then store it in a register. For icode:ifun and ra:rb, these registers split up from 8-bit into 4-bits since these four values are four-bit. For valC, it actually takes two 8-bit registers that would combine into a 16-bit register, as this is the max value we have for our registers in decode. It is important to note that the bits 0-7 are taken first, followed by 8-15 when loading valC. Finally, halt is implemented by checking after grabbing icode if the instruction is halt, which then outputs out of fetch into the main clock, and cuts off the clock's output to the whole circuit.



This is the PC increment circuit. It takes in the "need valC" and "need regID flag" from fetch and calculates valP. The equation for our circuit to do this is valP = 1 + r + 2*v, where r is 0 or 1 depending on if we need registers, and v is 0 or 1 depending on if we need valC. Since this comes directly from icode, valP will be calculated on the first cycle.



This is the new PC circuit. It takes in the value of valC, valP, ifun, icode, and the condition flag. The newPC is always going to be valP, unless the instruction is jmp or je. When it is jmp, it allows valC to pass through. When it is je, it first checks the condition flag, if the condition flag is not on, then it will not jump, so it will use valP. Otherwise, it uses valC.

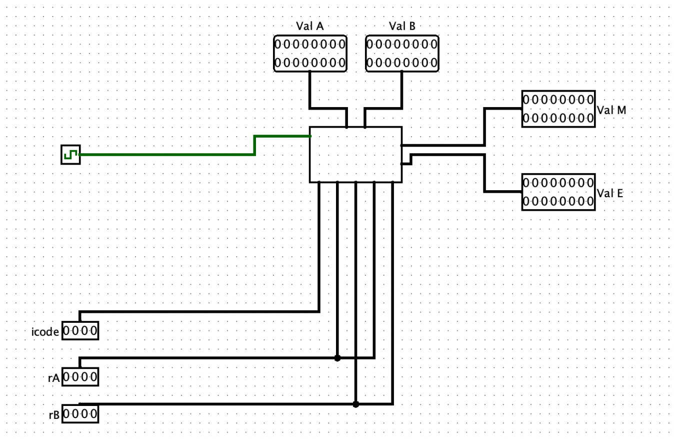


This is the complete 8-bit load store processor unit. It includes fetch, decode, execute, memory, PC, PC increment, new PC, and writeback. In order to implement all of these together, we had to make sure that each part was on the same page with clock cycles. Therefore, each part includes a counter that goes up to 11, which is the amount of cycles for each instruction besides halt, which would stop the entire processor.

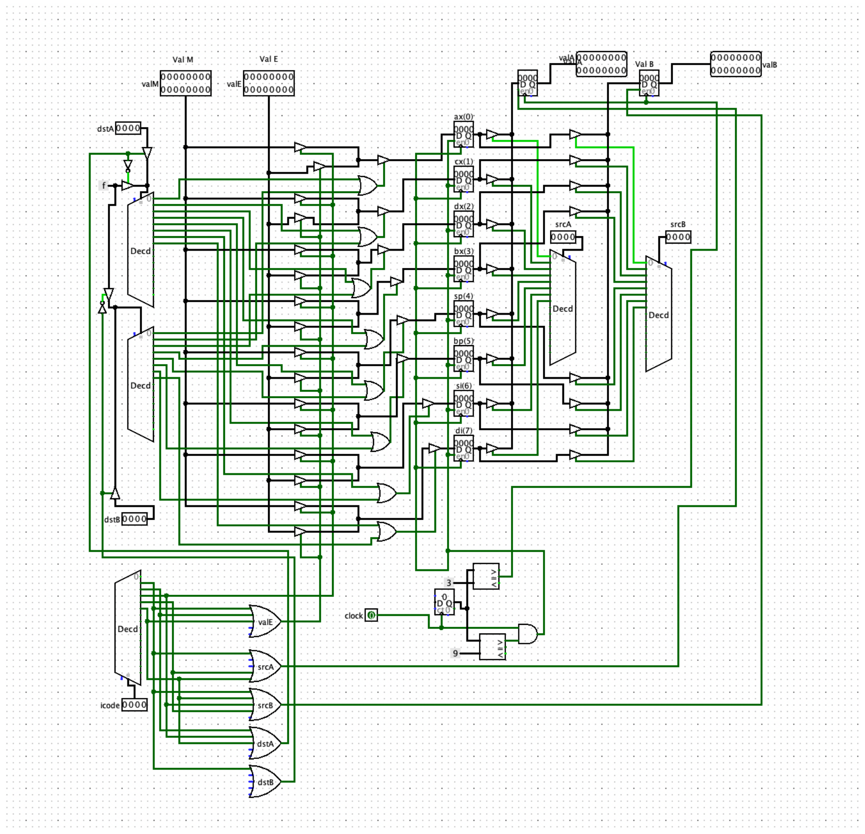
Decode/Writeback

By Trisha Narwekar

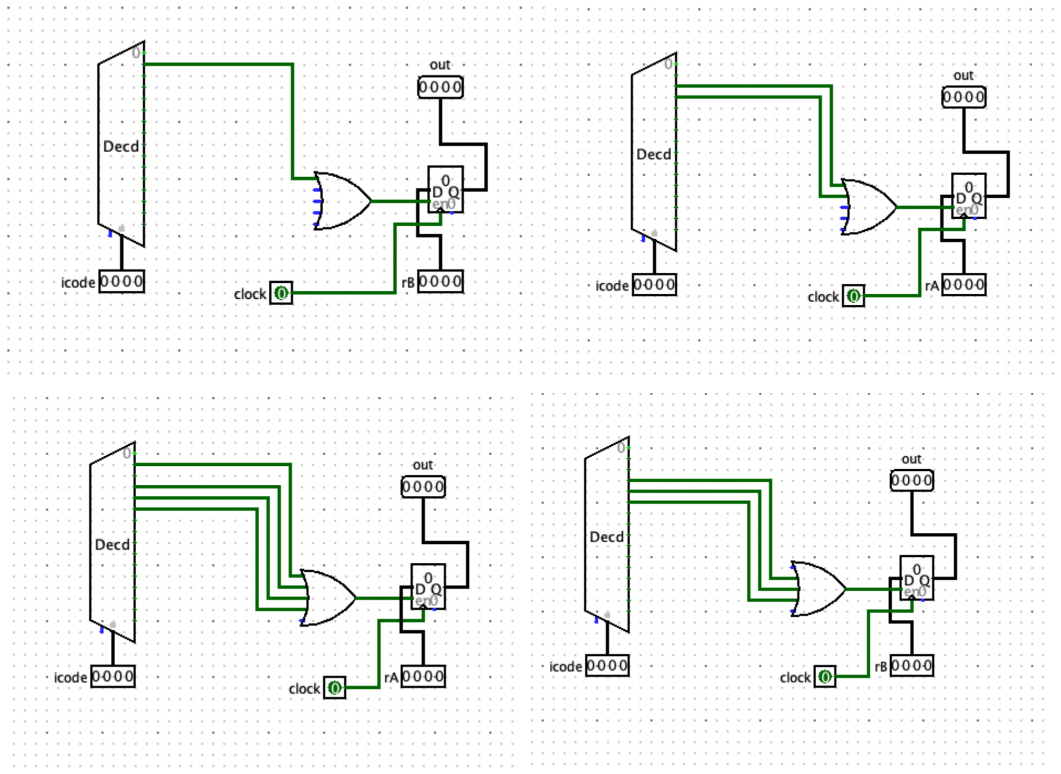
The main aspect that needed to be implemented for a Decode stage and for a Writeback stage is the register file. A register file holds multiple registers in order to read or write to certain outputs and from certain inputs. This feature is utilized by the stages mentioned previously to become the stage in which the values become a usable data set.



