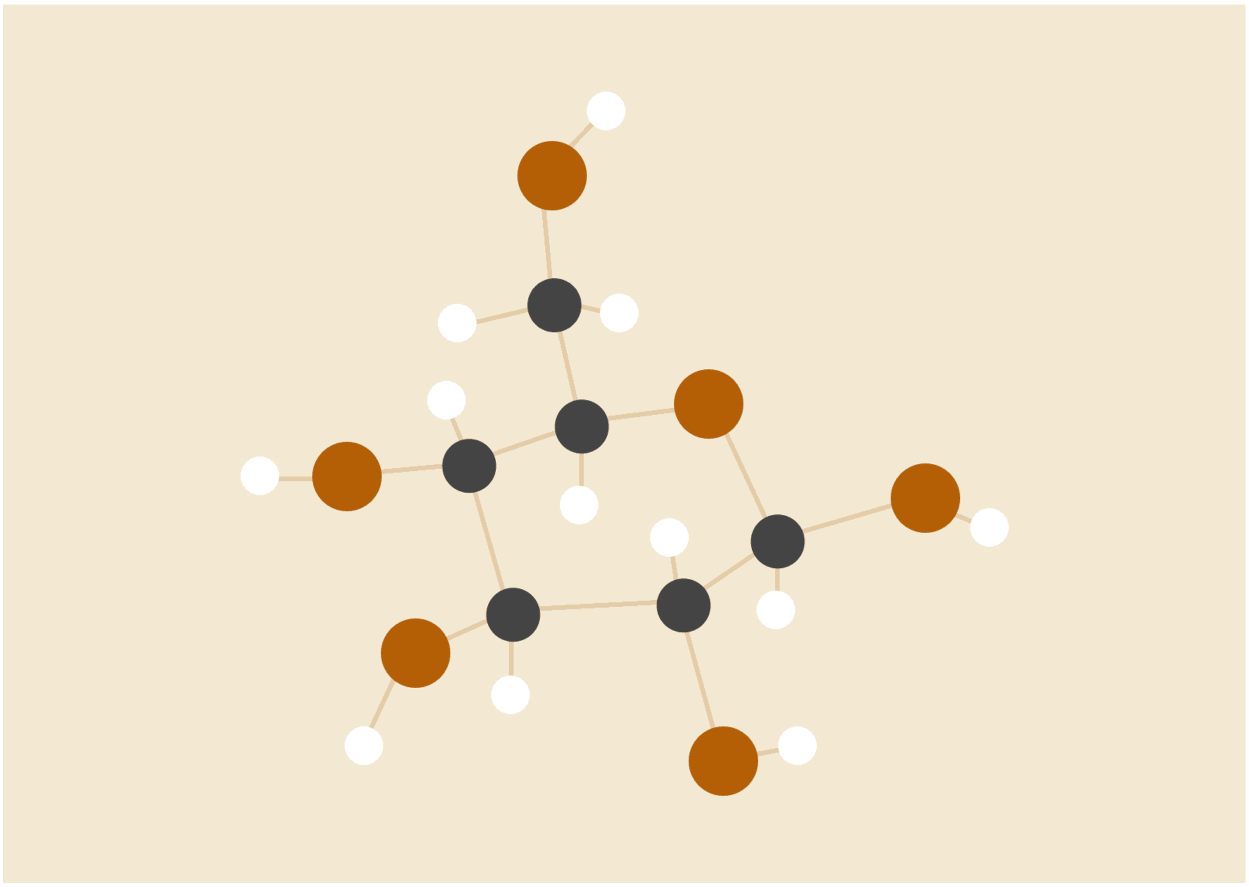
NAND2TETRIS



RITHVIK RAMASANI

IMT2020543

PROJECT 1

As part of project 1, it is already given to us, it consists of the implementation of all the basic elementary and multi-way variant chips.

PROJECT 2

As part of project 2, it is already given to us, it consists of the implementation of arithmetic chips like Half-adder, Full-adder, 16-bit adder, and Arithmetic logic unit (ALU).

PROJECT 3

As part of project 3, it is already given to us, it consists of the implementation of sequential logic, in a way it involves implementation of the memory unit by the use of flip-flops.

