

NAND2TETRIS Report

By: Siddharth Kothari, IMT2021019

Project Github Repository link:

<https://github.com/siddharth-kothari9403/Nand2Tetris-ComputX.git>

Project 4:

In this project we were supposed to learn the Hack Assembly language and build 2 programs using the language:

1. Fill program- it would fill the screen with white or black color, based on the key pressed by the keyboard. If no key was pressed, then the screen turns white and if a key was pressed, it turns black.
2. Mult program- it is a simple multiplier that multiplies two numbers. The Hack ALU does not have a mult instruction, so to multiply 2 numbers, we would perform repetitive addition.

Implementations:

Fill :

We keep a while true loop which will continue to listen to the keyboard register forever. The value of the keyboard is read. If no key is pressed (keyboard register value is zero) then we set color = white(all zeros, binary representation of 0) , and fill all the pixels of the screen to white color.

But if a key is pressed, (keyboard register has non zero value) then we set color = black (all ones, binary representation of -1), and do the same again.

Pseudo code:

color = white ; (0)

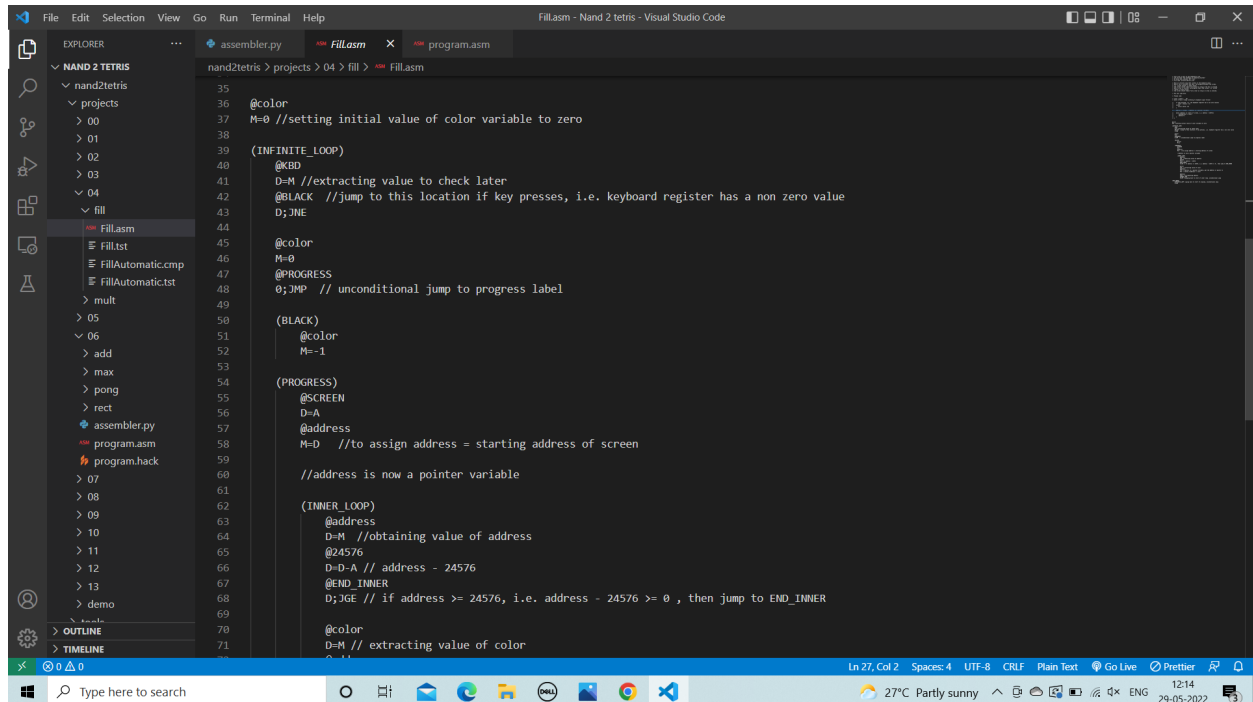
```
while (true){ //keep listening to keyboard input forever
```

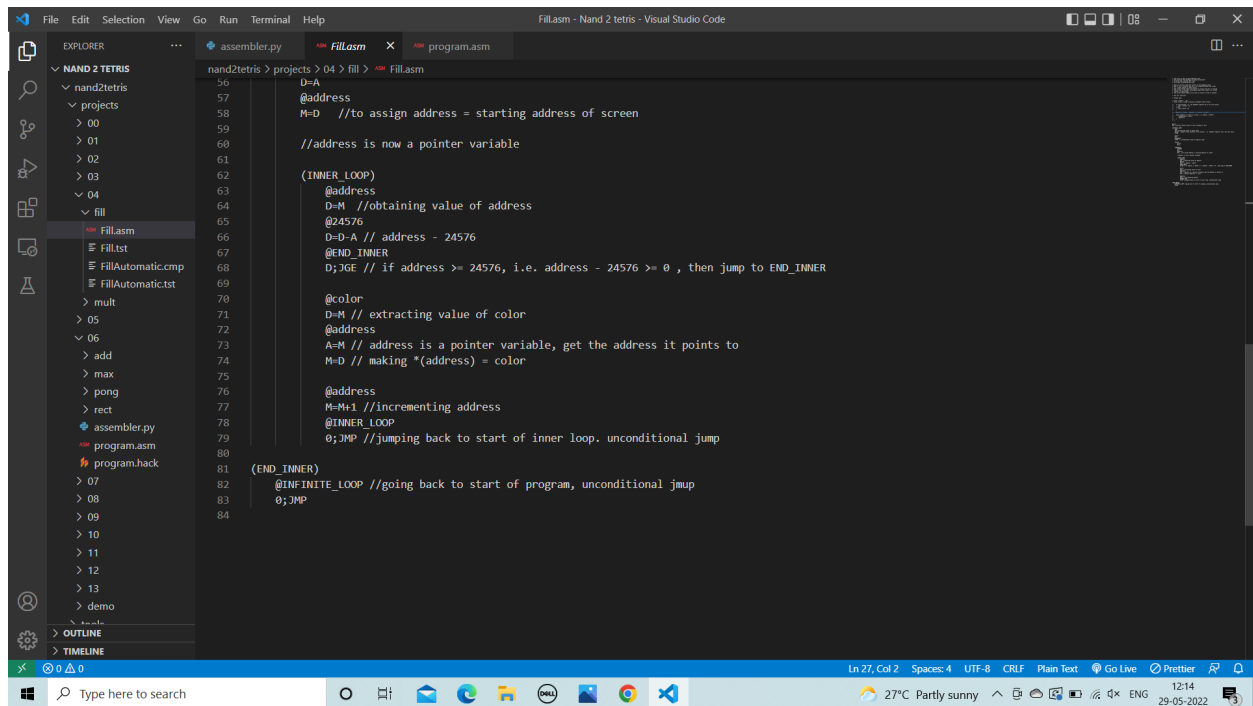
```
    if (key pressed, i.e. the keyboard register has a non zero value){  
        color = black; (-1)  
    }else{  
        color= white; (0)  
    }  
}
```

```
address = screen; //address is a pointer variable
```

```
while (address in limits of screen, i.e. address < 24576){  
    //filling the screen with the color  
    mem[address] = color;  
    address++;  
}  
}
```

Screenshots of code:





Mult:

This is a simple multiplier which will multiply two positive integer values and return the result. This is done by repetitive addition. The limits of the addition are :

