We took notes over the videos and created tables alongside helpful screenshots from the videos. This can be seen in the tables.pdf file. This was our primary resource of information that we used to construct our own y86 architecture.

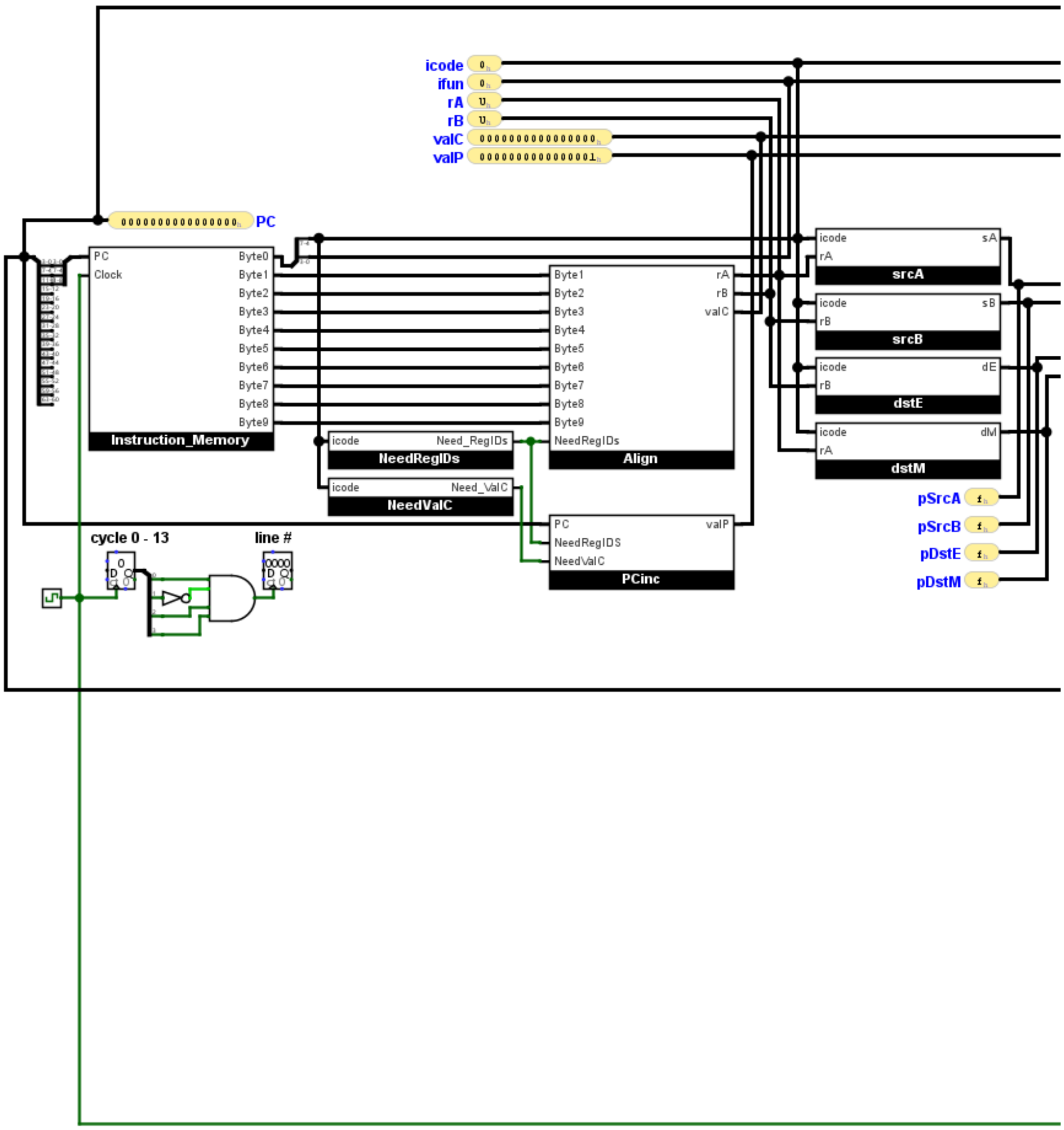
Now, we will take screenshots of each stage in the circuit and describe it very briefly.

Then, we will show the outputs of our y86 circuit of 3 example programs written in y86 assembly. One assembly file was written by Kevin Tang and it is the 2x2 multiplication program named lab6.ys. This is because he is in the honors section for this class. We will then compare the outputs of our y86 circuit to the output from executing the same object file using ./yis and prove they are equal.

Additionally, here is the same timing diagram as the pdf version.