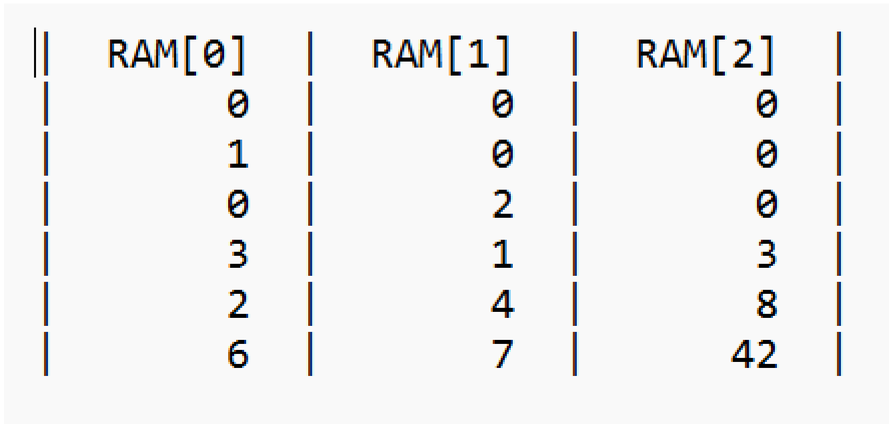
PROJECT 4

Basically, project 4 mainly deals with understanding the Hack assembly language developed in the construction of a hack computer. This hack architecture has 2 types of instructions.

- A instruction - @vaule
- C instruction - dest=comp;jump

This project has 2 programs to design in hack assembly language.

1. Multiplication Program - In this program, we need to multiply two numbers, which are stored in registers 0 and 1, and store it in register 2. I implemented this multiplication by addition in a loop. For each loop I decremented the value in register 1 and added the value in register 0 to value in register 2, which is initially 0, and updated it. Apart from updating each time, I checked whether either one of them is zero to terminate looping. Output result after running the test script in the simulator is shown below.



```
||  RAM[0]  |  RAM[1]  |  RAM[2]  |
|       0   |       0   |       0  |
|       1   |       0   |       0  |
|       0   |       2   |       0  |
|       3   |       1   |       3  |
|       2   |       4   |       8  |
|       6   |       7   |      42  |
```

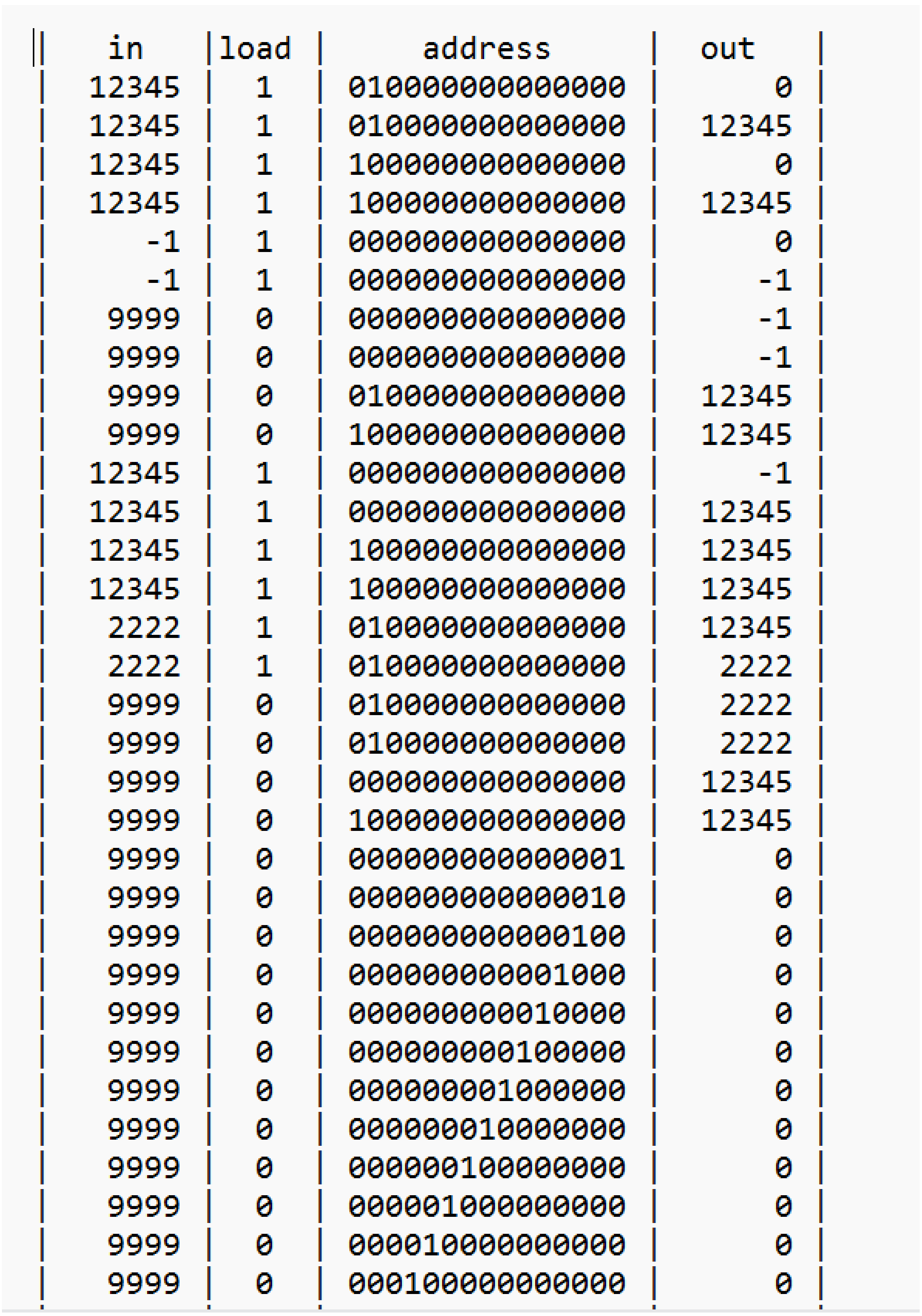
2. Fill Program - This is an interactive program. Based on the activity on the keyboard, i.e. when a key is pressed the whole screen in the simulator is turned to black. Initially, we set the color to white. We run an infinite loop to track keyboard events till we exit the simulation. When a key in the keyboard is pressed, we detect the event in the loop and by taking the starting pixel address and setting their color to black in a way change the magnitude of the set of 32 pixels to -1 i.e. all 32 one's.

