Sankalp Kothari (IMT2021028)

# Nand2Tetris project report

## Github repo link:

https://github.com/SankalpKothari0904/Nand2Tetris

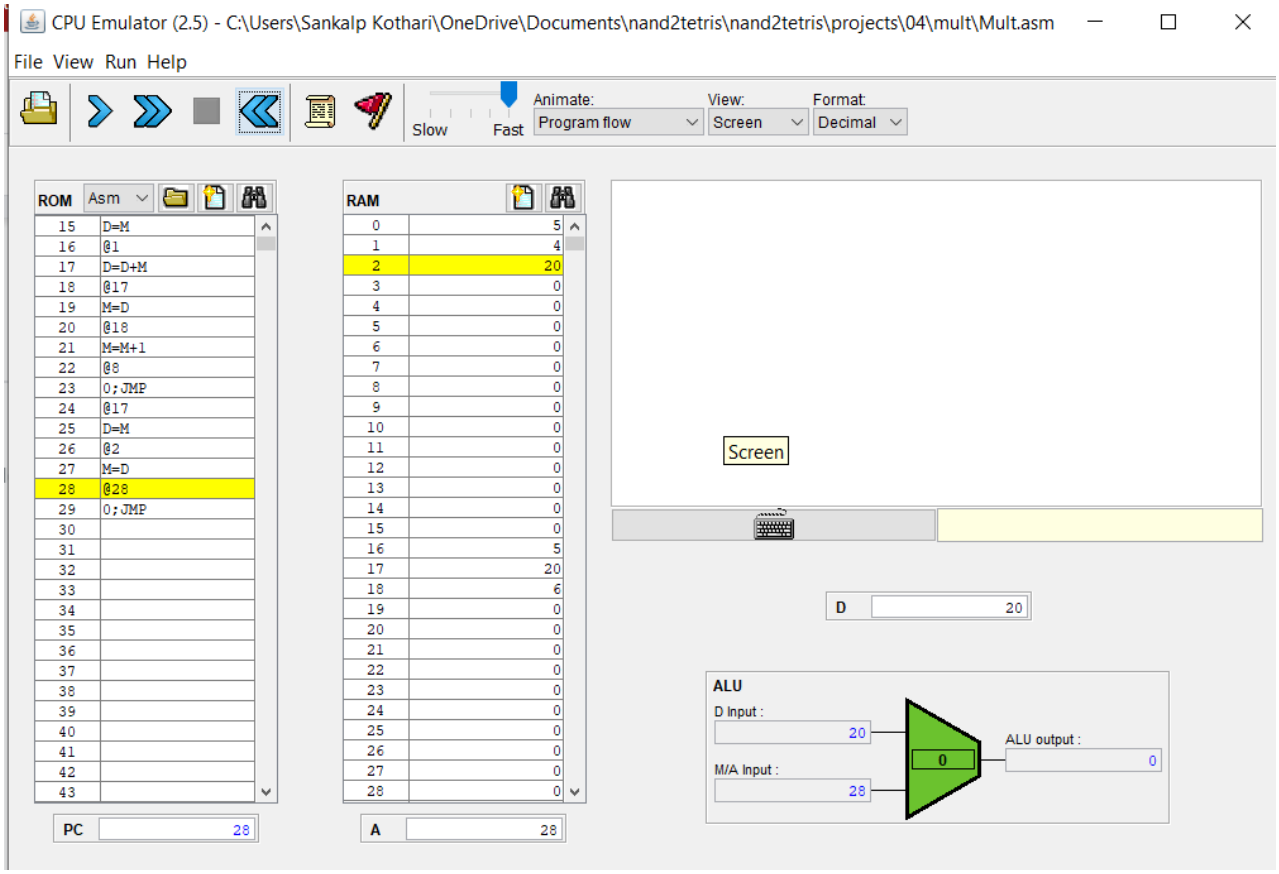
## Project 4:

### *Mult.asm :*

Multiply the values of R1 and R0 and store the result in R2.
We create a loop variable i, that we will use to add R1 to itself R0 number of times, and when i=R0, we terminate the loop.

