CS220 – Project 4: Machine Language Programming

Background

Each hardware platform is designed to execute a certain machine language, expressed using agreed-upon binary codes. Writing programs directly in binary code is a possible, yet an unnecessary, tedium. Instead, we can write such programs in a low-level symbolic language, called assembly, and have them translated into binary code by a program called an assembler. In this project you will write some low-level assembly programs, and will be forever thankful for high-level languages like C and Java. (Actually, assembly programming can be a lot of fun, if you are in the right mood; it's an excellent brain teaser, and it allows you to control the underlying machine directly and completely.)

Objective

To get a taste of low-level programming in machine language, and to get acquainted with the Hack computer platform. In the process of working on this project, you will become familiar with the assembly process — translating from symbolic language to machine-language — and you will appreciate visually how native binary code executes on the target hardware platform. These lessons will be learned in the context of writing and testing the two low-level programs described below.

Programs

Program	Description	Comments / Tests
Mult.asm	**Multiplication:** In the Hack framework, the top 16 RAM words (`RAM[0]` ... `RAM[15]`) are also referred to as the so-called *virtual registers* `R0` ... `R15`. With this terminology in mind, this program computes the value `R0*R1` and stores the result in `R2`.	For the purpose of this program, we assume that `R0>=0`, `R1>=0`, and `R0*R1<32768` (you are welcome to ponder where this value comes from). Your program need not test these conditions, but rather assume that they hold. To test your program, put some values in RAM[0] and RAM[1], run the code, and inspect RAM[2]. The supplied `Mult.tst` script and `Mult.cmp` compare file are deigned to test your program "officially", running it on several representative values supplied by us.
Fill.asm	**I/O handling:** This program illustrates low-level handling of the screen and keyboard devices. In particulate, the program runs an infinite loop that listens to the keyboard input. When a key is pressed (any key), the program blackens the screen, i.e. writes "black" in every pixel. When no key is pressed, the program clears the screen, i.e. writes "white" in every pixel.	You may choose to write code that blackens and clears the screen's pixels in any spatial/visual order, 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 fully cleared screen. We provide a test script (`Fill.tst`), but no compare file. The program should be checked visually by inspecting the simulated screen of the supplied *CPU Emulator*.

Mult Pseudo-Code

See pp. 249–250 of the textbook for a binary multiplication example. In the example on pg. 250, x is the multiplicand and y is the multiplier.

```
// Multiplicand and multiplier are pre-loaded into R0 and R1,
// respectively.
mask = 1;  // mask is used to isolate a single bit of the
           // multiplier.
i = 0;
R2 = 0;  // Product accumulated into R2

while (i < 16)
{
   // Isolate current multiplier bit; accumulate shifted multiplicand
   // in R0 into R2 if this bit is set.
   if ((mask & R1) != 0)
      R2 = R2 + R0;

   // Shift intuition: adding a number to itself is equivalent to
   // multiplying it by 2: x + x = 2x.  In binary, multiplying a
   // value by two results in all of the bits of the value being
   // shifted left one bit position, with a 0 being shifted into
   // the lsb position and the msb being lost.

   R0 = R0 + R0;  // Shift multiplicand left one bit position.
   i = i + 1;
   mask = mask + mask;  // Shift mask left one bit position.
}
```

Fill Pseudo-Code

```
sptr = SCREEN;  // Memory pointer to next screen memory location
                // to be filled with black or white pixels.
clear = -1;     // This is for the 10 "smooth user experience" points.
                // See implementation tip in Proposed Implementation
                // Section below.  "-1" is true in Hack; "0" is false.

while (1)
{
   if (*KBD != 0)  // Read the keyboard location's value.  if
   {               // !0, fill with black pixels.
      if (clear != 0)   // For "smooth user experience."
      {
         clear = false;
         sptr = SCREEN;
      }

      *sptr = -1;  // Write 16 black pixels to the screen memory location
                   // pointed to by sptr.
   }
   else  // Fill with white pixels.
   {
      if (clear == 0)   // For "smooth user experience."
      {
         clear = true;
         sptr = SCREEN;
      }

      *sptr = 0;  // Write 16 white pixels to the screen memory location
                  // pointed to by sptr.
   }

   sptr = sptr + 1;  // Increment sptr to point to next location
                     // in screen memory.

   // If we've reached the end of screen memory, start over.
   if (sptr >= SCREEN + 8192)
      sptr = SCREEN;
}
```

Proposed Implementation

1. Use a plain text editor to write the first assembly program. You can do it by loading and editing the supplied `mult/Mult.asm` file.