PROJECT 5

This project mainly deals with the implementation of hack architecture. It mainly consists of three parts.

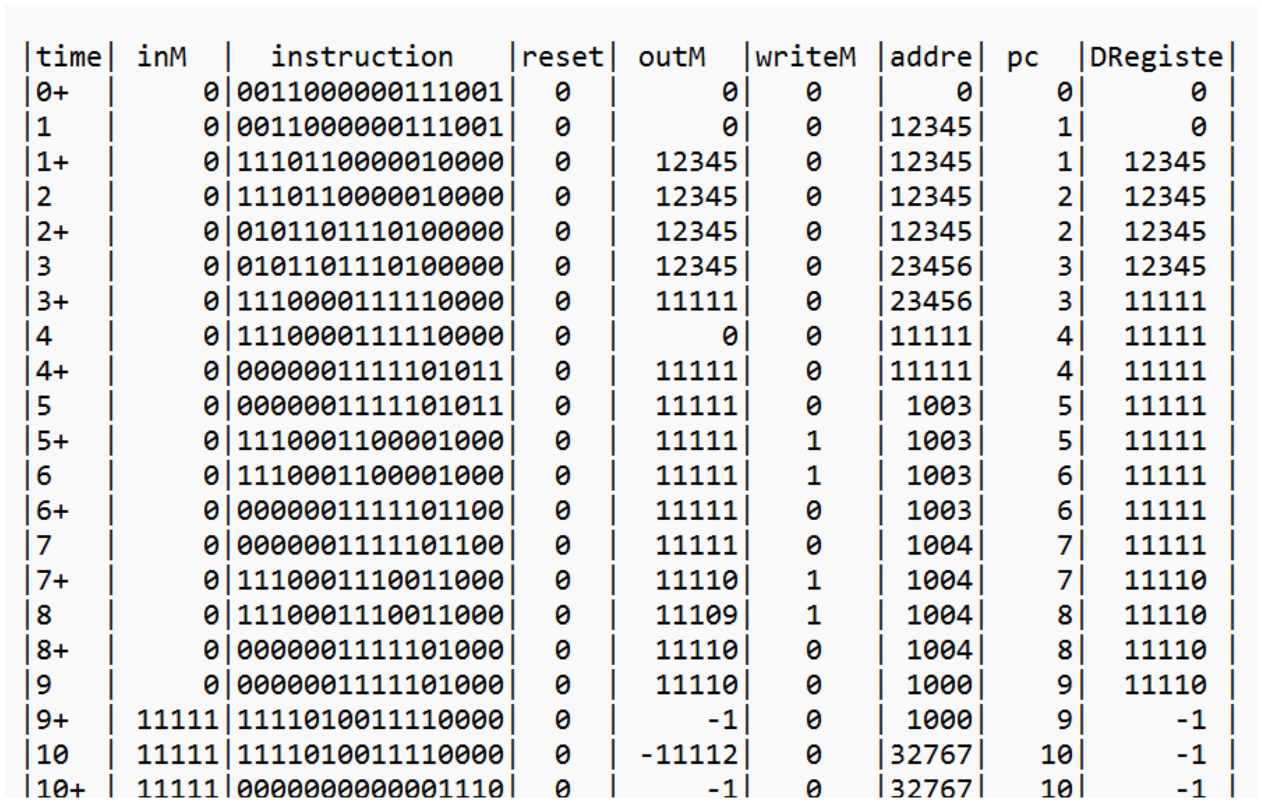
1. Memory Implementation - Memory in hack architecture in of 3 parts
    a. RAM - This basically occupies the first 16K registers.
    b. SCREEN - This is a map pointing to each pixel on screen, each pixel can be black or white depending on the value stored in its corresponding bit. This occupies the next 8K registers.
    c. Keyboard - This is 1 register memory, gives different values for different keys pressed on the keyboard.

