**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 1, 0 sequence of code is a possible, yet unnecessary and often error prone. 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 C and Java. *(****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 three low-level programs, as follows.

**Programs**

**Before you start implementing, make sure you have replaced the Compilers.jar and Hack.jar files in ../nand2tetris/tool/bin/lib with the new ones provided in this release.**

|  |  |  |
| --- | --- | --- |
| **Program** | **Description** | **Comments / Tests** |
| **Div.asm** | **Division**: in the Hack computer, 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.  Note that in the context of this program, we assume that (a) R0 and R1 are +ve non-zero integers, and (b) the modulus of R0/R1 is discarded. Your program need not test these conditions, but rather assume that they hold.  You may not use any bit shift operation in Div.asm  Loop will be used in division. | Start by using the supplied assembler to translate your Mult.asm program into a Div.hack file.  To test your program, load Div.hack into the CPUEmulator. Next, put some values in R0 and R1, run the code, and inspect R2.  Alternatively, use the supplied Div.tst script and Div.cmp compare file (that's how we test your program). These supplied files are designed to test your program by running it on several representative data values. |
| **Fill.asm** | **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. | Start by using the supplied assembler to translate your Fill.asm program into a Fill.hack file. 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.  The simple 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 FillAutomatic.tst script, along with the compare file FillAutomatic.cmp, are designed to test the Fill program automatically, as described by the test script documentation.  For completeness of testing, it is recommended to test the Fill program both interactively and automatically. |
| **Mult.asm** | Implement a program that multiply R0 and R1, and put the result in R2, with bit shift operations.  **Operator**: ‘<’ is left shift and  ‘>’ is right shift.  **Hint:** A pseudocode is given below. | This multiplication must be implemented using bit shifting only. Use of any other method will not receive full credit.  Use the supplied Mult.tst script and Mult.cmp compare file (that's how we test your program). These supplied files are designed to test your program by running it on several representative data values. |

**Contract**

Write and test the three 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 described in detail in Chapter 4.

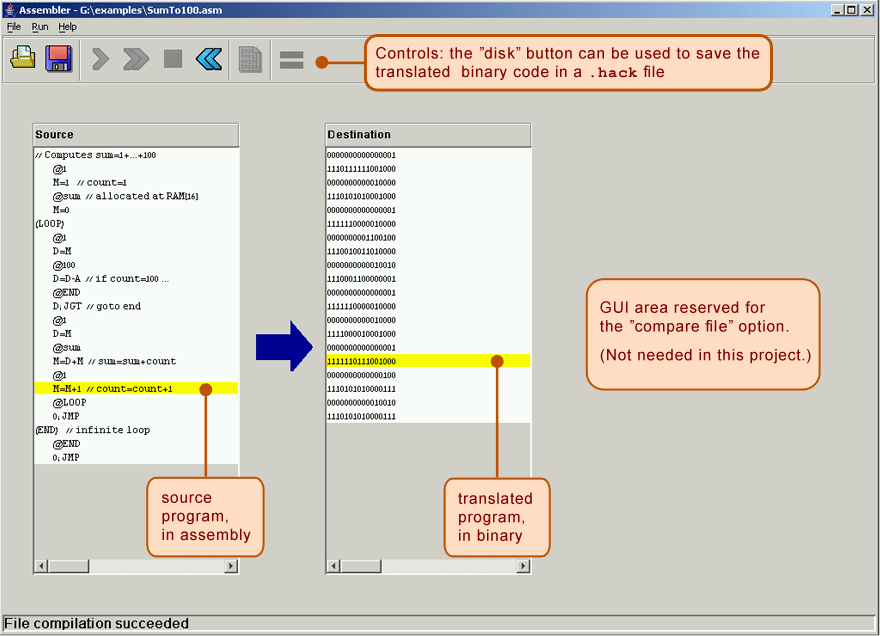
You will need two tools: the supplied assembler - a program that translates programs written in the Hack assembly language into binary Hack code, and the supplied CPU emulator - a program that runs binary Hack code on a simulated Hack platform. **As mentioned above, make sure you have replaced the Compilers.jar and Hack.jar files in ../nand2tetris/tool/bin/lib with the new ones.**

Two other related and useful resources are the supplied Assembler Tutorial and CPU Emulator Tutorial. We recommend going through these tutorials before starting to work on this project.

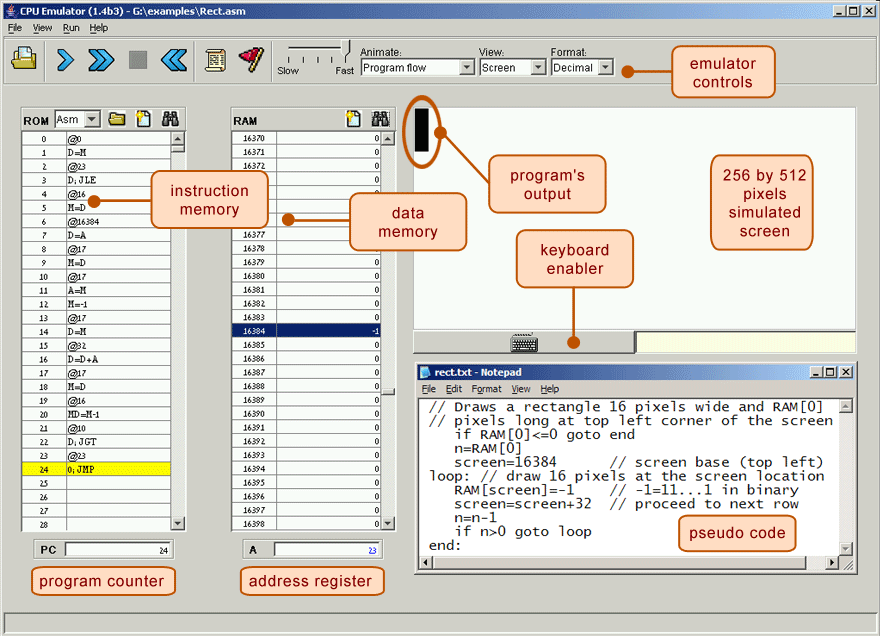
*Debugging tip: The Hack language is case-sensitive. A common error occurs when one writes, say, "@foo" and "@Foo" in different parts of one's program, thinking that both labels are treated as the same symbol. In fact, the assembler treats them as two different symbols. This bug is difficult to detect,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



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 only 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.

**High-level pseudocode for Mult.asm**

Function Mult(R0,R1){

int result, counter;

//You can create and access symbols with “@<*symbol name*>” command

counter=<*some number*>;

// Think about what is the correct number

result=0;

while(counter >=0 ){

int16 temp1 = 1<<counter; //1 shifts left this many bits; here << is used as a traditional symbol of left-shift in high-level languages

if(R1 & temp1 > 0 ) {

//Think about what does the above expression do?

result+=R0<<counter;

}

counter--;

}

return result; //Put this in R2

}

for(;;) //Infinite loop