SY303 Lab 4 **Machine Language**

Name: McKenzie Eshleman

Partner(s): Jon Robinson (outside of class)

Section: 3321

Objective: In this lab, you will build on your knowledge of fundamental computer components to write low-level code which moves binary data between these components. These programs will be coded in the Hack Assembly language using a text-based code editor such as vim. To receive full credit, properly commented code will be required (Hack Assembly comments begin with "//").

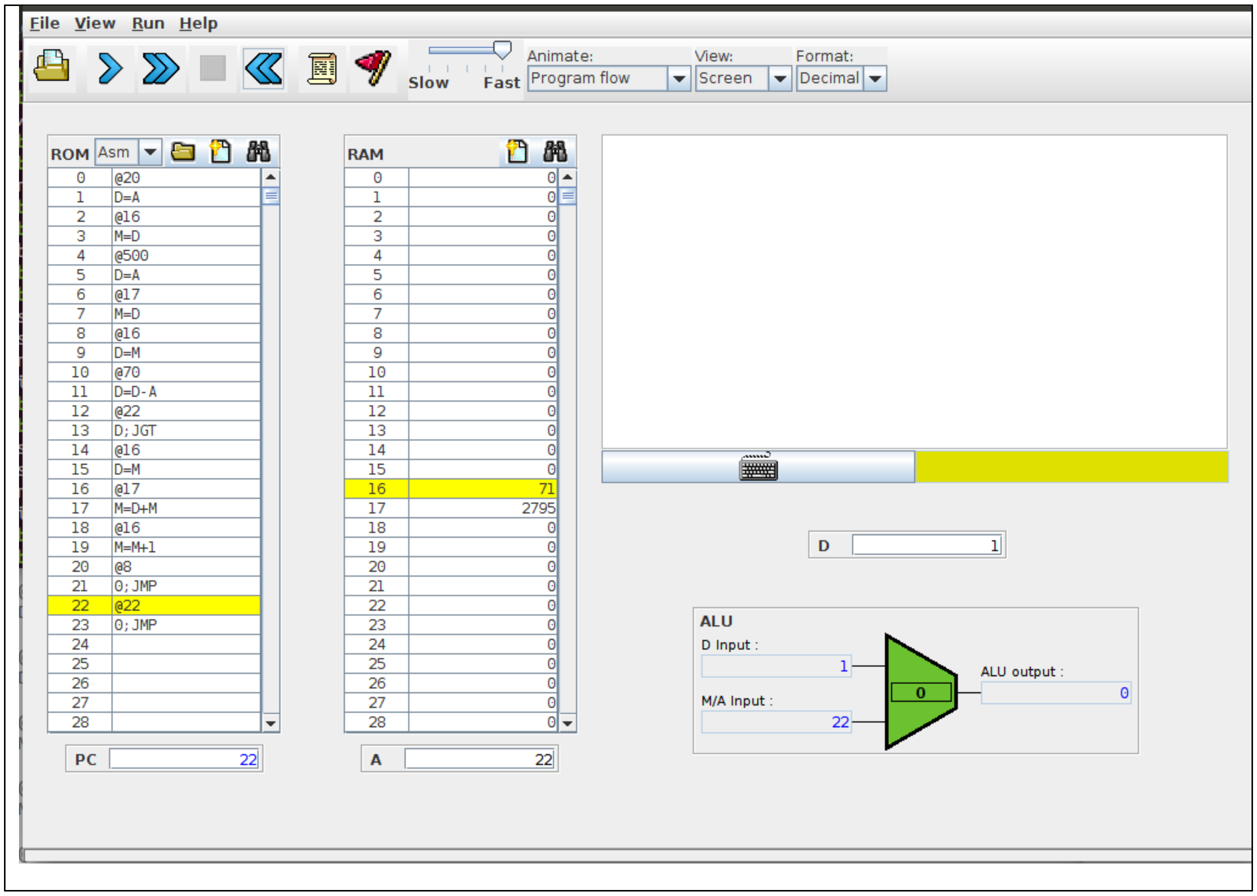
1) **Pre-Lab: Sum**
   a. Begin with the `Sum` assembly program on page 65 that your instructor demonstrated for you. Modify this code below to set the initial value of `sum` to 500 and then add the numbers from 20 to 70 (inclusive) instead. (10 pts)

```
//MIDN McKenzie Eshleman
//221938
//SY301 3321

//Adds the integers between 1 and 100


  @20    //load the address of the 20 in the A register
  D = A  // Sets D = to the value of 20

  @i
  M=D //M= RAM[20], giving the Ram address for the variable 20

  @500 //inputing the value of 500 into A
  D=A  //sets D to the value of 500 (which is the start variable for the sum)

  @sum   //load the SUM address into A
  M = 0  //set value of Sum into 0
 (LOOP)

  @i //loading the address of i into A
  D=M  //sets D = to the value of i

  @70 //load the constant value of 70 into A
  D=D-A //updates the value of D to be i-70

  @END //loads the end location from ROM into the A
  D; JGT //is i-70 > zero? if so then jump. only jumps when i =71
  @i  //loads the address of i into A
  D=M //makes D=i

  @sum //loads the address of sum into A
  M=D+M //updates the sum to be sum + i

  @i //reloads the add of i into A
  M=M+1 //increment 1

 @LOOP //load the ROM address of the start of the loop into A
 0;JMP //unconditional jump back to the start of the loop PC=LOOP

(END)

 @END //load the ROM address of this instruction into A
 0;JMP //unconditional jump back to this infinite loop
```