Here is the overall view of the Decode and Writeback stages. It shows the inputs that are necessary to decide what to write to and what to read from, and make the necessary changes.



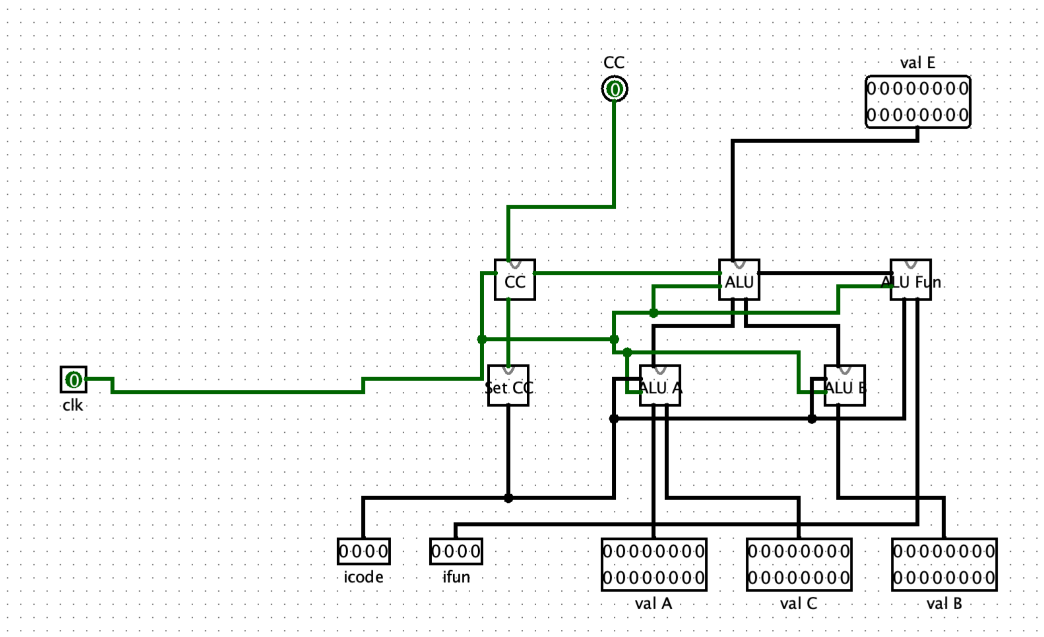
Within the overall view, there is a register file. In this register file, there are two write ports, as well as two read ports. For the write ports, it starts with the inputs which allow you to choose the register you want to write to by using a decoder to decode the 3 bits into one of the eight registers. Then, through the write ports, you can input the values you want to write into the register. Next, you decode the ports so that the register can take either of the write ports without error. On the other side, for the read ports, you have inputs which again choose the register. In addition, you have buses that decide which read port to display to, as well as which register to read from. The clock has to go through processes to make sure it waits for the stages prior to Decode or Writeback.



These images display the inner workings of each of the inputs. The four smaller inputs need to be decoded in order to take the Fetch output efficiently. They decode the inputs to match the register file and also give the stages that need more time, the time they need to run.

