Project 4: Machine Language

This project introduces low-level machine language programming. You will write some programs using the Hack *assembly language*, translate them into binary code using a supplied assembler, and test the resulting binary code by executing it. Normally, binary code is executed on the computer's hardware; since the Hack computer will be completed only in the next project, in this project you will execute binary code using a *CPU emulator*. The CPU emulator is a computer program that executes binary code using software emulation.

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.

Tools

Assembler: The supplied assembler is designed to translate 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.

CPU Emulator: The supplied CPU emulator is designed to execute code written in the Hack instruction set. During the code's execution, the emulator displays the current states of the Hack computer's memory, registers, and program counter. The emulator also displays the computer's screen, and allows entering inputs using the physical keyboard of the host PC.

Editor: The assembler of the online Nand2Tetris IDE features an editor for inspecting and writing assembly programs. If you elect to use the desktop Nand2Tetris tools, use a plain text editor of your choice for inspecting and writing assembly programs.

Executing machine language programs

Before writing assembly programs of your own, we recommend playing with existing programs. In particular, experiment with the following assembly programs, stored in the projects/6 folder:

Add.asm: Adds the constants 2 and 3, and puts the result in R0 (recall that Ri refers to RAM[i]).

Max.asm: Computes max(R0, R1) and puts the result in R2. Before executing, put some test values in R0 and R1.

Rect.asm: Draws a rectangle at the top left corner of the screen. The rectangle is 16 pixels wide, and R0 pixels high. Before executing, put some test value in R0.

Pong.asm: A classical single-player arcade game. A ball bounces repeatedly off the screen's "walls." The player attempts to hit the ball with a paddle, by pressing the left-arrow and right-arrow keys. For every successful hit, the player gains a point and the paddle shrinks a little, to make the game more challenging. If the player misses the ball, the game is over. To quit the game, press ESC.

Before starting the game, click “enable keyboard”. Use the speed slider for playing the game at different speeds, ranging from slow motion to quick and challenging.

Background note: The Pong program was developed using tools from Part II of the Nand to Tetris course and book. Specifically, the game software was written in the high-level, Java-like Jack language. Organized in several Jack classes, the Pong game implementation consists of some 300 lines of high-level Jack code. This program, *along with the Jack OS*, was compiled by the Jack compiler into a single Pong.asm file containing 28,000+ machine language instructions (lines of code). This is a large file, and it may take a minute or so to load, so be patient.

For each Prog.asm program mentioned above: (0) Read the program’s documentation, and inspect its code, (1) Load the file into the CPU emulator, (2) if the program expects to get inputs, put test values in the relevant RAM addresses, (3) execute the code, (4) inspect the program’s flow, and note how it impacts the computer's registers, program counter, and memory.

Program execution tips

The CPU emulator of the new online IDE (the recommended option) is self-explanatory.

If, alternatively, you wish to use the desktop CPU emulator, navigate to the folder on your PC where the Prog.asm file is located, and launch the CPU emulator by entering “CPUEmulator.sh” (Mac) or “CPUEmulator” (Windows) from the terminal / command line.

The CPU emulator has a builtin assembler: When you load an .asm file into the emulator, it translates each symbolic instruction into its binary version. In the process, all the symbolic variables and labels are translated into physical addresses. You can inspect the loaded code in its symbolic version (which is the default), or in its binary version (by choosing “bin” from the display format menu of the ROM panel).

Change the program execution speed by controlling the CPU emulator’s speed slider.

In the desktop version of the CPU Emulator, the *animation option* is set by default to “program flow”. This selection, which highlights the instruction which is currently executing, is good for debugging, but may be sluggish in other cases. In particular, when testing the Pong program and the Fill program (described below), and any program that uses the keyboard or the screen, select ‘No animation’ from the ‘Animation’ menu (meaning, no animation of code execution).