Clock	ON	off	ON	off	ON	off	ON	off	ON	off	On	off	ON	off	ON	off	ON	off	ON	off	ON	off	ON	off	On	off	ON	off
Cycle #	1		2		3		4		5		6		7		8		9		10		11		12		13		14	
Fetch	byte0 icode ifun		byte1 Ra Rb		byte2		byte3		byte4		byte5		byte6		byte7		byte8		byte9									
Decode			valA valB																									
Execute																					valE							
Memory																									valM			
Write Back																											dstE dstM	
PC update																											newPC	

Fetch Stage:



Instruction Memory:

- Contains a 4096x8 ROM that contains memory imported from a mem file.
- PC is the address that indicates where to start reading from ROM.
- We read 10 bytes per instruction, and therefore need 10 cycles to load each byte correctly.

NeedRegIDs and NeedvalC

- These modules take in icode (the first 4 bits of the first byte) and determine whether the corresponding instruction needs to specify register IDs and/or a val C.
- The tables and logic can be found in the tables.pdf

PCinc

- This module calculates the next line of operation based on the current instruction, outputting a valP based on this formula: valP = PC + 1 + (NeedRegIds) + 8(NeedvalC), where NeedsRegIds and NeedvalC are single bits.

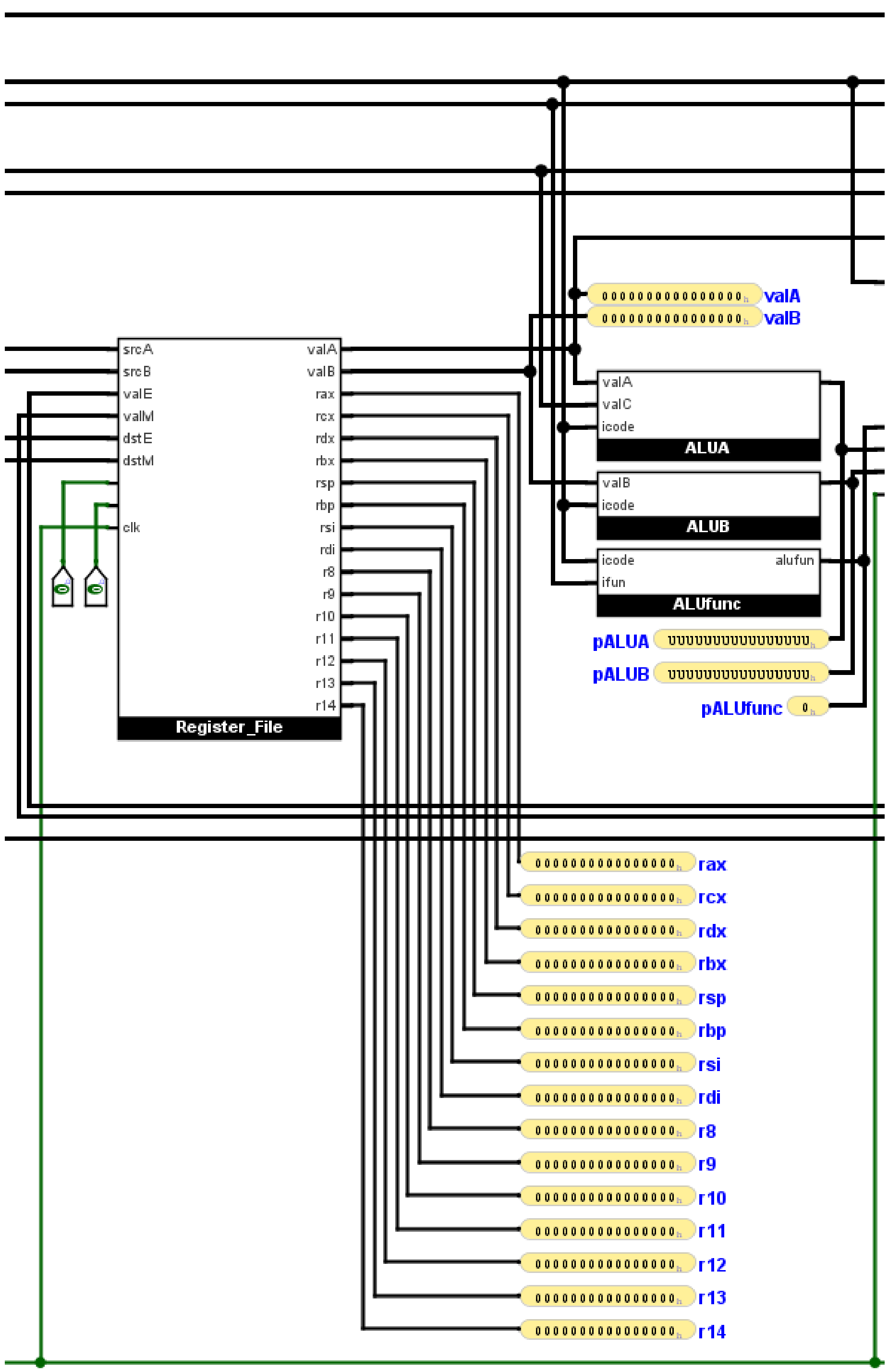
Align

- This module outputs 3 things, rA, rB, and valC. If the current icode does not need register IDs, then rA and rB will be set to 0xf which is an empty or trash register and valC will consist of the bytes 1-8. Otherwise, if the current icode does need register IDs, then valC will be the bytes 2-9.