b. Explain the changes you made. (5 pts)
The changes we made was to first make a start variable of 20 and then Set D to the value of 20. Then we made D(which is 20) equal to the address of 20. Then we created the variable 500, and set D equal to 500. These steps all occurred before the Loop because we are setting initial values. Within the loop we then loaded 70 into the address of A, and then we updated the value of D to be i-70. This will make the code Jump Greater than because we have reach 71 and will stop the code.

c. Under the “Animate:” drop down box in the CPU Emulator, select “Program & Data Flow”. Step through the first few iterations of the loop until you develop an intuition for how each instruction causes some physical movement or change. Then run the program to the end. What was the final total in the `sum` RAM location? (3 pts) The final number that the Sum outputs is 2,795

d. Include a screenshot below of your modified `Sum` program after it produces the final product. (2 pts)



Animate: Program flow
View: Screen
Format: Decimal

ROM (Asm)

0	@20
1	D=A
2	@16
3	M=D
4	@500
5	D=A
6	@17
7	M=D
8	@16
9	D=M
10	@70
11	D=D-A
12	@22
13	D; JGT
14	@16
15	D=M
16	@17
17	M=D+M
18	@16
19	M=M+1
20	@8
21	0; JMP
22	@22
23	0; JMP
24	
25	
26	
27	
28	

PC 22

RAM

0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0
10	0
11	0
12	0
13	0
14	0
15	0
16	71
17	2795
18	0
19	0
20	0
21	0
22	0
23	0
24	0
25	0
26	0
27	0
28	0

A 22

D 1

D Input : 1
M/A Input : 22
ALU output : 0

2) **Hack Assembly:**

Constants: What type of Hack Assembly instruction could move the constant value 226 into a CPU Register in just one instruction? (2 pts) The A instruction is used to set a 15-bit value to a CPU register.

A-Instruction / C-Instruction

Show the instruction below which could accomplish this task**.** (3 pts)

@226

D = M //D = register of 226

Direct Addressing: The **A (**Address**) Register** often sets up an access (read or write) of data memory (**RAM**). This is known as Direct Addressing. We want to retrieve the value that is already stored in **RAM** location 66 (RAM[66]), and move it into the **D** (Data) **Register** of our CPU for an upcoming ALU operation. Fill in the sequence of instructions below to accomplish this task. (5 pts)

@_66___________

D = _M____________

Now use direct addressing again to add the value in the **D Register** from above into the *Virtual Register R12*, which is really just a symbolic shortcut to describe RAM location 12. (5 pts)

@ __12____

__D__ = D + __M_

Computation: You have a program that is implementing a FOR loop and must decrement the loop control variable, currently stored in *Virtual Register R8* (which is actually located in **RAM**), because this loop will not terminate until that value in *R8* reaches 0. Fill in the Hack Assembly instructions to decrement the *R8* value. (5 pts)

@ 8

M = M - 1

3) Program: Mult

Purpose: Describe the function and purpose of this program in your own words. What are the inputs and outputs? (5 pts)

This program stores two values in R0 and R1 which is the top two Ram locations. The program then computes the product of R0*R1 and stores the result in R2.

Recall that the only arithmetic instructions available through our ALU are Addition and Subtraction. Conceptually, how can you use one or both of these instructions in some sequence to perform multiplication? (3 pts)

Conceptually to perform multiplication with only using addition and subtraction, I would have the program use addition to add a number to itself, until the number of times the value reaches the other number we are trying to multiply it with. This would then give us the value of two products multiplied.

