

|  |
| --- |
| Nand2tetris  Report |
|  |
| May 27  Authored by: Samarjeet Sanjay Wankhade |

# 4. Machine Language

|  |
| --- |
| a. Mult: *“Find even more easy-to-use tools on the Insert tab, such as to add a hyperlink or insert a comment”* |
| We aim to Multiply two numbers stored in R[0] and R[1] and  store the answer in R[2]. |
| Concepts Used:  We can write multiplication in form of repeated sum  Let a=R[0] and b=R[1]  a\*b= a+a+a+a+a+……+a 🡪 b number of times  Methods used:  We start by initially putting a zero in the answer field R[2].  Then continuously add (a) in the answer field (b) number of times with help of a loop.  After each iteration, we will decrement the value of (b), store it back in R[1], and then check if the value of b aka (R[1]) is equal to zero or not.  If found equal it will exit the loop and end the program.  Observation:  I have noticed that generally, pc should have gone through all the values and iterated through all of the ROM.  But whenever I ran the programs, it stops at a random pc value. The PC value is always greater than the number of actual instructions but programs end at random. i.e., 62, 122 as shown in the following image |

B. Fill

|  |
| --- |
| We aim to fill the screen with black color if a keyboard key is pressed |

Concepts used:

I used I while which runs for infinity to check if a key is pressed or not, if it is pressed then we fill all the values in the screen as “1” or else “0” and then continue the loop.

Method:

🡪I also stored the values required in processing the programs such as screen location, iterations, and iterator location, Beforehand.

🡪I used a while loop that runs for infinity, it detects if any key is pressed or not and goes and executes the phase or state whichever is applicable, with the help of keyboard value stored at location 24576.

🡪It fills “0” or “1” whatever the case may be all one by one.

For example, if it detects that a key is not pressed then it will go to a loop and fill all the values dedicated to the screening as “1” I used a simple for loop for it as well.

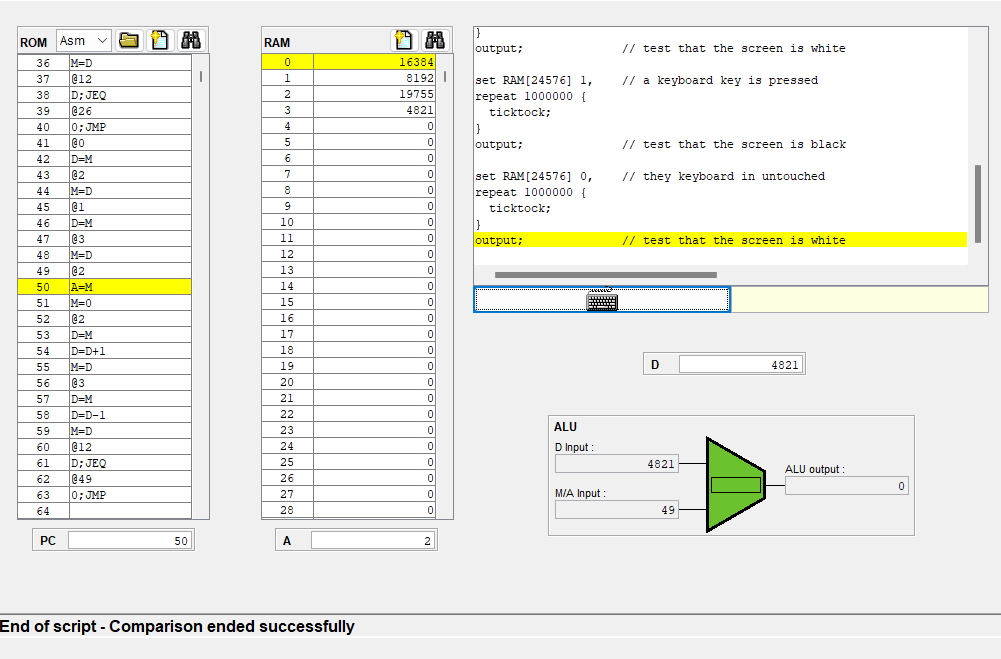
🡪I used a similar loop for the state where the key is pressed.