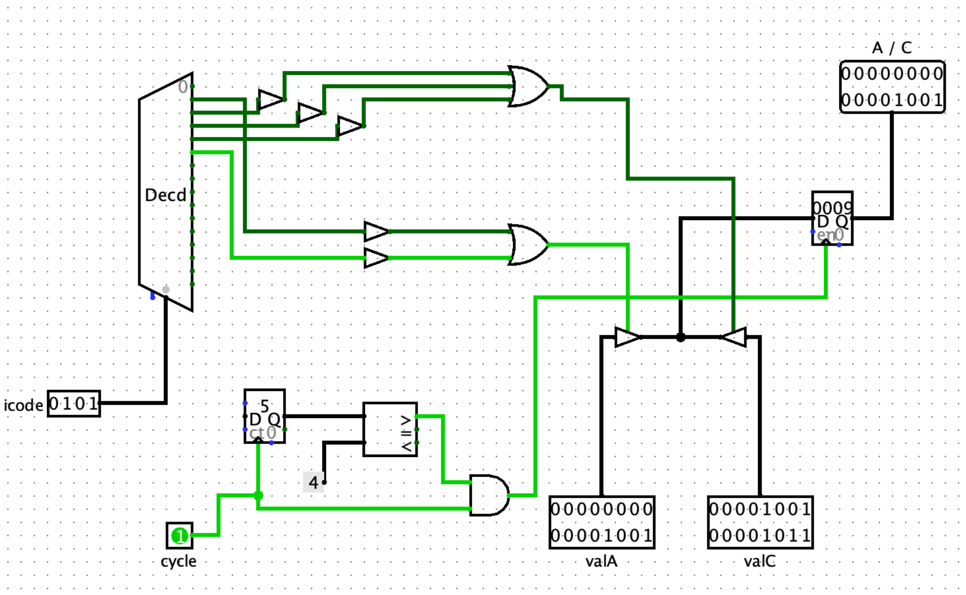
Execute

By Muneeb Hashmi

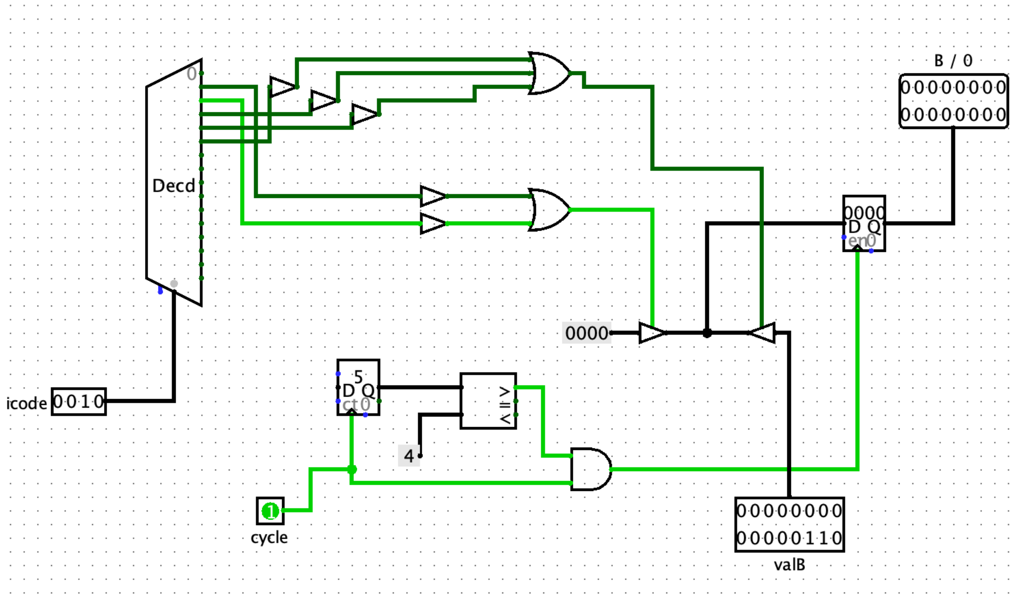


This is the main circuit for the execute block of our processor. The components include ALU A, ALU B, ALU Fun., the main ALU that combines these three ALUs, Set CC, and CC. The inputs needed for this stage are icode, ifun, and some combination of valA, valC, and valB (or zero). icode tells the processor which instruction to expect, and ifun further specifies, when needed, which function should be performed. For instance, in our opcode, 0x50 indicates addition. The first bit represents ifun, while the second bit represents icode. icode in this case is 5, which indicates that we are performing an arithmetic operation; ifun is 0, indicating addition. If ifun had a value of 1, we would perform multiplication instead. Another important component that isn't visible in the main circuit is a counter that counts the number of clock cycles that have occurred. This is present in every sub-circuit except Set CC, so it is better to explain it once rather than repeat myself. With our implementation, the fetch stage takes five clock cycles. Execute requires fetch to be completed first - as it needs the values of valA/B/C - so it can only start on the sixth clock cycle. Thus, a counter and comparator are needed to ensure that the number of clock cycles is greater than five before allowing anything to run (the sub ALUs can actually run on cycle 5, but the main ALU must wait until cycle 6; Set CC does not need to wait as it only takes in icode).

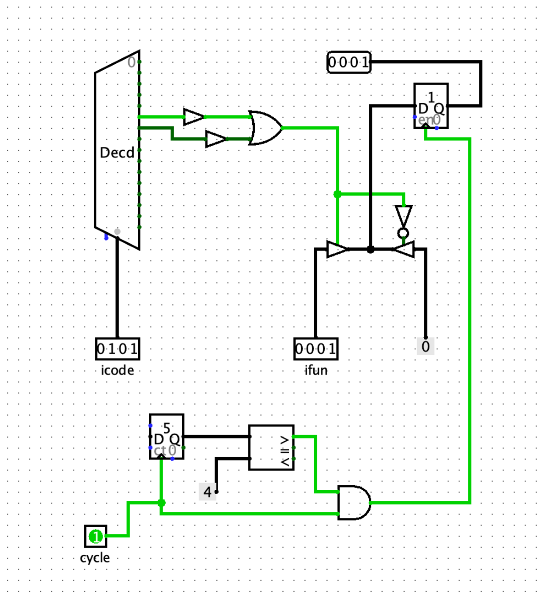
Beginning with the three "sub" ALUs, ALU A and ALU B have very simple functions: A selects either valA or valC, and B selects either valB or zero, depending on the value of icode. icode is decoded from four bits to 16 bits, so that we can apply logic in a simpler manner. The OR gates (shown below) stemming from the decoder check which bit is turned on (only one will be on at any given time), then send this information to one of two controlled buffers, which decide whether valA or valC will be chosen. For example, in the image below, icode is 0101, or 5. This indicates an operation (OP). In this case, the execute stage requires valA rather than valC. icode is decoded such that only the fifth bit is turned on, and, upon receiving this information, the controlled buffer for valA is switched on. valA is therefore the chosen value for the register, which is used to store the value for later clock cycles.



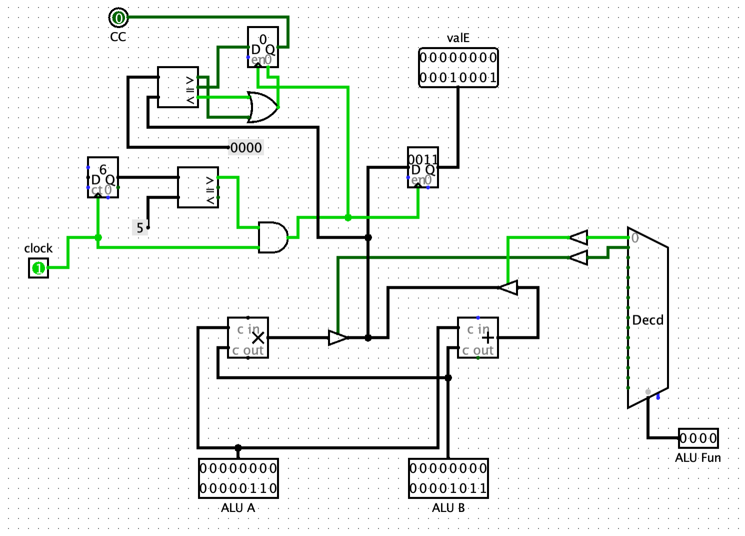
ALU B is very similar - it decides between valB and 0. In the image below, icode is 0010, representing the irmov instruction. In this case, execute does not require valB, so despite its value, 0 is chosen instead.

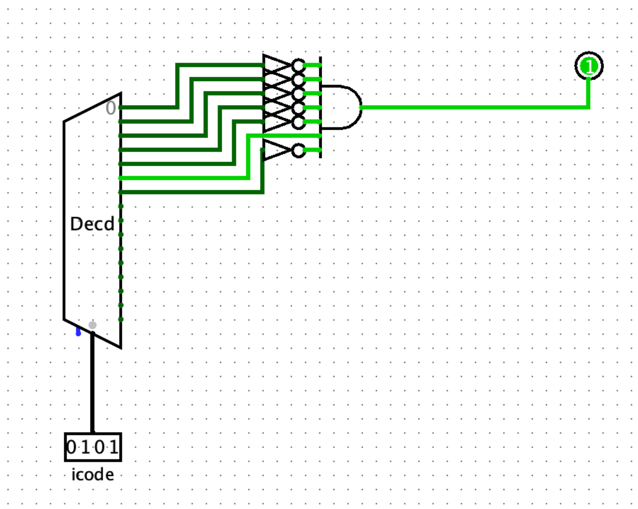


ALU Fun. on the other hand takes in icode and ifun, and decides which function to perform, if there is an option. Otherwise, it simply outputs 0. Looking at the image below, icode is 0101, so we have an operation. The processor now needs to know if this is an addition or multiplication (these are the only two operations we needed to implement). This information is provided by ifun: if ifun is 0, we have addition; if it is 1, we have multiplication. In this case, we have multiplication, so a value of 1 is passed to the register and output.



Moving on to the main ALU that takes in inputs from ALU A, ALU B, and ALU Fun., this is where the actual operations take place, the results of which are assigned/output to valE. As mentioned previously, only addition and multiplication were implemented. In the example below, icode (not shown) is 0100, representing st, or store, which is the same as Y86's rmmov instruction. In this instruction, the execute stage sets valE equal to valB + valC. Thus, the input from ALU A is that of valC, and the input from ALU B is that of valB. ALU Fun. can only be 0 in this case. The result of the addition is stored in a register, which then outputs to valE to be used later in the memory stage. The other output of this ALU is the CC, or condition code. This is useful for the OP instruction, which is the only instruction for which Set CC will return 1 (logic for Set CC is therefore very simple, as shown below). If Set CC is switched on, and the condition code from the ALU returns 1, the processor knows that an OP resulting in a zero value was performed, and it will update the PC value accordingly.

