

Nand to Tetris Project Report

By Barath S Narayan

Roll Number:IMT2021524

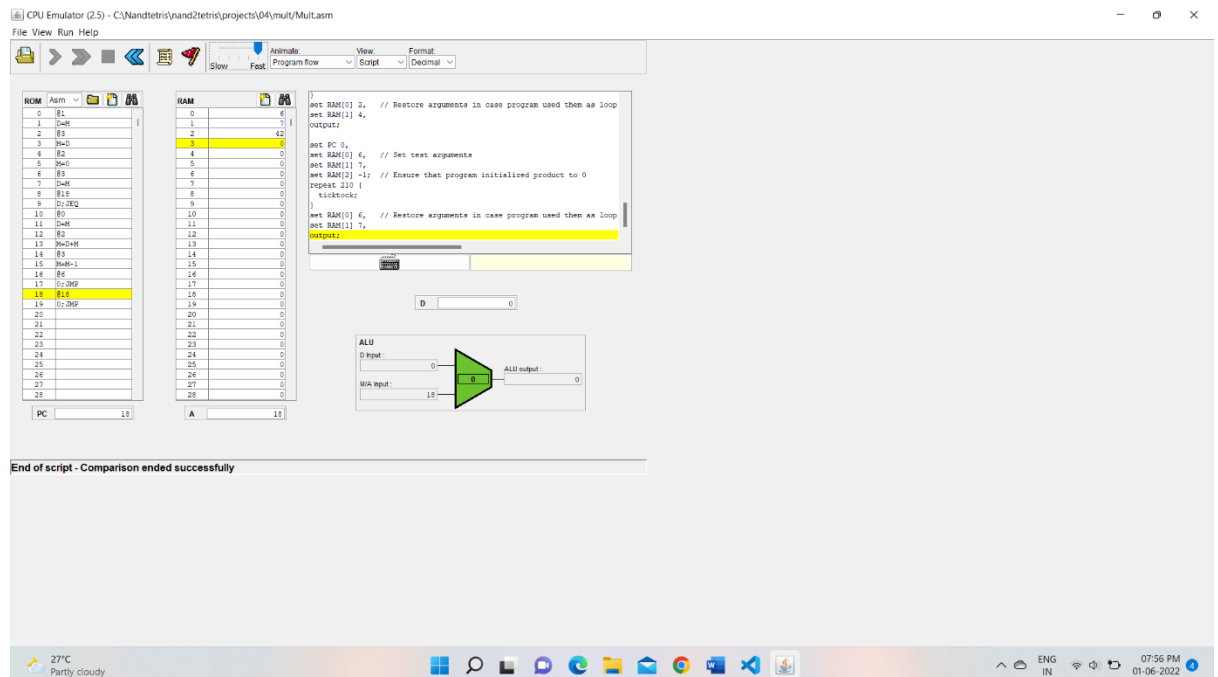
Project 4

Mult.asm

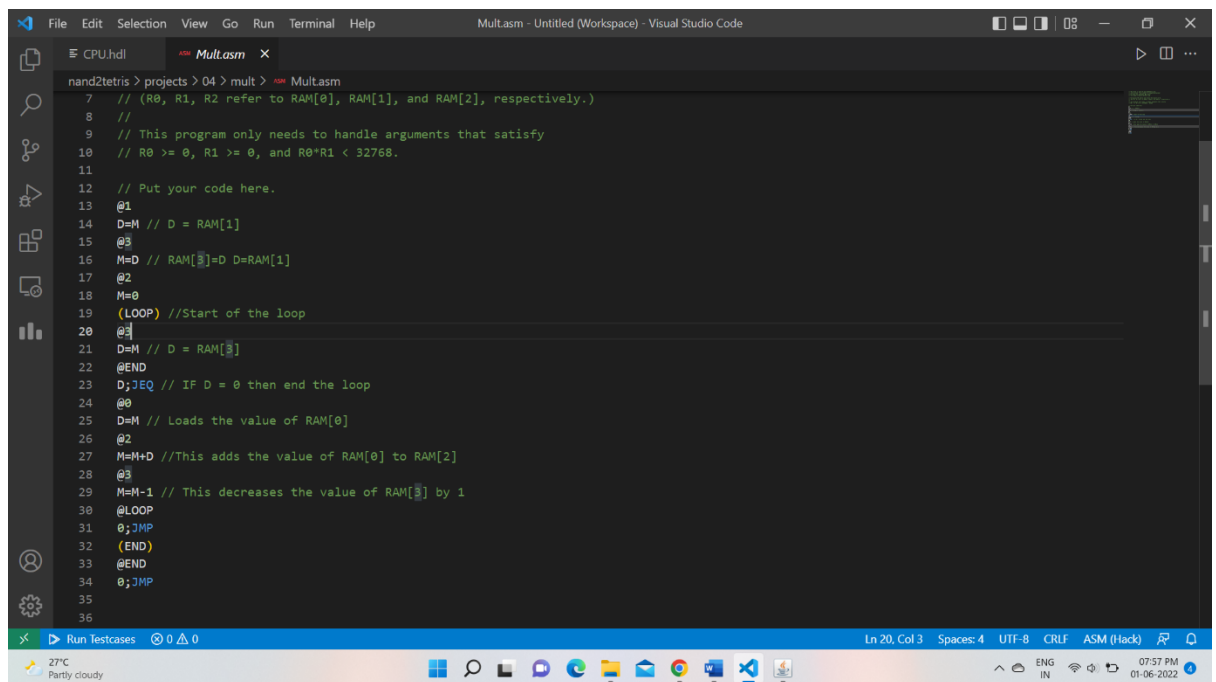
Since the hack computer does not have a multiply function we have to do loop addition.

To find $a * b$ we add 'a' b times using a loop . The value of a is written into memory location 0 while the value of b is written into memory location 1. Value at memory location 2(RAM[2]) is first set to 0. Then value at memory location 3 is set to value at memory location 1(RAM[1]). Then a loop is run where 'a' is added to memory location 2(RAM[2]) and value at memory location 3(RAM[3]) is decremented by 1. When the value at memory location 3 becomes 0 the program stops and memory location 2 contains $a * b$.

