CSCE-312 | Fall 2019

Project 4

Assembly Programming

Due Date: Submit on eCampus by **Friday, Nov 8th, 11:59 PM**

Grading:

(A) Project Demo [70%]: + Bonus Question (published separately)
You will be graded for correctness of the programs (asm) you have coded. We will be running test of all your asm codes using Nand2tetris software (Assembler and CPU Emulator).

(B) Project Quiz [30%]: Take Home, Will be announced on Saturday Nov 9th Morning
(Submit on eCampus by: Saturday, Nov 9th, 11:59 PM)

Deliverables & Submission

You need to turn in ONLY the asm files (div.asm, fill.asm, lcd.asm, mod.asm) for all the programs.

We will test with our own test and compare files

Put your **full name** in the introductory comment present in each ASM code.

Use relevant code comments and indentation in your code.

Zip all the required assembly (asm) files into a compressed file ***FirstName-LastName-UIN.zip***

Submit this zip file on eCampus.

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 binary 1, 0 sequence of code is a 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. ***(Actually, assembly programming can be highly rewarding, allowing direct and complete control of the underlying machine.)***

Objective

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 become 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 three low-level programs, as follows.

Programs

	Description	**Comments / Tests**
div.asm	Write a program to calculate the quotient from a division operation. The values of dividend **a** and divisor **b** are stored in RAM[0] (R0) and RAM[1] (R1), respectively. The dividend **a** is a non-negative integer, and the divisor **b** is a positive integer. Store the quotient in RAM[2] (R2). Ignore the remainder.	**Example:** if you are given two numbers 15 and 4 as dividend a (R0) and divisor b (R1), then the answer will be 3 stored in R2. ● Write your **div.asm** program using the Hack assembly language. ● Use the supplied Hack Assembler (tools) to translate your **div.asm** program, producing a **div.hack** file containing binary Hack instructions. ● Next, load the supplied **div.tst** script into the CPU Emulator (tools). This script loads the **div.hack** program, and executes it. Run the script. ● If you get any errors, debug and edit your **div.asm** program. Then assemble the program, re-run the **div.tst** script, etc. Note that the **div.tst** file is nearly complete to establish strong familiarity with testing methods, but it *may* be missing 'corner conditions'. Assure yourself of such cases and add as required.

mod.asm	Implement a program that calculates the modulo of two given numbers a and b, which is a%b in math. The value of a is stored in RAM[0] (R0), and the value of b is stored in RAM[1] (R1). The value a is non-negative integer and b is positive integer. The modulo value is stored in RAM[2] (R2).	**Example:** If you are given two numbers 18 and 4 stored in RAM registers, R0 and R1, then the (18 modulo 4) will be 2 stored in R2. Here, you are given starter tst and 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 div.
lcd.asm	Implement a program that calculates the **largest common divisor** (lcd) of two given non-negative integers, which are stored in RAM[0] (R0) and RAM[1] (R1). The lcd is stored in RAM[2] (R2).	Use Euclidean algorithm here. Here is a link showing you how Euclidean's algorithm works to find lcd of two numbers: https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/the-euclidean-algorithm Here, you are given starter tst and 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 div.
fill.asm	**I/O handling**: this program illustrates low-level handling of the screen and keyboard devices, as follows: The program runs an infinite loop that listens to the keyboard input. When a key is pressed (any key), the program darkens the screen, i.e. writes "1" in every bit for each pixel; the screen should remain fully dark as long as the key is pressed. When no key is pressed, the program clears the screen, i.e. writes "0" in every pixel; the screen should remain fully bright as long as no key is pressed.	Start by using the supplied assembler to translate your **fill.asm** program into a **fill.hack** file. **Implementation note:** Your program may darken and brighten the screen's pixels in any spatial/visual order, as long as pressing a key continuously for long enough results in a fully dark screen, and not pressing any key for long enough results in a fully bright screen. The simple **fill.tst** script, which comes with no compare file, is designed to do two things: (i) load the **fill.hack** program, and (ii) remind you to select 'no animation', and then test the program interactively by pressing and releasing some keyboard keys. The **fillAutomatic.tst** script, along with the compare file **fillAutomatic.cmp**, are designed to test the fill program automatically, as described by the test script documentation. For completeness of testing, it is recommended to test the fill program both interactively and automatically.

Contract

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

Resources

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

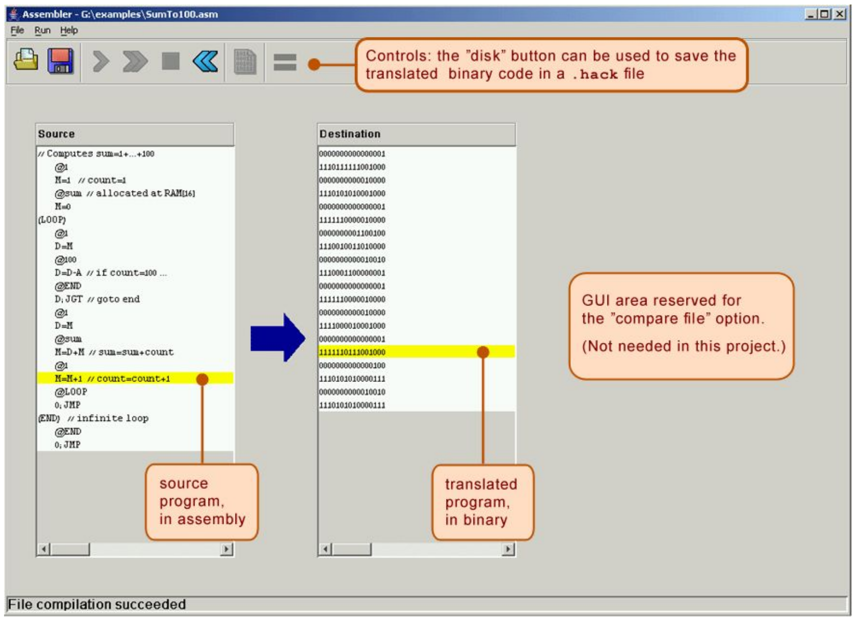
You will need two tools:

1. The supplied assembler - a program that translates programs written in the Hack assembly language into binary Hack code. **TUTORIAL:** PDF
2. The supplied CPU emulator - a program that runs binary Hack code on a simulated Hack platform. **TUTORIAL:** PDF

   Debugging tip: *The Hack 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.*

Tools

The supplied Hack Assembler can be used in either command mode (from the command shell), or interactively. The latter mode of operation allows observing the translation process in a visual and step-wise fashion, as shown below:

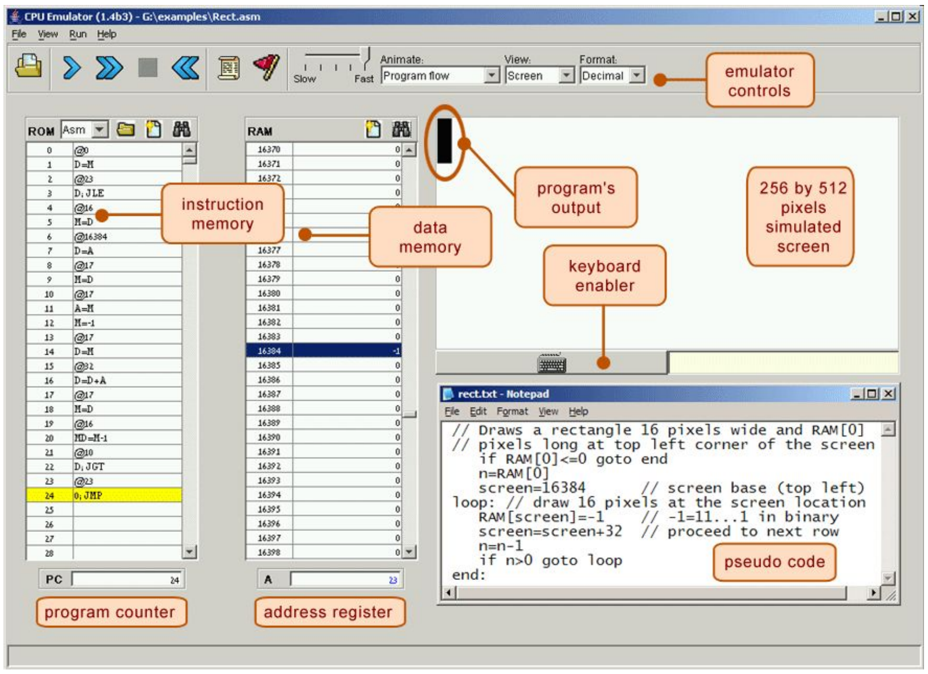

Assembler - G:\examples\SumTo100.asm
File Run Help
Controls: the "disk" button can be used to save the translated binary code in a .hack file
Source
Destination
GUI area reserved for the "compare file" option.
(Not needed in this project.)
source program, in assembly
translated program, in binary

```
// Computes sum=1+...+100
    @1
    M=1  // count=1
    @sum // allocated at RAM[16]
    M=0
(LOOP)
    @1
    D=M
    @100
    D=D-A // if count=100 ...
    @END
    D;JGT // goto end
    @1
    D=M
    @sum
    M=D+M // sum=sum+count
    @1
    M=M+1 // count=count+1
    @LOOP
    0;JMP
(END)  // infinite loop
    @END
    0;JMP
```

```
0000000000000001
1110111111001000
0000000000010000
1110101010001000
0000000000000001
1111110000010000
0000000001100100
1110010011010000
0000000000010010
1110001100000001
0000000000000001
1111110000010000
0000000000010000
1111000010001000
0000000000000001
1111110111001000
0000000000000100
1110101010000111
0000000000010010
1110101010000111
```

File compilation succeeded

The machine language programs produced by the assembler can be tested in two different ways. First, one can run the resulting .hack program in the supplied CPU emulator. Alternatively, one can run the same program directly on the Hack hardware, using the supplied hardware simulator used in projects 1-3. To do so, one can load the Computer.hdl chip (built in project 5) into the hardware simulator, and then proceed to load the binary code (from the .hack file) into the computer's Instruction Memory (also called ROM). Since we will only complete building the hardware platform and the Computer.hdl chip only in the next project, at this stage we recommend testing machine-level programs using the supplied CPU emulator.



Animate: Program flow
View: Screen
Format: Decimal

ROM Asm

```
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
25
26
27
28
```

PC 24

RAM

```
16370   0
16371   0
16372   0
        0
16377
16378
16379   0
16380   0
16381   0
16382   0
16383   0
16384  -1
16385   0
16386   0
16387   0
16388   0
16389   0
16390   0
16391   0
16392   0
16393   0
16394   0
16395   0
16396   0
16397   0
16398   0
```

A 23

```
// 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 fact, the CPU emulator even allows loading symbolic code written in assembly directly into the ROM, in which case the emulator translates the loaded code into binary code on the fly. This utility seems to render the supplied assembler unnecessary, but this is not the case. 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, as we illustrate below, or directly on the hardware platform, as we'll do in the next project.