Output result after running one of the test scripts in the simulator is shown below.



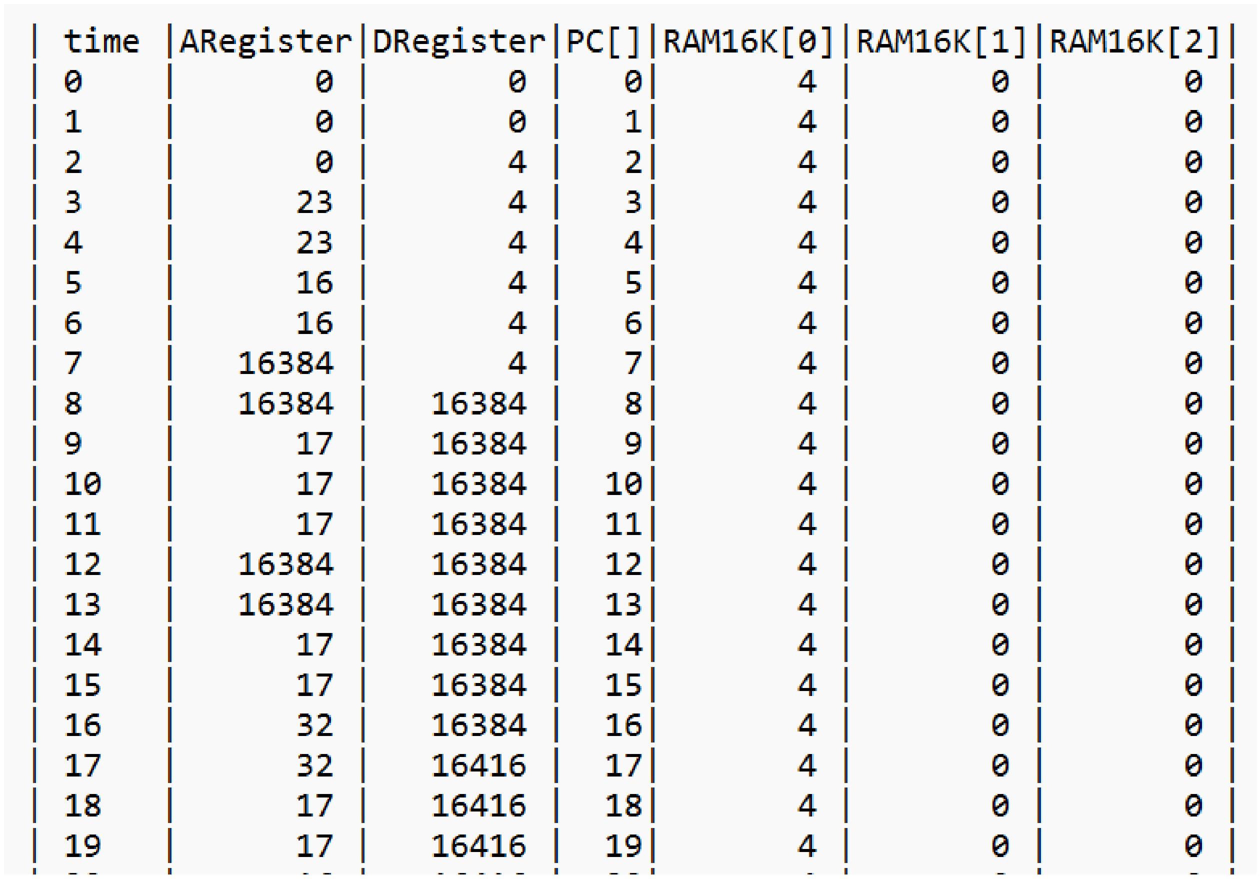
```
|   in   |load |     address     |  out   |
| 12345  |  1  | 010000000000000 |      0 |
| 12345  |  1  | 010000000000000 |  12345 |
| 12345  |  1  | 100000000000000 |      0 |
| 12345  |  1  | 100000000000000 |  12345 |
|     -1 |  1  | 000000000000000 |      0 |
|     -1 |  1  | 000000000000000 |     -1 |
|   9999 |  0  | 000000000000000 |     -1 |
|   9999 |  0  | 000000000000000 |     -1 |
|   9999 |  0  | 010000000000000 |  12345 |
|   9999 |  0  | 100000000000000 |  12345 |
|  12345 |  1  | 000000000000000 |     -1 |
|  12345 |  1  | 000000000000000 |  12345 |
|  12345 |  1  | 100000000000000 |  12345 |
|  12345 |  1  | 100000000000000 |  12345 |
|   2222 |  1  | 010000000000000 |  12345 |
|   2222 |  1  | 010000000000000 |   2222 |
|   9999 |  0  | 010000000000000 |   2222 |
|   9999 |  0  | 010000000000000 |   2222 |
|   9999 |  0  | 000000000000000 |  12345 |
|   9999 |  0  | 100000000000000 |  12345 |
|   9999 |  0  | 000000000000001 |      0 |
|   9999 |  0  | 000000000000010 |      0 |
|   9999 |  0  | 000000000000100 |      0 |
|   9999 |  0  | 000000000001000 |      0 |
|   9999 |  0  | 000000000010000 |      0 |
|   9999 |  0  | 000000000100000 |      0 |
|   9999 |  0  | 000000001000000 |      0 |
|   9999 |  0  | 000000010000000 |      0 |
|   9999 |  0  | 000000100000000 |      0 |
|   9999 |  0  | 000001000000000 |      0 |
|   9999 |  0  | 000010000000000 |      0 |
|   9999 |  0  | 000100000000000 |      0 |
```