04 > mult > Mult.asm

```
13
14  @R0
15  D=M
16  @a
17  M=D
18  @prod
19  M=0
20  @i
21  M=1
22  (LOOP)
23      @i
24      D=M
25      @a
26      D=D-M
27      @STOP
28      D;JGT
29      @prod
30      D=M
31      @R1
32      D=D+M
33      @prod
34      M=D
35      @i
36      M=M+1
37      @LOOP
38      0;JMP
39  (STOP)
40      @prod
41      D=M
42      @R2
43      M=D
44  (END)
45      @END
46      0;JMP
```




*Fill.asm :*

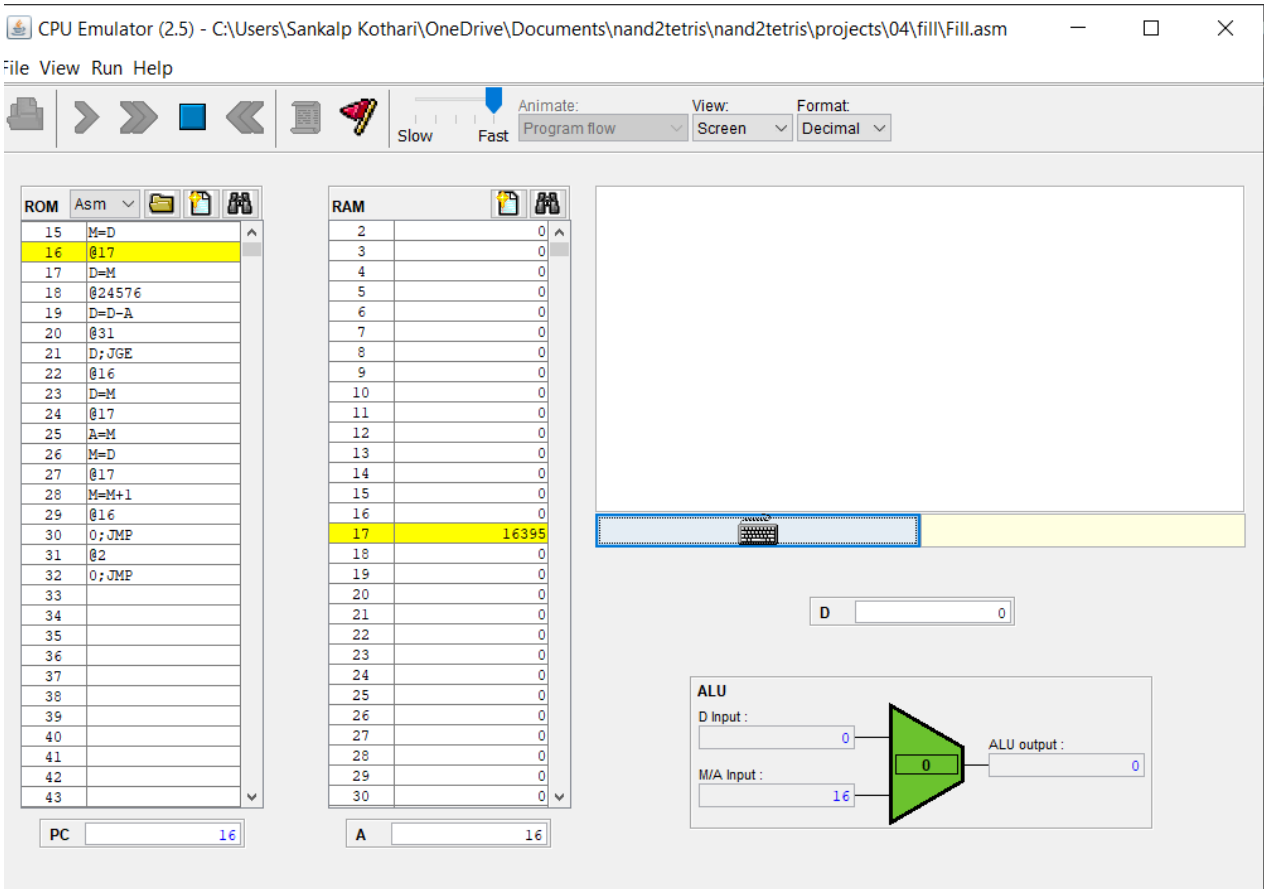
The screen should become black whenever a key is pressed, else white.
We run an infinite outer loop, loop1 that changes all the screen bits to 1 when any key is pressed on the keyboard, i.e., the keyboard value is non-zero.

A variable BorW decides whether the screen color needs to be changed or not. A second loop, loop2 changes all the screen bits to 1 (black) if the value of the variable BorW is -1.

