Project 6: Assembler

CS 220

Start: Oct. 16; Due: Nov. 1 at 11:55 pm

Background

Low-level machine programs are rarely written by humans. Typically, they are generated by compilers. Yet humans can inspect the translated code and learn important lessons about how to write their high-level programs better, in a way that avoids low-level pitfalls and exploits the underlying hardware better. One of the key players in this translation process is the assembler — a program designed to translate code written in a symbolic machine language into code written in binary machine language.

This project marks an exciting landmark in our Nand2Tetris odyssey: it deals with building the first rung up the software hierarchy, which will eventually end up in the construction of a compiler for a Java-like high-level language. But, first things first.

Objective

Write an Assembler program that translates programs written in the symbolic Hack assembly language into binary code that can execute on the Hack hardware platform built in the previous projects.

Contract

There are three ways to describe the desired behavior of your assembler:

1. When loaded into your assembler, a `Prog.asm` file containing a valid Hack assembly language program should be translated into the correct Hack binary code and stored in a `Prog.hack file`.

2. The output produced by your assembler must be identical to the output produced by the Assembler supplied with the Nand2Tetris Software Suite.

3. Your assembler must implement the translation specification given in Chapter 6, Section 2.

Resources

The relevant reading for this project is Chapter 6. Your assembler implementation can be written in any programming language, but check with me before using any programming language other than Java. We've provided starter code for an Eclipse Java project. Two useful tools are the supplied Assembler and the supplied CPU Emulator. These tools allow experimenting with a working assembler before setting out to build one yourself. In addition, the supplied assembler provides a visual line-level translation GUI, and allows code comparisons with the outputs that your assembler will generate. For more information about these capabilities, refer to the supplied Assembler Tutorial from Project 4.

Proposed Implementation

Section 6.3 includes a proposed, language-independent Assembler API, which can serve as your implementation's blueprint. We suggest building the assembler in two stages. First, write a basic assembler designed to translate assembly programs that contain no symbols. Next, extend your basic assembler with symbol handling capabilities, yielding the final assembler. The test programs that we supply below are designed to support this staged implementation strategy.

Section 6.3 mentions the usefulness of a hash table in implementing the symbol table. I found Java's `HashTable` class quite useful. I also used the `HashTable` class in my `Code` class; in fact, I used three of them.

Test Programs

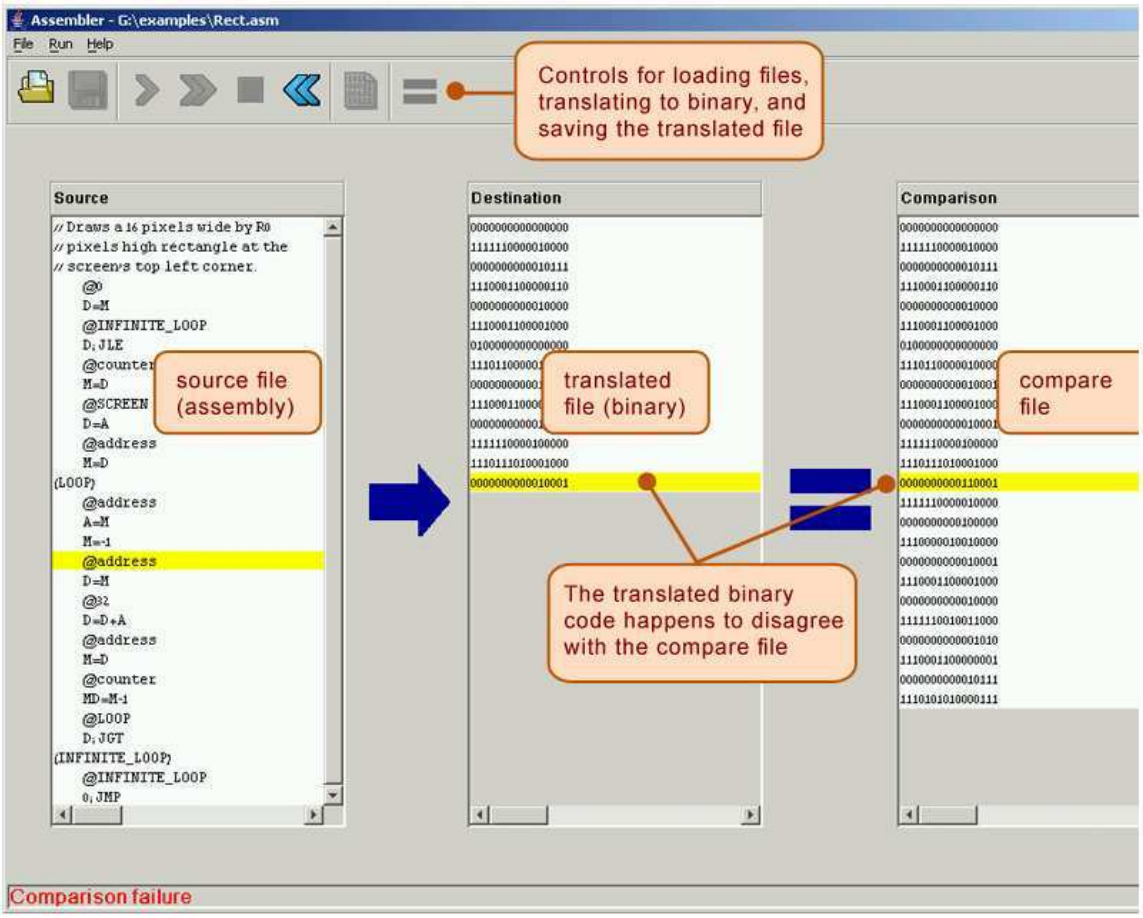
Each test program except the first one comes in two versions: `Prog.asm` is an assembly program; `ProgL.asm` is the very same program, Less the symbols (each symbol is replaced with an explicit memory address).