SrcA and SrcB

- After the first two cycles, we now have icode:ifun, and rA:rB. If rA or rB is not needed, then they are set to 0xf. Regardless, we have the necessary two bytes to determine the source register for the current operation. This leads to the decode stage.
- DstM and DstE are a part of the Write Back stage.

Decode Stage:



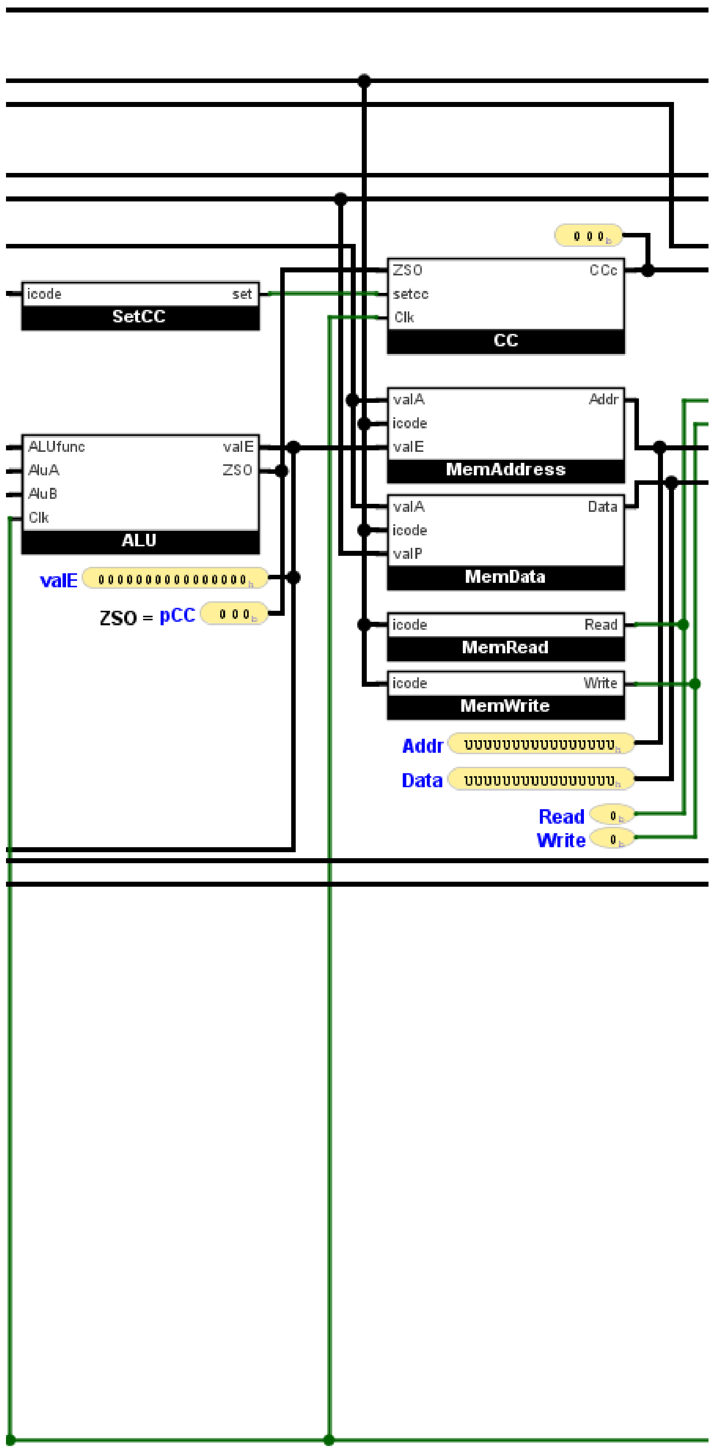
Register FIle

- This holds a 2 write, 2 read 64x16 register file. This means that there are 16 registers that can each hold 64 bits. During decode, we utilize both read ports, while in the write back stage we utilize the write ports.

Once valA and valB are retrieved based either from the register or a constant depending on the the icode, we still need valC (an input into ALUA) in order to make sure the ALU correctly executes. Therefore, we must wait until all 10 cycles are completed before moving onto the next cycle which we will execute.

- ALUA, ALUB, and ALUfunc all determine the operation the ALU should perform based on the current instruction's icode and ifun.

Execute:



ALU

- On the 11th cycle, we will perform the proper operation to the inputs, outputting valE, and the conditional flags Zero flag, Sign flag, and Overflow flag represented by bits in a 3-bit bus from left to right respectively.