Screenshots:



Execution of the program



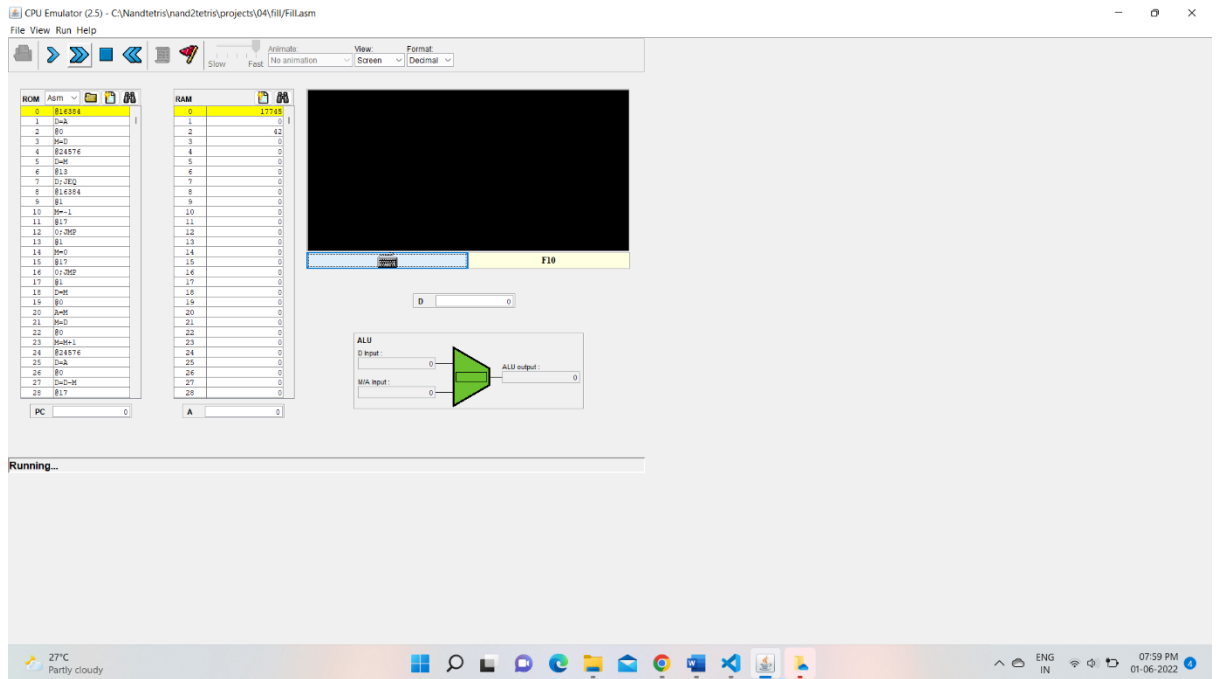
```
7 // (R0, R1, R2 refer to RAM[0], RAM[1], and RAM[2], respectively.)
8 //
9 // This program only needs to handle arguments that satisfy
10 // R0 >= 0, R1 >= 0, and R0*R1 < 32768.
11
12 // Put your code here.
13 @1
14 D=M // D = RAM[1]
15 @3
16 M=D // RAM[3]=D D=RAM[1]
17 @2
18 M=0
19 (LOOP) //Start of the loop
20 @3
21 D=M // D = RAM[3]
22 @END
23 D;JEQ // IF D = 0 then end the loop
24 @0
25 D=M // Loads the value of RAM[0]
26 @2
27 M=M+D //This adds the value of RAM[0] to RAM[2]
28 @3
29 M=M-1 // This decreases the value of RAM[3] by 1
30 @LOOP
31 @;JMP
32 (END)
33 @END
34 @;JMP
35
36
```

