Project 6: Assembler

Low-level programs written in symbolic machine language are called *assembly* programs. Programmers rarely write programs directly in machine language. Rather, programmers who develop high-performance programs (e.g. system software, mission-critical apps, and software for embedded systems) often inspect the assembly code generated by compilers. They do so in order to understand how their high-level code is actually deployed on the target hardware platform, and how that code can be optimized for gaining better performance. One of the key tools in this process is the program that translates code written in a symbolic machine language into code written in binary machine language. This program is called an *assembler*.

Objective

Develop an *assembler* that translates programs written in the Hack assembly language into Hack binary code. This version of the assembler assumes that the source assembly code is valid. Error checking, reporting and handling can be added to later versions of the assembler, but are not part of this project. If you have no programming experience, you can develop a manual assembly process, as described at the end of this document.

Contract

When supplied to your assembler as a command-line argument, a Prog.asm file containing a valid Hack assembly language program should be translated correctly into Hack binary code, and stored in a file named Prog.hack, located in the same folder as the source file (if a file by this name exists, it should be overridden). The output produced by your assembler must be identical to the output produced by the supplied assembler, as described below.

Test programs

Once again, the goal of the assembler is translating Prog.asm files into executable Prog.hack files. We provide four such test programs (same as those described in Project 4):

Add.asm: Adds the constants 2 and 3, and puts the result in R0.

Max.asm: Computes max(R0, R1) and puts the result in R2.

Rect.asm: Draws a rectangle at the top left corner of the screen. The rectangle is 16 pixels wide, and R0 pixels high.

Pong.asm: A classical single-player arcade game, described in detail in Project 4 (see the *executing machine language programs* section). This large assembly file will give your assembler a good stress-test.

Assembling programs

Before writing *your assembler*, we recommend playing with the *supplied assembler*. This will give you hands-on, visual, and step-wise exposure to the action of a correctly-implemented assembler.

If you are using the new Nand2Tetris IDE Online (the recommended option): Load any one of the four Prog.asm programs described above into the assembler, and translate it. Notice how the symbol table is built as part of the translation process. Ignore the "Compare code" panel for now. If you want, you can load the translated Prog.hack binary code into the CPU emulator, and proceed to execute it (before doing so, read the documentation in the Prog.asm file – some programs require putting test values in selected RAM addresses). Note: Executing the translated code is not part of the assembly process; It provides an indirect test that the assembler translates programs correctly.

Using the desktop Nand2Tetris Assembler is also possible. Launch the desktop assembler by entering "Assembler.sh" (Mac) or "Assembler" (Windows) from the terminal / command line. Then load Prog.asm into the "Source" panel and proceed to translate it.

Tools

The main tool needed for completing this project is the programming language in which you will implement your assembler. If you have no programming experience, you can develop a manual assembly process, as described at the end of this document. For both options – a program-based assembler or a manual assembler – you will also need the supplied assembler, for comparing the binary code generated by your assembler to the code generated by a correct assembler. If you wish to execute the code translated by your assembler and inspect its behavior, you can do so using the CPU emulator. Seeing the code produced by your assembler execute correctly can be gratifying.

Development plan

We recommend building and testing your assembler in two stages. First, write a basic assembler that translates programs that contain no symbolic references (i.e., neither variables nor labels). Then extend your assembler with symbol handling capabilities.

We supply test programs for both stages. Specifically: The Add.asm test program has no symbolic references. The remaining test programs (Max, Rect, and Pong) come in two versions: Prog.asm and ProgL.asm, which are with and without symbolic references, respectively.

With that in mind, test your stage I assembler on Add.asm, MaxL.asm, RectL.asm, and PongL.asm. Test your stage II, final assembler on Add.asm, Max.asm, Rect.asm, and Pong.asm.

Testing your assembler

Let Prog.asm be an assembly Hack program, e.g. one of the given test programs. There are essentially two ways to test if your assembler translates Prog.asm correctly. First, you can load the executable Prog.hack file generated by your assembler into the supplied CPU emulator, execute it, and check that the program does what is described in its documentation. Second, you can use the supplied assembler to compare the binary code that it produces (correctly) to the Prog.hack file

produced by *your* assembler. If your assembler is implemented correctly, the two code files must be identical. Otherwise, you will get a comparison error message.

If you are using the Nand2Tetris IDE Online, Load the Prog.asm test file into the assembler tool, and translate it. Next, load the Prog.hack code generated by *your assembler* into a plain text editor, and copy-paste it into the "Compare code" panel of the assembler tool. Finally, click the "Compare" button. If the two code files are identical, you will get a "Comparison successful" message. Otherwise, you will get an error message, indicating that your assembler has a problem.

If you are using the desktop Nand2Tetris assembler: Rename the Prog.hack file generated by your assembler to, say, Prog1.hack. Next, load Prog.asm into the desktop assembler, and translate it. Click the "equals" button, load the Prog1.hack file, and proceed to compare the two files.

Software update (relevant only if you are using the desktop assembler): We recently made some changes in the project 6 files. To make sure that you are using the latest version of these files, re-download the Nand to Tetris software package, and copy the downloaded projects/6 folder onto the projects/6 folder in your PC (the other project folders in your PC should not change). Do this before you start working on project 6.

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 more than one destination, simultaneously). However, some Hack assemblers flag these symbolic mnemonics as syntax errors, expecting instead MD=... and and AMD=... When developing *your* assembler, handle this issue by accepting either DM or MD as standing for the destination *d*-bits 011, and either ADM or AMD as standing for the *d*-bits 111.

Manual Assembler Option

If you have no programming background, you can apply and test your understanding of the assembly process by translating assembly programs manually. The task of the manual assembler is exactly the same as that of a program-based assembler: Getting an Xxx.asm file that contains a program written in the Hack assembly language as input, and translating it correctly into an Xxx.hack file written in Hack binary code, as output.

In order to perform the manual translation, all you need is a plain text editor. Use the editor to write the output file of the assembly process, i.e. a sequence of lines, each being a string of sixteen 0 and 1 characters.

How to carry out the manual translation process? To do so, follow the explanations and guidelines given in lecture / chapter 6. In a nutshell: ignore white space (comments and empty lines), make a "first pass" that builds a symbol table and adds the label symbols (generating no code), then make a "second pass" that completes the process by translating each instruction line-by-line, using the symbol table (and adding the symbolic variable names to the table in the process). Notice that the symbol table is not part of the assembler's output; it is a temporary tool used only during the assembly process.

We recommend building and testing your manual translation ability in two stages. First, develop an ability to translate assembly programs that contain no symbolic references (i.e. neither variables nor labels), and test this ability by translating the test programs Add.asm, MaxL.asm, and RectL.asm. Next, develop the ability to handle symbols and translate any given assembly program. Test this ability by translating the test programs Max.asm and Rect.asm (the Pong.asm test program is quite long, so translating it manually is not required). After each translation, test the resulting output – the Xxx.hack file that you wrote and saved using the editor – by following the guidelines given in the "Testing Your Assembler" section above.

Tutorials

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

Assembler tutorial (click *slideshow*)

For executing the translated .hack files:

CPU Emulator demo

CPU Emulator tutorial (click *slideshow*)