Project 6

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 to the hardware, and how that code can be optimized for gaining better performance. One of the players 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.

Resources

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 quite gratifying.

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 is overridden). The output produced by your assembler must be identical to the output produced by the supplied assembler.

Development plan

We suggest building and testing the 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.

Test programs

The first test program listed below has no symbolic references. The remaining test programs come in two versions: Prog.asm and ProgL.asm, which are with and without symbolic references, respectively.

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. Before running this program, put some 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 running this program, put a non-negative 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 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. Note: The Pong program was developed using tools presented in Part II of the course and the book. In particular, the game software was written in the high-level Jack language, and translated into the given Pong.asm file by the Jack compiler. Although the high-level Pong program is only about 300 lines of code, the executable Pong application is about 28,000 lines of binary code, most of which being the Jack operating system. Before running the code, select 'No animation' from the 'Animation' menu (meaning, no code execution highlighting). You can control the code's execution speed using the speed slider. The game will start after a few seconds, during which the OS initializes.

Testing

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 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.

The second testing technique is more direct: It compares the code generated by your assembler to the code generated by the supplied assembler. To begin with, rename the file generated by your assembler to Prog1.hack. Next, load Prog.asm into the supplied assembler, and translate it. If your assembler is working correctly, it follows that Prog1.hack must be identical to the Prog.hack file produced by the supplied assembler. This comparison can be done by loading Prog1.asm as a compare file. If needed, see the assembler tutorial.

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. 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 the lecture, or in chapter 6. In a nutshell: ignore white space (comments and empty lines), make a "first pass" that (i) builds a symbol table, (ii) adds the label symbols, and (iii) generates 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). All this is explained in detail in the lecture, and in chapter 6. 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" section described above.

References

Assembler Tutorial (click *slideshow*)

CPU Emulator demo (in case you want to execute the programs generated by your assembler)