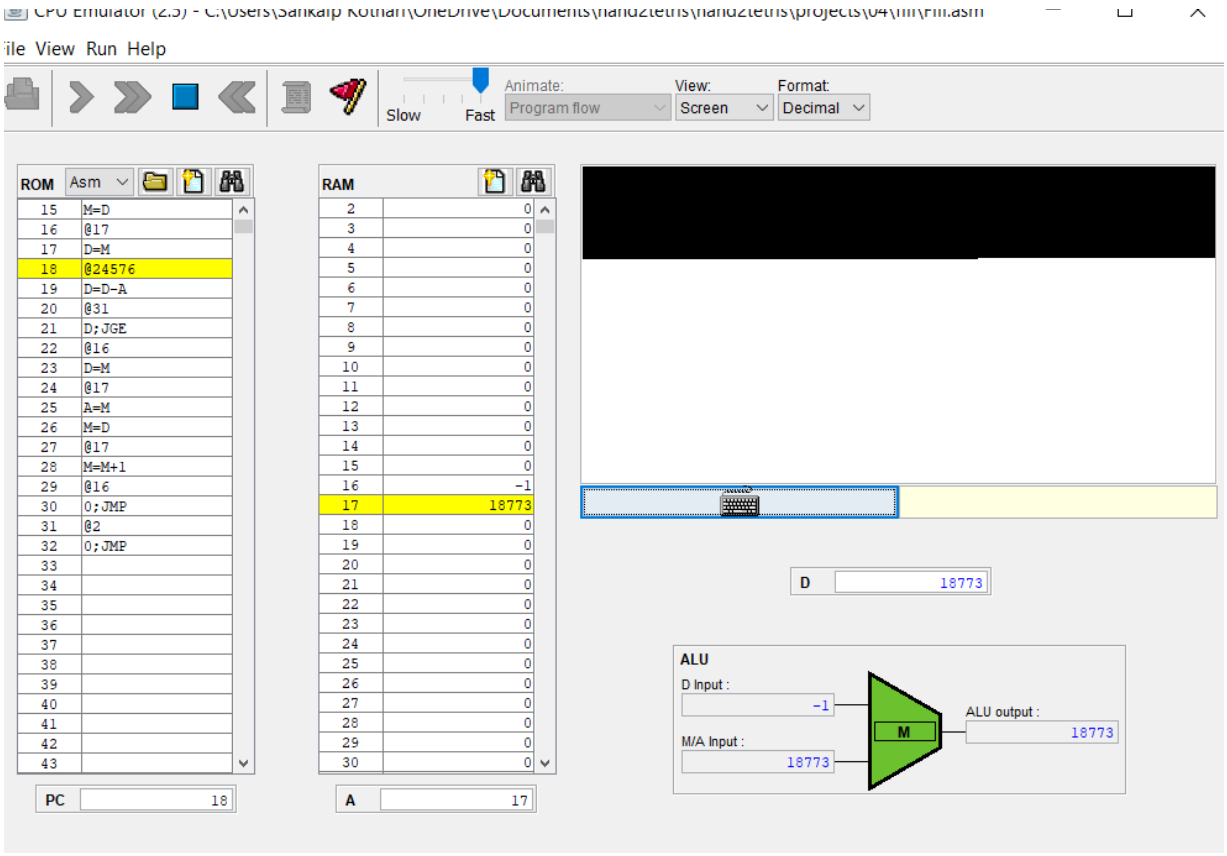
Code:

```
4 > fill > Fill.asm
16    @BorW
17    M=0     // variable black or white, 0 indicates white, -1 black
18    (LOOP1)
19        @KBD
20        D=M
21        @black
22        D;JNE
23        @BorW
24        M=0
25        @FILL_SCREEN
26        0;JMP
27        (black)
28            @BorW
29            M=-1
30        (FILL_SCREEN)
31            @SCREEN
32            D=A
33            @address
34            M=D
35            (LOOP2)
36                @address
37                D=M
38                @24576
39                D=D-A
40                @END2
41                D;JGE
42                @BorW
43                D=M
44                @address
45                A=M
46                M=D
47                @address
48                M=M+1
49                @LOOP2
50                0;JMP
51    (END2)
52        @LOOP1
53        0;JMP
```

No key pressed:




Key pressed: (halfway through the screen blackening process)