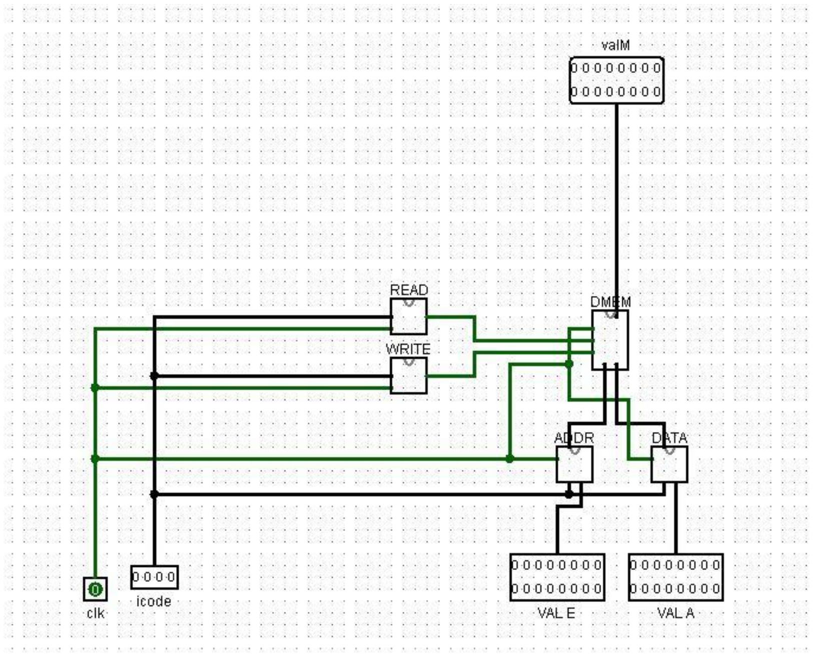




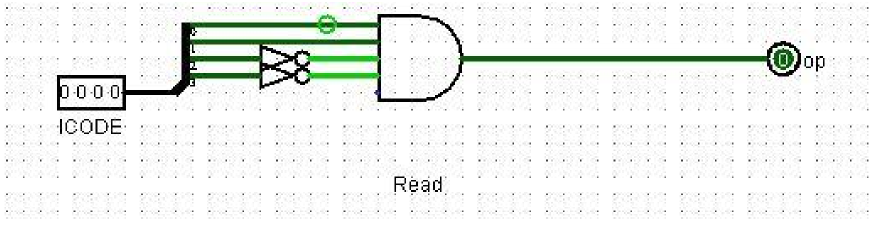
This covers all components necessary to implement the Execute stage of our architecture.

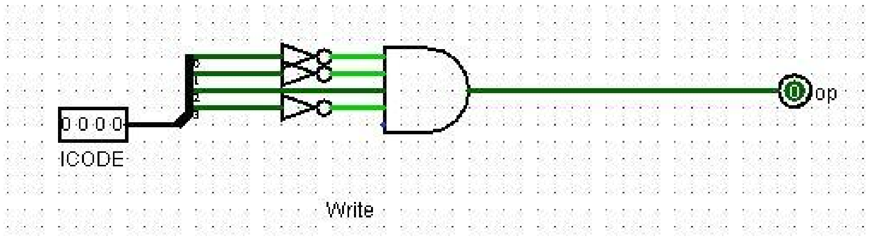
Memory

By Nick Martinez

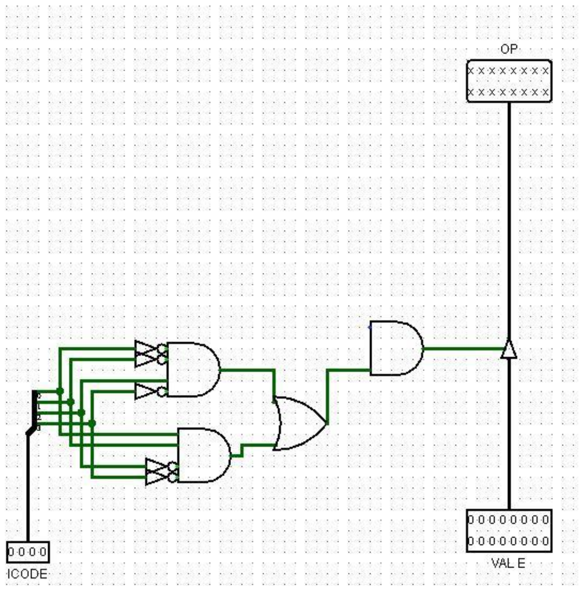


This is the main circuit for the memory block of our CPU. The memory block is composed of 5 units. There are two controller units called read and write which control what the main unit labeled DMEM does. DMEM pulls from the addresser and data units which basically just control what gets fed into DMEM.

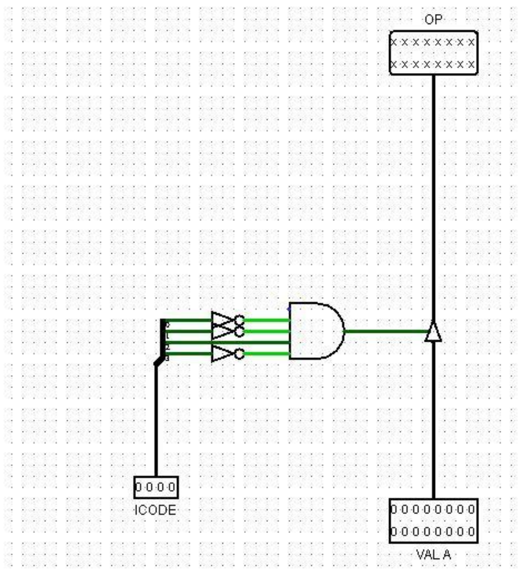




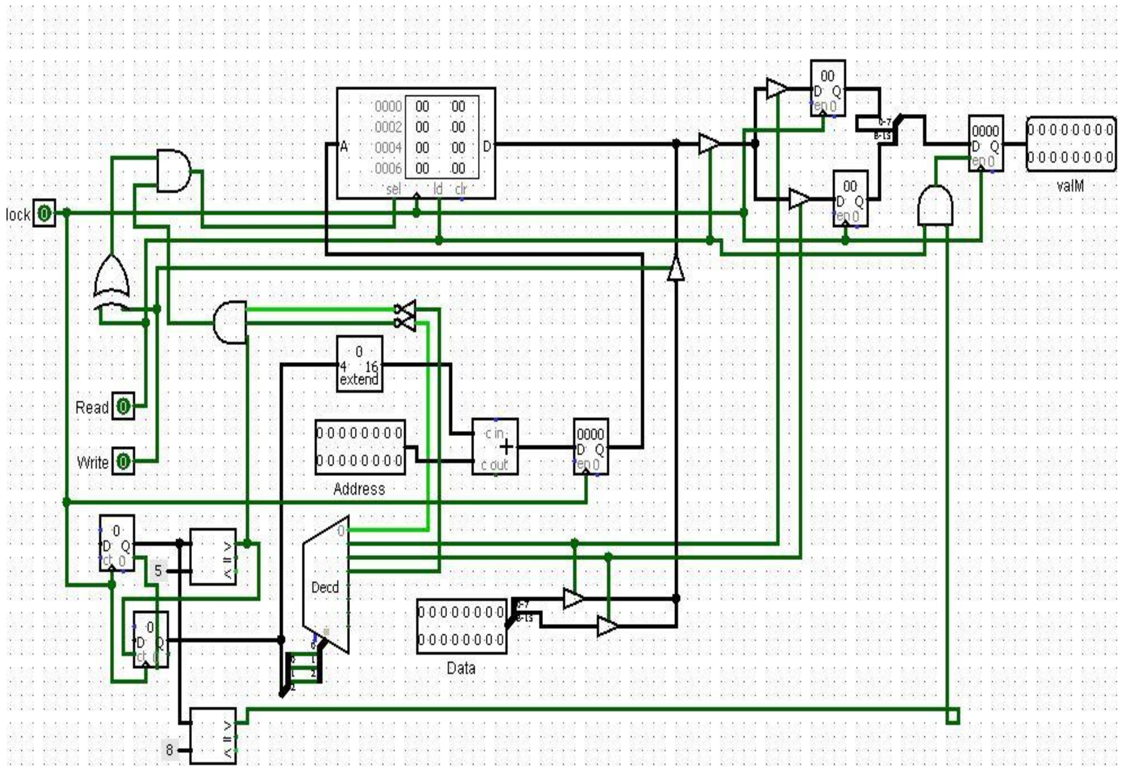
The read and write units both function pretty much the same. They use icode as an input and signal to the dmem unit if it needs to read the memory at valE or write valA at valE.