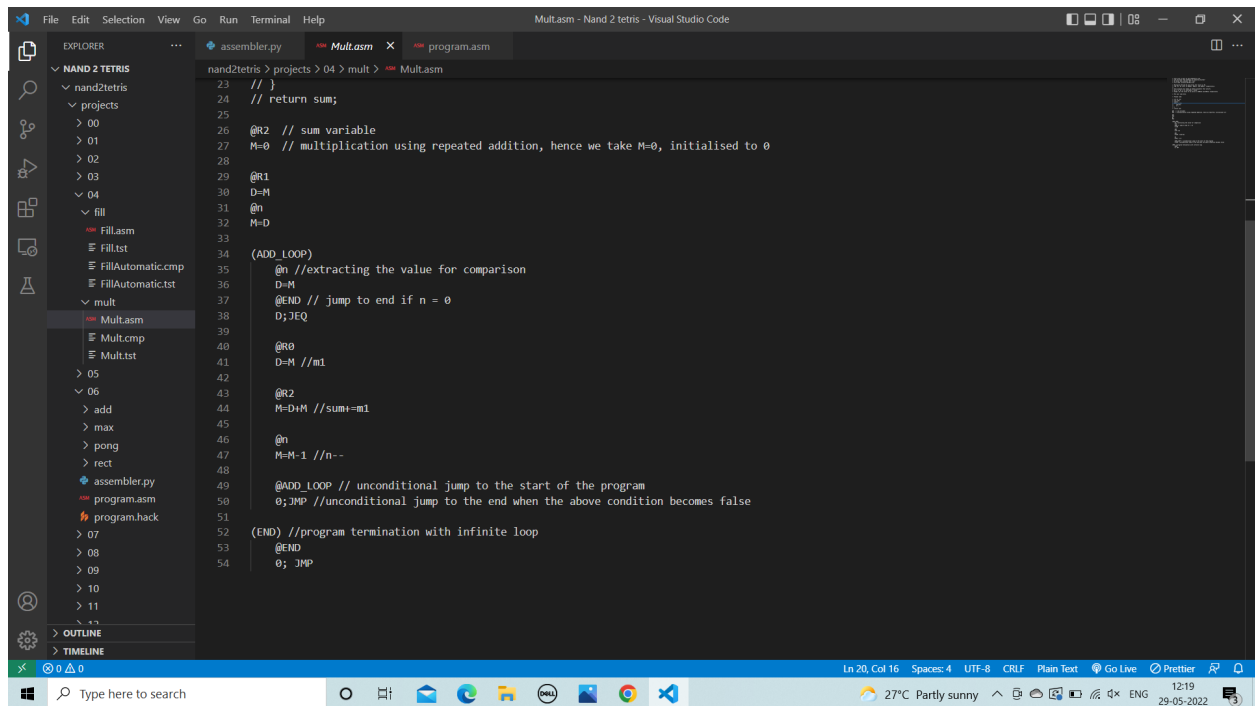
$R0 \leq 0, R1 \leq 0, R0.R1 < 32768$

Pseudo code:

```
int m1, m2;
int n=m2;
sum=0;
while (n>0){
    sum+=m1;
    n--;
}
```

return sum;

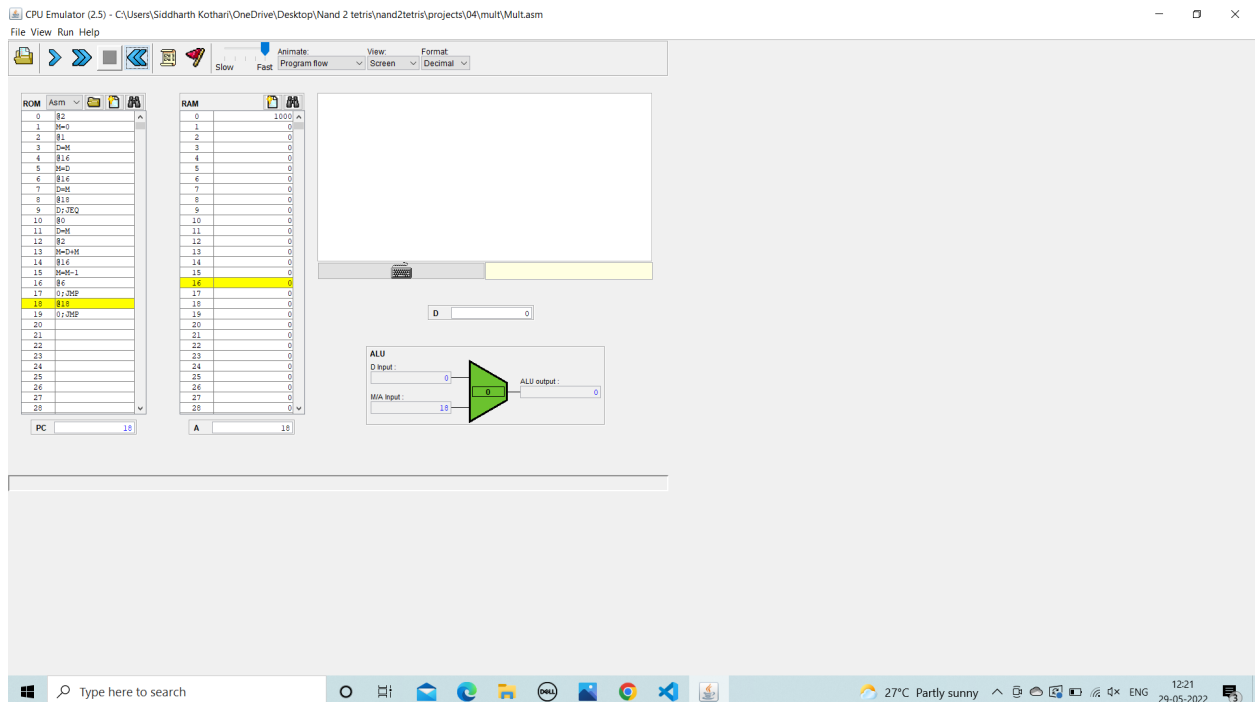
Screenshots of code:

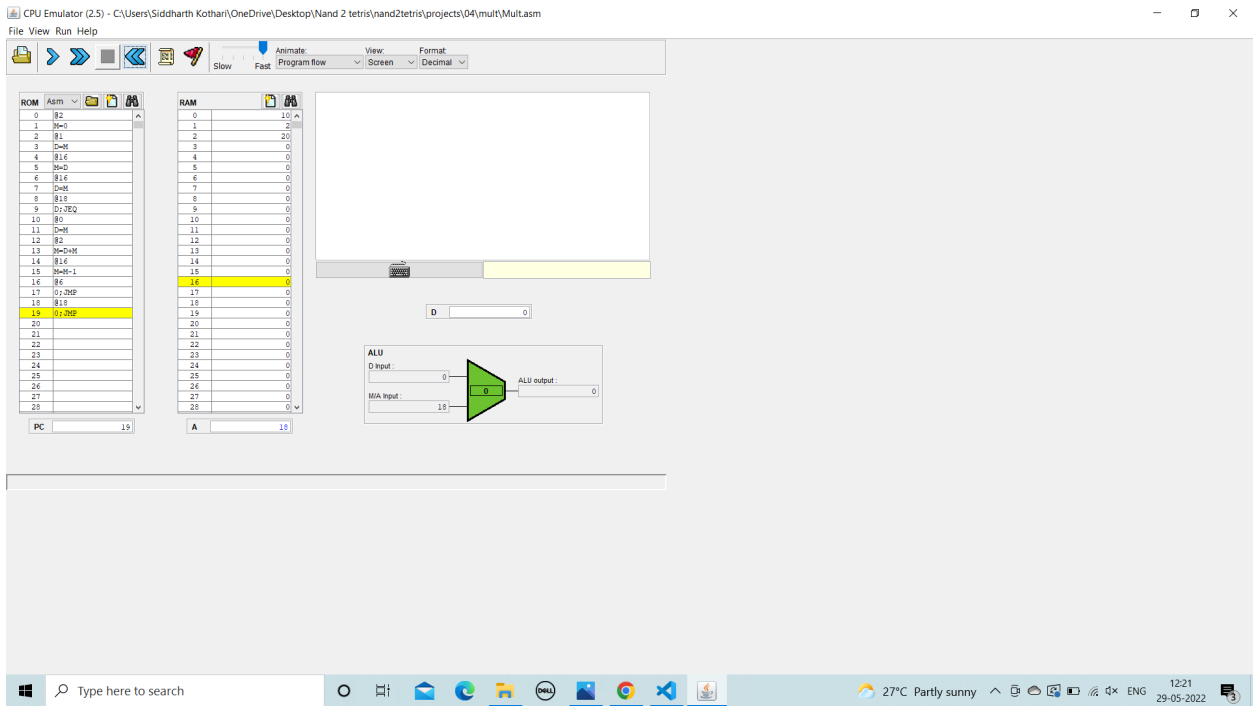
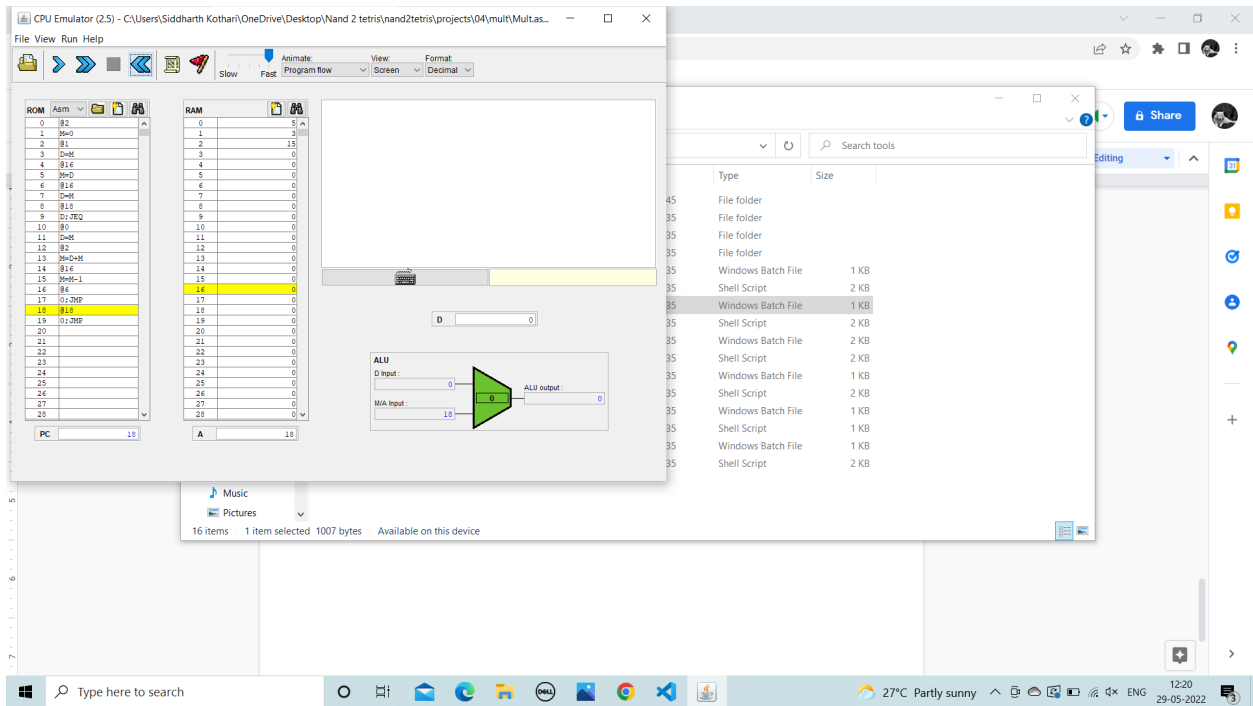


```
23 // }
24 // return sum;
25
26 @R2 // sum variable
27 M=0 // multiplication using repeated addition, hence we take M=0, initialised to 0
28
29 @R1
30 D=M
31 @n
32 M=D
33
34 (ADD_LOOP)
35 @n //extracting the value for comparison
36 D=M
37 @END // jump to end if n = 0
38 D;JEQ
39
40 @R0
41 D=M //m1
42
43 @R2
44 M=D+M //sum+=m1
45
46 @n
47 M=M-1 //n--
48
49 @ADD_LOOP // unconditional jump to the start of the program
50 0;JMP //unconditional jump to the end when the above condition becomes false
51
52 (END) //program termination with infinite loop
53 @END
54 0;JMP
```