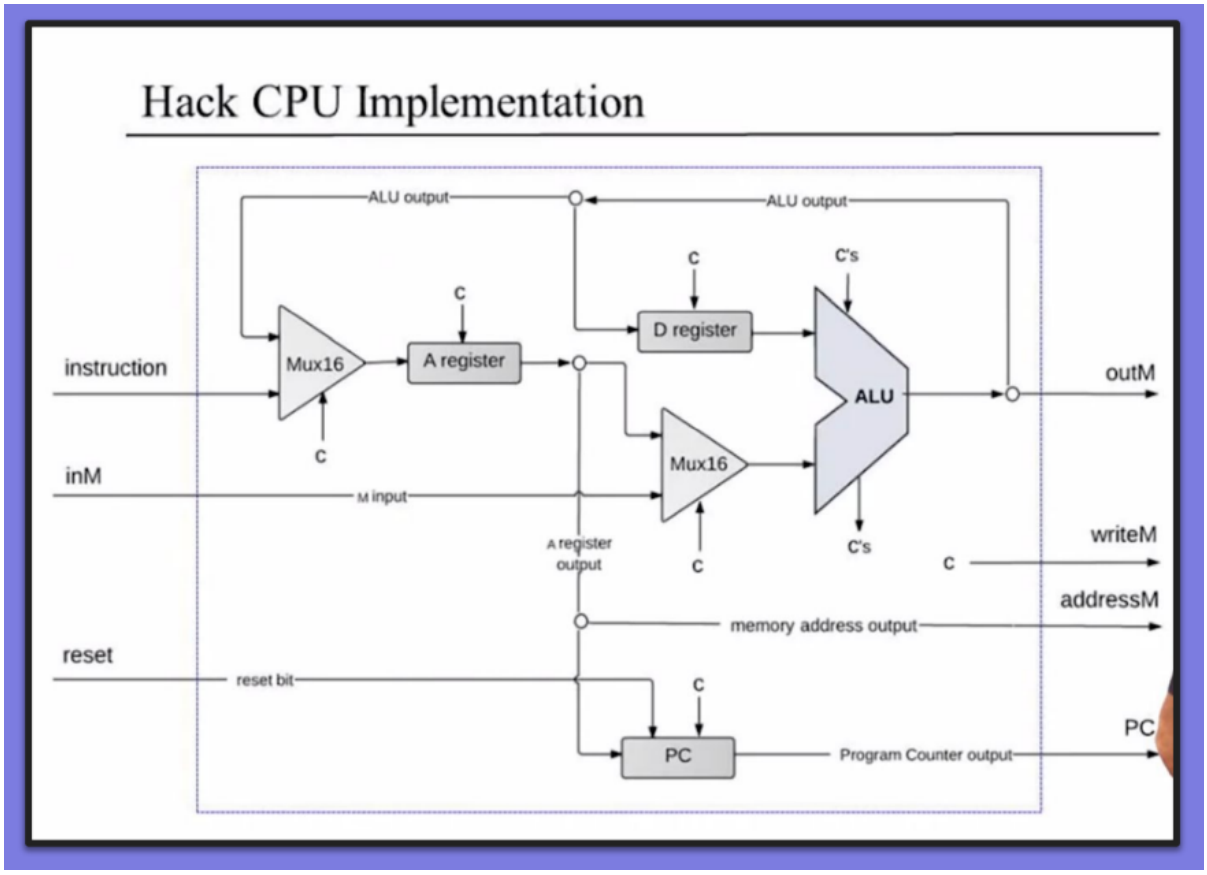



## Project 5:

In this project, we were supposed to make an entire Hack computer that would run Hack Machine Language Instructions by using inbuilt 32K ROM chip, a CPU and a Memory consisting of 16K RAM to store data, 8K built-in Screen and a 16 bit Keyboard register (built-in).

*CPU :*

Based on the following diagram:




Control signals:

- muxA: Controls the input to register A, acts as selector for mux before A in the diagram.
- writeA: determines whether or not anything has to be written to the A register or not.
- writeD: determines whether something needs to be written to D register or not.
- writeM: input sent to Memory.
- ALU control bits obtained directly from the instruction.
- The MSB of the instruction is the most important one as it determines the type of instruction.
- The PC is determined by the load control signal.

3 helper chips have been created in order to generate the aforementioned control signals.

1. Jump Helper:

Generates the load control signal that tells the next value of PC, i.e., the next instruction to be executed. It takes 5 inputs, 3 jump bits in a C-type instruction and zr and ng (2 ALU outputs). If load=0 or instruction is A type, we do not jump. Hence the value of load and instruction's MSB i.e. the 15th bit are taken in AND gate to determine whether or not to jump.

Truth table of load:

jump: 000 => no jump => load = 0 always

001 => jgt => load = 1 when zr and ng both are 0

010 => jeq => load = 1 when zr=1

011 => jge => load = 1 when ng=0

100 => jlt => load = 1 when ng=1

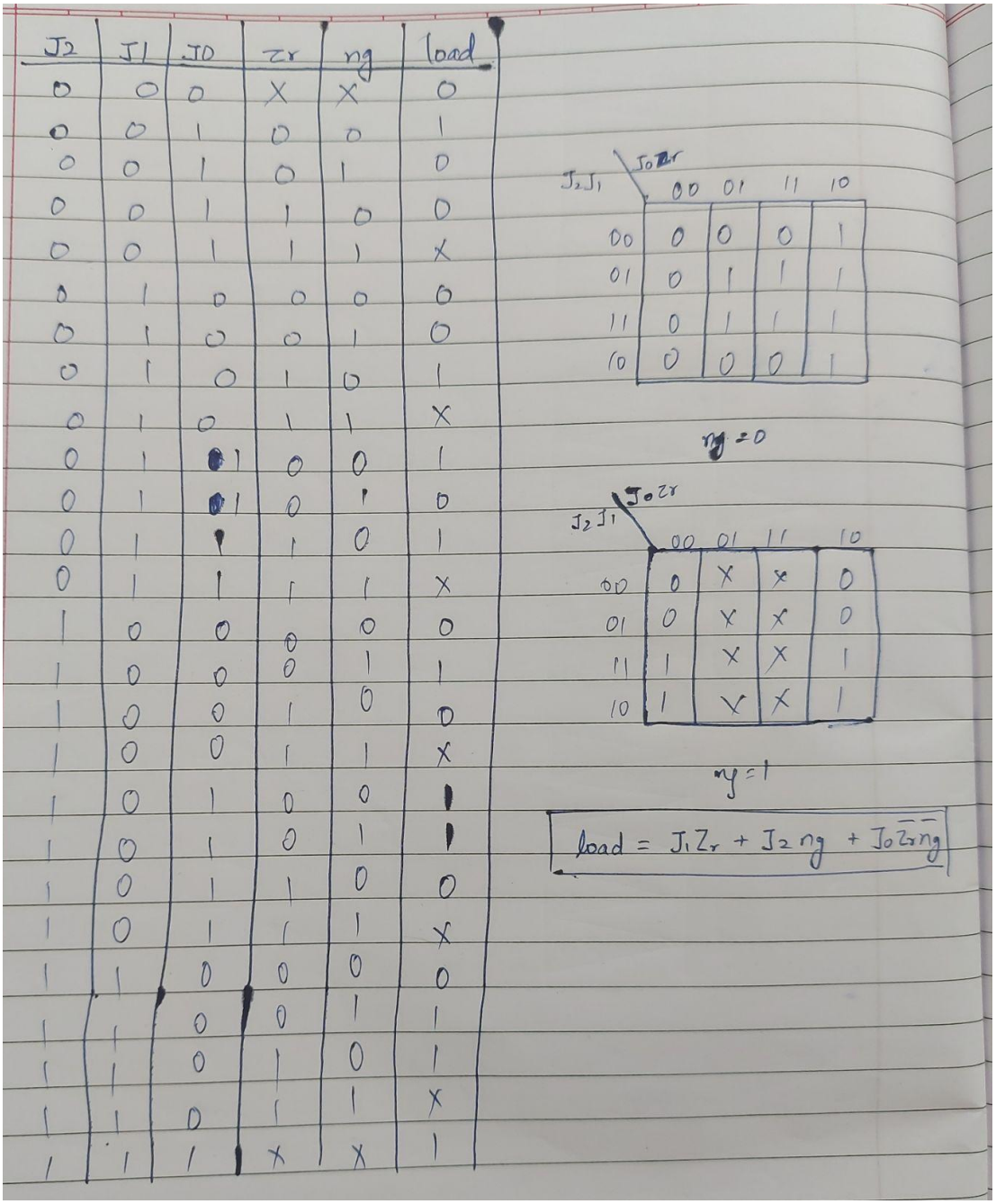
101 => jne => load = 1 when zr=0

110 => jle => load = 1 when ng=1 or zr=1

111 => jmp => load = 1 always

K-Map and corresponding helper chip logic:

```
14  CHIP Jump{
15      IN jump[3], zr, ng;
16      OUT load;
17
18      PARTS:
19      And(a=jump[1],b=zr,out=w1);
20      And(a=jump[2],b=ng,out=w2);
21
22      Not(in=zr,out=zrBar);
23      Not(in=ng,out=ngBar);
24
25      And(a=ngBar,b=zrBar,out=w3);
26      And(a=w3,b=jump[0],out=w4);
27
28      Or(a=w4,b=w2,out=w5);
29      Or(a=w5,b=w1,out=load);
30  }
```



| J2 | J1 | J0 | zr | ng | load |
|---|---|---|---|---|---|
| 0 | 0 | 0 | X | X | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | X |
| 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | X |
| 0 | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 | X |
| 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | X |
| 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | X |
| 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | X |
| 1 | 1 | 1 | X | X | 1 |

$J_2J_1$ \ $J_0zr$

| | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 0 | 1 |
| 01 | 0 | 1 | 1 | 1 |
| 11 | 0 | 1 | 1 | 1 |
| 10 | 0 | 0 | 0 | 1 |

ng = 0

$J_2J_1$ \ $J_0zr$

| | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | X | X | 0 |
| 01 | 0 | X | X | 0 |
| 11 | 1 | X | X | 1 |
| 10 | 1 | X | X | 1 |

ng = 1

$load = J_1Z_r + J_2 ng + J_0\bar{Z}_r\bar{ng}$

2. A-helper:

   This helper chip takes 2 inputs: instruction and destination and returns the 2 control signals muxA and writeA

instruction = 0, destination=0, muxA=1 , writeA=1
instruction = 0, destination=1, muxA=1 , writeA=1
instruction = 1, destination=0, muxA=X , writeA=0
instruction = 1, destination=1, muxA=0 , writeA=1

```
CHIP AHelper{
    IN instruction,destination;
    OUT muxA,writeA;

    PARTS:
    Not(in=instruction,out=muxA);
    Not(in=instruction,out=instructionBar);
    Or(a=instructionBar,b=destination,out=writeA);
}
```

