Project 4

In this project you will write and execute some low-level machine language programs. In particular, you will write programs in the Hack *assembly language*, use an *assembler* to translate them into binary code, and test the resulting code. Since the Hack computer will be built only in the next project, in this project you will run the programs on a *CPU Emulator*, designed to execute binary code written in the Hack instruction set.

Objectives

- Get a hands-on taste of low-level programming in machine language;
- Get acquainted with the Hack instruction set, before building the Hack computer in project 5;
- Get acquainted with the assembly process, before building an assembler in project 6.

Tasks

Write and test two programs:

Mult.asm (example of an arithmetic task): The inputs of this program are the values stored in R0 and R1 (RAM[0] and RAM[1]). The program computes the product R0 * R1 and stores the result in R2 (RAM[2]). Assume that R0 ≥ 0, R1 ≥ 0, and R0 * R1 < 32768 (your program need not test these conditions). The supplied Multi.test script and Mult.cmp compare file are designed to test your program on some representative values.

Fill.asm (example of an input/output task): This program runs an infinite loop that listens to the keyboard. When a key is pressed (any key), the program blackens the entire screen by writing "black" in every pixel. When no key is pressed, the program clears the screen by writing "white" in every pixel. You may choose to blacken and clear the screen in any spatial pattern, as long as pressing a key continuously for long enough will result in a fully blackened screen, and not pressing any key for long enough will result in a cleared screen. This program has a test script (Fill.tst) but no compare file – it should be checked by visibly inspecting the simulated screen in the CPU emulator.

You will write these assembly programs using a plain text editor, translate them into Hack binary code using an assembler, and execute them using a CPU Emulator.

Tools

Editor: Use any editor that handles plain text files.

Assembler: The supplied nand2tetris/tools/Assembler can be used for translating Xxx.asm source files containing symbolic Hack instructions into Xxx.hack files containing binary code that can be executed by computers and emulators that implement the Hack instruction set. To launch the assembler, enter “Assembler.sh” (Mac) or “Assembler” (Windows) from the command line. The assembler will be launched in a graphical window, featuring visual panels containing the source code and the translated code, and various controls.

CPU Emulator: The supplied nand2tetris/tools/CPUEmulator can be used for executing and testing the binary Xxx.hack files programs created by the assembler. To launch the emulator, enter

"CPUEmulator.sh" (Mac) or "CPUEmulator" (Windows) from the command line. The emulator will be launched in a graphical window displaying the current states of the Hack computer's instruction memory (ROM), data memory (RAM), registers (A and D), and program counter (PC). The emulator also displays the current state of the computer's screen, and allows entering inputs through the physical keyboard of your PC.

Steps

The projects/04 folder contains two sub-folders containing skeletal Mult.asm and Fill.asm programs. These are the assembly programs that you have to complete, translate, and test. We recommend proceeding as follows:

0. Navigate to the folder on your PC where the Xxx.asm file is located. Launch the Assembler and the CPUEmulator *from that current folder.*
1. Use a plain text editor to edit the Xxx.asm file and save its new version in the current folder.
2. Use the Assembler to translate the Xxx.asm file. If there are syntax errors, go to step 1.
3. Save the resulting Xxx.hack file in the current folder.
4. Informal testing: Load the Xxx.hack file into the CPU Emulator. If the Xxx program is expected to get inputs and / or write outputs in designated RAM locations, enter test values in these locations, run the program, and inspect the values that the program writes as it executes. If there are run-time errors, go to step 1.
5. Formal testing: Load the supplied Xxx.tst file into the CPU Emulator, and execute it. If there are run-time errors, go to step 1.

Assembly tips

Command-line assembly: The interactive assembler is a convenient starter. At some point you may want to switch to using a faster and less verbose program. To do so, enter "Assembler.sh Xxx.asm" (Mac) or "Assembler Xxx.asm" (Windows) from the command line. This is the same assembler, without the fancy GUI. If Xxx.asm is error-free, you will get no feedback, and a file named Xxx.hack will be created in the current folder (if you translate again, the file's new version will override the previous one). Otherwise, you will get an error message.

Built-in assembly: The CPU Emulator is equipped with a built-in assembler. This feature allows loading not only Xxx.hack files, but also symbolic Xxx.asm files, into the emulator. When an Xxx.asm file is loaded, it is translated to binary code on the fly (the emulator's ROM panel has a menu that allows switching between symbolic and binary views of the loaded code). We recommend *not* using this service in this project, since it is difficult to debug programs that are assembled implicitly. Instead, use either the interactive or the command-line assembler.

Error reporting in all versions of the Assembler is minimal, and often results in obscure error messages. Welcome to the rough world of low-level debugging.

Known bug: According to the Hack language C-instruction specification, two of the possible eight destinations are DM=... and ADM=... (these directives allow storing the ALU output in several destinations, simultaneously). However, the supplied Hack assembler flags these symbolic mnemonics as syntax errors, expecting instead MD=... and and AMD=.. This bug will be fixed in the next version of the supplied Hack assembler. For now, follow this guideline: If the assembly code that you write needs to use the destination DM=... or ADM=..., use MD=... or AMD=... instead.

Testing tips

The CPU Emulator has a simulated *execution speed* which is set by default to "medium". If you want, you can control the simulated execution speed by moving the speed slider from slow to fast.

The CPU Emulator's *animation option* is set by default to "program flow". This selection causes the emulator to highlight the instruction which is currently executing. This animation option is good for debugging, but may be sluggish for simulating the actual program behavior. In particular, when testing the Fill program, we recommend setting the "animate" menu to "no animation".

The Fill program has an optional test consisting of the files FillAutomatic.tst and FillAutomatic.cmp. For more information about this optional test, read the documentation of the former file.

Programming tips

The symbolic Hack language is case sensitive. A common programming error occurs when one writes, say, @foo and @Foo in different parts of the program, thinking that both instructions refer to the same symbol. In fact, the assembler will treat them as two unique symbols.

Another common error is using lower case, or spaces, when writing instructions. For example, either m=1 or M = 1 results in syntax errors. The correct syntax is M=1

Best practice:

- Use upper-case letters for labels, like LOOP, and lower-case letters for variables, like sum.
- Indent your code: Start all lines that are not label declarations a few characters to the right.
- Write comments, when comments are needed.
- See the programs in the lecture or in the book, and follow their example.
- As always, strive to write elegant, efficient, and self-explanatory programs.

References

CPU Emulator demo

Assembler tutorial (click *slideshow*)

CPU Emulator tutorial (click *slideshow*)