2. CPU Implementation - If the instruction coming is A instruction, then we directly store instruction value in A register and if it is a C instruction, we store the ALU output in A register. Now, based on the instruction we decide to send A or M into the ALU. As we have decided what the operands are, now we need to find what computation should be done and this done from instruction bits. Second, input is basically computed from the instruction itself, as it says whether to load ALU output to D register or not and this loaded value in D register is the second input to ALU. Now, the ALU computation part is done. Moving to the jump part, depending on the instruction provided we compute what jump is to be performed and depending on the jump we update the value of the PC. This completes the implementation of the CPU part. Output result after running one of the test scripts in the simulator is shown below.



```
|time| inM  |  instruction   |reset| outM  |writeM |addre| pc  |DRegiste|
|0+  |     0|0011000000111001|  0  |     0|   0   |    0|    0|     0  |
|1   |     0|0011000000111001|  0  |     0|   0   |12345|    1|     0  |
|1+  |     0|1110110000010000|  0  | 12345|   0   |12345|    1|  12345 |
|2   |     0|1110110000010000|  0  | 12345|   0   |12345|    2|  12345 |
|2+  |     0|0101101110100000|  0  | 12345|   0   |12345|    2|  12345 |
|3   |     0|0101101110100000|  0  | 12345|   0   |23456|    3|  12345 |
|3+  |     0|1110000111110000|  0  | 11111|   0   |23456|    3|  11111 |
|4   |     0|1110000111110000|  0  |     0|   0   |11111|    4|  11111 |
|4+  |     0|0000001111101011|  0  | 11111|   0   |11111|    4|  11111 |
|5   |     0|0000001111101011|  0  | 11111|   0   | 1003|    5|  11111 |
|5+  |     0|1110001100001000|  0  | 11111|   1   | 1003|    5|  11111 |
|6   |     0|1110001100001000|  0  | 11111|   1   | 1003|    6|  11111 |
|6+  |     0|0000001111101100|  0  | 11111|   0   | 1003|    6|  11111 |
|7   |     0|0000001111101100|  0  | 11111|   0   | 1004|    7|  11111 |
|7+  |     0|1110001110011000|  0  | 11110|   1   | 1004|    7|  11110 |
|8   |     0|1110001110011000|  0  | 11109|   1   | 1004|    8|  11110 |
|8+  |     0|0000001111101000|  0  | 11110|   0   | 1004|    8|  11110 |
|9   |     0|0000001111101000|  0  | 11110|   0   | 1000|    9|  11110 |
|9+  | 11111|1111010011110000|  0  |    -1|   0   | 1000|    9|     -1 |
|10  | 11111|1111010011110000|  0  |-11112|   0   |32767|   10|     -1 |
|10+ | 11111|0000000000001110|  0  |    -1|   0   |32767|   10|     -1 |
```