The picture above shows the address unit, it basically just outputs valE to the DMEM unit while the icode equals load or store.



Here the data unit works pretty much the same as the address unit. It outputs valA to DMEM while the icode for the store operation is present.



The dmem unit does most of the heavy lifting in the memory block for sure. It takes in the read and write signals from their respective units. Address and data are also attained from their respective units. A combination of counters and comparators is used to wait for the data to be read in and also to tell the RAM which byte address to read from/write to. valE(the address value) won't be ready until after 5 clock cycles have passed. If writing data, it takes an additional

4 cycles to write both bytes to the RAM at the address. If reading from the ram, it takes 5 additional clock cycles to read both of the bytes at the address and output them to valM.

C Code/Assembly/Binary

By Patricio Bunt

The first step required to generate the input binary for our CPU, was to create a C program to demonstrate the intended results/functionality. This C code is shown in Snippet 1.

Snippet 1: Adding two 2x2 matrices in C

```
void main()
{
  int a[2][2] = {{1,7},{4,2}};
  int b[2][2] = {{9,3},{6,8}};
  int c[2][2];

  int* pa = *a;
  int* pb = *b;
  int* pc = *c;
  int count = 4;

  for (int i = 0; i < count; i++){
     *pc = *pa+ *pb;
     pa +=1;
     pb +=1;
     pc +=1;
  }
}
```

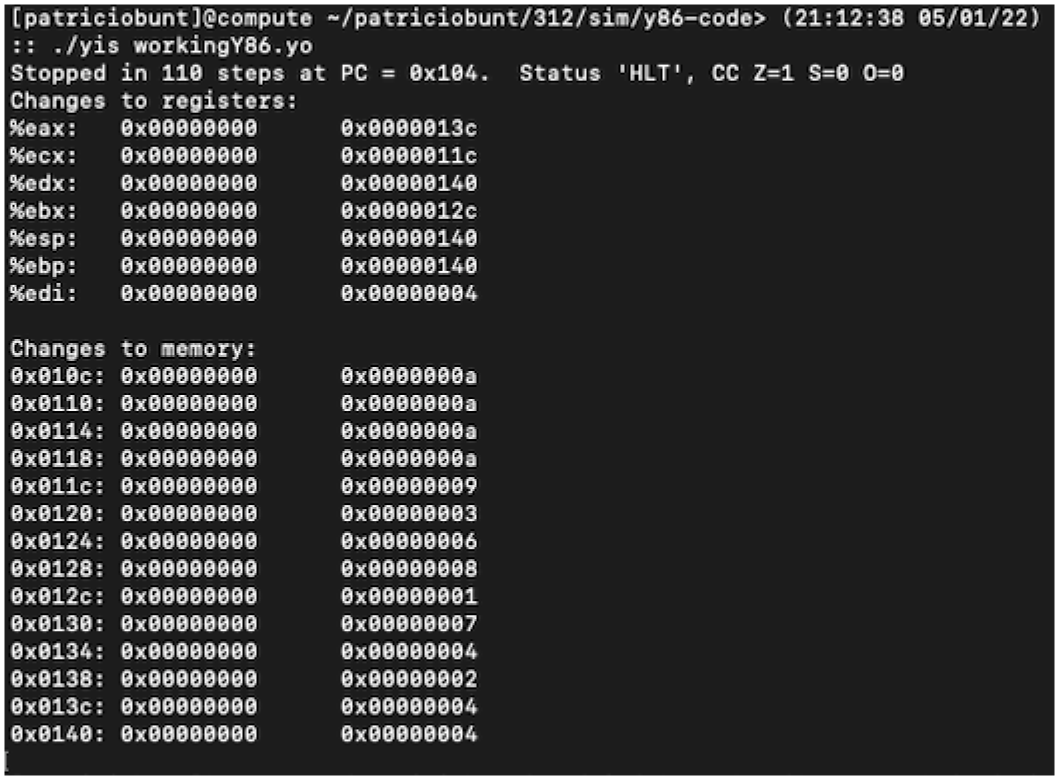
After this C code was generated, the respective x86 code was generated. Using this x86 code as a reference, I was able to reconstruct the program using the y86 assembly language instruction set. This y86 code is shown in Snippet 2 and its output in Image 1.

Snippet 2- Adding two 2x2 matrices in Y86

```
init:
 irmovl 0x140, %ebp # initializing base pointer
 irmovl 0x140, %esp # initializing stack pointer
 irmovl $1, %esi #esi = 1
 rmmovl %esi, -20(%ebp) #a[0][0] = esi
 irmovl $7, %esi #esi = 7
 rmmovl %esi, -16(%ebp) #a[0][1] = esi
 irmovl $4, %esi #esi = 4
 rmmovl %esi, -12(%ebp) #a[1][0] = esi
 irmovl $2, %esi #esi = 2
 rmmovl %esi, -8(%ebp) #a[1][1] = esi
 irmovl $9, %esi #esi = 9
 rmmovl %esi, -36(%ebp) #b[0][0] = esi
 irmovl $3, %esi #esi = 3
 rmmovl %esi, -32(%ebp) #b[0][1] = esi
 irmovl $6, %esi #esi = 6
 rmmovl %esi, -28(%ebp) #b[1][0] = esi
 irmovl $8, %esi #esi = 8
 rmmovl %esi, -24(%ebp) #b[1][1] = esi
 irmovl $4, %esi #esi = 4
 rmmovl %esi, -4(%ebp) # count = 4, we subtract 4 to make it point to count
 irmovl $0, %esi #esi = 0
 rmmovl %esi, (%ebp) #i = esi = 0
 rrmovl %ebp, %eax #eax = ebp, next we subtract 20 to make it point to 'a' matrix
 irmovl $-20, %esi #esi = 20
 addl %esi, %eax #eax = a[][]
 rrmovl %ebp, %ebx #ebx = ebp, next we subtract 36 to make it point to 'b' matrix
 irmovl $-36, %esi
 addl %esi, %ebx #ebx = b[][]
 rrmovl %ebp, %ecx #ecx = ebp, next we subtract 52 to make it point to 'c' matrix
 irmovl $-52, %esi
 addl %esi, %ecx #ecx = c[][]
 rrmovl %ebp, %edx #edx = ebp, next we subtract 0 to make it point to 'i'
 irmovl $0, %esi
 addl %esi, %edx #edx = i
L1:
 irmovl $0, %esi # clearing esi
 mrmovl (%eax), %esi #esi = *pa
 irmovl $0, %edi # clearing edi
 mrmovl (%ebx), %edi #edi = *pb
 addl %esi, %edi #edi = *pa + *pb
 rmmovl %edi, (%ecx) #*pc = *pa + *pb
 irmovl $4, %esi #esi = 4
 addl %esi, %eax #pa +=1
 addl %esi, %ebx #pb +=1
 addl %esi, %ecx #pc +=1
 irmovl $1, %esi #esi = 1
 irmovl $0, %edi #clear edi
 mrmovl (%edx), %edi #edi = *i
 addl %esi, %edi #i++
 rmmovl %edi, (%edx) #setting value of i in mem
 mrmovl -4(%ebp), %esi #esi = count
 subl %edi, %esi #edi = count - i
 je L2 #jump if count == i (i.e. looped 'count' times)
 jmp L1 # if count != i, loop
L2:
 halt # terminate

.pos 0x140
Stack:
```