3. writeHelper:

Again takes 2 inputs and returns single output writeD which tells whether or not the value has to be written to D register.
Can also be used for register M, only the destination bits change.

```
CHIP WriteHelper{
    IN instruction,DHelper;
    OUT writeD;
    PARTS:
    And(a=instruction,b=destination,out=writeD);
}

// this can be used for both D and M registers,
// only destination bits are different,
// rest functionality is same
```

Using all these helper chips, CPU logic is as follows:

```
CHIP CPU {

    IN  inM[16],         // M value input  (M = contents of RAM[A])
        instruction[16], // Instruction for execution
        reset;           // Signals whether to re-start the current
                         // program (reset==1) or continue executing
                         // the current program (reset==0).

    OUT outM[16],        // M value output
        writeM,          // Write to M?
        addressM[15],    // Address in data memory (of M)
        pc[15];          // address of next instruction

    PARTS:

    AHelper(instruction=instruction[15], destination=instruction[5], muxA=muxA, writeA=writeA);
    Mux16(a=ALUOut, b=instruction, sel=muxA, out=dataA, out[0..14]=addressM);
    Register(in=dataA, load=writeA, out=outA);

    WriteHelper(instruction=instruction[15], destination=instruction[4], writeD=writeD);
    Register(in=ALUOut, load=writeD, out=outD);

    Mux16(a=outA, b=inM, sel=instruction[12], out=outAM);
```

```
    ALU(
        x=outD,
        y=outAM,
        zx=instruction[11],
        nx=instruction[10],
        zy=instruction[9],
        ny=instruction[8],
        f=instruction[7],
        no=instruction[6],
        out=outM,
        out=ALUOut,
        zr=zero,
        ng=neg
    );

    WriteHelper(instruction=instruction[15], destination=instruction[3], out=writeM);

    Jump(jump=instruction[0..2], zr=zero, ng=neg, load=load1);
    And(a=load1, b=instruction[15], out=pcload);

    Not(in=pcload, out=notpcload);
    PC(in=outA, load=pcload, inc=notpcload, reset=reset, out[0..14]=pc);
}
```

## *Memory :*

In the Memory, the 15 bit address is passed as input, along with control signal load, and 16 bit value to be written. One value is is obtained which is value at that memory location

We use a demultiplexer here to determine which part of the memory we are accessing. If the first 2 address bits are 00 or 01 (i.e. address<=16383) , we are accessing some location inside RAM. If the first 2 bits are 10 (i.e. address >=16384 and <24576) we are accessing the screen bits. Else, if both bits are 11, (i.e. address = 24576) we are accessing the keyboard.

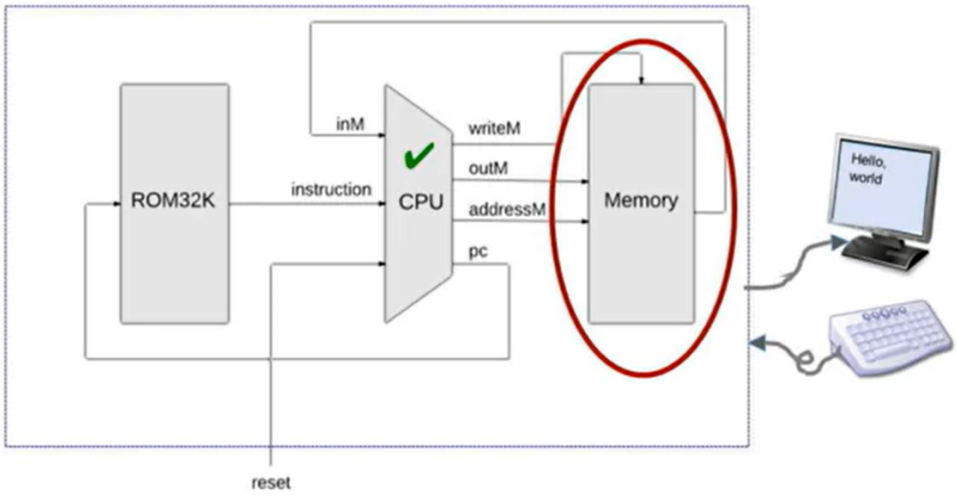
We perform demultiplexing and multiplexing by the same address bits.