SetCC

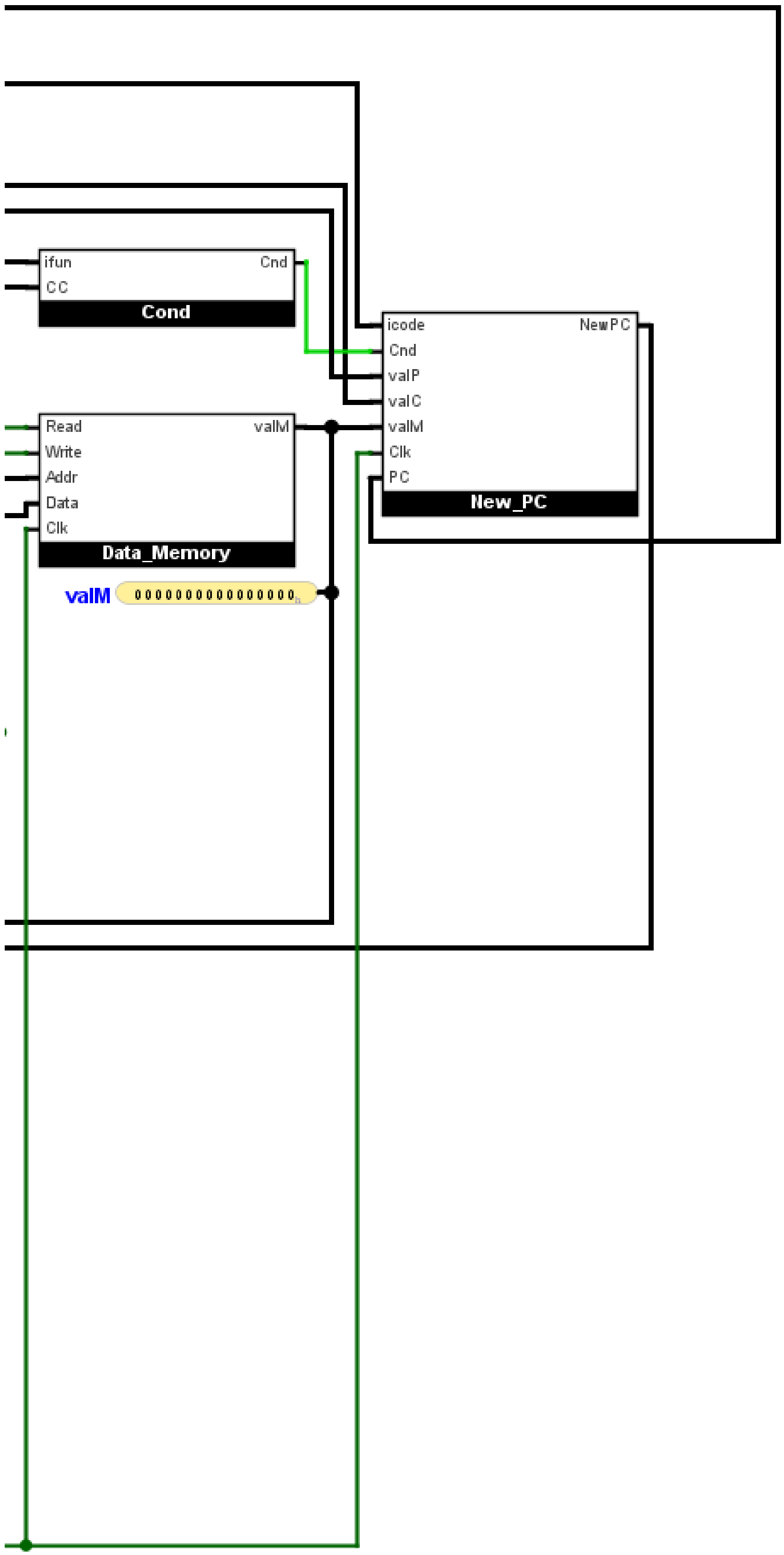
- This module only will enable the CC to change its value when the current instruction is OPq.

CC

- This module holds a register that contains the most recent OPq conditional statement,

Once we have calculated valE, we can now perform the Memory Stage on the 12th cycle.
The MemAddress, MemData, MemRead, and MemWrite are just more logic modules that select the correct inputs/enable for the memory based on the icode.

Memory:


ifun
Cnd
CC
Cond
icode
NewPC
Cnd
valP
valC
valM
Clk
PC
New_PC
Read
valM
Write
Addr
Data
Clk
Data_Memory
valM
0000000000000000

Data Memory

- This holds the main memory of the y86 architecture, a 4096x64 RAM. Based on the MemAddr, MemData, and MemRead/Write modules, valM will be correctly computed and outputted.

Cond

- This is primarily used for the jXX instruction.
- The Cond module determines the type of jump instruction based on ifun and determines whether or not the condition to jump is true or false. Other instructions will not use the Cond module.

The entire circuit has been shown, so the following stages refer back to the images above.

Write Back:

- Once ValE (cycle 11) and ValM (cycle 12) has been computed correctly, we can now update the register file based on the current instruction on the 13th cycle. As mentioned above, DstE, DstM determine which register to write which data into. If the current instruction does not need to write back, the DstE and/or DstM are set to 0xf to write to the trash register.