Image 1. Output of Y86 Matrix Addition code.



```
[patriciobunt]@compute ~/patriciobunt/312/sim/y86-code> (21:12:38 05/01/22)
:: ./yis workingY86.yo
Stopped in 110 steps at PC = 0x104.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x0000013c
%ecx:   0x00000000      0x0000011c
%edx:   0x00000000      0x00000140
%ebx:   0x00000000      0x0000012c
%esp:   0x00000000      0x00000140
%ebp:   0x00000000      0x00000140
%edi:   0x00000000      0x00000004

Changes to memory:
0x010c: 0x00000000      0x0000000a
0x0110: 0x00000000      0x0000000a
0x0114: 0x00000000      0x0000000a
0x0118: 0x00000000      0x0000000a
0x011c: 0x00000000      0x00000009
0x0120: 0x00000000      0x00000003
0x0124: 0x00000000      0x00000006
0x0128: 0x00000000      0x00000008
0x012c: 0x00000000      0x00000001
0x0130: 0x00000000      0x00000007
0x0134: 0x00000000      0x00000004
0x0138: 0x00000000      0x00000002
0x013c: 0x00000000      0x00000004
0x0140: 0x00000000      0x00000004
```

Once this Y86 code was created, we could begin to work on the binary input that will be used for our custom CPU. This was a bit tricky however, as we decided to implement the multiply command, namely “mul” into our ISA, however this command is not valid in the Y86 architecture. Inversely, the code from Snippet 2 requires the use of the subtraction command, namely “subl”, which we did not implement into our ISA. For this reason, a workaround had to be thought up.

In order to avoid the use of invalid commands, a separate Y86 code was created, which did not use the “subl” command, and reserved a spot for the “mul” command. This spot was reserved using the “addl” command, as the instruction is structured in the same manner as the “mul” command. This permitted us to compile the Y86 code, and have an output which we could then manually manipulate in accordance with what our CPU would take as input. This output can be seen in Snippet 3.

Snippet 3 - Compiled Y86 Matrix Addition code designed for use with custom CPU

```
0x000:              | init:
0x000: 30f544020000 |   irmovl 0x244, %ebp #
0x006: 30f444020000 |   irmovl 0x244, %esp
0x00c: 30f601000000 |   irmovl $1, %esi #esi = 1
0x012: 4065ecffffff |   rmmovl %esi, -20(%ebp) #a[0][0] = esi
0x018: 30f607000000 |   irmovl $7, %esi #esi = 7
0x01e: 4065f0ffffff |   rmmovl %esi, -16(%ebp) #a[0][1] = esi
0x024: 30f604000000 |   irmovl $4, %esi #esi = 4
0x02a: 4065f4ffffff |   rmmovl %esi, -12(%ebp) #a[1][0] = esi
0x030: 30f602000000 |   irmovl $2, %esi #esi = 2
0x036: 4065f8ffffff |   rmmovl %esi, -8(%ebp) #a[1][1] = esi
0x03c: 30f609000000 |   irmovl $9, %esi #esi = 9
0x042: 4065dcffffff |   rmmovl %esi, -36(%ebp) #b[0][0] = esi
0x048: 30f603000000 |   irmovl $3, %esi #esi = 3
0x04e: 4065e0ffffff |   rmmovl %esi, -32(%ebp) #b[0][1] = esi
0x054: 30f606000000 |   irmovl $6, %esi #esi = 6
0x05a: 4065e4ffffff |   rmmovl %esi, -28(%ebp) #b[1][0] = esi
0x060: 30f608000000 |   irmovl $8, %esi #esi = 8
0x066: 4065e8ffffff |   rmmovl %esi, -24(%ebp) #b[1][1] = esi
0x06c: 30f604000000 |   irmovl $4, %esi #esi = 4
0x072: 4065fcffffff |   rmmovl %esi, -4(%ebp) #count = esi = 4
0x078: 30f600000000 |   irmovl $0, %esi #esi = 0
0x07e: 406500000000 |   rmmovl %esi, (%ebp) #i = esi = 0
0x084: 2050         |   rrmovl %ebp, %eax #eax = ebp, then subtract 48 (ADDR 00d0)
0x086: 30f6ecffffff |   irmovl $-20, %esi
0x08c: 6060         |   addl %esi, %eax #eax = a[][]
0x08e: 2053         |   rrmovl %ebp, %ebx #ebx = ebp, then subtract 64 (ADDR 00c0)
0x090: 30f6dcffffff |   irmovl $-36, %esi
0x096: 6063         |   addl %esi, %ebx #ebx = b[][]
0x098: 2051         |   rrmovl %ebp, %ecx #ecx = ebp, then subtract 80 (ADDR 00b0)
```

```
0x09a: 30f6ccffffff |   irmovl $-52, %esi
0x0a0: 6061         |  addl %esi, %ecx #ecx = c[][]
0x0a2: 2052         |  rrmovl %ebp, %edx #edx = ebp, then subtract 28 (ADDR 00e4)
0x0a4: 30f600000000 |   irmovl $0, %esi
0x0aa: 6062         |  addl %esi, %edx #edx = i
0x0ac:              | L1:
0x0ac: 30f600000000 |   irmovl $0, %esi
0x0b2: 506000000000 |   mrmovl (%eax), %esi #esi = *pa
0x0b8: 30f700000000 |   irmovl $0, %edi
0x0be: 507300000000 |   mrmovl (%ebx), %edi #edi = *pb
0x0c4: 6067         |  addl %esi, %edi #edi = *pa + *pb
0x0c6: 407100000000 |   rmmovl %edi, (%ecx) #*pc = *pa + *pb
0x0cc: 30f604000000 |   irmovl $4, %esi #esi = 4
0x0d2: 6060         |  addl %esi, %eax #pa +=1
0x0d4: 6063         |  addl %esi, %ebx #pb +=1
0x0d6: 6061         |  addl %esi, %ecx #pc +=1
0x0d8: 30f601000000 |   irmovl $1, %esi #esi = 1
0x0de: 30f700000000 |   irmovl $0, %edi #clear edi
0x0e4: 507200000000 |   mrmovl (%edx), %edi #edi = *i
0x0ea: 6067         |  addl %esi, %edi #i++
0x0ec: 407200000000 |   rmmovl %edi, (%edx) #setting value of i in mem
0x0f2: 30f6ffffffff |   irmovl $-1, %esi # making esi -1, to multiply it with i
0x0f8: 6067         |  addl %esi, %edi #multiply edi (i) by -1 [CHANGE addl opcode to mul
opcode]
0x0fa: 5065fcffffff |   mrmovl -4(%ebp), %esi #esi = count
0x100: 6076         |  addl %edi, %esi #esi = count + -i
0x102: 730c010000   |  je L2 ##jump if count==i
0x107: 70ac000000   |  jmp L1 #jump if count>i     (i<count)
0x10c:              | L2:
0x10c: 00           |  halt
                    |
0x244:              | .pos 0x244
0x244:              | Stack:
```

