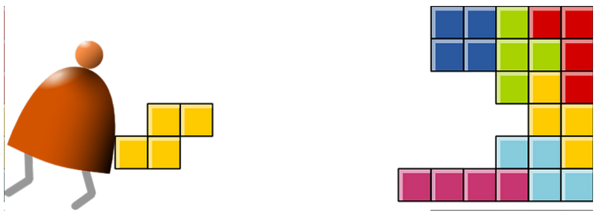
From Nand to Tetris

Building a Modern Computer From First Principles



Project 4: Machine Language Programming

Background

Every hardware platform is designed to execute commands in a certain machine language, expressed using agreed-upon binary codes. Writing programs directly in binary code is a possible, yet unnecessary. Instead, we can write such programs using a low-level symbolic language, called assembly, and have them translated into binary code by a program called assembler. In this project you will write some low-level assembly programs, and will be forever thankful for high-level languages like Java and Python. (Actually, assembly programming can be highly rewarding, allowing direct and complete control of the underlying machine.)

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 two low-level programs, as follows.

Programs

Program	Description	Guidelines / Tests
Program Mult.asm	Description **Multiplication**: in the Hack computer, the top 16 RAM words (RAM[0]...RAM[15]) are also referred to as R0...R15. With this terminology in mind, this program computes the value R0*R1 and stores the result in R2. The program assumes that R0>=0, R1>=0, and R0*R1<32768. Your program need not test these conditions, but rather assume that they hold.	Test Scripts Use a plain text editor to write your Mult.asm program using the Hack assembly language. Use the supplied Hack Assembler to translate your Mult.asm program, producing a Mult.hack file containing binary Hack instructions. Next, load the supplied Mult.tst script into the CPU Emulator. This script loads the Mult.hack program, and executes it. Run the script. If you get any errors, debug and edit your Mult.asm program. Then assemble the program, re-run the Mult.txt script, etc.
Program Fill.asm	Description **I/O handling**: this program illustrates low-level handling of the screen and keyboard devices, as follows. 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; the screen should remain fully black as long as the key is pressed. When no key is pressed, the program clears the screen, i.e. writes "white" in every pixel; the screen should remain fully clear as long as no key is pressed. Implementation note: your program may blacken and clear the screen's pixels in any spatial/visual order, as long as pressing a key continuously for long enough results in a fully blackened screen, and not pressing any key for long enough results in a fully cleared screen.	Test Scripts Write, test, and debug your Fill.asm program by following the same guidelines given above for the Mult program. The supplied Fill.tst script, which comes with no compare file, is designed to do two things: (i) load the Fill.hack program, and (ii) remind you to select 'no animation', and then test the program interactively by pressing and releasing some keyboard keys. The supplied FillAutomatic.tst script, along with the supplied compare file FillAutomatic.cmp, are designed to test the Fill program automatically, as described by the test script documentation. For completeness of testing, test the Fill program both interactively and automatically.

Contract

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

Resources

The Hack assembly language is documented and discussed in Chapter 4.

The two relevant tools for this project are the supplied *CPU emulator*, which is required, and the supplied *assembler*, which is optional. Explanation: When an .asm file is loaded into the CPU emulator, the program translates the symbolic instructions into binary Hack instructions on the fly, obviating the need for a separate assembly process.

See the CPU emulator tutorial (PPT, PDF) and, optionally, the assembler tutorial (PPT, PDF).

The project 4 files are available in your nand2tetris/projects/04 folder.

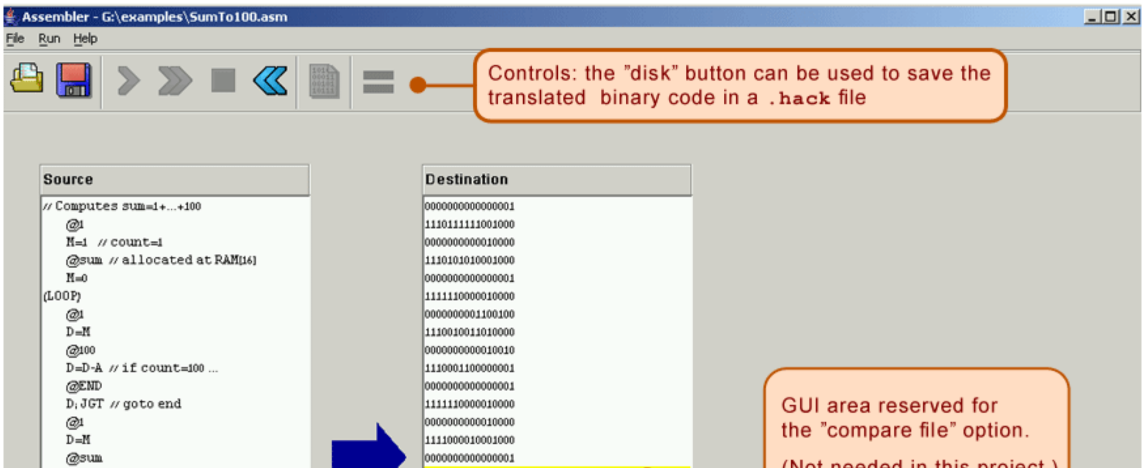
Proposed Implementation

1. Use a plain text editor to write/edit the first assembly program. Start by loading and editing the supplied skeletal projects/04/mult/Mult.asm file.
2. If you want, you can use the supplied assembler for translating your Mult.asm program into Hack binary code. This activity is optional, since the CPU Emulator translates loaded .asm files on the fly.
3. Use the supplied CPU emulator to load, and then execute, your Mult program. This can be done either interactively, or using the supplied Mult.tst script. If there are any errors, goto step 1.
4. Repeat stages 1-3 for the second program (Fill.asm), working in the projects/04/fill folder.

