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 key 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, following the specifications and APIs described in the relevant lecture and book chapter 6. In addition, you will need to use the supplied assembler, and, optionally, the CPU emulator, both of which are available in nand2tetris/tools on your computer. You will use the supplied assembler for comparing the binary code generated by your assembler to the code generated by the supplied assembler. And, if you wish to execute that translated code and inspect its behavior, you can do so using the CPU emulator.

Contract

When given to your assembler as a command-line argument, a Prog.asm file containing a valid Hack assembly language program should be translated into the correct 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., no variables and labels). Then extend your assembler with symbol handling capabilities.

Test programs

The first test program listed below has no symbolic references (i.e. no variables and labels). 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.

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 20,000 lines of binary code, most of which being the Jack operating system. Running this interactive program in the supplied CPU emulator is a slow affair, so don't expect a high-powered Pong game. This slowness is actually a virtue, since it enables tracking the graphical behavior of the program. As you'll develop the software hierarchy described in Part II of the course and the book, the game will run much faster.

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 it's doing what it's supposed to be doing.

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.

References

Assembler Tutorial (click *slideshow*)

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