PC Update:

- Once the instruction has been fully executed, we can finally update the PC to the next line of instruction on the 14th cycle. The new value of PC depends solely on the current instruction's icode. This value will then be sent back to the fetch stage and it will be used as the next address to read the next instruction from instruction memory, Asum Object File

```
                              | # Execution begins at address 0
0x000:                        |   .pos 0
0x000: 30f40002000000000000 |   irmovq stack, %rsp      # Set up stack pointer
0x00a: 803800000000000000   |   call main       # Execute main program
0x013: 00                   |   halt            # Terminate program
                              |
                              | # Array of 4 elements
0x018:                        |   .align 8
0x018: 0d000d000d000000     | array:    .quad 0x000d000d000d
0x020: c000c000c0000000     |   .quad 0x00c000c000c0
0x028: 000b000b000b0000     |   .quad 0x0b000b000b00
0x030: 00a000a000a00000     |   .quad 0xa000a000a000
                              |
0x038: 30f71800000000000000 | main: irmovq array,%rdi
0x042: 30f60400000000000000 |   irmovq $4,%rsi
0x04c: 805600000000000000   |   call sum        # sum(array, 4)
0x055: 90                   |   ret
                              |
                              | # long sum(long *start, long count)
                              | # start in %rdi, count in %rsi
0x056: 30f80800000000000000 | sum:  irmovq $8,%r8        # Constant 8
0x060: 30f90100000000000000 |   irmovq $1,%r9        # Constant 1
0x06a: 6300                 |   xorq %rax,%rax       # sum = 0
0x06c: 6266                 |   andq %rsi,%rsi       # Set CC
0x06e: 708700000000000000   |   jmp     test         # Goto test
0x077: 50a70000000000000000 | loop: mrmovq (%rdi),%r10   # Get *start
0x081: 60a0                 |   addq %r10,%rax       # Add to sum
0x083: 6087                 |   addq %r8,%rdi        # start++
0x085: 6196                 |   subq %r9,%rsi        # count--.  Set CC
0x087: 747700000000000000   | test: jne    loop          # Stop when 0
0x090: 90                   |   ret                  # Return
                              |
                              | # Stack starts here and grows to lower addresses
0x200:                        |   .pos 0x200
0x200:                        | stack:
```

NOTE: reading these output follow the example at the bottom of our tables.pdf - these can be confusing

To give another example:
The value shown in the asum output below for %rax is 0x0000abcdabcdabcd
It is actually the value: 0xcdabcdabcdab0000
- However the value for %rsp is 0x0000000000000200 which is simply the value 0x200
- Even we are a bit confused on how to read the output properly every time without looking at the object file

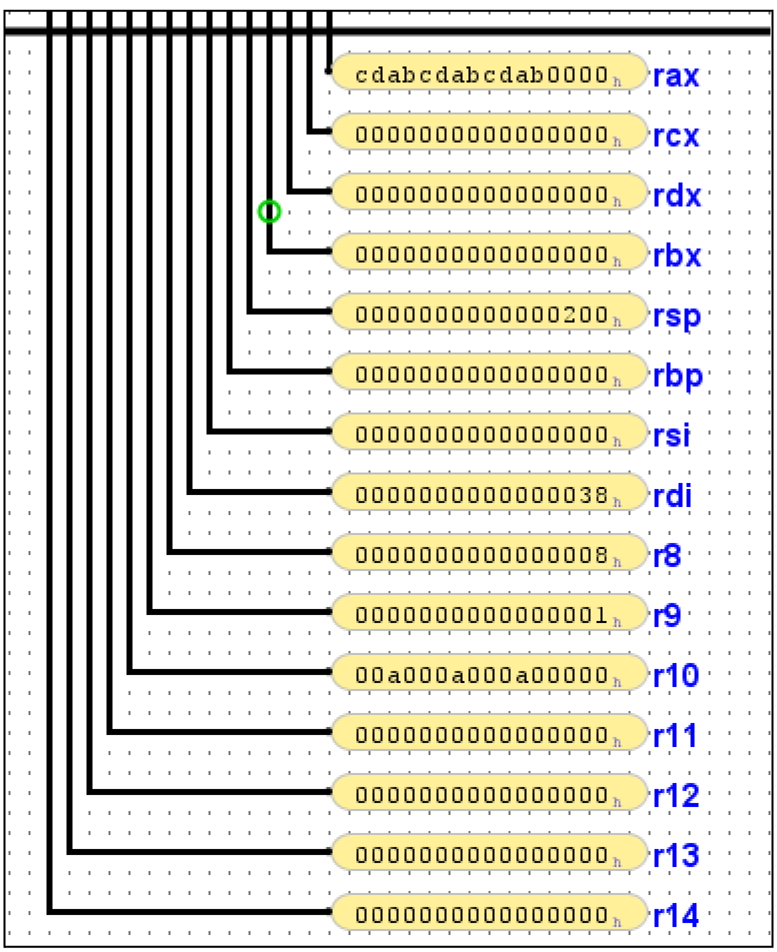
Asum ouptut

```
[kevintang7215]@linux2 ~/sim/misc> (19:08:36 05/02/23)
:: ./yis asum.yo
Stopped in 34 steps at PC = 0x13.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%rax:	0x0000000000000000	0x0000abcdabcdabcd
%rsp:	0x0000000000000000	0x0000000000000200
%rdi:	0x0000000000000000	0x0000000000000038
%r8:	0x0000000000000000	0x0000000000000008
%r9:	0x0000000000000000	0x0000000000000001
%r10:	0x0000000000000000	0x0000a000a000a000

Changes to memory:
0x01f0:	0x0000000000000000	0x0000000000000055
0x01f8:	0x0000000000000000	0x0000000000000013

[kevintang7215]@linux2 ~/sim/misc> (19:11:55 05/02/23)
::
```