Screenshots of some sample outputs:

The arguments are present in RAM locations 0 and 1, and the result is stored in location 2.





Project 5:

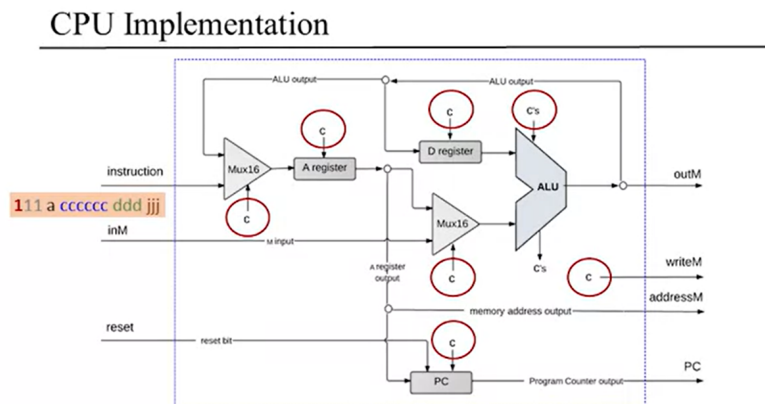
In this project we were supposed to make a CPU which will run the Hack Machine Language instructions using some built in chips that we were provided for project 1,2,3 and a Memory consisting of the following- 1. A 16K RAM which stores data; 2. An 8K RAM which serves as the Screen (for this we use a built in chip); 3. A Keyboard Register (another built in chip); and make sure that the addressing of these registers is proper.

Using these 2 chips and a built in ROM 32K chip, we would build the Hack Computer.

Implementations:

CPU:

The CPU Model is based on this diagram.



Implementation tips:

- Chip-parts: Mux16, ARegister, DRegister, PC, ALU, ...
- Control: use HDL subscribing to route instruction bits to the control bits of the relevant chip-parts.

The control signals for this computer are:

1. muxA: Controls the input to the A register, and is the selector for the Mux before A register.
2. writeA: Controls whether or not to write anything to the A register
3. writeD: Controls whether or not to write to D register or not
4. writeM: Is also an output to be sent to the Memory.
5. Apart from these, the ALU control bits are directly obtainable from the instruction, and so is the selector for the Mux before the second ALU input.
6. The instruction[15] bit is the most important info that controls the entire functioning of the ALU.
7. PC is controlled by the load control signal.

We have created 3 helper chips to generate control signals muxA, writeA, writeD, writeM, and load.

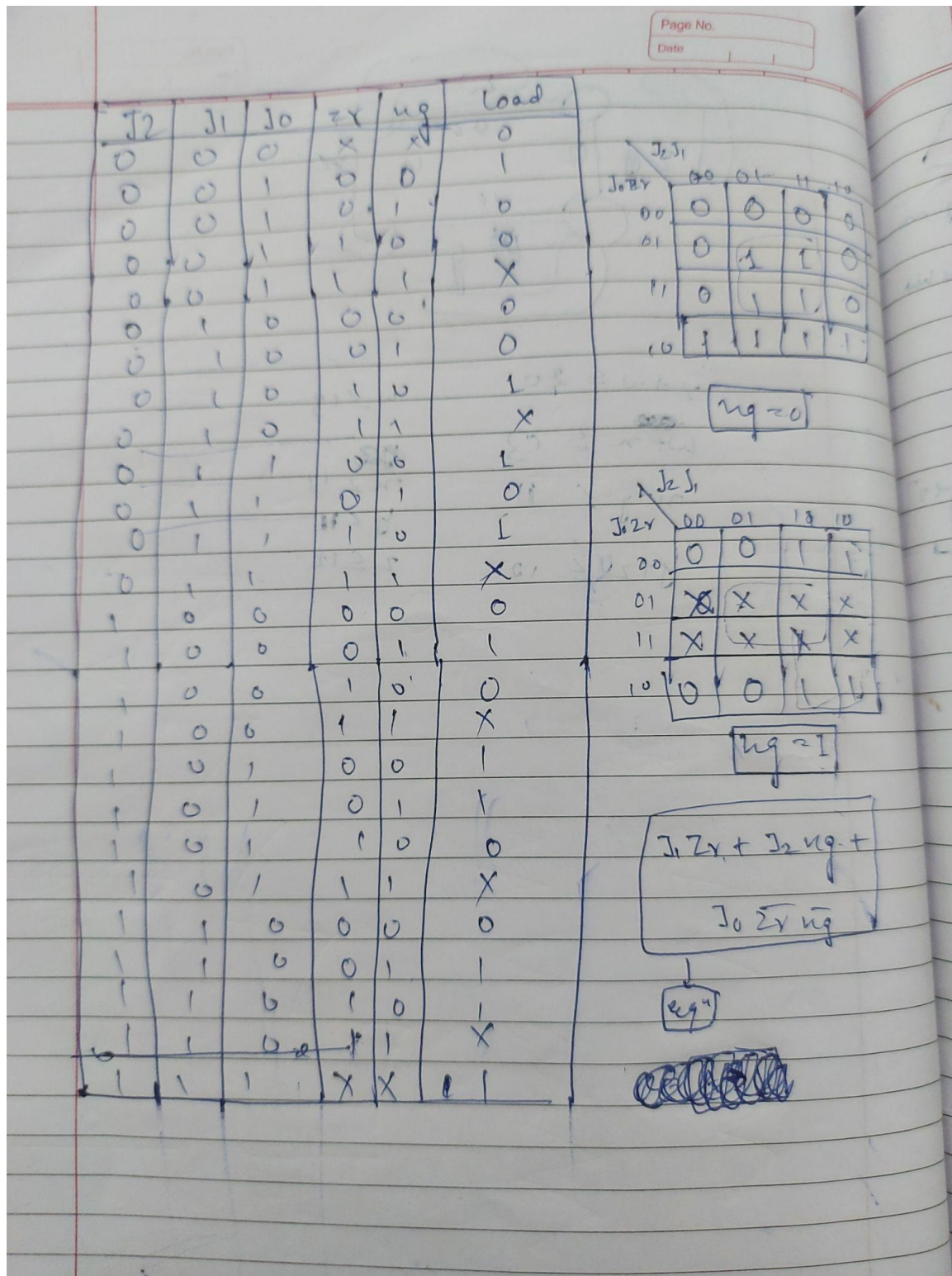
Jump Helper:

Generates the load control signal. Takes 5 inputs, the 3 jump bits and the 2 ALU outputs zr, and neg. Based on these values, it tells whether to jump or not. Load = 1 and instruction= C type will tell that we are supposed to jump to the address mentioned in A register. Load=0 or instruction type= A tells that we shouldn't jump. Hence the value of Load is in an AND gate with instruction[15] to tell where to get next instruction from.

Load is generated based on the following truth table:

J2	J1	J0	Meaning	Load
0	0	0	No Jump	Always 0
0	0	1	JGT	1 when zr AND ng = 0, else 0
0	1	0	JEQ	1 when zr = 1, else 0
0	1	1	JGE	1 when ng=0 else 0
1	0	0	JLT	1 when ng=1 else 0
1	0	1	JNE	1 when zr = 0, else 1
1	1	0	JLE	1 when zr OR ng = 1 else 0
1	1	1	JGT	Always 1

The K Map and equation for this chip is as follows:



According to this K Map, the chip logic is as follows:


```

17
18  Chip JumpHelper {
19      IN jump[3], zr, ng;
20      OUT load;
21
22      PARTS:
23      And(a=jump[1], b=zr, out=j1AndZr);
24      And(a=jump[2], b=ng, out=j2AndNg);
25
26      Not(in=zr, out=zrBar);
27      Not(in=ng, out=ngBar);
28
29      And(a=zrBar, b=ngBar, out=bars);
30      And(a=jump[0], b=bars, out=j0AndZrBarAndNgBar);
31
32      Or(a=j1AndZr, b=j2AndNg, out=inter);
33      Or(a=inter, b=j0AndZrBarAndNgBar, out=load);
34  }

```

WriteToAHelper:

This generates 2 control signals, muxA and writeA, based on the following truth tables:

// Truth table: (when muxA=0, we write from the output of ALU, when muxA=1, we write from the instruction obtained)

// instr = 1 -> c type, instr = 0 -> a type

// | instr | dest | muxA | writeA |

// | 0 | 0 | 1 | 1 |

// | 0 | 1 | 1 | 1 |

// | 1 | 0 | X | 0 |

// | 1 | 1 | 0 | 1 |

The code:

```
Chip generateA {  
  IN instr, dest;  
  OUT muxA, writeA;  
  
  PARTS:  
  Not(in=instr, out=muxA);  
  Not(in=instr, out=notInstr);  
  Or(a=notInstr, b=dest, out=writeA);  
}
```

WriteHelper:

This generates one output, write, which tells whether to write or not to the location. This is used to generate both writeD and writeM control signals.

Truth Table:

```
// Truth table:  
// instr = 1 -> c type, instr = 0 -> a type  
// | instr | dest | write |  
// | 0 | 0 | 0 |  
// | 0 | 1 | 0 |  
// | 1 | 0 | 0 |  
// | 1 | 1 | 1 |
```

Code:

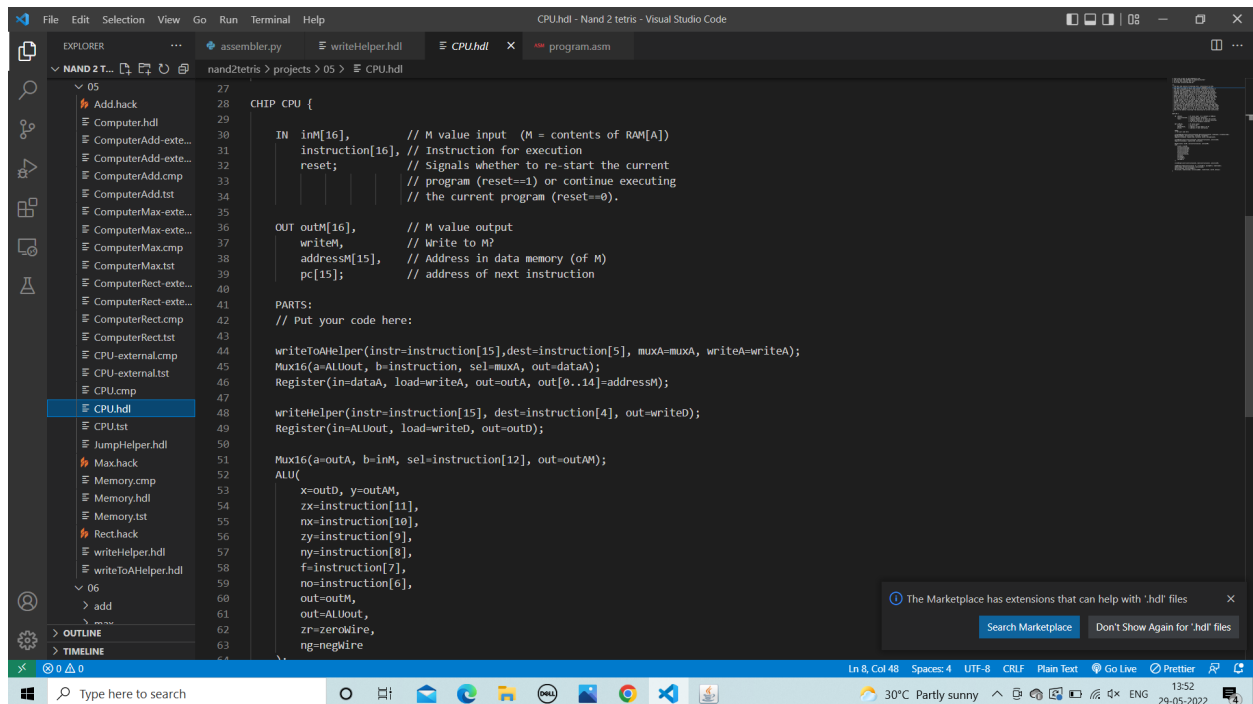
```
Chip writeHelper {  
  IN instr, dest;  
  OUT write;
```

PARTS:

And(a=instr, b=dest, out=write);