Tasks

After playing with the assembly programs mentioned above, you are ready to develop assembly programs of your own. Write and test the following 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). Your code should not change the values of R0 and R1. The supplied Mult.test script and Mult.cmp compare file are designed to test your program on the CPU emulator, using some representative R0 and R1 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 order (top-down, bottom-up, spiral, etc.), 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.

The projects/4 folder contains skeletal Mult.asm and Fill.asm programs. These are the assembly programs that you have to develop. For each Xxx.asm program we also supply Xxx.tst and Xxx.cmp files, for testing the evolving programs on the CPU emulator.

Developing assembly programs

If you are using the Nand2Tetris IDE Online (the recommended option):

0. To start working on the assembly program Xxx.asm, load the Xxx.asm file into the assembler's editor (the "source" panel).

1. Write/edit the Xxx.asm file. All the changes will be saved automatically in the browser's memory. If you wish to save a copy on your local PC, click the *download* button. The current version of all the project's Xxx.asm files will be downloaded to your PC as one zip file.

2. Use the "translate" controls to translate the assembly program into binary code. Notice that the assembler creates and displays a *symbol table*. In this project there is no need to review or understand this symbol table, or the translation process (we'll do it in project 6). For now, simply translate the program; If there are syntax errors, go to step 1.

3. Click the "load" control of the "binary code" panel. This action will launch the CPU emulator, and load the binary code into it.

4. Informal testing: If the code expects to read values from RAM addresses, enter test values in these addresses; If the code expects to get inputs from the keyboard, click "enable keyboard". Next, execute the code using the step/run/reset controls. If there are run-time errors, or incorrect outputs, go to step 1.

5. Formal testing: Notice that the test panel is already loaded with an Xxx.tst test script. Execute the test script. If there are run-time errors, or incorrect outputs, go to step 1.

Using the desktop Nand2Tetris assembler is also possible. Follow these steps:

0. Start by launching the three tools that you will need for writing and executing assembly programs. Launch the assembler by entering "Assembler.sh" (Mac) or "Assembler" (Windows) from the terminal / command line. The assembler will be launched in a graphical window. Next, launch the CPU emulator by entering "CPUEmulator.sh" (Mac) or "CPUEmulator" (Windows) from the terminal / command line. Finally, to start working on the assembly program Xxx.asm, load the projects/4/Xxx/Xxx.asm file into a plain text editor of your choice. At this point you will have three active windows: the editor, the assembler, and the CPU emulator.

1. Write/edit the Xxx.asm file in the editor, and save it in the current folder.

2. Load the Xxx.asm file into the assembler, and translate it. If there are syntax errors, go to step 1.

3. Save the translated Xxx.hack file in the current folder (this step is optional).

4. Load the Xxx.asm file (or the Xxx.hack file, it does not matter) into the CPU Emulator.

5. Informal testing: If the code expects to read values from RAM addresses, enter test values in these addresses; If the code expects to get inputs from the keyboard, click "enable keyboard". Next, execute the code using the step/run/reset controls. If there are run-time errors, or incorrect outputs, go to step 1.

6. Formal testing: click the "load script" control, load the projects/4/Xxx/Xxx.tst file into the CPU Emulator, and execute it. If there are run-time or incorrect outputs, go to step 1.

The Fill.asm 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.

Assembly tips

A command-line assembler is also available. If you wish to use it, enter "Assembler.sh Xxx.asm" (Mac) or "Assembler Xxx.asm" (Windows) from the terminal / command line. 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.

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, some Hack assemblers flag these destination codes as syntax errors, expecting instead MD=... and and AMD=...
Until we fix this bug, follow this guideline: If the assembly code that you write needs to use the destination DM=... or ADM=..., use MD=... or AMD=... instead.

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 treats 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, as needed, to make your code clear.
- 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.

Tutorials

The tutorials below focus on using the desktop version of the tools used in this project. Tutorials for the online simulator, the preferred tool for this project, will be available soon. However, you can apply the principles from these tutorials to perform similar actions in the online tools.

CPU Emulator demo

Assembler tutorial (click *slideshow*)

CPU Emulator tutorial (click *slideshow*)