Our output



1e8	0000000000000000
1f0	0000000000000055
1f8	0000000000000013
200	0000000000000000
208	0000000000000000

Poptest object file

```
                            | # Test of Pop semantics for Y86-64
0x000: 30f40001000000000000 |   irmovq $0x100,%rsp   # Initialize stack pointer
0x00a: 30f0cdab000000000000 |   irmovq $0xABCD,%rax
0x014: a00f                 |   pushq  %rax          # Put known value on stack
0x016: b04f                 |   popq   %rsp          # Either get 0xABCD, or 0x100
0x018: 00                   |   halt
```

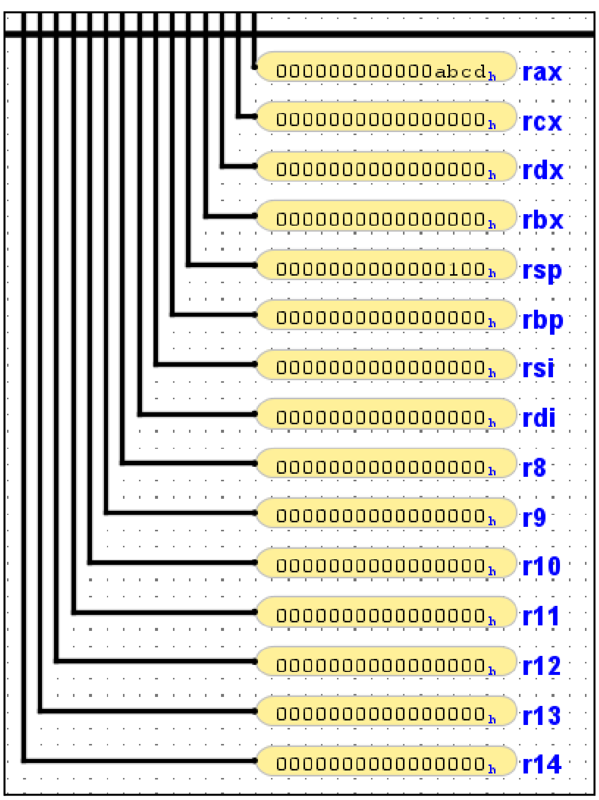
Poptest Ouput

```
[kevintang7215]@linux2 ~/sim/misc> (18:35:32 05/02/23)
:: ./yis poptest.yo
Stopped in 5 steps at PC = 0x18.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%rax:   0x0000000000000000      0x000000000000abcd
%rsp:   0x0000000000000000      0x000000000000abcd

Changes to memory:
0x00f8: 0x0000000000000000      0x000000000000abcd

[kevintang7215]@linux2 ~/sim/misc> (19:08:36 05/02/23)
::
```

Our Output - Note: our %rsp contains the value 0x100 since it prioritizes writing valM over valE when writing to the same register.



0b0	0000000000000000
0b8	0000000000000000
0c0	0000000000000000
0c8	0000000000000000
0d0	0000000000000000
0d8	0000000000000000
0e0	0000000000000000
0e8	0000000000000000
0f0	0000000000000000
0f8	000000000000abcd
100	0000000000000000
108	0000000000000000
110	0000000000000000

