IWKS 3300: NAND to Tetris
Spring 2019

John K. Bennett

Virtual Machine Part 2: Program Control

Foundations of Global Networked Computing:
Building a Modern Computer From First Principles

This course is based upon the work of Noam Nisan and Shimon Schocken. More information can be found at (www.nand2tetris.org).
Where We Are

Abstract design
- Human Thought (Chapters 9, 12)
  - Machine Code (Chapters 4 - 5)
  - Computer Architecture (Chapters 4 - 5)
  - Hardware Platform (Chapters 1 - 3)
  - Gate Logic (Chapters 1 - 3)

Software Hierarchy
- Compiler (H.L. Language & Operating Sys. (Chapters 10 - 11))
  - Virtual Machine (Chapters 7 - 8)

Hardware Hierarchy
- Chips & Logic Gates (abstract interface)
  - Assembler (abstract interface)

We are (still) here
The Big Picture

VM language

CISC machine language
RISC machine language
Other digital platforms, each equipped with its VM implementation
Any computer
Hack computer

A Java-based emulator is included in the course software suite
Implemented in Projects 7-8

Some language
Some Other language
Jack language

Some compiler
Some Other compiler
Jack compiler

Chapters 1-6
Chapters 7-8
Chapters 9-13
Our Game Plan

**Goal:** Specify and implement a VM model and language:

<table>
<thead>
<tr>
<th>Arithmetic / Boolean commands</th>
<th>Program flow commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>label (declaration)</td>
</tr>
<tr>
<td>sub</td>
<td>goto (label)</td>
</tr>
<tr>
<td>neg</td>
<td>if-goto (label)</td>
</tr>
<tr>
<td>eq</td>
<td></td>
</tr>
<tr>
<td>gt</td>
<td></td>
</tr>
<tr>
<td>lt</td>
<td></td>
</tr>
<tr>
<td>and</td>
<td></td>
</tr>
<tr>
<td>or</td>
<td></td>
</tr>
<tr>
<td>not</td>
<td></td>
</tr>
</tbody>
</table>

**Memory access commands**

- pop x (pop into x, a variable)
- push y (push y, a variable or constant)

**Program flow commands**

- **Last Week**
  - label (declaration)
  - goto (label)
  - if-goto (label)

- **Today**
  - function (declaration)
  - call (a function)
  - return (from a function)

**Our game plan today:**

(a) describe the VM abstraction (above)
(b) suggest how to implement it over the Hack platform.
The Compilation Challenge

Source code (high-level language)

```java
class Main {
    static int x;

    function void main() {
        // Inputs and multiplies two numbers
        var int a, b, c;
        let a = Keyboard.readInt("Enter a number");
        let b = Keyboard.readInt("Enter a number");
        let c = Keyboard.readInt("Enter a number");
        let x = solve(a, b, c);
        return;
    }
}

// Solves a quadratic equation (sort of)
function int solve(int a, int b, int c) {
    var int x;
    if (~(a = 0))
        x = (-b + sqrt(b*b - 4*a*c)) / (2 * a);
    else
        x = -c / b;
    return x;
}
```

Target code

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111100000010000
0000000000010000
1111110000010000
0000000000010001
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000010001
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000010001
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000010001
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000010001
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000010001
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000010001
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000010001
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000010001
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000010001
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000010001
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000010001
0000000000010000
1110111111001000...
```

Our ultimate goal:
Translate high-level programs into executable code.

Let’s focus here for a bit
We will develop the Jack language compiler later in the course.

We now turn to describe how to complete the implementation of the VM language.

That is -- how to translate each VM command into assembly commands that perform the desired semantics.
Typical compiler’s source code input:

```
// Computes x = (-b + sqrt(b^2 - 4*a*c)) / 2*a
if (~(a = 0))
  x = (-b + sqrt(b * b - 4 * a * c)) / (2 * a)
else
  x = -c / b
```

How to translate this high-level code into assembly language?

- In a two-tier compilation model, the overall translation challenge is broken between a front-end compilation stage and a subsequent back-end translation stage.
- In our Hack-Jack platform, all of the above sub-tasks (handling arithmetic / Boolean expressions and program flow / function calling commands) are done by the back-end, i.e. by the VM translator.
Program Flow Commands in the VM Language

**VM code example:**