Initialization: It's important to realize that you are not choosing the numbers to be multiplied and you are <u>not responsible</u> for initializing Memory with their values. The test script will place values into Virtual Registers *R0* and *R1* before your program begins. Your program may manipulate these values as necessary to accomplish the implementation you described above. When your program completes, the correct product needs to be in *R2*.

HINT: <u>BEWARE of edge cases</u>! Make sure your plan still works if either input value is zero! You should be using the CPU emulator to check these values as your program runs.

We will build the solution for **Mult** logically, one step at a time.

Step 1) Initialize the **product** variable. From the <u>Contract</u> details in section 4.4 of the book, which virtual register (in **RAM**) will be the variable used to hold your final **product**? (1 pts)

Product Variable	RAM[2]

Now write Hack Assembly to initialize this variable to start at 0. (For good style and readability, you should indent lines of code which are not labels) (4 pts)

```
@R2
M=0 //sets the value of R2 to zero
```

Step 2) Set a label to mark the instruction at the beginning of your loop. You will return to this point via a jump command after every loop iteration. (3 pts)

```
(LOOP)
```

Step 3) The Contract describes two more virtual registers which are the operands for the addition problem. Choose one to be your **addend** and the other to be your **loop control** variable. (2 pts)

Addend	R1
Loop Control	R0

Now check the value of your **loop control** variable. If it is already down to zero, then have your program jump to a label which you will place at the end of the loop block. (3 pts)

```
@R0
D=M
@END
D;JEQ //if R0 is zero then jump to the end of the loop

@R1
D=M
@END
D;JEQ //if R1 is equal to zero then jump to end of loop
```

Step 4) Add the value in your **Addend** variable to the value in your **Product** variable. (5 pts)

```
@2
M=D+M //add Ram[1] to total, Ram[0] times
@counter
M=M+1 //incrementor for the counter
```

Step 5) Decrement your **loop control** variable. (2 pts)

```
@0
D=D-M //reduces the counter by R0
```

Step 6) Now, jump (unconditionally) back to the label at which your loop began. (5 pts)

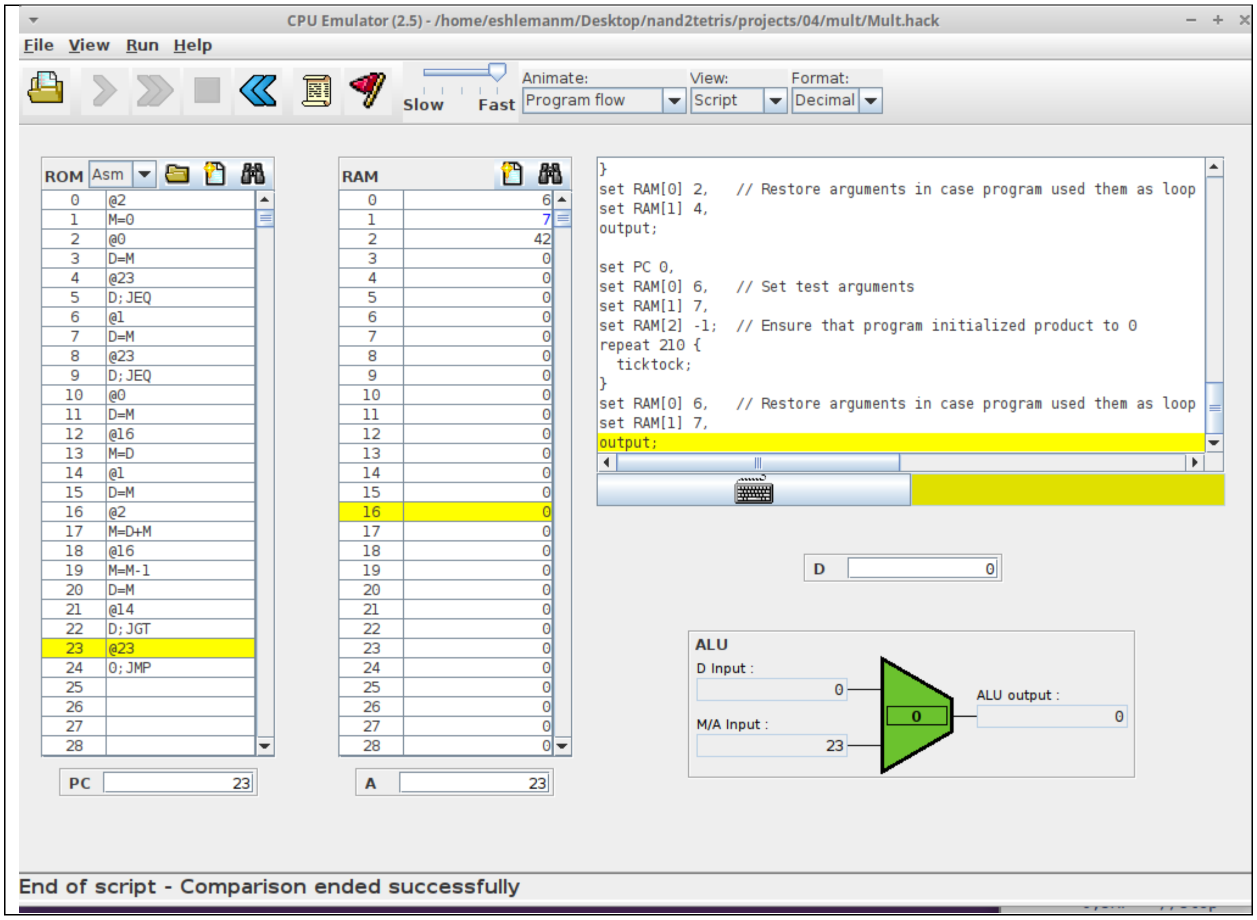
```
D=M    //sets R1 equal to M
@LOOP//Loop address
D;JGT   //Jumps back through the loop
```