Observations:

Programs run but have some response delays. So basically, does not work smoothly.

Also, we needed a total of 8192 bits to be changed starting from 16384.

Keyboard inputs are stored at 24576.



# 5. Computer Architecture

A. Memory

|  |
| --- |
| Assemble another chipset to make Memory |

Concepts Used:

According to the Question, we can see that the last two bits of the address decide which component we want to access, i.e.(Memory, screen ) but for Keyboard, we also need to check the all-other bits.

Method used:

I used a Dmux based on the last two bits of address input to choose between screen and memory locations for inputting the change or whatever the input is (where the change is needed to be made). And then used a mux to select between screen output and memory output also based on the last two bits of addresses.

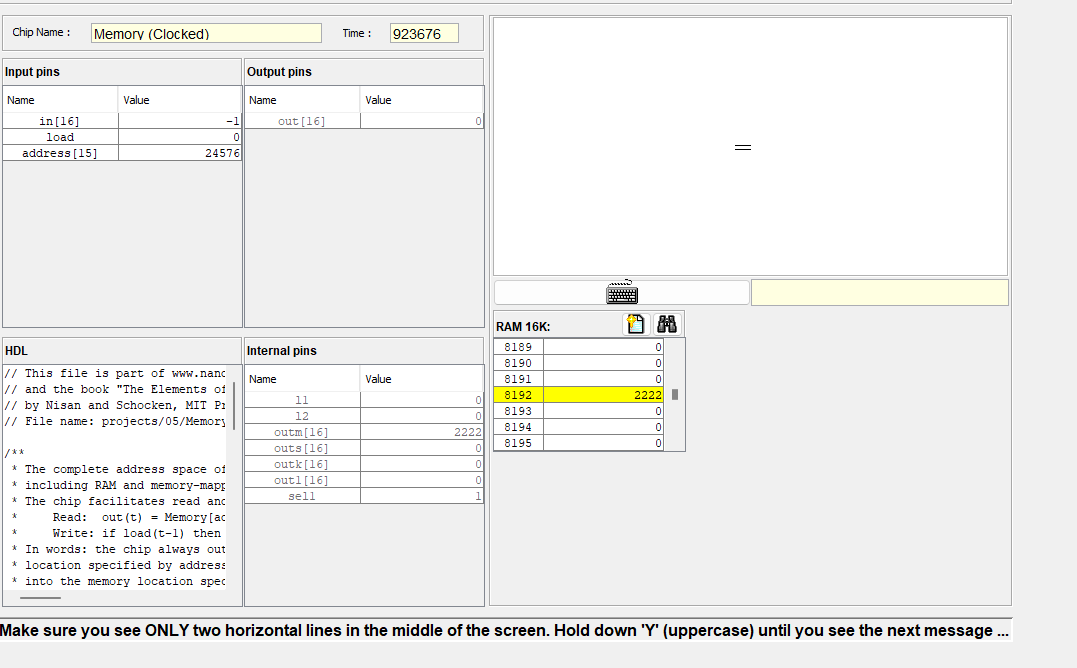
Finally, choose between the keyboard and the earlier selection(screen or memory whatever the case may be), Also based last two digits of the addresses field.

Observation:

Note: If the last two bits of the address are 00, 01 then it is memory access.

If 10 then it’s screen access.

If 11 then it is supposedly Keyboard access.



# B. CPU

|  |
| --- |
| Assemble another chipset to make a CPU |

Concepts Used: The diagram given in the reference material.

Instruction types whichever applicable, based on ixxxaccccccdddjjj

Format to decode the control signals.

Method used:

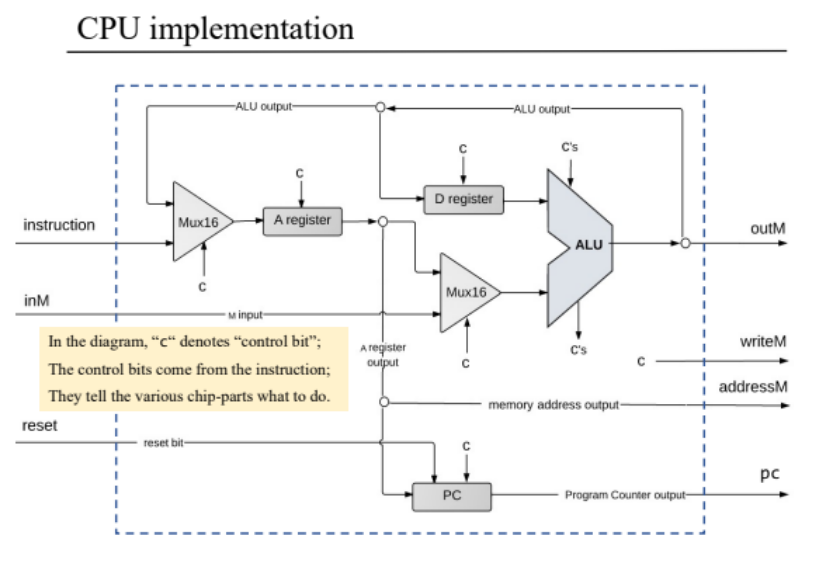
Observing and predicting the probable working of the system, I decoded the control bits.

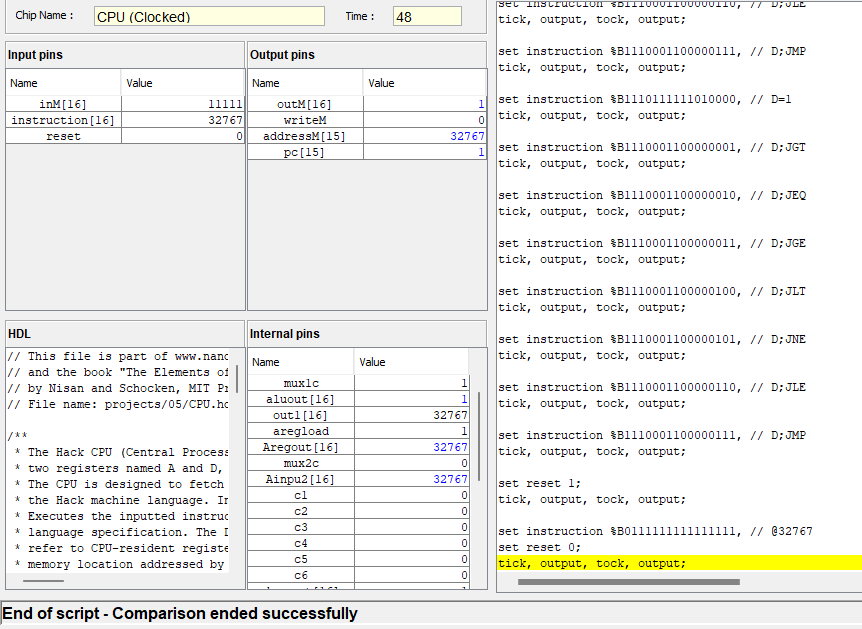
Making required logic and joining them.

Observations:

The last bit decides many things i.e.(mux1, mux2 selections, jumps, “Aregload”, and “Dregload”).

whether it is a jump or not, pc should always be incremented irrespective of the state.