```vm
function mult 1
    push constant 0
    pop local 0
label loop
    push argument 0
    push constant 0
    eq
if-goto end
    push argument 0
    push 1
    sub
    pop argument 0
    push argument 1
    push local 0
    add
    pop local 0
    goto loop
label end
    push local 0
return
```

In the VM language, the program flow abstraction is delivered using three commands:

- **label c**  // label declaration
- **goto c**   // unconditional jump to the
   // VM command following the label c
- **if-goto c**  // pops the topmost stack element;
   // if it’s not zero, jumps to the
   // VM command following the label c

**How to translate these 3 abstractions into assembly?**

- **label** declarations and **goto** directives can be effected directly by assembly commands
- The VM Translator must emit one or more assembly commands that performs these semantics on the Hack platform
- Today’s lecture will describe how
Subroutines are a programming artifact of most modern languages

- Basic idea: the given language can be extended at will by user-defined commands (aka subroutines / functions / methods ...)
- Important: the language’s built-in primitive commands and the user-defined commands have the same look-and-feel
- This transparent extensibility is the one of the most important abstractions provided by high-level programming languages
- The challenge: how to implement this abstraction, allowing program control to flow seamlessly from one subroutine to another and back

```c
// Compute x = (-b + sqrt(b^2 - 4*a*c)) / 2*a
if ~(a = 0))
    x = (-b + sqrt(b * b - 4 * a * c)) / (2 * a)
else
    x = -c / b
```
The invocation of the VM’s primitive commands and subroutines follow the same rules:

- The caller pushes the necessary argument(s) onto the stack and calls the command / function for its effect.
- The called command / function is responsible for removing the argument(s) from the stack, and for popping onto the stack the result of its execution.

**Called code, aka “callee” (example)**

```plaintext
function mult 1
  push constant 0
  pop local 0 // result (local 0) = 0
label loop
  push argument 0
  push constant 0
  eq
  if-goto end // if arg0 == 0, jump to end
  push argument 0
  push constant 1
  sub
  pop argument 0 // arg0--
  push argument 1
  push local 0
  add
  pop local 0 // result += arg1
  goto loop
label end
  push local 0 // push result
  return
```

**Calling code (example)**

```plaintext
... // computes ((7 + 2) * 3) - 5
push constant 7
push constant 2
add
push constant 3
call mult 2
push constant 5
sub
...
```

VM subroutine call-and-return commands
**Q:** Why this particular syntax?

**A:** Because it simplifies the VM implementation (as we will see in a moment).
Function Call-and-return Conventions

### Calling function

```plaintext
function demo 3
  ...
push constant 7
push constant 2
add
push constant 3
call mult 2
...
```

### called function aka “callee” (example)