Lab6 object file - the matrix multiplication program written by Kevin Tang

```
                               | # Execution begins address 0
0x000:                         |     .pos 0
0x000: 30f4ff0f000000000000    |     irmovq stack, %rsp # Set up stack
0x00a: 801400000000000000      |     call main      # Execute main
0x013: 00                      |     halt           # Terminate
                               |
0x014:                         | main:
                               |     #-----------------------Matrix A
0x014: 30f70100000000000000    |     irmovq $1, %rdi # Set rdi to 1 = A[0][0]
0x01e: 30f80200000000000000    |     irmovq $2, %r8  # Set r8 to 2 = A[0][1]
0x028: 30f90300000000000000    |     irmovq $3, %r9  # Set r9 to 3 = A[1][0]
0x032: 30fa0400000000000000    |     irmovq $4, %r10 # Set r10 to 4 = A[1][1]
                               |
                               |     #-----------------------Matrix B
0x03c: 30fb0500000000000000    |     irmovq $5, %r11 # Set r11 to 5 = B[0][0]
0x046: 30fc0600000000000000    |     irmovq $6, %r12 # Set r12 to 6 = B[0][1]
0x050: 30fd0700000000000000    |     irmovq $7, %r13 # Set r13 to 7 = B[1][0]
0x05a: 30fe0800000000000000    |     irmovq $8, %r14 # Set r14 to 8 = B[1][1]
                               |
                               |     # Result matrix C is in memory 0x00 to 0x18
                               |     # rax = a hold
                               |     # rbx = b hold
                               |     # rcx = desination
                               |     # rdx = destination holder
                               |     # rsi is iterator
                               |
                               |     #-----------------------Compute C[0][0]
0x064: 30f60100000000000000    |     irmovq $1, %rsi
                               |
0x06e: 2070                    |     rrmovq %rdi, %rax
0x070: 20b3                    |     rrmovq %r11, %rbx
0x072: 2031                    |     rrmovq %rbx, %rcx
0x074: 806501000000000000      |     call multiply
0x07d: 2032                    |     rrmovq %rbx, %rdx
                               |
0x07f: 2080                    |     rrmovq %r8, %rax
0x081: 20d3                    |     rrmovq %r13, %rbx
0x083: 2031                    |     rrmovq %rbx, %rcx
0x085: 806501000000000000      |     call multiply
0x08e: 6032                    |     addq %rbx, %rdx
                               |
0x090: 30f10000000000000000    |     irmovq $0, %rcx
0x09a: 40210000000000000000    |     rmmovq %rdx, (%rcx)
                               |
                               |     #-----------------------Compute C[0][1]
0x0a4: 30f60100000000000000    |     irmovq $1, %rsi
                               |
0x0ae: 2070                    |     rrmovq %rdi, %rax
0x0b0: 20c3                    |     rrmovq %r12, %rbx
0x0b2: 2031                    |     rrmovq %rbx, %rcx
0x0b4: 806501000000000000      |     call multiply
```

```
0x0bd: 2032                 |     rrmovq %rbx, %rdx
                            |
0x0bf: 2080                 |     rrmovq %r8, %rax
0x0c1: 20e3                 |     rrmovq %r14, %rbx
0x0c3: 2031                 |     rrmovq %rbx, %rcx
0x0c5: 806501000000000000   |     call multiply
0x0ce: 6032                 |     addq %rbx, %rdx
                            |
0x0d0: 30f10800000000000000 |     irmovq $8, %rcx
0x0da: 40210000000000000000 |     rmmovq %rdx, (%rcx)
                            |
                            |     #----------------------Compute C[1][0]
0x0e4: 30f60100000000000000 |     irmovq $1, %rsi
                            |
0x0ee: 2090                 |     rrmovq %r9, %rax
0x0f0: 20b3                 |     rrmovq %r11, %rbx
0x0f2: 2031                 |     rrmovq %rbx, %rcx
0x0f4: 806501000000000000   |     call multiply
0x0fd: 2032                 |     rrmovq %rbx, %rdx
                            |
0x0ff: 20a0                 |     rrmovq %r10, %rax
0x101: 20d3                 |     rrmovq %r13, %rbx
0x103: 2031                 |     rrmovq %rbx, %rcx
0x105: 806501000000000000   |     call multiply
0x10e: 6032                 |     addq %rbx, %rdx
                            |
0x110: 30f11000000000000000 |     irmovq $16, %rcx
0x11a: 40210000000000000000 |     rmmovq %rdx, (%rcx)
                            |
                            |     #----------------------Compute C[1][1]
0x124: 30f60100000000000000 |     irmovq $1, %rsi
                            |
0x12e: 2090                 |     rrmovq %r9, %rax
0x130: 20c3                 |     rrmovq %r12, %rbx
0x132: 2031                 |     rrmovq %rbx, %rcx
0x134: 806501000000000000   |     call multiply
0x13d: 2032                 |     rrmovq %rbx, %rdx
                            |
0x13f: 20a0                 |     rrmovq %r10, %rax
0x141: 20e3                 |     rrmovq %r14, %rbx
0x143: 2031                 |     rrmovq %rbx, %rcx
0x145: 806501000000000000   |     call multiply
0x14e: 6032                 |     addq %rbx, %rdx
                            |
0x150: 30f11800000000000000 |     irmovq $24, %rcx
0x15a: 40210000000000000000 |     rmmovq %rdx, (%rcx)
                            |
0x164: 90                   |     ret
                            |
                            | #----------------------Multiplication with repeated addition
0x165:                      | multiply:
0x165: 6160                 |     subq %rsi, %rax
0x167: 707001000000000000   |     jmp test
0x170:                      | test:
0x170: 747a01000000000000   |     jne loop
0x179: 90                   |     ret
0x17a:                      | loop:
0x17a: 6013                 |     addq %rcx, %rbx
0x17c: 706501000000000000   |     jmp multiply
                            |
                            | # Stack starts here and grows to lower addresses
0xfff:                      |   .pos 0xfff
0xfff:                      | stack:
```