Screenshot of code mult.asm

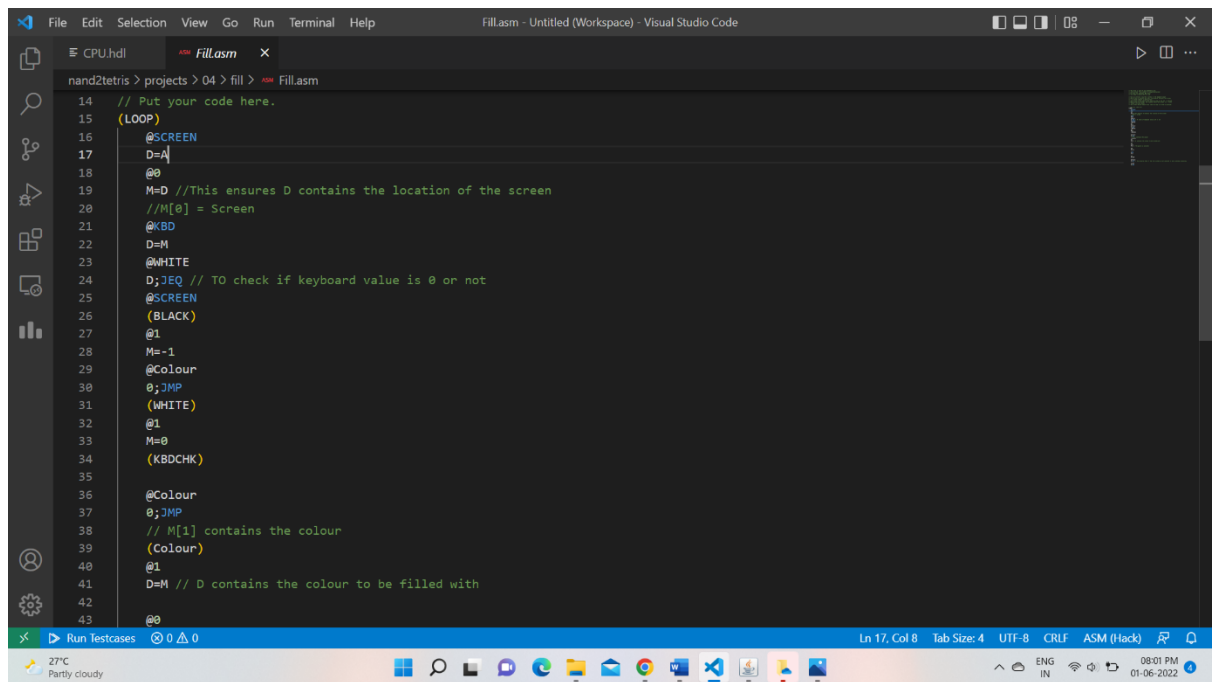
Fill.asm

This program makes the screen black if a key is pressed and white when it is not pressed. The program first stores the location of screen in RAM[0]. Then the program checks if the keyboard is pressed or not . If the keyboard is pressed it stores 1 at RAM[0] , otherwise it stores 0 at RAM[0] and runs a loop to colour the screen. Every time the loop is executed It makes the value at RAM[RAM[0]] 0 or -1 depending on the keyboard. The value at memory location 0 is then incremented and is checked if it becomes equal to the address of the keyboard . The loop stops at the point where they are equal and the program goes back to the start to see if a button is pressed or not.

Screenshots:



Execution of the program



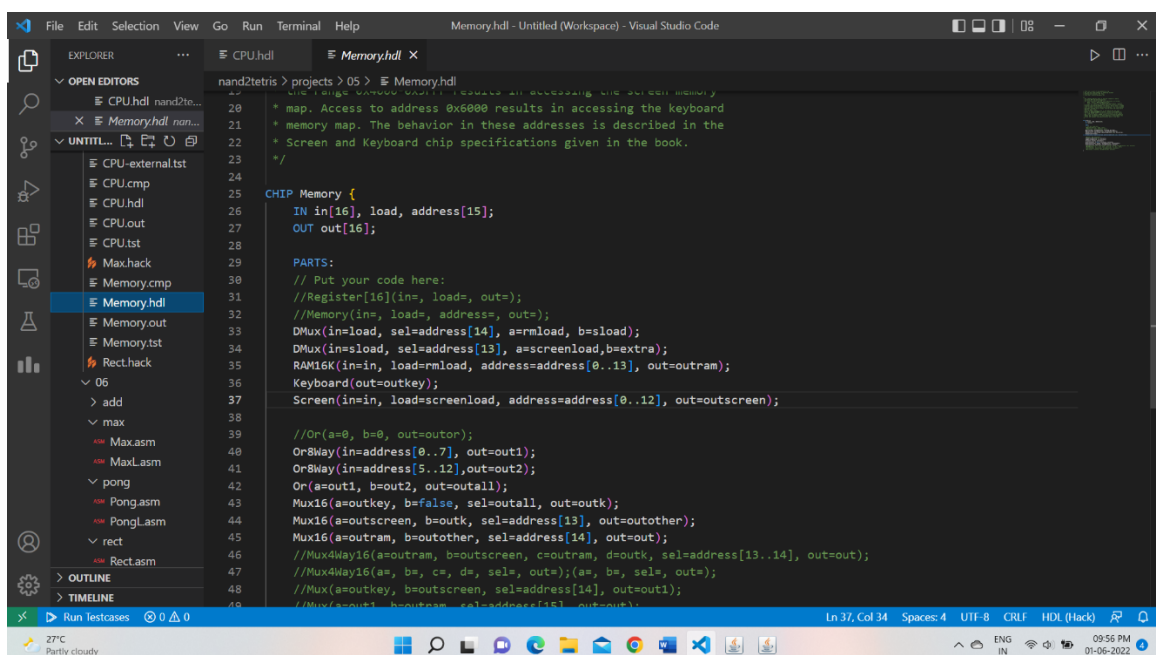
Screenshot of the program Fill.asm

Project 5

Memory.hdl

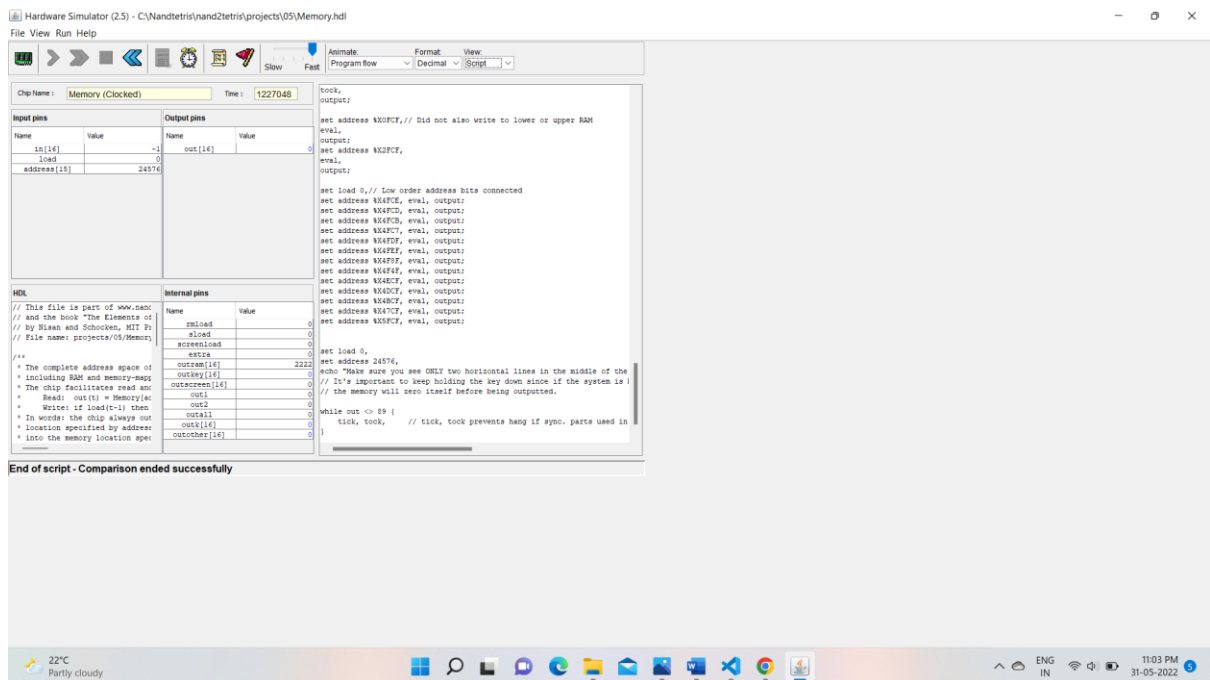
The file represents the data memory of the hack computer. It is created using the built-in Ram16K module ,built-in keyboard, built-in screen. We use two Muxes to choose the hardware component based on the input address. The load for the RAM module and the Screen module is selected using DMuxes for which load is the input and the outputs are ramload, screenload and extra. This ensures that only one of the components has load value equal to one. This way, only one of the component is connected to the output and the correct result is given.