}

Now that we have gone over the control signals, the CPU chip logic is as follows.
This follows from the diagram described above.



```
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).

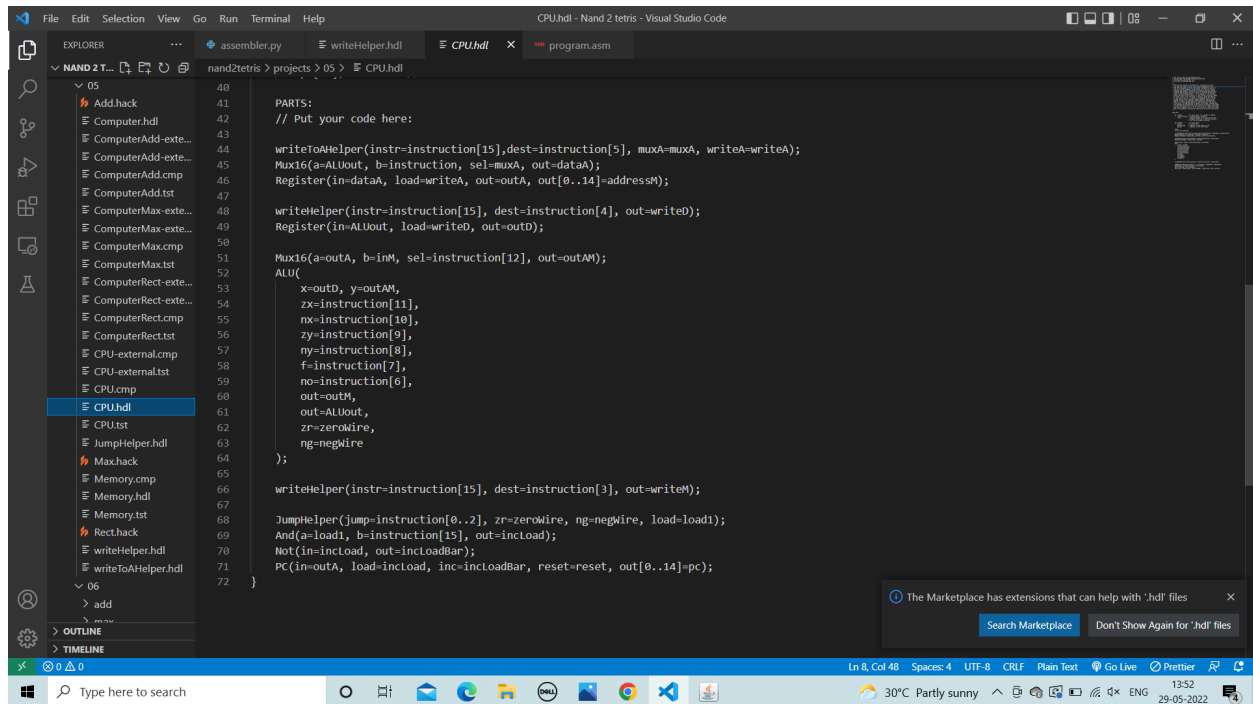
    OUT outM[16], // M value output
    writeM, // Write to M?
    addressM[15], // Address in data memory (of M)
    pc[15]; // address of next instruction

    PARTS:
    // Put your code here:

    writeToAHelper(instr=instruction[15],dest=instruction[5], muxA=muxA, writeA=writeA);
    Mux16(a=ALUout, b=instruction, sel=muxA, out=dataA);
    Register(in=dataA, load=writeA, out=outA, out[0..14]=addressM);

    writeHelper(instr=instruction[15], dest=instruction[4], out=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=zeroWire,
        ng=negWire
    );
}
```



Memory:

The Memory Model is based on the following:

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

Now we would require demultiplexing here. If the address is <16384 , i.e. the first 2 bits of the address are 00 or 01, we need to access the RAM. If the address is ≥ 16384 and <24576 , (first 2 bits are 10) we need to access the Screen Chip. Finally if the address is equal to 24576, (first 2 bits are 11), then we need to access the keyboard.

Both demultiplexing and multiplexing are done by the same address bits.

The code is as follows:

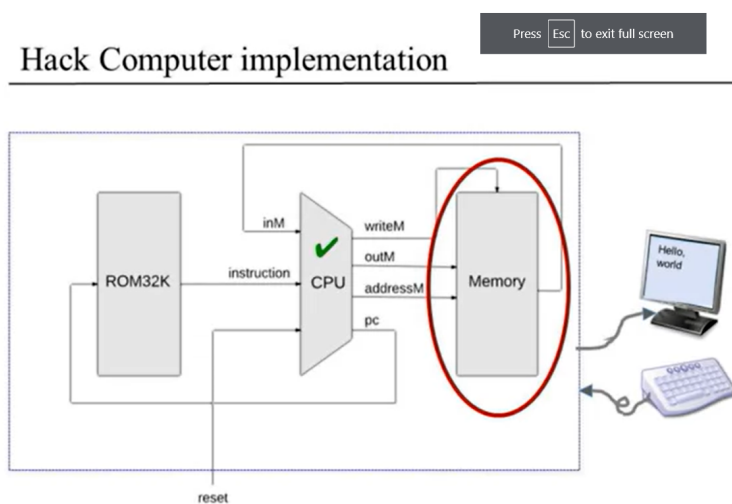
```

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
32      DMux4Way(in=load, sel=address[13..14], a=RAM1, b=RAM2, c=scrn, d=kbd);
33      Or(a=RAM1, b=RAM2, out=RAMSel);
34
35      RAM16K(in=in, load=RAMSel, address=address[0..13], out=outRAM);
36      Screen(in=in, load=scrn, address=address[0..12], out=outScreen);
37      Keyboard(out=outKBD);
38
39      Mux4Way16(a=outRAM, b=outRAM, c=outScreen, d=outKBD);
40  }

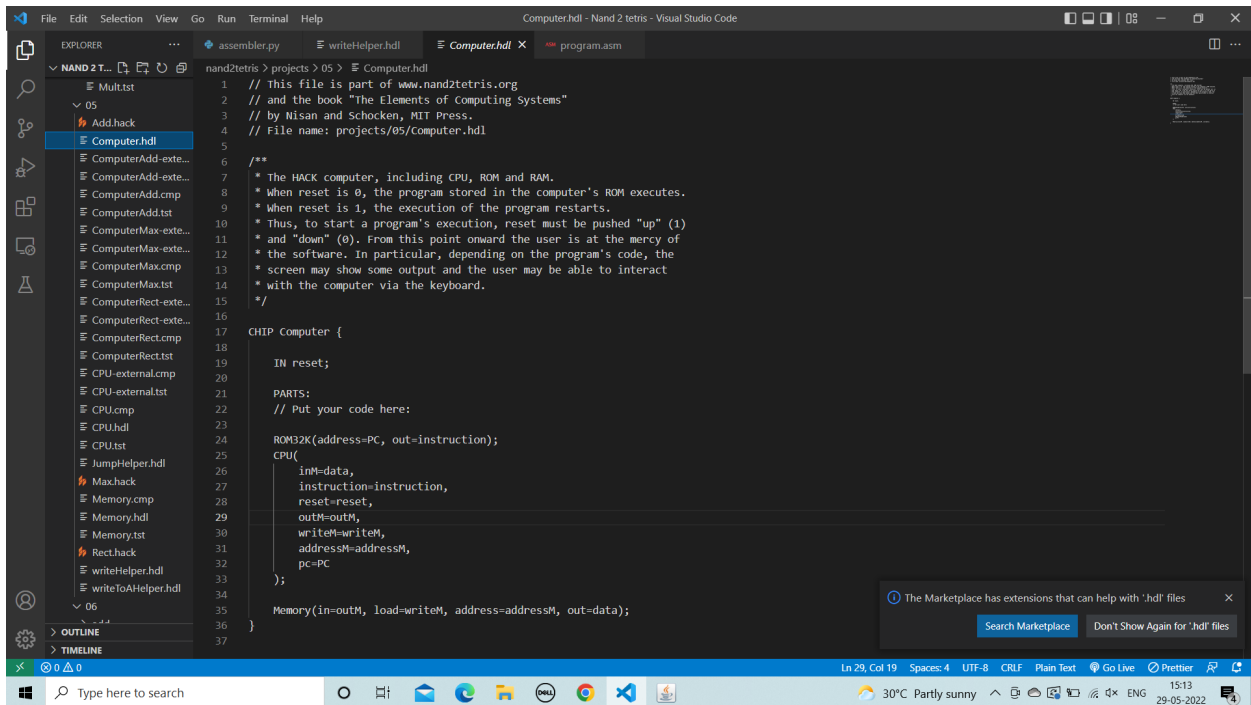
```

Computer:

The computer is just a combination of CPU, Memory and ROM32K chips, according to the following diagram.



The code is as follows:



The screenshot shows the Visual Studio Code editor with the 'Computer.hdl' file open. The Explorer sidebar on the left shows the project structure under 'NAND2TETRI...'. The main editor area displays the following code:

```
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     IN reset;
19
20     PARTS:
21     // Put your code here:
22
23     ROM32K(address=PC, out=instruction);
24     CPU(
25         inM=data,
26         instruction=instruction,
27         reset=reset,
28         outM=outM,
29         writeM=writeM,
30         addressM=addressM,
31         pc=PC
32     );
33     Memory(in=outM, load=writeM, address=addressM, out=data);
34 }
35
36
37
```

Project 6:

Till now, we had done the writing of the instructions in hack assembly and the designing of the CPU and Computer to run the instructions in binary. Now we write the assembler, the program to convert the assembly instructions to the binary instructions.

I have written the assembler in Python language.

How to run:

Command line : `python3 assembler.py`

The program takes 1 input: the name of the assembly file it has to read instructions from. Eg. `program.asm` or `add.asm` etc.

