CSCE-312 | Project 4

Assembly Programming

Project Submission: 100 points

Grading

Project Submission [100%]:

You will be graded for correctness of the functions you have designed and coded. TA's will run tests on all the submitted assembly program files downloaded from your Canvas submission using Nand2tetris software (**CPU Emulator**). So, please make sure to test and verify your programs before finally submitting on Canvas.

Rubric:

You are provided with the *starter test collateral* for all programs which means you will need to extend the test collateral to fully verify the functionality of your programs: We will test your programs with our own test collateral. Partial credit will be provided for programs that do not pass the entire suite of tests.

Deliverables & Submission

You are required to turn in **the completed ASM files for all the functions implemented**. **You do NOT need to turn in your TST or CMP files**. Put your **full name** in the introductory comment present in each file. Use relevant code comments and indentation to give you the ability to debug and to give the TA's the tools to understand your code for any consideration of partial credits. Zip all the required files into a compressed file ***FirstName-LastName-UIN-P4.zip*** . **Submit this zip file on Canvas.**

Late Submission Policy: Refer to the Syllabus

Background

Every hardware platform is designed to execute commands in a certain machine language, expressed using agreed-upon binary codes. Writing programs directly in a binary 1, 0 sequence of code is possible, yet unnecessary and often error prone. Instead, we can write such programs using a low-level symbolic language called assembly and have them translated into binary code by a program called an assembler. In this project you will write some low-level assembly programs and will be forever thankful for high-level languages like C and Java. ***Assembly programming can be highly rewarding, allowing direct and complete control of the underlying machine; languages like C come close to being 'machine-aware' but are not as powerful in utilizing the capabilities of the underlying machine-like the assembly language.***

Main Objective

Our main objective is to get a taste of low-level programming in machine language, and to get acquainted with the Hack Computer platform. In the process of working on this project, you will also become peripherally familiar with the assembly process - translating from symbolic language to machine-language - and you will appreciate visually how native binary code executes on the target hardware platform. These lessons will be learned in the context of writing and testing four low-level programs, as follows.

Programs (Please read the specs very carefully; some examples are in the comments column to reinforce key points)

Name	Description	Comments / Tests
calc.asm (40 pts)	Write a program "***calc.asm***" to implement a **calculator application** in the HACK assembly language. Here are the specs: ● The values **a** and **b** on which the calc program will perform various operations are stored in RAM[0] (R0) and RAM[1] (R1), respectively. ● The different operations that are to be performed on the operands **a** and **b** are **Addition**, **Subtraction**, **Multiplication** and **Division** respectively. ● There is a third input **c** from the user that indicates the choice of the operation the user wishes to perform. The input **c** is stored in RAM[2] (R2), and can take the possible values of **1 (addition), 2 (subtraction), 3 (multiplication) & 4 (division)** for the respective operations. ● For Addition and Subtraction operations assume that the operands can be positive or negative. **For the Multiplication operation, at most one operand can be negative. For the Division operation both operands must be positive and greater than 0.** Your program must test for these exceptions to the input stipulations, and **exit with a -1 stored in RAM[1024]**	**Examples:** ● If you are given two numbers 14 and 4 as **a** (R0) and **b** (R1), respectively, with the operation choice input **c** (R2) as 4 indicating Division, the result will be 3 stored in R3 as the quotient and 2 stored in R4 as the remainder. ● If **a** (R0) is 15, **b** (R1) is 4, and **c** (R2) is 1 indicating Addition, then 19 will be stored in R3 as the result. ● If **a** (R0) is 15, **b** (R1) is 4, and **c** (R2) is 2 indicating Subtraction, then 11 will be stored in R3 as the result of subtracting **b** (R1) from **a** (R0). Always subtract **b** from **a** for a subtraction operation (i.e., **result** = **a - b**). ● If **a** (R0) is -15, **b** (R1) is 4, and **c** (R2) is 3 indicating Multiplication, then -60 will be stored in R3 as the multiplied result. *As a variation of this example, if **a** (R1) is -15, **b** (R1) is -4, and **c** (R2) is 3 indicating Multiplication, then the program discovers an exception to the input operand conditions and exits after storing -1 in RAM[1024].* ● Write your **calc.asm** program using the Hack Assembly Language. ● Next, load **calc.tst** script into the CPU Emulator (tools). This script loads the **calc.asm** program and executes it. Run the script. ● If you get any errors, debug and edit your **calc.asm** program. Then assemble the program, re-run the **calc.tst** script, etc.