Screenshots:



```
20 * map. Access to address 0x5000 results in accessing the keyboard
21 * memory map. The behavior in these addresses is described in the
22 * Screen and Keyboard chip specifications given in the book.
23 */
24
25 CHIP Memory {
26     IN in[16], load, address[15];
27     OUT out[16];
28
29     PARTS:
30         // Put your code here:
31         // Register[16](in, load, out);
32         // Memory(in, load, address, out);
33         DMux(in=load, sel=address[14], a=ramload, b=sload);
34         DMux(in=sload, sel=address[13], a=screenload, b=extra);
35         RAM16K(in=in, load=ramload, address=address[0..13], out=outram);
36         Keyboard(out=outkey);
37         Screen(in=in, load=screenload, address=address[0..12], out=outscreen);
38
39         //Or(a=0, b=0, out=outor);
40         Or8Way(in=address[0..7], out=out1);
41         Or8Way(in=address[5..12], out=out2);
42         Or(a=out1, b=out2, out=outall);
43         Mux16(a=outkey, b=false, sel=outall, out=outk);
44         Mux16(a=outscreen, b=outk, sel=address[13], out=outother);
45         Mux16(a=outram, b=outother, sel=address[14], out=out);
46         //Mux4Way16(a=outram, b=outscreen, c=outram, d=outk, sel=address[13..14], out=out);
47         //Mux4Way16(a=, b=, c=, d=, sel=, out=);(a=, b=, sel=, out=);
48         //Mux(a=outkey, b=outscreen, sel=address[14], out=out1);
49         //Mux(a=out1, b=outscreen, sel=address[15], out=out);
```

Screenshot of the program Memory.hdl



Screenshot of testing of Memory.hdl

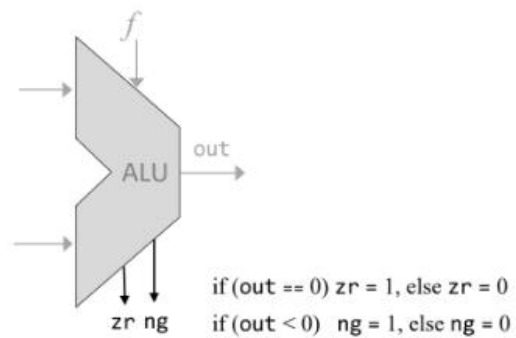
CPU.hdl

This file represents the CPU of the hack computer. It first sends the value of inM and output of the ALU to a mux whose select line is complement of the 15th bit of the instruction input. The output of this mux is connected to the A register. The load value of the A register is loada whose value is got by complement(instruction[15]) or instruction[15]&instruction[5]. The output of ARegister is connected to a second Mux. The second input for the mux is inM. The second mux takes select line as instruction[12]. This selects between the output of the A register and input Memory. The output of the Mux is sent to the ALU.

Similar to the first Mux which uses the aluout, the DRegister also takes aluout as input, load (which is calculated as instruction[15]&instruction[4]). The output of this also becomes an input for the ALU.

The ALU takes instruction[11..6] as control bits of the ALU (The control bits are zx, nx, zy, ny, f, no). There are three outputs which are received. They are out, zr, and ng. Zr is the output which denotes whether the output is zero or not. The output ng denotes whether out is negative. The PC module takes output of ARegister as input. The load for the pc function is got by the following conditions

jump	j1	j2	j3	condition
null	0	0	0	no jump
JGT	0	0	1	if (ALU out > 0) jump
JEQ	0	1	0	if (ALU out = 0) jump
JGE	0	1	1	if (ALU out ≥ 0) jump
JLT	1	0	0	if (ALU out < 0) jump
JNE	1	0	1	if (ALU out ≠ 0) jump
JLE	1	1	0	if (ALU out ≤ 0) jump
JMP	1	1	1	Unconditional jump