Important: The .asm file should be in the same location as the assembler program.

It will then read the assembly instructions and will write the binary instructions into a file by the name of program.hack.

This assembler is complete and will resolve any symbols and remove comments and white spaces from the assembly code, and then translate it into binary format.

The Logic:

Part 1: Handling White Spaces

The logic of the assembler is simple. First we read all the lines in the code as it is. We just remove any extra white spaces from the instructions.

We run 2 for loops after that:

The first one is to remove those lines which contain only comments and no code, and the empty strings which are left after removing white spaces(spaces and newline characters).

At the end of this loop, the only lines of code left contain either only the instructions or instructions with comments after them in the same line.

Eg. D=D+M

O;JMP // Unconditional jump

These are the only 2 instructions left.

Now in the second for loop, we split the lines at the // character, which indicates a comment, we remove it and then append the instruction back to a new list after again removing unnecessary white spaces.

In this way, all comments and white spaces are removed and we are left with a list containing only the instructions.

Part 2: Handling Labels

This is the first pass over the assembly code. We only access the labels and add their corresponding addresses to the symbol table, and then remove them from the list of code as they are not actual code.

This leaves us with a code with labels and their corresponding addresses added to the symbol table, and labels removed.

Part 3: Actual Conversion to Binary:

Over here we will resolve the A and C instructions, add variables to the symbol table and resolve all symbols to numbers, and get the binary representation of them.

A instruction:

- Starts with the @ sign.
- If the content following @ is a number, convert it to integer
- Else if it is a symbol which is already not resolved, then first add it to the symbol table and then resolve it to integer
- Else if it is a symbol and is already there in the symbol table, then resolve it to integer directly.
- Now after obtaining the integer, simply get its 15 bit binary representation and add a zero to it(opcode of A type). The instruction is resolved.

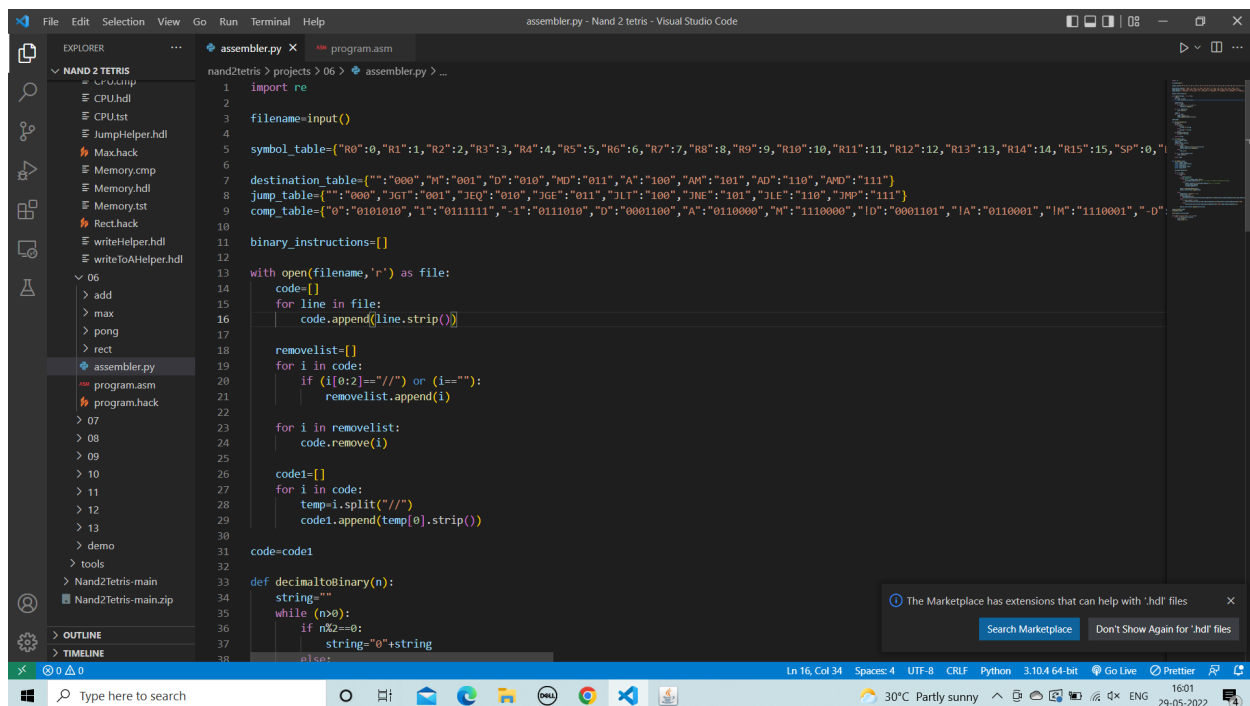
C instruction:

- General format: dest = comp ; jump
- dest and jump both are optional fields, but atleast one out of them will always be present.
- Therefore there are 2 kinds of c instructions: where all 3 fields are present, and where only 2 are present (either dest and comp, or comp and jump).
- When 3 are there, the association is easy. Splitting at = and ; gives 3 values. The first is dest, second is comp, third is jump.
- When 2 are there, split at =. If dest bits are there, it is split into 2, else it is split into one. This is the defining criteria.

- If it is split into 2 at = , then the first half represents dest bits, and the second half represents comp bits. The jump bits would be 000.
- If it was not split, then the first half represents comp bits and second half represents jump bits. The dest bits would be 000.
- The instruction is resolved.

For both these instructions we keep appending to the list of binary instructions and finally write the list separated by newline characters to the **program.hack** file.

The code:



```

1  import re
2
3  filename=input()
4
5  symbol_table={"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,"
6
7  destination_table={"": "000", "M": "001", "D": "010", "MD": "011", "A": "100", "AM": "101", "AD": "110", "AMD": "111"}
8  jump_table={"": "000", "JGT": "001", "JEQ": "010", "JGE": "011", "JLT": "100", "JNE": "101", "JLE": "110", "JMP": "111"}
9  comp_table={"0": "0101010", "1": "0111111", "-1": "0111010", "D": "0001100", "A": "0110000", "M": "1110000", "LD": "0001101", "LA": "0110001", "LM": "1110001", "-D":
10
11 binary_instructions=[]
12
13 with open(filename,'r') as file:
14     code=[]
15     for line in file:
16         code.append(line.strip())
17
18     removelist=[]
19     for i in code:
20         if (i[0:2]=="//") or (i==""):
21             removelist.append(i)
22
23     for i in removelist:
24         code.remove(i)
25
26     code1=[]
27     for i in code:
28         temp=i.split("//")
29         code1.append(temp[0].strip())
30
31     code=code1
32
33 def decimalToBinary(n):
34     string=""
35     while (n>0):
36         if n%2==0:
37             string="0"+string
38         else:

```

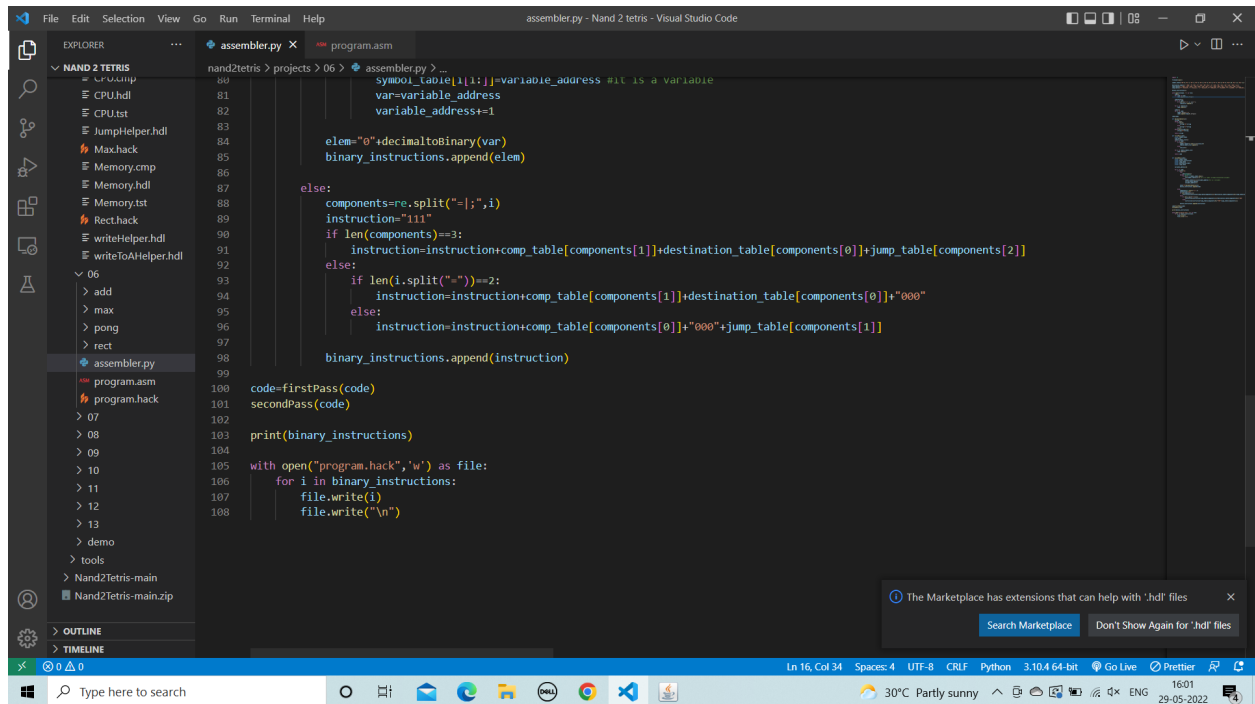
```
def decimalToBinary(n):
    string=""
    while (n>0):
        if n%2==0:
            string="0"+string
        else:
            string="1"+string
        n=n//2
    while(len(string)<15):
        string="0"+string
    return string

def firstPass(code):
    global symbol_table
    line_no=0
    remove_labels_list=[]
    for i in code:
        if i[0]=="(":
            symbol_table[i[1:len(i)-1]]-line_no+1
            remove_labels_list.append(i)
        else:
            line_no+=1
    for i in remove_labels_list:
        code.remove(i)
    return code

def secondPass(code):
    global symbol_table
    global binary_instructions
    global comp_table
    global destination_table
    global jump_table
```

```
variable_address=16

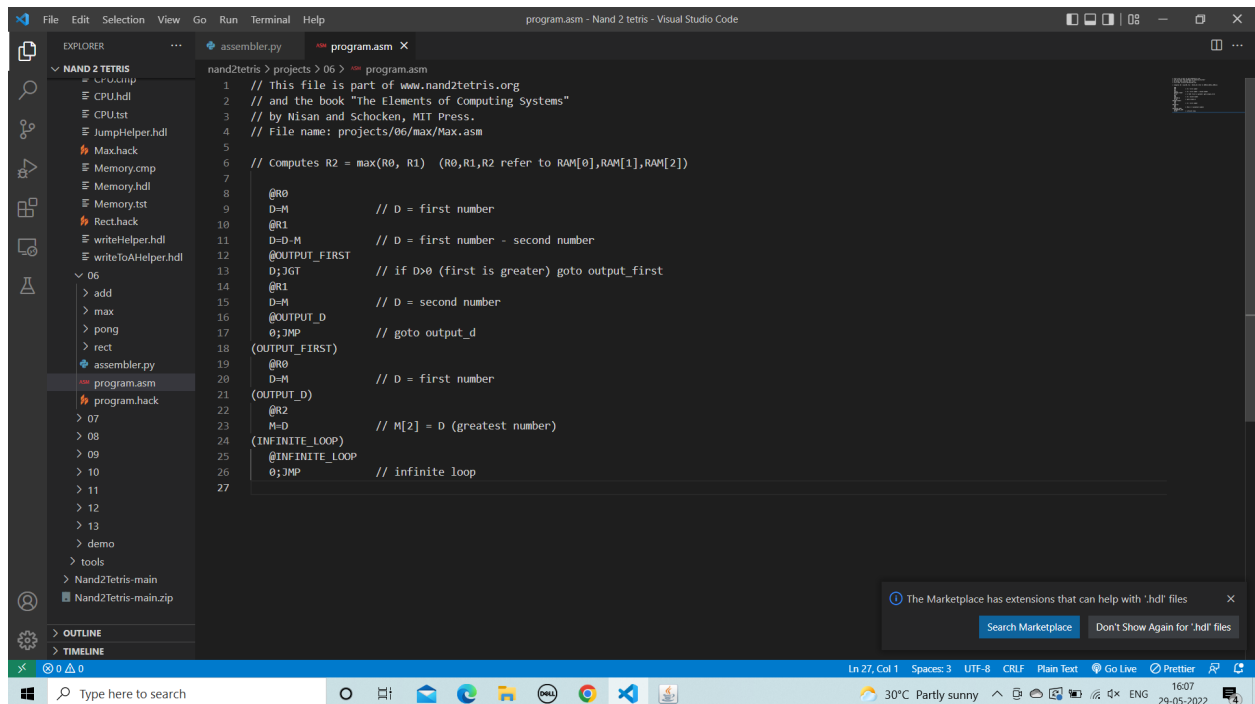
for i in code:
    if i[0]=="@":
        try:
            var=int(i[1:])
        except ValueError:
            if i[1:] in symbol_table.keys():
                var=symbol_table[i[1:]] #it is a label/ already encountered variable
            else:
                symbol_table[i[1:]]-variable_address #it is a variable
                var=variable_address
                variable_address+=1
        elem="0"+decimalToBinary(var)
        binary_instructions.append(elem)
    else:
        components=re.split("-",i)
        instruction="111"
        if len(components)==3:
            instruction=instruction+comp_table[components[1]]+destination_table[components[0]]+jump_table[components[2]]
        else:
            if len(i.split("-"))==2:
                instruction=instruction+comp_table[components[1]]+destination_table[components[0]]+"000"
            else:
                instruction=instruction+comp_table[components[0]]+"000"+jump_table[components[1]]
        binary_instructions.append(instruction)
```



```
180 symbol_table[i[i:i]] = variable_address #it is a variable
181 var = variable_address
182 variable_address += 1
183
184 elem = "0" + decimaltoBinary(var)
185 binary_instructions.append(elem)
186
187 else:
188     components = re.split("=", i)
189     instruction = "111"
190     if len(components) == 3:
191         instruction = instruction + comp_table[components[1]] + destination_table[components[0]] + jump_table[components[2]]
192     else:
193         if len(i.split("=")) == 2:
194             instruction = instruction + comp_table[components[1]] + destination_table[components[0]] + "000"
195         else:
196             instruction = instruction + comp_table[components[0]] + "000" + jump_table[components[1]]
197     binary_instructions.append(instruction)
198
199 code = firstPass(code)
200 secondPass(code)
201
202 print(binary_instructions)
203
204 with open("program.hack", "w") as file:
205     for i in binary_instructions:
206         file.write(i)
207         file.write("\n")
```

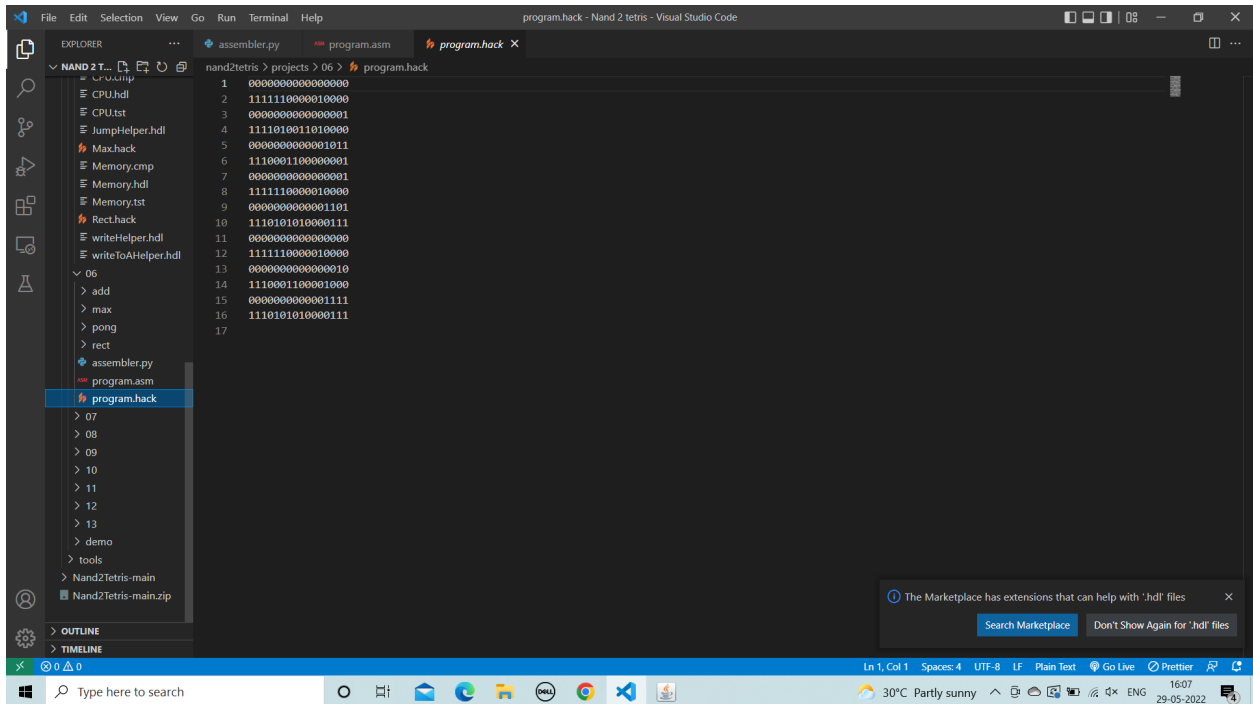
Codes generated for some .asm files:

Code:



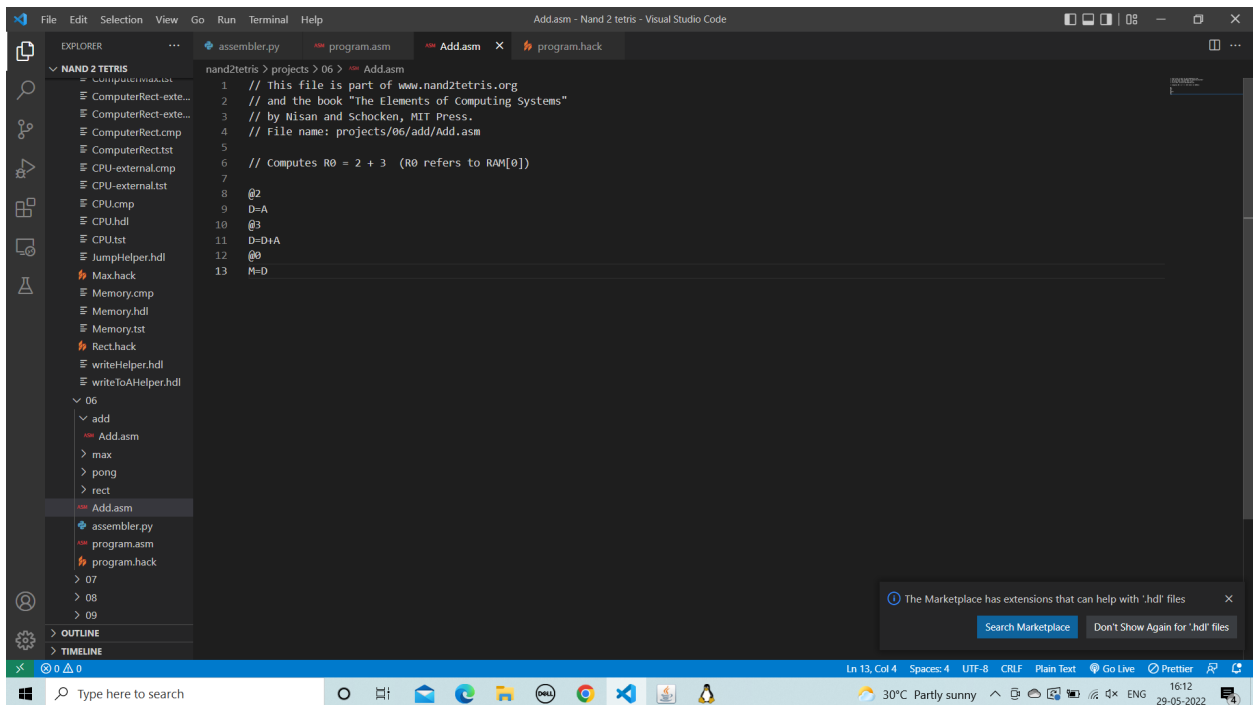
```
1 // This file is part of www.nand2tetris.org
2 // and the book "The Elements of Computing Systems"
3 // by Nisan and Schocken, MIT Press.
4 // File name: projects/06/max/Max.asm
5
6 // Computes R2 = max(R0, R1) (R0,R1,R2 refer to RAM[0],RAM[1],RAM[2])
7
8 @R0
9 D=M
10 @R1
11 D=D-M
12 @OUTPUT_FIRST
13 D;JGT
14 @R1
15 D=M
16 @OUTPUT_D
17 0;JMP
18 (OUTPUT_FIRST)
19 @R0
20 D=M
21 (OUTPUT_D)
22 @R2
23 M=D
24 (INFINITE_LOOP)
25 @INFINITE_LOOP
26 0;JMP
27
```

Output:

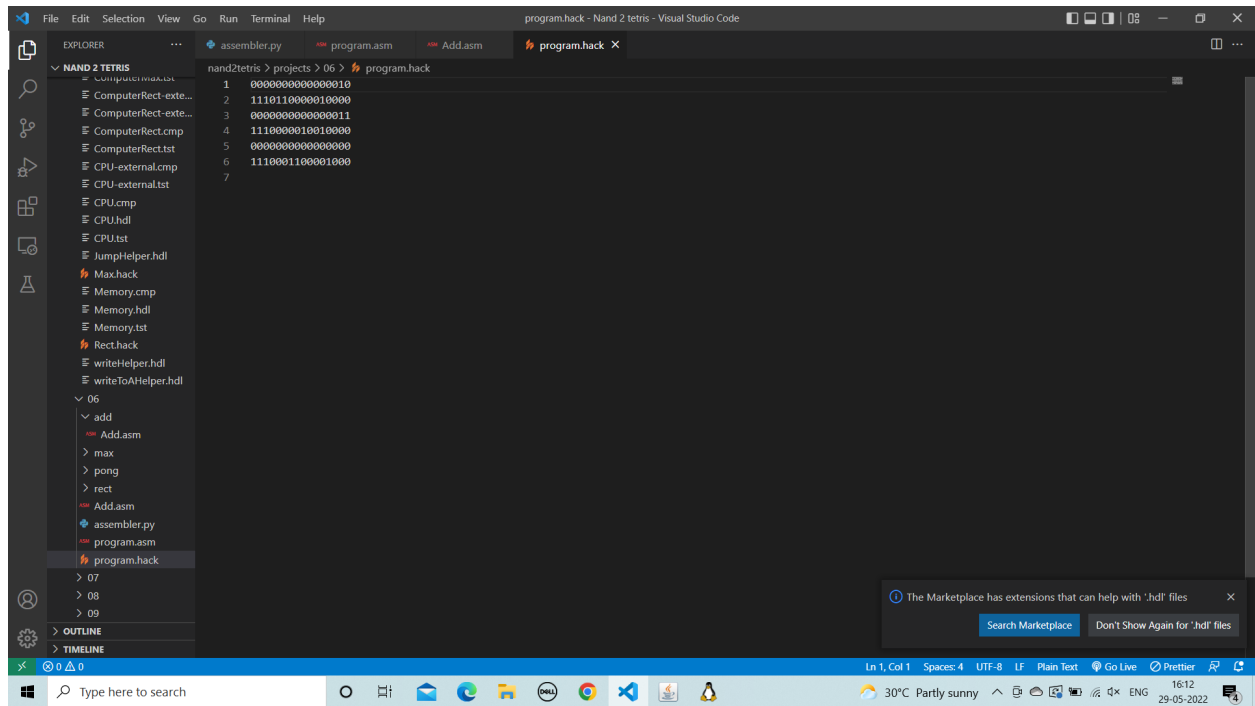


```
1 0000000000000000
2 1111110000010000
3 0000000000000001
4 1111010011010000
5 0000000000010111
6 1110001100000001
7 0000000000000001
8 1111110000010000
9 0000000000001101
10 1110101010000111
11 0000000000000000
12 1111110000010000
13 0000000000000010
14 1110001100010000
15 0000000000011111
16 1110101010000111
17
```

Code:



```
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/06/add/Add.asm
5
6 // Computes R0 = 2 + 3 (R0 refers to RAM[0])
7
8 @2
9 D=A
10 @3
11 D=D+A
12 @0
13 M=D
```



Thank You