CSCE-312 | Lab Exercise 4

Basic HACK Programs

Submission

Grading

The grade for this assignment is 100% based on the correctness of the assembly code you submit. The grader will use the CPU Emulator tool, along with the given test scripts, to verify the correctness of your code. Since test scripts are included for all programs, **no partial credit will be given for programs that are not *fully* functional:** We want you to get all tests passing to realize complete understanding of the implemented functions. if a program passes only part of its test cases, you will receive 0 points for that program.

Please refer to the Syllabus for the late submission policy.

Deliverables

You need to turn in:

- Completed ASM files for all implemented programs.
    - Put your full name and UIN in the header comment present in each ASM file
    - You do not need to submit the .tst or .cmp files

Zip all the required HDL files into a compressed file FirstName-LastName-UIN-LE4.zip and submit this zip file on CANVAS. If you do not follow these steps properly, you may receive a 0.

Background

This exercise serves as an introduction to HACK assembly programming. Hack is a simplified assembly language that is relatively easy to pick up; however, it can still be quite a puzzle to write assembly programs. Starting with these small programs will hopefully help get you used to the way assembly code works. See the last section below for some helpful notes on Hack programming.

Objective

Implement the following assembly programs. They are listed in recommended order (increasing complexity), though you can approach them in any order you choose. Note that our test scripts have limits on the number of instructions they will execute for each case; if your program does not produce the desired output within the allotted limit, it will fail the test case. We will grade your submission using the same time limits, so we recommend that you not change them for your local testing. With that said, given the simplicity of these tasks, you should have no trouble staying within the set number of instructions.

Program [points]	**Description**
add.asm [25]	This program should simply add two numbers (stored in RAM[0] and RAM[1]) and store the result in RAM[2]. Note that you may not overwrite the values in RAM[0] or RAM[1].
complement.asm [25]	This program should check whether two numbers (stored in RAM[0] and RAM[1]) are *complements*, i.e., every bit that is *on* in one is *off* in the other (a = !b, bitwise NOT). Note that you may not overwrite the values in RAM[0] or RAM[1].
addn.asm [50]	This program adds together a specified amount of numbers and stores the value in RAM[0]. The amount of numbers, *n (<16)*, is given in the initial value of RAM[0]; the numbers themselves are stored starting in RAM[1] and going up in consecutive addresses up to RAM[n]. You may overwrite the data in the RAM as much as necessary, as long as the correct value ends up in RAM[0] when your program is done. *Please note that for this problem, the value of n is less than 16.*

Program Details

Add

This program does exactly what it says – add two numbers together. It's a basic intro to Hack assembly and doesn't require any sort of control flow (i.e., jumps). It's just to get familiar with the basics of register use and arithmetic.

Complement

This program requires a small amount of control flow, in the form of an if-statement: if the two inputs are complements, then the output (`RAM[2]`) should be set to 1; otherwise, it should be set to 0. Note that Hack doesn't explicitly support advanced control flow features like if-statements; instead, you'll have to implement this behavior using comparison-based jumps.

As mentioned in the starter ASM file, we're checking for binary complements: whether the active bits in the two numbers are the opposite. There are a couple of ways to state this more precisely:

- (`a ^ b = 0b111…`), where `^` is bitwise XOR
- (`a & b = 0` and `a | b = 0b111…`), where "`&`" is bitwise AND and "`|`" is bitwise OR
- (`a + b = 0b111…`), where "+" is just normal addition

Note that `0b111…` in 2's complement is -1.

Add-N

This program requires some more advanced control flow. In order to add a runtime-specified (i.e., given as a parameter to the program, not a fixed constant) count of numbers, you'll need to implement a loop that runs a variable number of times. You can do this however you'd like; I'd recommend you start by imagining how to write this program in a very basic high-level pseudocode, then figure out how to turn that into assembly. Remember that Hack supports `@variables`; they'll likely be very helpful for this problem.

The goal of the program is to add numbers in an "array" (i.e., consecutive locations in memory) starting at RAM index 1. The amount of numbers is specified in the initial value of `RAM[0]`. This value should be overwritten by the program to hold the sum of the numbers. You may overwrite the rest of RAM however you want; we will only check the value `RAM[0]` when evaluating your program's correctness.

Helpful Hack Assembly Notes

We're assuming you've looked at the many resources we've provided (e.g., the Project 4 FAQ), but we want to remind you of a few *motifs* (helpful patterns) that show up in many Hack assembly programs.

- `@label -> JMP`
    - All jump instructions set the program counter to whatever value is stored in the `A` register; in other words, they jump to the instruction at that offset. Thus, prior to a jump, you'll pretty much always want to load a specific program point into the `A` register; this is accomplished with the `@label` notation, where a `(label)` of the same name should show up elsewhere in the code. This is the point where the jump will lead to.
- `@address -> D=M -> @something` (or `@address -> M=D -> @something`)
    - The Hack platform is severely limited in the number of registers: we only have `A` and `D` (and `M`, which is dependent on `A`). As such, you'll have to load/store your "working data" fairly often.
- `(END) -> @END -> 0;JMP`
    - This is an infinite loop; it often shows up at the end of Hack programs to ensure the computer doesn't keep incrementing the program counter indefinitely after our program completes its normal operations.

Also remember that you can declare and use variables using `@variablename` notation; if a `@something` statement doesn't have a corresponding `(something)` label, then it will create a variable (i.e., allocate a unique address in memory). These are very helpful when dealing with internal program state, e.g., a loop index.