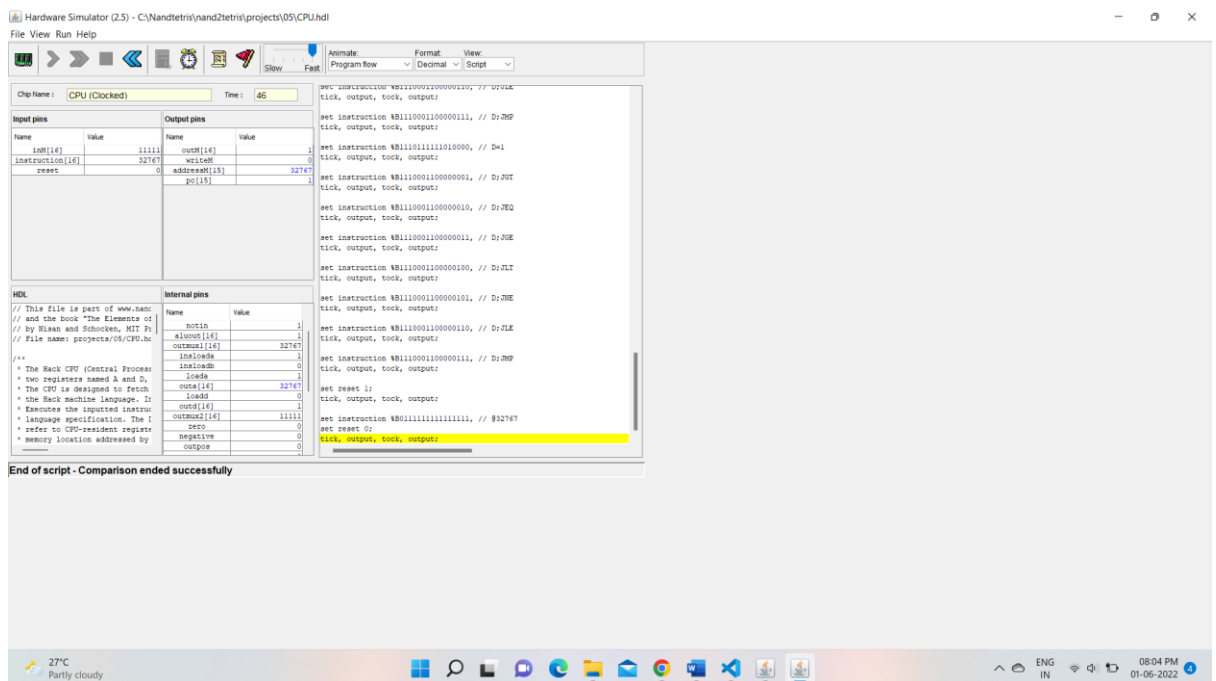
Jump decision:

$J(j1, j2, j3, zr, ng) = 1$ if *condition* is true, 0 otherwise

J can be computed using gate logic, And then help compute the address of the next instruction

The pc value of the next instruction is got using these conditions. These modules make up the CPU of the hack computer.

Screenshots:



Screenshot of testing of CPU.hdl

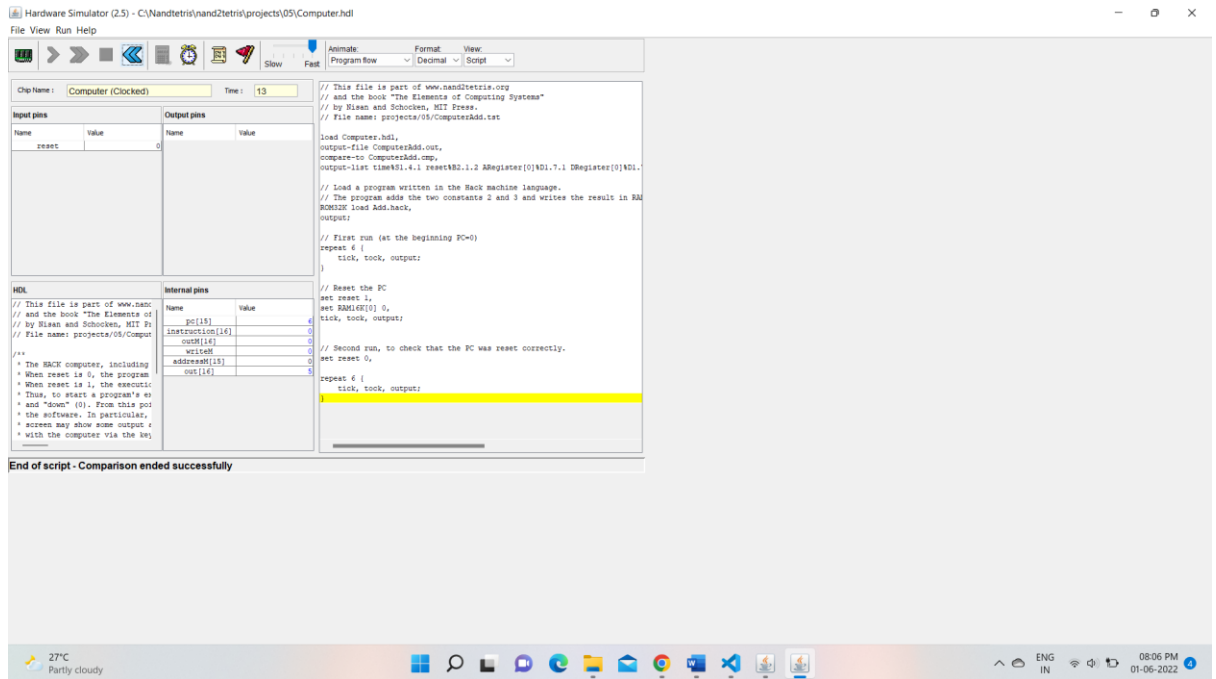
```
File Edit Selection View Go Run Terminal Help CPU.hdl - Untitled (Workspace) - Visual Studio Code
CPU.hdl x Fillasm
nand2tetris > projects > 05 > CPU.hdl
43 Not(in=instruction[15], out=notin);
44 Mux16(a=aluout, b=instruction, sel=notin, out=outmux1);
45
46 //A register load
47 Not(in=instruction[15], out=insloada);
48 And(a=instruction[15], b=instruction[5], out=insloadb);
49 Or(a=insloada, b=insloadb, out=loada);
50
51 //A register
52 ARegister(in=outmux1, load=loada, out=outa, out[0..14]=addressM);
53
54 //D register load
55 And(a=instruction[15], b=instruction[4], out=loadd);
56
57 //D Register
58 DRegister(in=aluout, load=loadd, out=outd);
59
60 Mux16(a=outa, b=inM, sel=instruction[12], out=outmux2);
61
62 ALU(x=outd, y=outmux2, zx=instruction[11], nx=instruction[10], zy=instruction[9], ny=instruction[8], f=instruction[7], no=instruction[6]
63 And(a=instruction[15], b=instruction[3], out=writeM);
64
65 Or(a=zero, b=negative, out=outpos);
66 Not(in=outpos, out=pos);
67
68 And(a=instruction[15], b=instruction[0], out=j3);
69 And(a=instruction[15], b=instruction[1], out=j2);
70 And(a=instruction[15], b=instruction[2], out=j1);
71
72 And(a=j3, b=pos, out=out1);
```