	when any of these exceptions are discovered. ● **The final result of Addition, Subtraction, Multiplication are stored in RAM[3] (R3).** ● For the Division operation the program must preserve both the Quotient and the Remainder as follows: ○ **The Quotient must be stored in RAM[3] (R3) and the Remainder (if there is one) must be stored in RAM[4] (R4).**	You are given starter calc.tst and calc.cmp files. Complete these files with additional test cases (edge cases related to addition and subtraction of positive and negative numbers, small and large numbers on numerator and denominator, multiplication of positive numbers with negative numbers, exception conditions on input operands, etc.) to the best of your abilities and test your code against them.
palin.asm (30 pts)	Write a program "***palin.asm***" that determines if the given input **a** is a **palindrome**. The value of **a** is stored in RAM[0] (R0). The input **a** is a positive integer of 5 decimal digits (e.g., 41234). Assume this condition to always be true; the number of digits will not exceed that limit, nor will there be fewer digits, and the number is always positive. You must extract the decimal digits from the number first and then compare between them to figure out whether the number is palindrome or not. **After extracting the digits, you must store them in registers R6-R2 in a reverse order (see example on the right, and convince yourself of this by examining the palin.cmp file). Finally, after palindrome computation you will store 1 in RAM[1] (R1) if the number is palindrome else you will store 0 in R1.** **TIP**: You may need to use 10 for extracting the digits as shown in the suggested algorithm on the right. **Therefore, as a small aid, you will be provided with a helper number of 10 stored in R8 (you may verify this in the provided starter tst file).**	**Examples:** ● If you are given the value of input **a** as 12321 stored in RAM register R0, then your program will parse the value of **a** and end up storing the digits 1 in R2, 2 in R3, 3 in R4, 2 in R5, and 1 in R6 respectively. The program will then test for palindrome and store 1 in R1 to indicate ***successful*** detection of a Palindrome. ● If you are given the value of input **a** as 12345 stored in RAM register R0, then your program will parse the value of **a** and end up storing the digits 5 in R2, 4 in R3, 3 in R4, 2 in R5, and 1 in R6 respectively. The program will then test for palindrome and store 0 in R1 to indicate ***unsuccessful*** detection of a Palindrome. You are given starter palin.tst and palin.cmp files. Complete these files with additional test cases to the best of your abilities and test your code against them. **Follow similar instructions for the testing procedure as noted above in the case of calc exercise.** **TIP: Never** try to store a number greater than **32767** inside **R0** while debugging as that is not allowed due to the size of each register being 16 bits.

		Suggested Algorithm for extracting the digits of a number (Assume the number is **a**): 1. **c = a % 10** (% is the modulus operator) 2. **a = a – c** (c is a digit in **a**) 3. store **c** in the specified memory register. **4. a = a / 10 (/** is the division operator resulting in quotient being assigned to **a)** 5. Loop back and repeat steps 1-3 if a >0 else stop if a == 0. The array of registers where you have stored the value of **c** in every iteration of the loop will now contain the digits of the number. Try the above algorithm by hand to verify it!
gcd.asm (30 pts)	Implement a program ***"gcd.asm"*** that calculates the **greatest common divisor** (gcd) of two given integers input numbers, which are stored in RAM[0] (R0) and RAM[1] (R1). Assume both numbers are positive and greater than 0. The gcd is stored in RAM[2] (R2).	Use the Euclidean algorithm here. Here is a link showing you how Euclid's algorithm works to find the **gcd** of two numbers: https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/the-euclidean-algorithm Here again, you are given starter gcd.tst and gcd.cmp files. Complete these files with additional test cases to the best of your abilities and test your code against them. **Follow similar instructions for the testing procedure as noted above in the case of calc.**

Contract

Write and test the programs described above. When executed on the supplied CPU emulator, your programs must generate the results mandated by the specified tests.