2. Use the supplied Assembler (in either batch or interactive mode) to translate your

program. If you get syntax errors, go to step 1. If there are no syntax errors, the assembler will produce a file called `mult/Mult.hack`, containing binary instructions written in the Hack machine language.

3. Use the supplied CPU Emulator to load, and then test, the translated `Mult.hack` code. This can be done either interactively, or batch-style using the supplied `Mult.tst` script. If you get run-time errors, go to step 1.

4. Repeat steps 1–3 for the second program (`Fill.asm`), working in the `fill` directory. Implementation tip: If you're struggling with the `Fill.asm` program's "smooth user experience" logic, implement it first without this logic and get that working. Then, work on adding this logic. At the worst, you'll lose 5% for not having this logic working.

Debugging tip: The Hack language is case-sensitive. A common error occurs when one writes, say, `@foo` and `@Foo` in different parts of the program, thinking that both labels are treated as the same symbol. In fact, the assembler treats them as two different symbols. This is a nasty, difficult-to-detect bug.

Criteria for Success

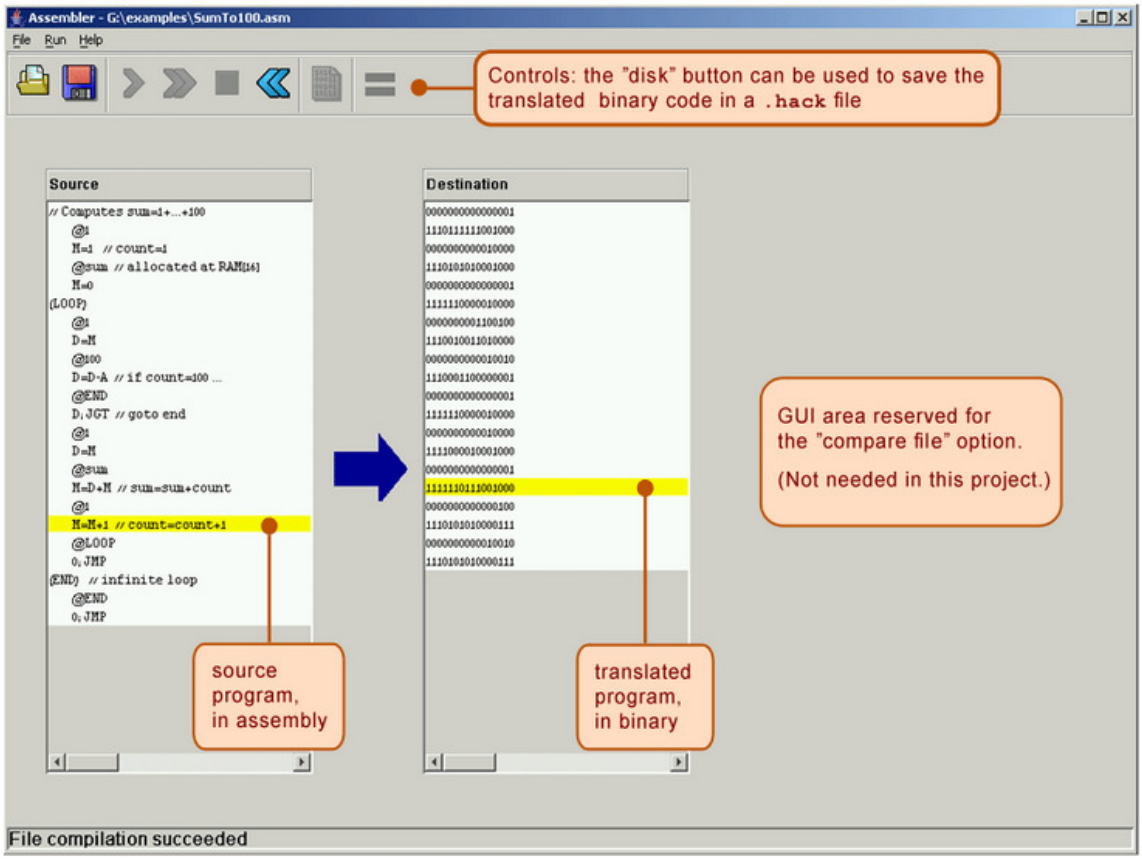
Write and test the two programs described above. When executed on the CPU Emulator, your programs should generate the results mandated by the specified tests.