Screenshot of the program CPU.hdl

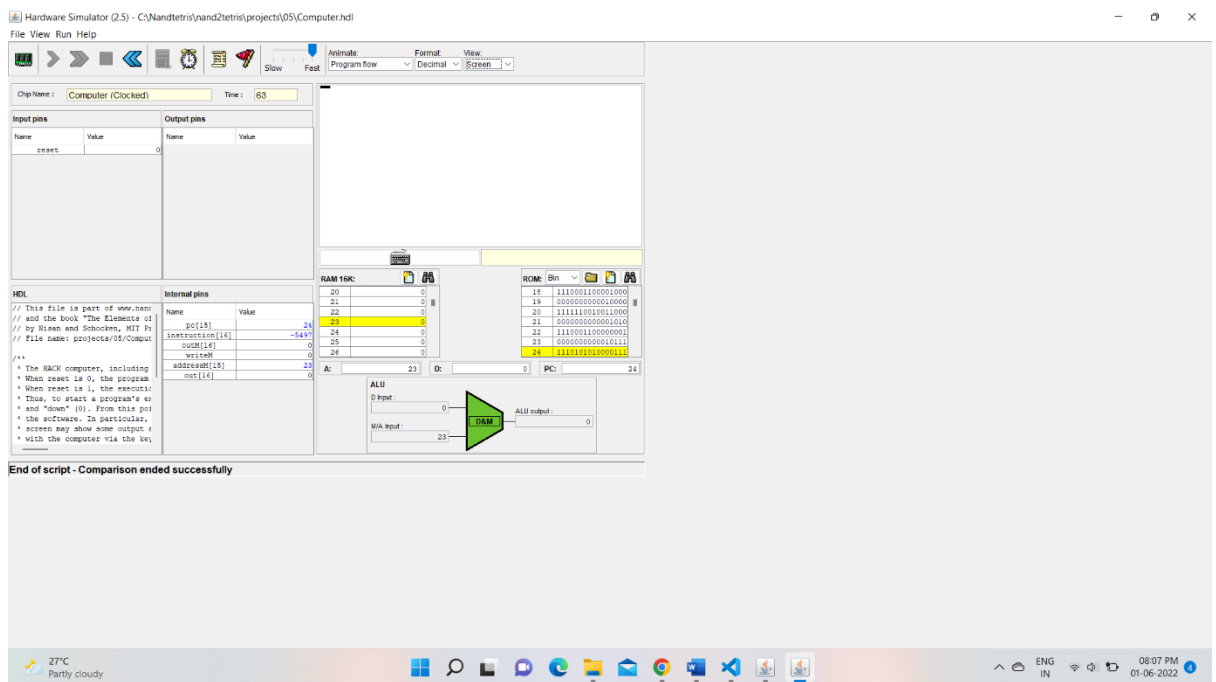
Computer.hdl

This file represents the hack computer . It contains the instruction memory (ROM32K) , CPU and data memory (RAM16K). ROM32K takes address as pc and the output is instruction . The output of the ROM32K becomes instruction of CPU. The inM of the ALU is the output of the memory of the computer. The reset value is got from the input reset of the computer module. The outM output pin of the CPU becomes input for the Memory module. The addressM pin of the CPU is connected to address pin of the Memory. The CPU also calculates the next pc values which is then supplied to the ROM32K.

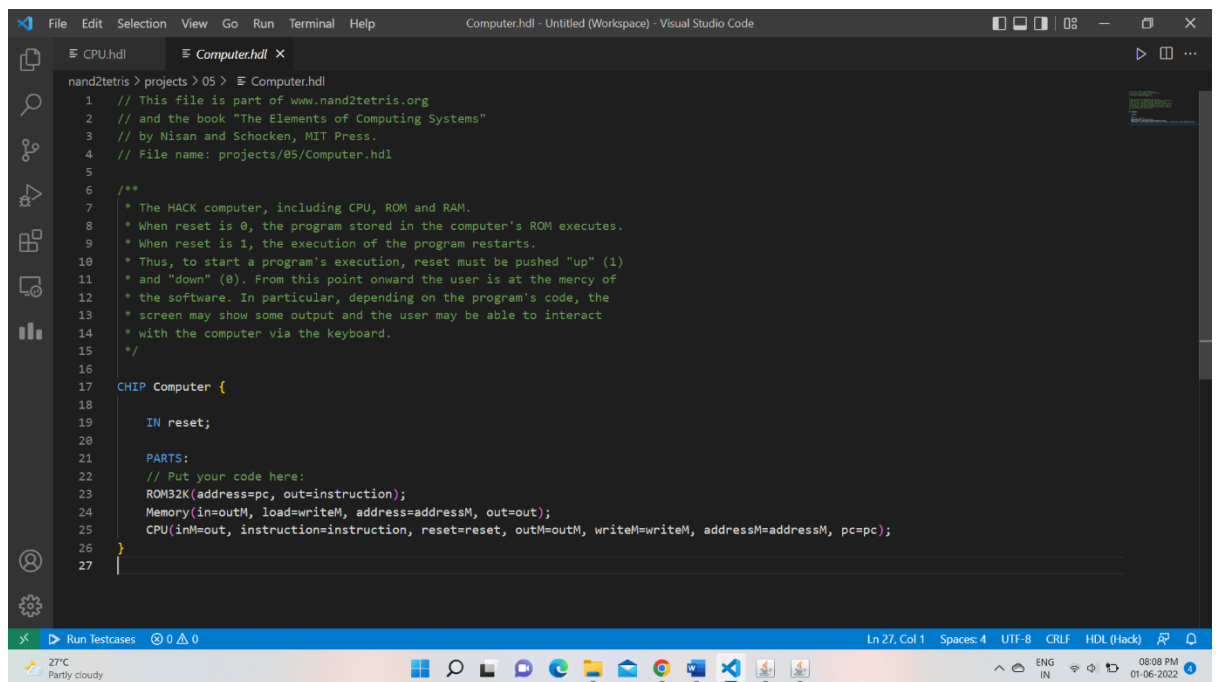
Screenshots:



Screenshot of testing of Computer.hdl



Screenshot of testing of Computer.hdl



```
1 // This file is part of www.nand2tetris.org
2 // and the book "The Elements of Computing Systems"
3 // by Nisan and Schocken, MIT Press.
4 // File name: projects/05/Computer.hdl
5
6 /**
7  * The HACK computer, including CPU, ROM and RAM.
8  * When reset is 0, the program stored in the computer's ROM executes.
9  * When reset is 1, the execution of the program restarts.
10  * Thus, to start a program's execution, reset must be pushed "up" (1)
11  * and "down" (0). From this point onward the user is at the mercy of
12  * the software. In particular, depending on the program's code, the
13  * screen may show some output and the user may be able to interact
14  * with the computer via the keyboard.
15  */
16
17 CHIP Computer {
18
19     IN reset;
20
21     PARTS:
22     // Put your code here:
23     ROM32K(address=pc, out=instruction);
24     Memory(in=outM, load=writeM, address=addressM, out=out);
25     CPU(inM=out, instruction=instruction, reset=reset, outM=outM, writeM=writeM, addressM=addressM, pc=pc);
26
27 }
```

Screenshot of program Computer.hdl

Project 6

Assembler.py

This program converts Hack assembly into binary so that it can be used by the computer. The file which is to be assembled has to be written correctly into the program. The program first removes the white spaces and comments present in the assembly code and is written into 'abc.txt'. After this the labels and the variables appropriate value is stored in a dictionary. Then the code scans each line of 'abc.txt' file and each line is split into A, C and label instructions. For A instructions the value after '@' is converted into a 15 bit binary number. This is appended to '0' as A instructions start with '0'. The binary form of the instruction is written into 'abc1.hack'. The C instruction has first three bits as '111'. Then the next bits are of the form 'acccccddjjj'. The bits are got according to the following conditions given in the picture below.

Translating C-instructions

Symbolic syntax: *dest = comp ; jump*

Binary syntax: `1 1 1 a c c c c c d d j j j`

comp	c	c	c	c	c	dest	d	d	d	effect: the value is stored in:
0	1	0	1	0	1	0	0	0	0	the value is not stored
1	1	1	1	1	1	1	0	0	1	RAM[A]
-1	1	1	1	0	1	0	0	1	0	D register
D	0	0	1	1	0	0	0	1	1	D register and RAM[A]
A	M	1	1	0	0	0	1	0	0	A register
!D		0	0	1	1	0	1	0	1	A register and RAM[A]
!A	!M	1	1	0	0	0	1	1	0	A register and D register
-D		0	0	1	1	1	1	1	1	A register, D register, and RAM[A]
-A	-M	1	1	0	0	1	1	1	1	
D+1		0	1	1	1	1	1	1	1	
A+1	M+1	1	1	0	1	1	1	1	1	
D-1		0	0	1	1	1	1	1	0	
A-1	M-1	1	1	0	0	1	1	1	0	
D+A	D+M	0	0	0	0	1	0	0	0	
D-A	D-M	0	1	0	0	1	1	1	1	
A-D	M-D	0	0	0	1	1	1	1	1	
D&A	D&M	0	0	0	0	0	0	0	0	
D A	D M	0	1	0	1	0	1	0	1	

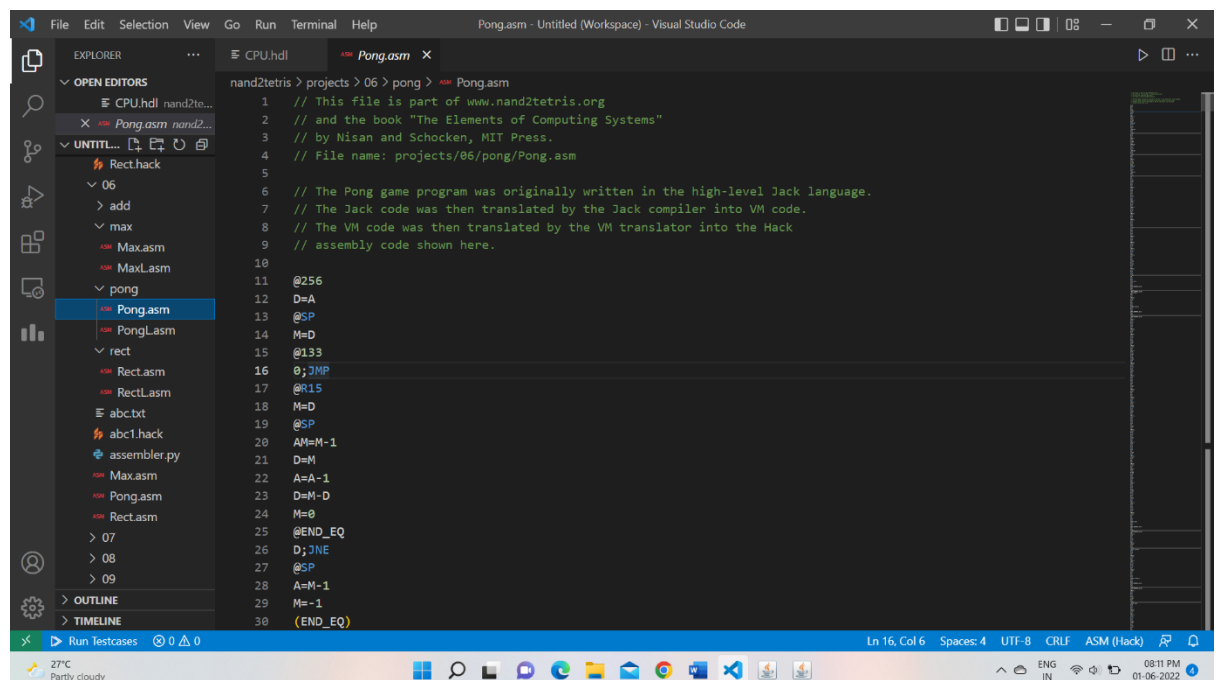
$a == 0 \quad a == 1$

Binary:

Example: `D = D+1 ; JLE` → `1110011111010110`

The a, c, d and j bits of the C instruction are got using separate functions which finds the appropriate value of these bits and returns them according to the assembly instruction. The a, c, d and j bits are concatenated and the binary representation of the instruction is written into abc1.hack file. For label instructions the corresponding value of the label is retrieved. This file can be used to run the program in the CPU Emulator . For the assembler to work, the program which is to be assembled has to be on the same folder as the assembler.