Having this, we were then able to remove all of the excess unnecessary clutter, such as the comments, as well as the memory addresses at each line of code. In order to input the code into our Logisim CPU, each bit had to be separated by a space, and finally the OPcodes had to be changed for every single command according to the ISA seen in Table 1. In certain cases, the order in which the registers and memory addresses had to be reversed as well, due to the nature of our ISA. The final input code that is to be put into the CPU is seen in Snippet 4.

Table 1. Instruction Set Architecture of our CPU vs. Y86

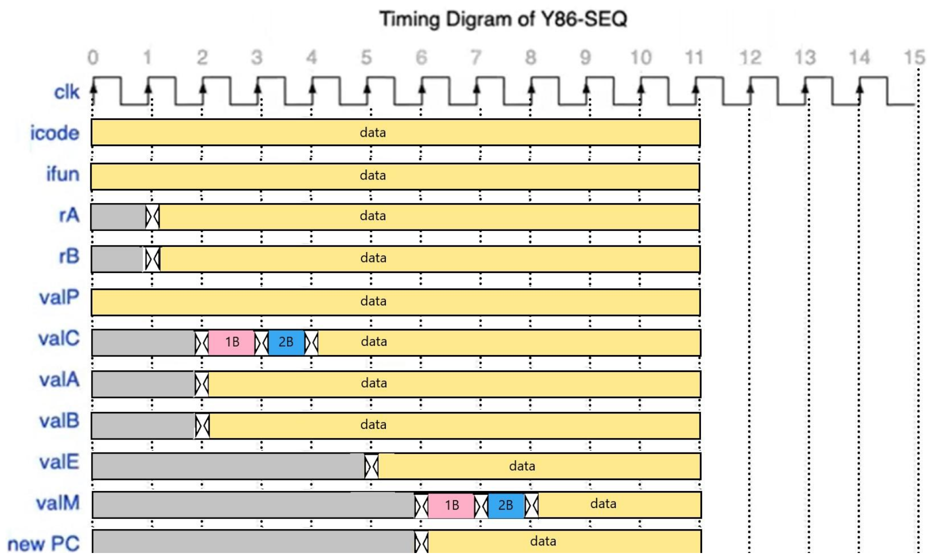
INSTRUCTION	OPCODE	Y86	OPCODE
halt	0x00	halt	0x00
rrmov regA, regB	0x10	rrmov	0x20
irmov imm, reg A	0x20	irmovl	0x30
ld regA, <memory address>	0x30	mrmov	0x50
st regA, <memory address>	0x40	rmmov	0x40
add regA, regB	0x50	add	0x60
mul regA, regB	0x51	mul	INVALID
jmp <memory address>	0x60	jmp	0x70
jz <memory address>	0x61	lg	0x73

Snippet 4 - Final binary input using custom ISA

```
v2.0 raw
20 5f 40 01
20 4f 40 01
20 6f 01 00
40 65 ec ff
20 6f 07 00
40 65 f0 ff
20 6f 04 00
40 65 f4 ff
20 6f 02 00
40 65 f8 ff
20 6f 09 00
40 65 dc ff
20 6f 03 00
40 65 e0 ff
20 6f 06 00
40 65 e4 ff
20 6f 08 00
```

```
40 65 e8 ff
20 6f 04 00
40 65 fc ff
20 6f 00 00
40 65 00 00
10 50
20 6f ec ff
50 06
10 53
20 6f dc ff
50 36
10 51
20 6f cc ff
50 16
10 52
20 6f 00 00
50 26
20 6f 00 00
30 60 00 00
20 7f 00 00
30 73 00 00
50 76
40 71 00 00
20 6f 04 00
50 06
50 36
50 16
20 6f 01 00
20 7f 00 00
30 72 00 00
50 76
40 72 00 00
20 6f ff ff
51 76
30 65 fc ff
50 67
30 72 00 00
61 c0 00
60 78 00
00
```