Resources

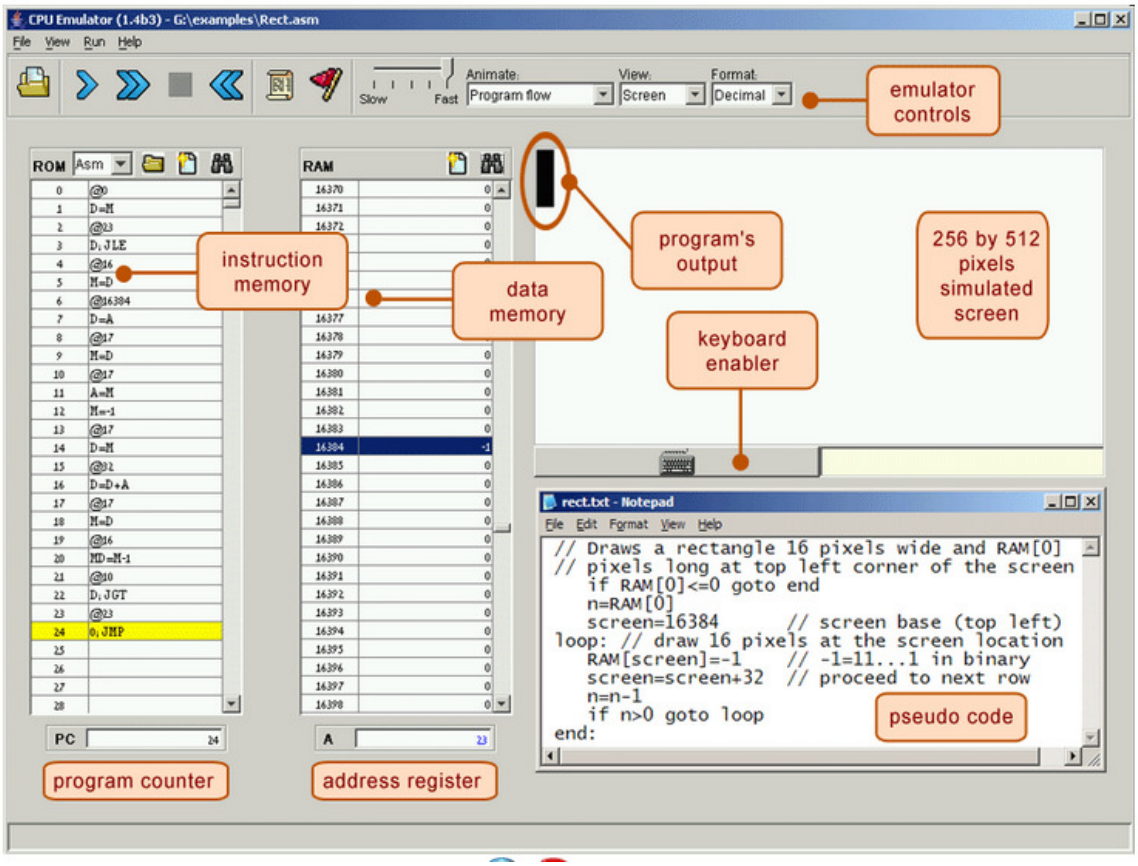
The Hack assembly language is described in detail in Chapter 4. You will need two tools: the supplied Assembler — a program that translates programs written in the Hack assembly language into binary Hack code, and the supplied CPU Emulator — a program that runs binary Hack code on a simulated Hack platform. Two other related and useful resources are the supplied Assembler Tutorial and the CPU Emulator Tutorial. We recommend going through these tutorials before starting to work on this project. The project files are available in a ZIP archive file available on the course web site.

Tools

The supplied Hack Assembler can be used in either command mode (from the command shell) or interactively. The latter mode of operation allows observing the translation process in a visual and step-wise fashion, as shown below:



The machine language programs produced by the assembler can be tested using the supplied CPU Emulator. This CPU Emulator includes a ROM (also called Instruction Memory) representation, into which the binary code is loaded, and a RAM representation, which holds data. For ease of use, the emulator enables the user to view the loaded ROM-resident code in either binary mode, or in symbolic / assembly mode. In fact, the CPU Emulator even allows loading a program written in assembly directly into the ROM, in which case the program is translated into binary code on the fly. This utility seems to render the supplied assembler unnecessary, but this is not the case. First, the supplied assembler shows the translation process visually, for instructive purposes. Second, the assembler generates a persistent binary file. This file can be executed either on the CPU Emulator, as we illustrate below, or directly on the hardware platform, as we'll do in the next project.



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.

Submit all your ASM files as a single ZIP archive.