3. Computer Implementation - This a simple union of Memory, CPU and ROM. ROM is basically just a 32K registrars plain memory.Output result after running one of the test scripts in the simulator is shown below.



```
| time  |ARegister|DRegister|PC[]|RAM16K[0]|RAM16K[1]|RAM16K[2]|
| 0     |       0 |       0 |   0|       4 |       0 |       0 |
| 1     |       0 |       0 |   1|       4 |       0 |       0 |
| 2     |       0 |       4 |   2|       4 |       0 |       0 |
| 3     |      23 |       4 |   3|       4 |       0 |       0 |
| 4     |      23 |       4 |   4|       4 |       0 |       0 |
| 5     |      16 |       4 |   5|       4 |       0 |       0 |
| 6     |      16 |       4 |   6|       4 |       0 |       0 |
| 7     |   16384 |       4 |   7|       4 |       0 |       0 |
| 8     |   16384 |   16384 |   8|       4 |       0 |       0 |
| 9     |      17 |   16384 |   9|       4 |       0 |       0 |
| 10    |      17 |   16384 |  10|       4 |       0 |       0 |
| 11    |      17 |   16384 |  11|       4 |       0 |       0 |
| 12    |   16384 |   16384 |  12|       4 |       0 |       0 |
| 13    |   16384 |   16384 |  13|       4 |       0 |       0 |
| 14    |      17 |   16384 |  14|       4 |       0 |       0 |
| 15    |      17 |   16384 |  15|       4 |       0 |       0 |
| 16    |      32 |   16384 |  16|       4 |       0 |       0 |
| 17    |      32 |   16416 |  17|       4 |       0 |       0 |
| 18    |      17 |   16416 |  18|       4 |       0 |       0 |
| 19    |      17 |   16416 |  19|       4 |       0 |       0 |
```

PROJECT 6

This project involves construction of an Assembler to hack assembly language. This assembler functionality is to convert assembly language code to machine language code i.e. binary.

In this project, I designed an assembler in java language. This assembler mainly consists of 3 functions along with another supporting class. Initially, before designing these functions, we create hashmaps of all the possible computations, destinations and jumps in a C instruction along with default variables map mapping from their respective assembly code to binary code. Firstly, findSymbols() function basically goes through the whole code and creates a hashmap of all the references used in the code, references are nothing but symbols. These symbols are found by matching with the patterns in which they exist.

Now, with the use of this findSymbols() function, we develop the asmToHack() function that is used to convert that asm codes to binary codes. Firstly, we check if the code is an A instruction, if it is, then we check if it is a reference for upcoming jump and convert it to binary based on its corresponding value in its hashmap. If it is not for jump, then we check if it is just a value and convert it to binary. If none of them is matching, then it should be a user defined variable, then we check in all hashmaps if it's found we assign its corresponding binary code, if it's not then it is its first discovery, so we assign it a new address which is 16+No. Of newly discovered symbols. Now, if it's not an A instruction, we check if it's a C instruction, we compute the destination, computation, and jump code from the instruction and validate them if everything goes correctly we consider it as a C instruction and return it. If it does not match with any above, we rule it as the wrong instruction format.

Now, with the help of these two functions we design a converter() function that removes all the comments and spaces and calls the required functions to convert the provided asm file to hack file, which is in binary format. This concludes the design of assembler for hack assembly language.

GitHub Repository Link