code:

```
CHIP Memory {
    IN in[16], load, address[15];
    OUT out[16];

    PARTS:
    // Put your code here:

    DMux4Way(in=load, sel=address[13..14], a=RAM1, b=RAM2, c=scr, d=kbd);
    Or(a=RAM1,b=RAM2,out=ram);

    RAM16K(in=in, load=ram, address=address[0..13], out=RAMout);
    Screen(in=in, load=scr, address=address[0..12],out=SCRout);
    Keyboard(out=KBDout);

    Mux4Way16(a=RAMout,b=RAMout,c=SCRout,d=KBDout);
}
```

## *Computer :*

The computer is simply a connection of the ROM32K, CPU and Memory chips.




```
CHIP Computer {

    IN reset;

    PARTS:
    // Put your code here:

    ROM32K(address=PC, out=instruction);
    CPU(
        inM=data,
        instruction=instruction,
        reset=reset,
        out=outM,
        writeM=writeM,
        addressM=addressM,
        pc=PC
    );

    Memory(in=outM, load=writeM, address=addressM, out=data);
}
```

## Project 6:

So far, we wrote instructions in hack assembly and made a computer that would run the binary equivalents of these instructions. Now we make the assembler, that does the job of translating these hack assembly language instructions to binary.

It removes all the comments, white spaces, labels etc and translates the variables, labels etc to their corresponding memory locations.

How to run:
Assembler has been written in python. Terminal command: python3 assembler.py

It takes 1 input: name of file to be assembled. Eg. program.asm etc.

**Warning:** The file must be in the same location as the assembler.py file

All the instructions are translated and written in their binary representation in a file called program.hack

Removing the white spaces:
We first remove the blank lines, then run 2 loops to remove the lines that have only comments, or blank spaces (newline characters, tabs etc), and the subsequent empty strings that are created as a result.Then we have only the lines that have either only code or ones that have both code and comments remaining.

In the second for loop, we remove the lines of code that have comments written and then we remove any trailing whitespace.

Adding Labels:
We now traverse through the assembly code and look for label declarations, we then add them to the symbol table (the table that contains all the various predefined and user defined labels and symbols) and add them there with the corresponding addresses and remove them from the assembly instructions.

Conversion of instructions to binary:

Now we resolve the assembly instructions as A and C type, resolve the variable references and get the binary representation of the instructions.

A type:

- Starts with @
- If it is followed by a number, convert to an integer
- If it is a variable or symbol that is already in the table, resolve it directly.
- If it is not in the table, add it to the table and then resolve to integer.
- Convert this address to a 15 bit binary string and add a 0 before it (opcode of A type instruction)

C type:

- General format: dest = comp ; jump
- All 3 fields may be present in instruction, or maybe 2 out of three ( dest and comp or comp and jump)
- If all 3 are present, split at = and ; We get 3 items, 1st is destination, second is computation and third is jump.
- If there are 2 components, and there is no semicolon (dest and comp present, not jump) we split at = and get 2 items. First is dest, second is comp, and jump=000
- If there are 2 components, and there is no = sign (jump and comp present, not dest) we split at ; and get 2 items. First is comp, second is jump, and dest=000
- Values of other bits can be seen from this table:

Symbolic syntax: *dest* = *comp* ; *jump*

Binary syntax: 1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3

| *comp* | | c1 | c2 | c3 | c4 | c5 | c6 |
|---|---|---|---|---|---|---|---|
| 0 | | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | | 1 | 1 | 1 | 1 | 1 | 1 |
| -1 | | 1 | 1 | 1 | 0 | 1 | 0 |
| D | | 0 | 0 | 1 | 1 | 0 | 0 |
| A | M | 1 | 1 | 0 | 0 | 0 | 0 |
| !D | | 0 | 0 | 1 | 1 | 0 | 1 |
| !A | !M | 1 | 1 | 0 | 0 | 0 | 1 |
| -D | | 0 | 0 | 1 | 1 | 1 | 1 |
| -A | -M | 1 | 1 | 0 | 0 | 1 | 1 |
| D+1 | | 0 | 1 | 1 | 1 | 1 | 1 |
| A+1 | M+1 | 1 | 1 | 0 | 1 | 1 | 1 |
| D-1 | | 0 | 0 | 1 | 1 | 1 | 0 |
| A-1 | M-1 | 1 | 1 | 0 | 0 | 1 | 0 |
| D+A | D+M | 0 | 0 | 0 | 0 | 1 | 0 |
| D-A | D-M | 0 | 1 | 0 | 0 | 1 | 1 |
| A-D | M-D | 0 | 0 | 0 | 1 | 1 | 1 |
| D&A | D&M | 0 | 0 | 0 | 0 | 0 | 0 |
| D\|A | D\|M | 0 | 1 | 0 | 1 | 0 | 1 |
| a=0 | a=1 | | | | | | |