Lab6 output

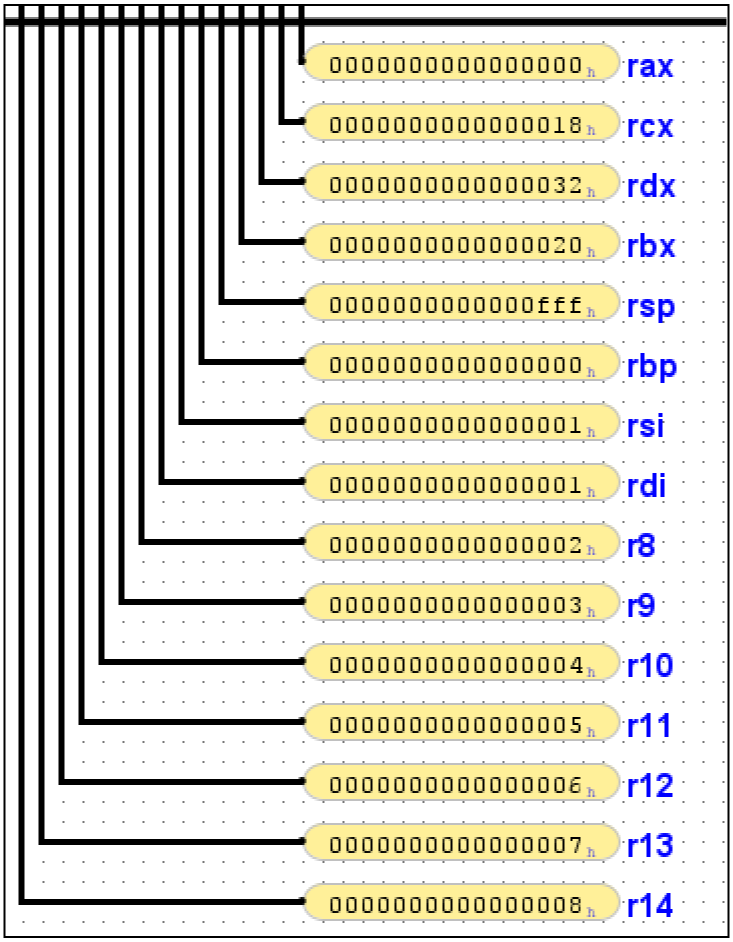
```
[kevintang7215]@linux2 ~/sim/misc> (18:28:39 05/02/23)
:: ./yas lab6.ys

[kevintang7215]@linux2 ~/sim/misc> (18:34:19 05/02/23)
:: ./yis lab6.yo
Stopped in 156 steps at PC = 0x13.  Status 'HLT', CC Z=0 S=0 O=0
Changes to registers:
%rcx:   0x0000000000000000      0x0000000000000018
%rdx:   0x0000000000000000      0x0000000000000032
%rbx:   0x0000000000000000      0x0000000000000020
%rsp:   0x0000000000000000      0x0000000000000fff
%rsi:   0x0000000000000000      0x0000000000000001
%rdi:   0x0000000000000000      0x0000000000000001
%r8:    0x0000000000000000      0x0000000000000002
%r9:    0x0000000000000000      0x0000000000000003
%r10:   0x0000000000000000      0x0000000000000004
%r11:   0x0000000000000000      0x0000000000000005
%r12:   0x0000000000000000      0x0000000000000006
%r13:   0x0000000000000000      0x0000000000000007
%r14:   0x0000000000000000      0x0000000000000008

Changes to memory:
0x0000: 0x000000000ffff430      0x0000000000000013
0x0008: 0x0000000014800000      0x0000000000000016
0x0010: 0x0001f73000000000      0x000000000000002b
0x0018: 0xf830000000000000      0x0000000000000032
0x0fe8: 0x0000000000000000      0x4e00000000000000
0x0ff0: 0x0000000000000000      0x1300000000000001

[kevintang7215]@linux2 ~/sim/misc> (18:34:20 05/02/23)
::
```

Our output - Note: the resulting Matrix is stored in memory
C[0][0] at 0x0
C[0][1] at 0x8
C[1][0] at 0x10
C[1][1] at 0x18



000	0000000000000013
008	0000000000000016
010	000000000000002b
018	0000000000000032
020	0000000000000000
028	0000000000000000
030	0000000000000000
038	0000000000000000
040	0000000000000000
048	0000000000000000
050	0000000000000000
058	0000000000000000
060	0000000000000000
068	0000000000000000
070	0000000000000000