Step 7) Finally, place a label here to represent the destination from your conditional jump above (the condition where the loop control variable was 0). (2 pts)

```
(END)
@END
0;JMP
```

Step 8) Combine all the code segments you created in each step above and place the full program in your **Mult.asm** file.

Step 9) Assemble the program with the Assembler tool, then test it with the CPU Emulator. Include a screenshot below of your solution completed successfully in the CPU Emulator. (5 pts)


CPU Emulator (2.5) - /home/eshlemanm/Desktop/nand2tetris/projects/04/mult/Mult.hack
File View Run Help
Slow Fast
ROM
RAM
D
ALU
D Input :
M/A Input :
ALU output :

Animate: Program flow View: Script Format: Decimal

ROM (Asm):

```
0  @2
1  M=0
2  @0
3  D=M
4  @23
5  D; JEQ
6  @1
7  D=M
8  @23
9  D; JEQ
10 @0
11 D=M
12 @16
13 M=D
14 @1
15 D=M
16 @2
17 M=D+M
18 @16
19 M=M-1
20 D=M
21 @14
22 D; JGT
23 @23
24 0; JMP
25
26
27
28
```

RAM:

0	6
1	7
2	42
3	0
4	0
5	0
6	0
7	0
8	0
9	0
10	0
11	0
12	0
13	0
14	0
15	0
16	0
17	0
18	0
19	0
20	0
21	0
22	0
23	0
24	0
25	0
26	0
27	0
28	0

PC 23 A 23

```
}
set RAM[0] 2,   // Restore arguments in case program used them as loop
set RAM[1] 4,
output;

set PC 0,
set RAM[0] 6,   // Set test arguments
set RAM[1] 7,
set RAM[2] -1;  // Ensure that program initialized product to 0
repeat 210 {
  ticktock;
}
set RAM[0] 6,   // Restore arguments in case program used them as loop
set RAM[1] 7,
output;
```

D 0

D Input : 0
M/A Input : 23
0
ALU output : 0

End of script - Comparison ended successfully

4) **In both Sum.asm and Mult.asm, you were required to create a loop in assembly syntax using labels and jump statements. Compare this method to the way you write loops in high level languages using the FOR and WHILE keywords. Can *any* FOR or WHILE loop be implemented using our assembly instructions instead?** (10 pts)

We first implemented a jump, which was initiated if our R0 or R1 values were zero. This is similar to using a while conditional loop, instead we set them to be jumps to a certain point in our program. We then used a jump statement to add R0 for an iteration of R1 times. This is similar to using a for loop in a high level language. Based off our code I believe that any for or while loop can be implemented using our assembly instructions instead.

5) **Both the Sum program and the Mult Program were easier to write because you were able to refer to commonly used data with variables (a type of symbolic notation). Section 4.2.4 covers the types of symbols available in the Hack Assembly language. In**

your own words describe the <u>importance of symbols</u>, the <u>ways they can be used</u>, and <u>how they simplify programming</u>, even in high level languages. (10 pts)

Symbols play a crucial role in the Hack assembly language because the symbols are predefined, this means that we can use the symbols as virtual registers or predefined pointers and we do not have to code them. We can use these symbols as registers for Ram[0..15], and for four predefined pointers that refer to the RAM addresses of 0 to 4. This helps simplify programming because we do not have to code these pointers and registers they are already done for us.