CSCE-312 (504-505) | Sprint 2019

Project 4

Assembly Programming

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

Grading:

(A) Project Demo [50%]: Offline
You will be graded for correctness of the programs (asm) you have coded. We will be running offline test of all your asm codes using Nand2tetris software (Assembler and CPU Emulator). The same simulator you will be using to check your chips in the course of this project. So, make sure to test and verify your codes before finally submitting on eCampus.

***Rubric**:* The rubric allocation for chips is shown at the end of this document. Each chip needs to pass all its test cases to get the points, else you will receive 0 on that chip.

(B) Code Review/Q&A [50%]: To be held with the LIVE Demo
Code review of randomly selected chips. The questions can involve drawing circuit diagram of randomly selected chips. Should not be difficult for you if you have understood the core inner workings of your project.

Deliverables & Submission: Turn in a zip file in the format *FirstName-LastName-UIN.zip* containing the 5 ASM files(div.asm, Fill.asm and Mult.asm, lcd.asm, mod.asm), no need to submit the hack files. Also, include this cover sheet with your signature below. Put your full name in the introductory comments present in each asm file. Use relevant code comments and indentation in your code. Submit this zip file on eCampus.

Late Submission Policy: Refer to the Syllabus

First Name: Last Name: UIN:

Any assignment turned in without a fully completed cover page will NOT BE GRADED.
Please list all below all sources (people, books, web pages, etc) consulted regarding this assignment:

CSCE 312 Students	Other People	Printed Material	Web Material (URL)	Other
1.	1.	1.	1.	1.
2.	2.	2.	2.	2.
3.	3.	3.	3.	3.

Please consult the Aggie Honor System Office for additional information regarding academic misconduct – it is your responsibility to understand what constitutes academic misconduct and to ensure that you do not commit it.

I certify that I have listed above all the sources that I consulted regarding this assignment, and that I have not received nor given any assistance that is contrary to the letter or the spirit of the collaboration guidelines for this assignment.

Submission Date: ____________________
Printed Name (in lieu of a signature): __________________________

<u>Background</u>

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 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.)***

<u>Objective</u>

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.

<u>Programs</u>

Before you start implementing these programs, make sure you have replaced the <u>Compilers.jar</u> and <u>Hack.jar</u> files in ../nand2tetris/tools/bin/lib with the new ones provided in this release.

Program	**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[R0] and RAM[R1], respectively. The dividend a is a non-negative integer, and the divisor b is a positive integer. Store the quotient in RAM[R2]. Ignore the remainder.	Example: if you are given two numbers 15 and 4 as dividend a and divisor b, then the answer will be 3 stored in RAM[R2].
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),	Start by using the supplied assembler to translate your Fill.asm program into a Fill.hack file. Implementation note: your program may blacken and clear the screen's pixels in any spatial/visual order, as long as pressing a key continuously for long enough results in a fully blackened screen, and not pressing any key for long enough results in a fully cleared screen.

	the program blackens the screen, i.e. writes "black" in every pixel; the screen should remain fully black as long as the key is pressed. When no key is pressed, the program clears the screen, i.e. writes "white" in every pixel; the screen should remain fully clear as long as no key is pressed.	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.
mult.asm	Implement a program to multiply two numbers R0 and R1, and put the result in R2, with bit shift instructions. **Operator**: '<' is left shift and '>' is right shift. **Hint:** A pseudo code is shared later in this document.	**This multiplication must be implemented using bit shifting only. Use of any other method will not receive full credit.** Use the supplied Mult.tst script and Mult.cmp compare file (that's how we test your program). These supplied files are designed to test your program by running it on several representative data values. **Usage of < and > operators:** **1. shift m to left by n bits** $m < n$ **2. shift m to right by n bits** $m > n$ **Notice, the *m* and *n* here can only be 1 or values loaded from register. In your Mult.asm program, you may want to do use it like:** **@tempReg1** **D=M;** **@tempReg2** **M = M<D** **----------------------** **@tempReg2** **D=M** **@tempReg1** **M=1<D** **(I provided an example test.asm program showing you how this works here, please use the CPUEmulator to run it step by step to understand it). The program test.asm will shift a number in register R[0] to left by the number in register R[1] and then shift to right 1 bit each**

		time until it becomes 0 and R[2] stores the final result 0.
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[R0], and the value of b is stored in RAM[R1]. Value a is non-negative integer and b is positive integer. The modulo value is stored in RAM[R2].	Use the algorithm of your choice. The easiest way to do it is to do subtraction continuously until you find the result of subtracting b from a is smaller than b, then that's the result your program should give as output to RAM[R2].
lcd.asm	Implement a program that calculates the largest common divisor (lcd) of two given non-negative integers, which are stored in RAM[R0] and RAM[R1]. The lcd is stored in RAM[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

Contract

Write and test the five 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 (Use TAMU mail id to access).