c. Computer

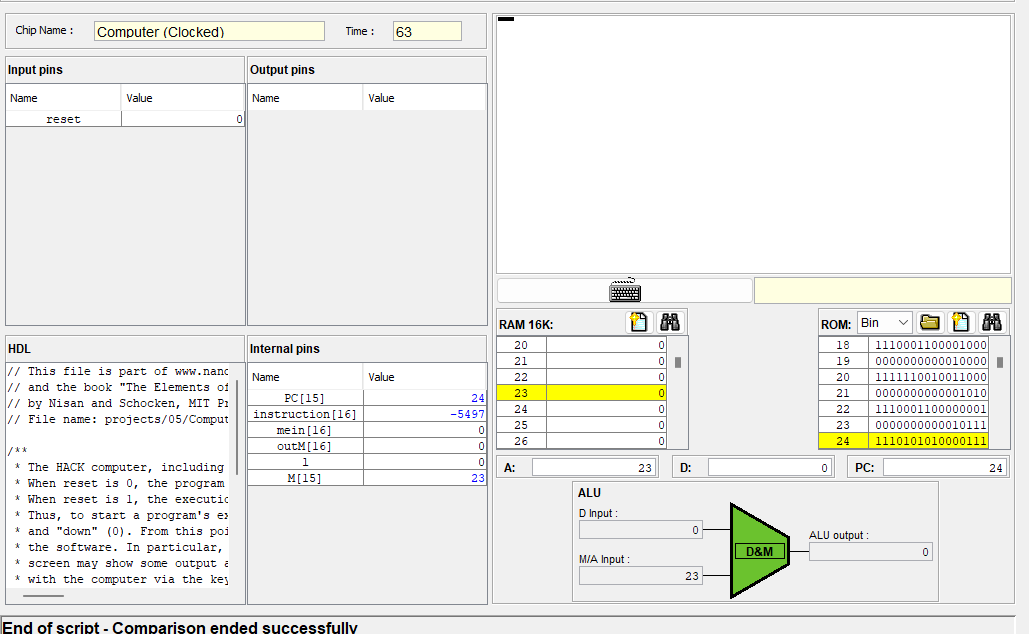
|  |
| --- |
| The aim is to join the chipset to make the computer |

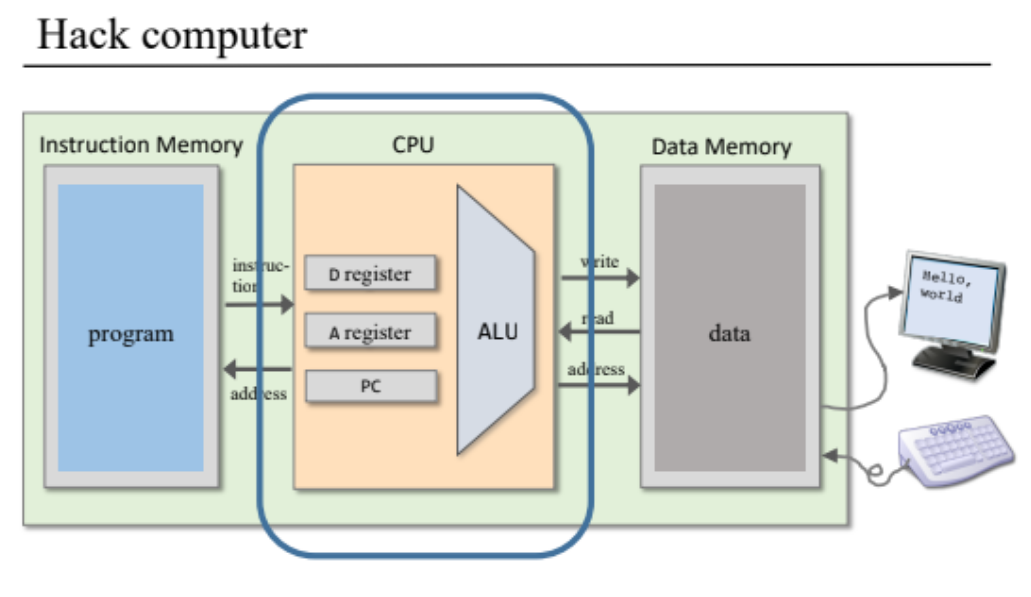
Concepts used: The diagram in the reference material

Methods Used:

Joining all the components

Observation: it is so cool





# 6. Assembling

|  |
| --- |
| The aim is to make an assembler |

Concepts used:

Hard coding the values.

Recognizing the comments.

Removal of extra spaces.

Recognizing the different types and formats of instruction.

Recognizing labels or variable names.

Method Used: For recognizing the comments I detected the presence of “//” and just neglected all of these.

I removed all the extra space using a while loop.

I also stored all the instructions of type ( x y z ) or variable name in a HashMap with a value based on which position the instruction occurs i.e., if the location is at the 10th instruction it will get a value of 10.

Labels without a location will get a value starting from 16

Converting all the instructions commented indented into proper plain instructions.

Also adding the values of format @R1--@R15.

Adding the values for THIS, THAT, ACG.

Observations:

THIS, THAT, ACG, etc. are custom instruction that refers to the value between 0-5

We should store the location of a label that has not occurred yet starting from 16.