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

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:



Controls: the "disk" button can be used to save the translated binary code in a `.hack` file

```
// Computes sum=1+...+100
    @1
    M=1  // count=1
    @sum  // allocated at RAM[16]
    M=0
(LOOP)
    @1
    D=M
    @100
    D=D-A  // if count=100 ...
    @END
    D;JGT  // goto end
    @1
    D=M
    @sum
```

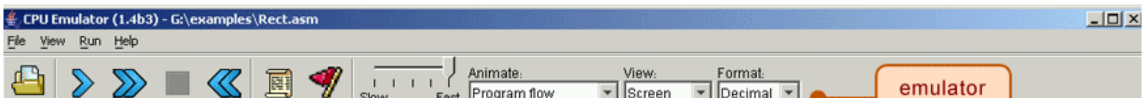
```
0000000000000001
1110111111001000
0000000000010000
1110101010001000
0000000000000001
1111110000010000
0000000001100100
1110010011010000
0000000000010010
1110001100000001
0000000000000001
1111110000010000
0000000000010000
1111000010001000
0000000000000001
```

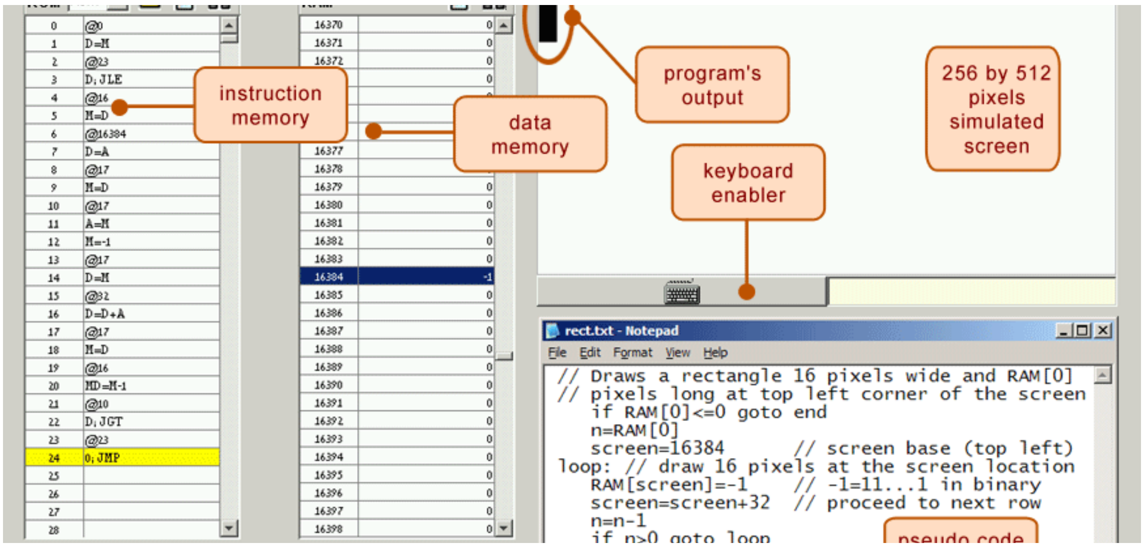
GUI area reserved for the "compare file" option.

(Not needed in this project.)

The machine language programs produced by the assembler can be tested in two different ways. First, one can run the resulting .hack program in the supplied CPU emulator. Alternatively, one can run the same program directly on the Hack hardware, using the supplied hardware simulator used in projects 1-3. To do so, one can load the Computer.hdl chip (built in project 5) into the hardware simulator, and then proceed to load the binary code (from the .hack file) into the computer's Instruction Memory (also called ROM). Since we will complete building the hardware platform and the Computer.hdl chip only in the next project, at this stage we recommend testing machine-level programs using the supplied CPU emulator.

The supplied 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 symbolic code written in assembly directly into the ROM, in which case the emulator translates the loaded code 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.





```
0   @0
1   D=M
2   @23
3   D;JLE
4   @16
5   M=D
6   @16384
7   D=A
8   @17
9   M=D
10  @17
11  A=M
12  M=-1
13  @17
14  D=M
15  @32
16  D=D+A
17  @17
18  M=D
19  @16
20  MD=M-1
21  @10
22  D;JGT
23  @23
24  0;JMP
25
26
27
28
```

```
16370  0
16371  0
16372  0
16377
16378
16379  0
16380  0
16381  0
16382  0
16383  0
16384  -1
16385  0
16386  0
16387  0
16388  0
16389  0
16390  0
16391  0
16392  0
16393  0
16394  0
16395  0
16396  0
16397  0
16398  0
```

```
// Draws a rectangle 16 pixels wide and RAM[0]
// pixels long at top left corner of the screen
    if RAM[0]<=0 goto end
    n=RAM[0]
    screen=16384       // screen base (top left)
loop: // draw 16 pixels at the screen location
    RAM[screen]=-1     // -1=11...1 in binary
    screen=screen+32   // proceed to next row
    n=n-1
    if n>0 goto loop
```