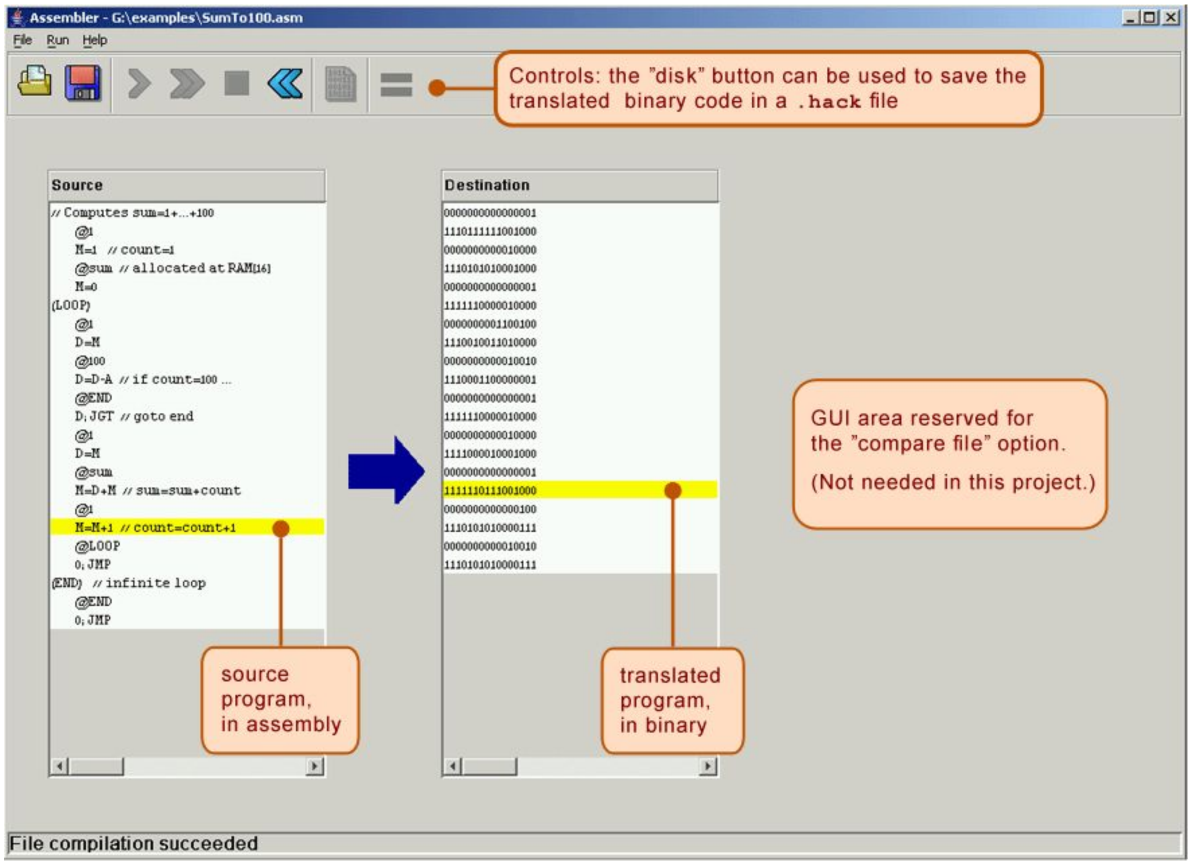
You will need two tools: the supplied assembler - a program that translates programs written in the Hack assembly language into binary Hack code, and the supplied CPU emulator - a program that runs binary Hack code on a simulated Hack platform. **As mentioned above, make sure you have replaced the Compilers.jar and Hack.jar files in ../nand2tetris/tool/bin/lib with the new ones.**

Two other related and useful resources are the supplied Assembler Tutorial and CPU Emulator Tutorial. We recommend going through these tutorials before starting to work on this project.