Screenshots:



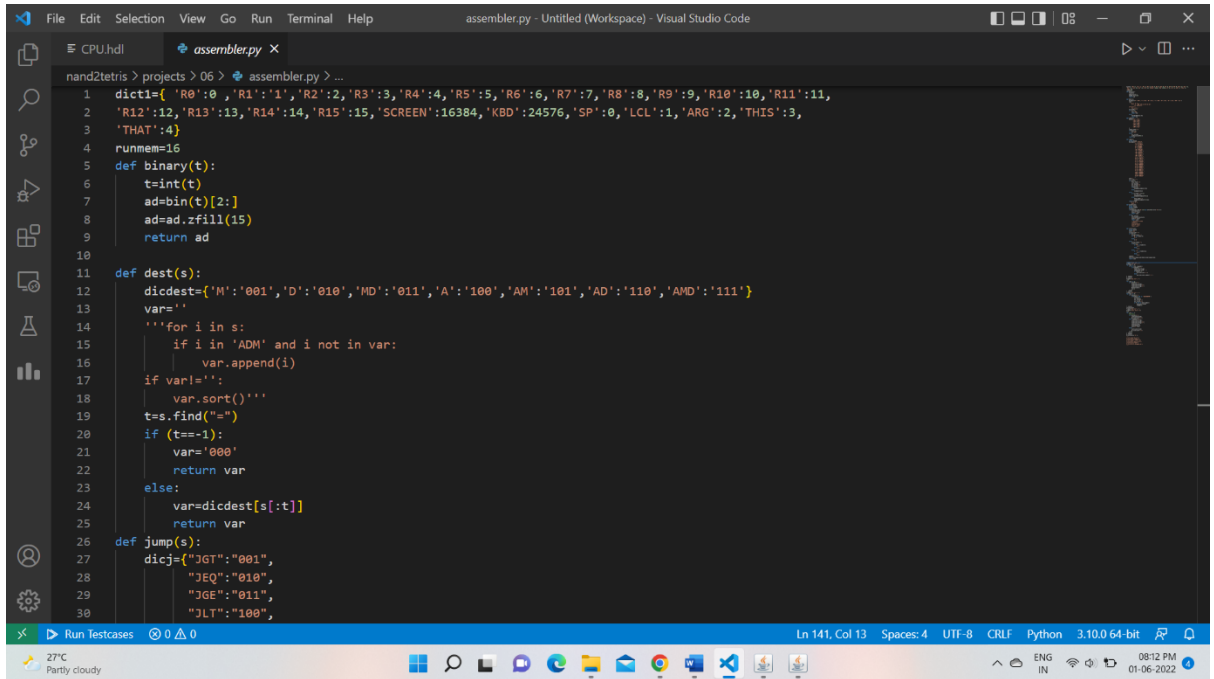
Screenshot of assembly program Pong.asm

```
1 @256
2 D=A
3 @SP
4 M=D
5 @133
6 0;JMP
7 @R15
8 M=D
9 @SP
10 AM=M-1
11 D=M
12 A=A-1
13 D=M-D
14 M=0
15 @END_EQ
16 D;JNE
17 @SP
18 A=M-1
19 M=-1
20 (END_EQ)
21 @R15
22 A=M
23 0;JMP
24 @R15
25 M=D
26 @SP
27 AM=M-1
28 D=M
29 A=A-1
30 D=M-D
```

Screenshot of file 'abc.txt'

```
1 0000000100000000
2 1110110000010000
3 0000000000000000
4 1110001100001000
5 000000010000101
6 111010101000111
7 000000000001111
8 1110001100001000
9 0000000000000000
10 1111110010101000
11 1111110000010000
12 1110110010100000
13 1111000111010000
14 1110101010001000
15 0000000000010011
16 111000110000101
17 0000000000000000
18 1111110010100000
19 11101101010001000
20 000000000001111
21 1111110000010000
22 111010101000111
23 000000000001111
24 1110001100001000
25 0000000000000000
26 1111110010101000
27 1111110000010000
28 1110110010100000
29 1111000111010000
30 1110101010001000
```

Screenshot of 'abc1.hack'



```
1 dict1={ 'R0':0, 'R1':1, 'R2':2, 'R3':3, 'R4':4, 'R5':5, 'R6':6, 'R7':7, 'R8':8, 'R9':9, 'R10':10, 'R11':11,
2 'R12':12, 'R13':13, 'R14':14, 'R15':15, 'SCREEN':16384, 'KBD':24576, 'SP':0, 'LCL':1, 'ARG':2, 'THIS':3,
3 'THAT':4}
4 runmem=16
5 def binary(t):
6     t=int(t)
7     ad=bin(t)[2:]
8     ad=ad.zfill(16)
9     return ad
10
11 def dest(s):
12     dicdest={'M':'001', 'D':'010', 'MD':'011', 'A':'100', 'AM':'101', 'AD':'110', 'AMD':'111'}
13     var=""
14     for i in s:
15         if i in 'ADM' and i not in var:
16             var.append(i)
17     if var!="":
18         var.sort()
19     t=s.find("=")
20     if (t!=-1):
21         var='000'
22         return var
23     else:
24         var=dicdest[s[:t]]
25         return var
26 def jump(s):
27     dicj={"JGT":"001",
28 "JEQ":"010",
29 "JGE":"011",
30 "JLT":"100",
```

Screenshot of program assembler.py

Github repository URL:

<https://github.com/narayanbarath-007/Nand2Tetris>