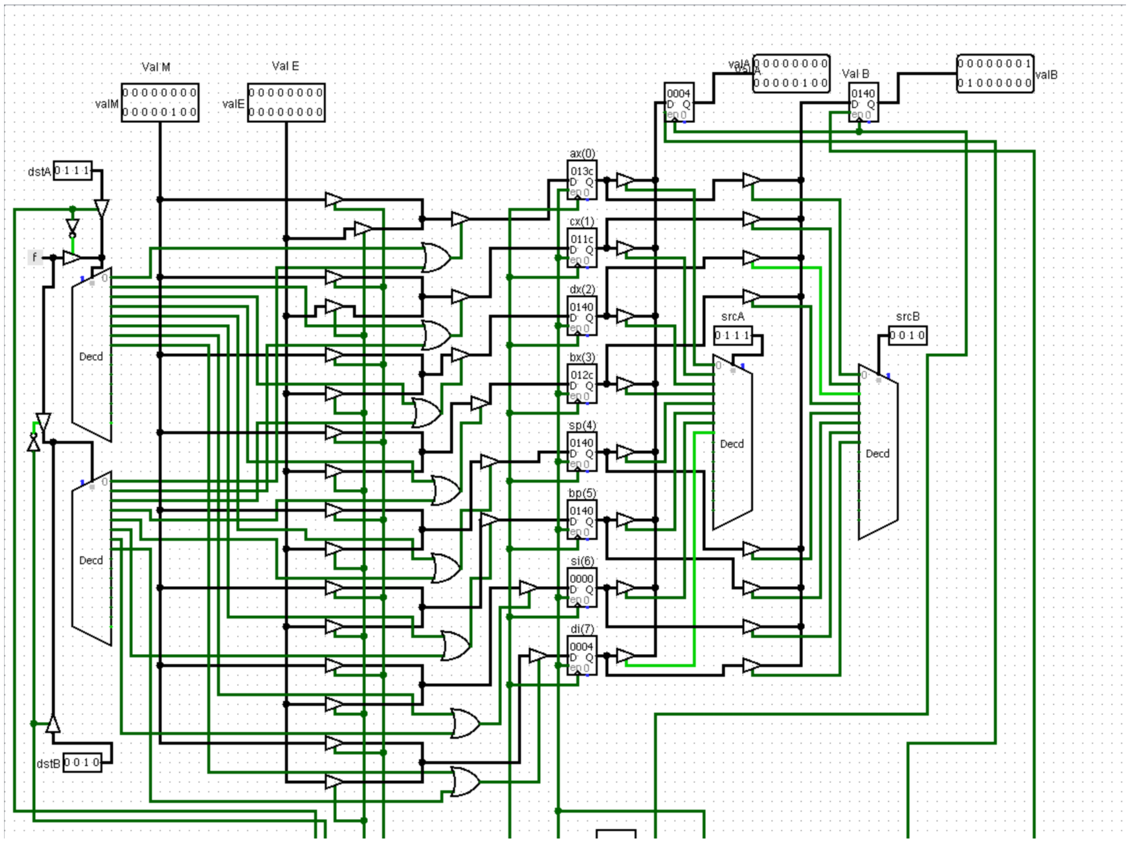
Timing Diagram



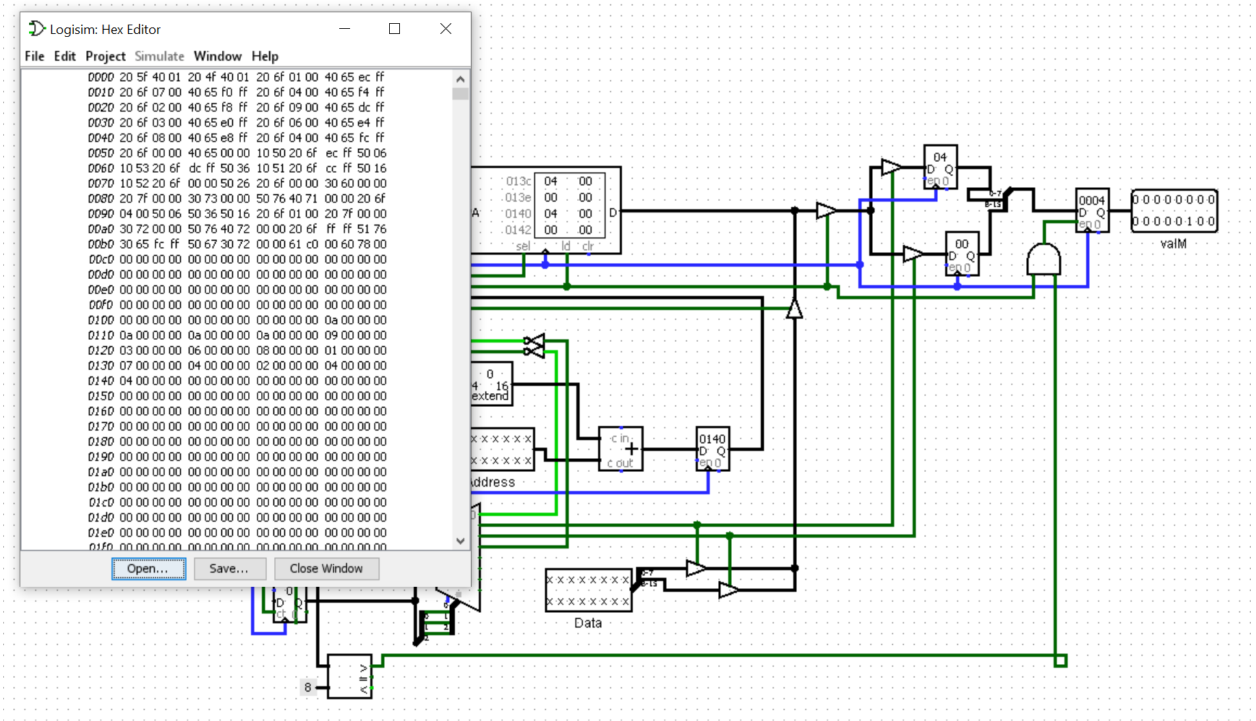
Input Selection Table

	Fetch	Fetch	Decode	Decode	Writeback	Writeback	Execution	Execution	Execution	Execution	Memory	Memory	Memory	Memory
Instructions	need_regids	need_valc	srcA	srcB	dstA	dstB	ALU A	ALU B	ALU fun.	set CC	Mem Read	Mem Write	Mem Addr	Mem Data
	(0/1)	(0/1)	(rA)	(rB)	(rA)	(rB)	valA/valC	valB/0x0	0/1	0/1	0/1	0/1	valE/valA	valA
halt	0	0	x	x	x	x	x	x	x	x	x	x	x	x
rrmov rA, rB	1	0	rA	rB	x	rB	valA	0x0	0	0	0	0	x	x
irmov V, rA	1	1	x	x	rA	x	valC	0x0	0	0	0	0	x	x
ld regA, rB(V)	1	1	x	rB	rA	x	valC	valB	0	0	1	0	valE	x
st regA, rb(V)	1	1	rA	rB	x	x	valC	valB	0	0	0	1	valE	valA
add rA, rB	1	0	rA	rB	rA	x	valA	valB	0	1	0	0	x	x
mul rA, rB	1	0	rA	rB	rA	x	valA	valB	1	1	0	0	x	x
jmp V	0	1	x	x	x	x	x	x	0	0	0	0	x	x
jz V	0	1	x	x	x	x	x	x	1	0	0	0	x	x

Expected output of running process with CPU input



This shows the registers after running the instructions from our program.



```
0000 20 5f 40 01  20 4f 40 01  20 6f 01 00  40 65 ec ff
0010 20 6f 07 00  40 65 f0 ff  20 6f 04 00  40 65 f4 ff
0020 20 6f 02 00  40 65 f8 ff  20 6f 09 00  40 65 dc ff
0030 20 6f 03 00  40 65 e0 ff  20 6f 06 00  40 65 e4 ff
0040 20 6f 08 00  40 65 e8 ff  20 6f 04 00  40 65 fc ff
0050 20 6f 00 00  40 65 00 00  10 50 20 6f  ec ff 50 06
0060 10 53 20 6f  dc ff 50 36  10 51 20 6f  cc ff 50 16
0070 10 52 20 6f  00 00 50 26  20 6f 00 00  30 60 00 00
0080 20 7f 00 00  30 73 00 00  50 76 40 71  00 00 20 6f
0090 04 00 50 06  50 36 50 16  20 6f 01 00  20 7f 00 00
00a0 30 72 00 00  50 76 40 72  00 00 20 6f  ff ff 51 76
00b0 30 65 fc ff  50 67 30 72  00 00 61 c0  00 60 78 00
00c0 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
00d0 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
00e0 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
00f0 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
0100 00 00 00 00  00 00 00 00  00 00 00 00  0a 00 00 00
0110 0a 00 00 00  0a 00 00 00  0a 00 00 00  09 00 00 00
0120 03 00 00 00  06 00 00 00  08 00 00 00  01 00 00 00
0130 07 00 00 00  04 00 00 00  02 00 00 00  04 00 00 00
0140 04 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
0150 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
0160 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
0170 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
0180 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
0190 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
01a0 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
01b0 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
01c0 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
01d0 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
01e0 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
01f0 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
```

This shows the memory after running our instructions from our program.