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:

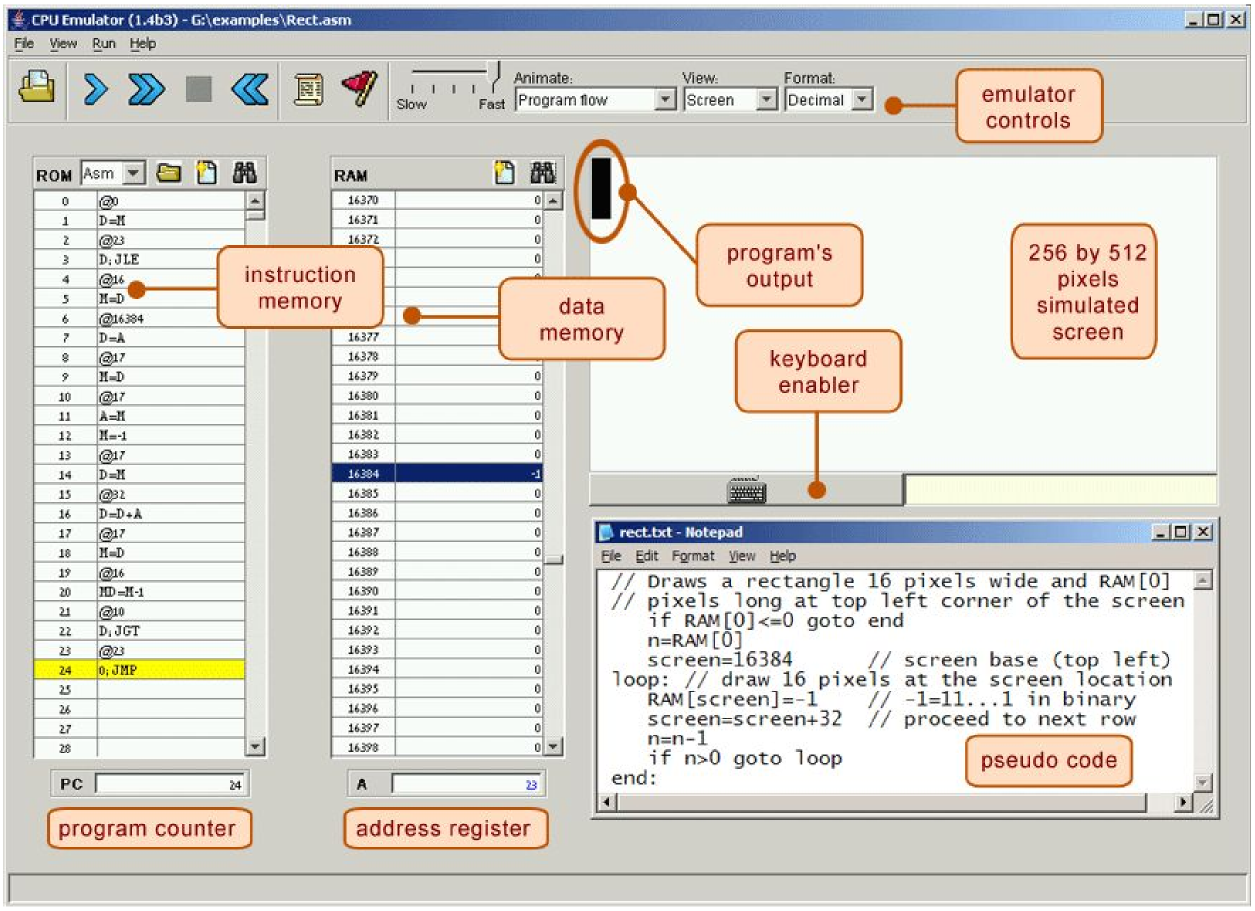

Assembler - G:\examples\SumTo100.asm
File Run Help
Controls: the "disk" button can be used to save the translated binary code in a .hack file
Source
Destination
GUI area reserved for the "compare file" option.
(Not needed in this project.)
source program, in assembly
translated program, in binary

```
// 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.



ROM:

```
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
```

RAM:

```
16370   0
16371   0
16372   0
16373   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
```

```
// 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.

High-level pseudocode for Mult.asm

```
Function Mult(R0,R1){
int result, counter;
//You can create and access symbols with “@<symbol name>” command
counter=<some number>;
// Think about what is the correct number
result=0;
while(counter >=0 ){
      int16 temp1 = 1<<counter; //1 shifts left this many bits; here << is used
      as a traditional symbol of left-shift in high-level languages
      if(R1 & temp1 > 0 ) {
                  //Think about what does the above expression do?
                  result+=R0<<counter;
            }
            counter--;
}
return result; //Put this in R2
}
for(;;) //Infinite loop
```