Symbolic Program	Without Symbols	Description
Add.asm		Adds up the constants 2 and 3 and puts the result in `R0`.
Max.asm	MaxL.asm	Computes `max(R0,R1)` and puts the result in R2.
Rect.asm	RectL.asm	Draws a rectangle at the top-left corner of the screen. The rectangle is 16 pixels wide and `R0` pixels high.
Pong.asm	PongL.asm	A single-player *Pong* game. A ball bounces off the screen's "walls". The player attempts to hit the ball with a paddle by pressing the *left* and *right* arrow keyboard keys. For each successful hit, the player gains one point and the paddle shrinks a little, to make the game slightly more challenging. If the player misses the ball, the game is over. To quit the game, press the `ESC` key.

The Pong program supplied above was written in the Java-like high-level Jack language and translated into the Hack assembly language by the Jack compiler (Jack and the Jack compiler are described in Chapter 9 and in Chapters 10–11, respectively). Although the original Jack program is only about 300 lines of Jack code, the executable Pong code is naturally much longer. 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 your eye to track the graphical behavior of the program. As we continue to build the software platform in the next few projects, Pong and and other games will run much faster.

Tools

The supplied Hack Assembler shown below is guaranteed to generate correct binary code. This guaranteed performance can be used to test if another assembler, say the one written by you, also generates correct code. The following screen shot illustrates the comparison process:



The comparison logic: Let `Prog.asm` be some program written in the symbolic Hack assembly language. Suppose we translate this program using the supplied assembler, producing a binary file called `Prog.hack`. Next, we use another assembler (e.g. the one that you wrote) to translate the same program into another file, say `MyProg.hack`. Now, if the latter assembler is working correctly, it follows that `Prog.hack == MyProg.hack`. Thus, one way to test a newly written assembler is as follows:

1. load into the supplied visual assembler `Prog.asm` as a source program and `MyProg.hack` as a compare file,
2. translate the source program, and
3. compare the resulting binary code with the compare file (see the figure above). If the comparison fails, the assembler that generated `MyProg.hack` must be buggy; otherwise, it may be OK.

Submission and Assessment

If you can't finish the project on time, submit what you've managed to do, and relax. All the projects in this course are highly modular, with incremental test files. Each hardware project consists of many chip modules (`*.hdl` programs), and each software project consists of many software modules (classes and methods). It is best to treat each project as a modular problem set, and try to work out as many problems as you can. You will get partial credit for partial work.

What if your chip or program is not working? It's not the end of the world. Hand in whatever you did, and explain what works and what doesn't in a README file. If you want, you can also

supply test files that you developed, to demonstrate working and non-working parts of your project. Instead of trying to hide the problem, be explicit and clear about it. You will get partial credit for your work.

See the next page for the assessment rubric. Submit the following as a single ZIP archive in GoucherLearn:

1. A README file containing the names of all group members. This file may also contain other information, as described above.

2. All files necessary to build your Assembler program from source. If you use the starter Eclipse Java project, export your project into a ZIP archive and submit the archive.

3. Nothing else.

You will lose points for not following these submission instructions.

Project 6: Assembler

Student name(s): _____

Grading method: As usual with programming assignments, we look for elegance, clarity, reasonable documentation, and neatness.

Assembler	***Working?***		***Comments***
Working?	/ 75		
Well built?	/ 25		
Total	/ 100		

Total grade: _____