| *dest* | d1 | d2 | d3 | effect: the value is stored in: |
|---|---|---|---|---|
| null | 0 | 0 | 0 | The value is not stored |
| M | 0 | 0 | 1 | RAM[A] |
| D | 0 | 1 | 0 | D register |
| MD | 0 | 1 | 1 | RAM[A] and D register |
| A | 1 | 0 | 0 | A register |
| AM | 1 | 0 | 1 | A register and RAM[A] |
| AD | 1 | 1 | 0 | A register and D register |
| AMD | 1 | 1 | 1 | A register, RAM[A], and D register |

| *jump* | j1 | j2 | j3 | effect: |
|---|---|---|---|---|
| null | 0 | 0 | 0 | no jump |
| JGT | 0 | 0 | 1 | if out > 0 jump |
| JEQ | 0 | 1 | 0 | if out = 0 jump |
| JGE | 0 | 1 | 1 | if out ≥ 0 jump |
| JLT | 1 | 0 | 0 | if out < 0 jump |
| JNE | 1 | 0 | 1 | if out ≠ 0 jump |
| JLE | 1 | 1 | 0 | if out ≤ 0 jump |
| JMP | 1 | 1 | 1 | Unconditional jump |

We then store all these instructions that we have obtained in the binary format and then write them to the program.hack file.

Code of assembler:

```
import re

symbolTable={"R0":0,"R1":1,"R2":2,"R3":3,"R4":4,"R5":5,"R6":6,"R7":7,"R8":8,"R9":9,
"R10":10,"R11":11,"R12":12,"R13":13,"R14":14,"R15":15,"SP":0,"LCL":1,"ARG":2,"THIS":3,
"THAT":4,"SCREEN":16384,"KBD":"24576"}

dest_dict={"":"000","M":"001","D":"010","A":"100","MD":"011","AM":"101","AD":"110","AMD":"111"}

jump_dict={"":"000","JGT":"001","JEQ":"010","JGE":"011","JLT":"100","JNE":"101","JLE":"110","JMP":"111"}

comp_dict={"0":"0101010","1":"0111111","-1":"0111010","D":"0001100","A":"0110000","M":"1110000",
"!D":"0001101","!A":"0110001","!M":"1110001","-D":"0001111","-A":"0110011","-M":"1110011",
"D+1":"0011111","A+1":"0110111","M+1":"1110111","D-1":"0001110","A-1":"0110010",
"M-1":"1110010","D+A":"0000010","D+M":"1000010","D-A":"0010011","D-M":"1010011","A-D":"0000111",
"M-D":"1000111","D&A":"0000000","D&M":"1000000","D|A":"0010101","D|M":"1010101"}

file1=input()
with open(file1,'r') as file:
    code=[]
    for line in file:
        code.append(line.strip())
    removelist=[]
    for i in code:
        if (i[0:2]=="//" or i==""):
            removelist.append(i)
    for i in removelist:
        code.remove(i)
    code1=[]
    for i in code:
        temp=i.split("//")
        code1.append(temp[0].strip())
file.close()
code=code1
binaryInstructions=[]
```

```
def getBinary(n):
    s=""
    while (n!=0):
        if (n%2==0):
            s="0"+s
        else:
            s="1"+s
        n=n//2
    while (len(s)<15):
        s="0"+s
    return s

def Assembler():
    global code
    global symbolTable
    global binaryInstructions
    global jump_dict
    global dest_dict
    global comp_dict
    line_no=0
    remLabels=[]
    vars=16
    for i in code:
        if (i[0]=="("):
            symbolTable[i[1:len(i)-1]]=line_no+1
            remLabels.append(i)
        else:
            line_no+=1
    for i in remLabels:
        code.remove(i)
    for i in code:
```

```
    for i in code:
        if i[0]=="@":
            try:
                v=eval(i[1:len(i)])
            except ValueError:
                if i[1:] in symbolTable.keys():
                    v=symbolTable[i[1:]]
                else:
                    symbolTable[i[1:]]=vars
                    v=vars
                    vars+=1
            instr="0"+getBinary(v)
            binaryInstructions.append(instr)
        else:
            c=re.split("=|;",i)
            instr="111"
            if (len(c)==3):
                instr=instr+comp_dict(c[1])+dest_dict(c[0])+jump_dict(c[2])
            else:
                if len(i.split("="))==2:
                    instr=instr+comp_dict(c[1])+dest_dict(c[0])+"000"
                else:
                    instr=instr+comp_dict(c[0])+"000"+jump_dict(c[1])
            binaryInstructions.append(instr)

Assembler()

with open("program.hack",'w') as write:
    for i in binaryInstructions:
        write.write(i)
        write.write('\n')
```