IMPORTANT: While it is impractical to achieve optimal implementation (as measured by the # of lines executed) of any reasonable complexity ASM code, it is nonetheless critical to aim for efficiency. We have provided reasonable runtime bounds for testing. These bounds are implemented in the form of **'repeat'** statements followed by a *number* in the tst files. The starter tst files provided with this project give you a flavor of the size of these repeats, but they are not an absolute prescription by any means. As you explore this further, you will come to realize that runtime efficiency of your program is largely (BUT NOT ENTIRELY) determined by the size of the 'loop' structures in your programs. Therefore, it is imperative that you pay close attention to writing efficient loops (as measured by the number of lines of asm code in the loop structures).

The grading for each program will be based solely on the correctness of your code

Resources

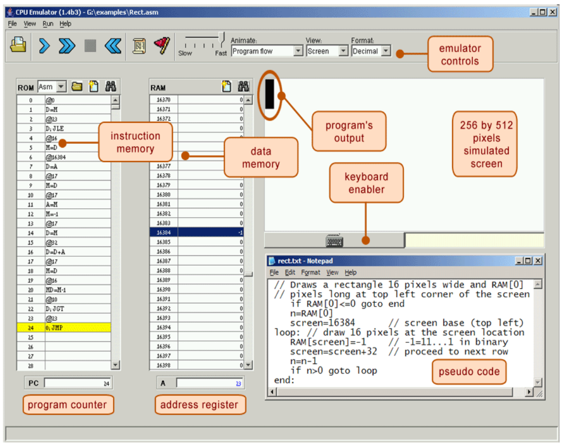
The Hack assembly language is described in detail in **Chapter 4 .**

You will need the following tool: **CPU emulator** - a program that runs binary (hack) or symbolic (asm) Hack code on an emulated Hack platform. **TUTORIAL:** PDF

> ***Debugging tip:*** *The Hack assembly language is case-sensitive. A common error occurs when one writes, say, "@foo" and "@Foo" in different parts of one's program, thinking that both labels are treated as the same symbol. In fact, the assembler treats them as two different symbols. This bug is difficult to detect, so you should be aware of it.*
>
> ***Further tips on using the CPU Emulator and debugging common errors are provided in the Project 4 FAQ document posted in the Project 4 repository.***

Tools

The CPU Emulator tool is used to run and test the assembly programs.



```
0   @0
1   D=M
2   @23
3   D;JLE
4   @16
5   M=D
6   @16384
7   D=A
8   @17
9   M=D
10  @17
11  A=M
12  M=-1
13  @17
14  D=M
15  @32
16  D=D+A
17  @17
18  M=D
19  @16
20  MD=M-1
21  @10
22  D;JGT
23  @23
24  0;JMP
```

```
// Draws a rectangle 16 pixels wide and RAM[0]
// pixels long at top left corner of the screen
   if RAM[0]<=0 goto end
   n=RAM[0]
   screen=16384      // screen base (top left)
loop: // draw 16 pixels at the screen location
   RAM[screen]=-1    // -1=11...1 in binary
   screen=screen+32  // proceed to next row
   n=n-1
   if n>0 goto loop
end:
```

The supplied CPU Emulator includes a ROM (also called Instruction Memory) representation, into which the binary code is loaded, and a RAM representation, which holds data. For ease of use, the emulator enables the user to view the loaded ROM-resident code in either binary mode, or in symbolic / assembly mode. In this project we are recommending you load the symbolic/assembly code directly into the CPU Emulator, in which case the emulator translates the loaded code into binary code on the fly. However one can take the indirect path of first converting the .asm (symbolic) into the .hack (binary) file using the Assembler, and then loading the .hack file into the CPU Emulator. The benefit of using the latter approach is instructive. First, the supplied assembler shows the translation process visually, for instructive purposes. Second, the assembler generates a persistent binary file. This file can be executed either on the CPU emulator or directly on the hardware platform, as we'll do in our next project!