```plaintext
function mult 1
  push constant 0
  pop local 0 // result (local 0) = 0
label loop
  ...
label end
push local 0 // push result
return
```

Although not obvious in this example, every VM function has a private set of 5 memory segments (local, argument, this, that, pointer)

These resources exist only as long as the function is running.

### Call-and-return programming convention

- The caller must push the necessary argument(s), call the callee, and wait for it to return
- Before the callee terminates (returns), it must push a return value
- At the point of return, the callee’s resources are recycled, the caller’s state is re-instated, execution continues from the command just after the call
- Caller’s net effect: the arguments were replaced by the return value (just like with primitive commands)

### Behind the scene

- Recycling and re-instating subroutine resources and states is a major headache
- Some agent (either the VM or the compiler) should manage it behind the scene “like magic”
- In our implementation, the magic is VM / stack-based.
The Hack VM Function Call-and-return Protocol

The caller’s view:

- Before calling a function $g$, I must push onto the stack as many arguments as are needed by $g$.
- Next, I invoke the function using the command `call g nArgs`.
- After $g$ returns:
  - The arguments that I pushed before the call have been removed from the stack, and a return value is (always) present on the top of the stack.
  - All my memory segments (local, argument, this, that, pointer) have been restored to the same state as they were before the call to $g$.

The callee’s ($g$’s) view:

- When I start executing, my argument segment has been initialized with actual argument values passed by the caller.
- My local variables segment has been allocated and initialized to zero.
- The static segment that I see has been set to the static segment of the VM file to which I belong, and the working stack that I see is empty.
- Before exiting, I must push a value onto the stack and then use the command `return`.

Blue = VM function writer’s responsibility

Green = “magic,” provided by the VM implementation (you)
When function $f$ calls function $g$, the VM implementation must:

- Save the return address; this is the address of the instruction just after the call
- Save all virtual segment pointers of $f$
- Allocate, and initialize to 0, as many local variables as will be needed by $g$
- Set the local and argument segment pointers of $g$
- Transfer control to $g$.

When $g$ terminates and control returns to $f$, the VM implementation:

- Clears $g$’s arguments and other junk from the stack (by resetting SP)
- Restores the virtual segment pointers of $f$
- Transfers control back to $f$
  (jump to the saved return address).
Implementation of the VM's Stack on the Hack RAM

**Global stack:**
the entire RAM area dedicated for holding the stack

**Working stack ("stack frame"):**
The stack that the current function sees

- At any point of time, only one function (the *current function*) is executing; other functions may be waiting up the calling chain
- Shaded areas: irrelevant to the current function
- The current function sees only the working stack, and has access only to its memory segments
- The rest of the stack holds the saved states of all the functions up the calling hierarchy.

**ARG**
- argument 0
- argument 1
- . . .
- argument nArgs-1
- saved returnAddress
- saved LCL
- saved ARG
- saved THIS
- saved THAT
- local 0
- local 1
- . . .
- local nVars-1

**LCL**
- working stack of the current function

**SP**
- frames of all the functions up the calling chain

Arguments pushed by the caller for the current function

Saved state of the calling function. Used by the VM implementation to restore the segments of the calling function just after the current function returns.

Local variables of the current function
Implementing the \textit{call g nArgs} Instruction

// In the course of implementing the code of $f$
// (the caller), we arrive to the command \textit{call g nArgs}.
// we assume that $nArgs$ arguments have been pushed
// onto the stack prior to the call.
// Now we generate a new symbol for the returnAddress,
// and save the current stack frame segment pointers:
push returnAddress // saves the return address
push LCL           // saves the LCL of $f$
push ARG           // saves the ARG of $f$
push THIS          // saves the THIS of $f$
push THAT          // saves the THAT of $f$
ARG = SP-nArgs-5   // repositions ARG for $g$
LCL = SP           // repositions LCL for $g$
goto g             // transfers control to $g$
returnAddress:     // the generated symbol

The VM implementation must emit the above logic in Hack assembly language, as we will now describe.
Implementing the function \textit{g nVars} Instruction

\begin{itemize}
  \item \textbf{function g nVars}
  \end{itemize}

// To implement the command \textit{function g nVars},
// we first initialize the local variables:

g:
  repeat nVars times:
    push 0
  // ...
  // the rest of g's code ...
  // ...
return

The VM implementation must emit the above logic in Hack assembly language.
Implementing the \textit{return} Instruction

// In the course of implementing the code of \textit{g},
// we eventually arrive at the command \textit{return}.
// We assume that a (1) return value has been pushed
// onto the stack (push zero if no return value).
// We effect the following logic:
frame = LCL          // (2) frame is a temp. variable
retAddr = *(frame-5) // (3) retAddr is a temp. var.
*ARG = RAM[SP-1]     // (4) repositions return value
SP=ARG+1            // (5) restores caller’s SP
THAT = *(frame-1)   // (6) restores caller’s THAT
THIS = *(frame-2)    // (6) restores caller’s THIS
ARG = *(frame-3)     // (6) restores caller’s ARG
LCL = *(frame-4)     // (6) restores caller’s LCL
goto retAddr        // (7) goto returnAddress

The VM implementation must emit the above logic in Hack assembly language.
## “Standard” Hack Memory Map

<table>
<thead>
<tr>
<th>RAM[n]</th>
<th>Register n</th>
<th>Seg. Name</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>R0</td>
<td>SP</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>1</td>
<td>R1</td>
<td>LCL Base</td>
<td>Ptr. To LCL Vars (in Stack Frame)</td>
</tr>
<tr>
<td>2</td>
<td>R2</td>
<td>ARG Base</td>
<td>Ptr. To Args (in Stack Frame)</td>
</tr>
<tr>
<td>3</td>
<td>R3</td>
<td>THIS Base</td>
<td>Pointer 0 (ptr to This segment)</td>
</tr>
<tr>
<td>4</td>
<td>R4</td>
<td>THAT Base</td>
<td>Pointer 1 (ptr to That segment)</td>
</tr>
<tr>
<td>5</td>
<td>R5</td>
<td>TEMP</td>
<td>Temp 0</td>
</tr>
<tr>
<td>6</td>
<td>R6</td>
<td></td>
<td>Temp 1</td>
</tr>
<tr>
<td>7</td>
<td>R7</td>
<td></td>
<td>Temp 2</td>
</tr>
<tr>
<td>8</td>
<td>R8</td>
<td></td>
<td>Temp 3</td>
</tr>
<tr>
<td>9</td>
<td>R9</td>
<td></td>
<td>Temp 4</td>
</tr>
<tr>
<td>10</td>
<td>R10</td>
<td></td>
<td>Temp 5</td>
</tr>
<tr>
<td>11</td>
<td>R11</td>
<td></td>
<td>Temp 6</td>
</tr>
<tr>
<td>12</td>
<td>R12</td>
<td></td>
<td>Temp 7</td>
</tr>
<tr>
<td>13</td>
<td>R13</td>
<td>N/A</td>
<td>GP register</td>
</tr>
<tr>
<td>14</td>
<td>R14</td>
<td>N/A</td>
<td>GP register</td>
</tr>
<tr>
<td>15</td>
<td>R15</td>
<td>N/A</td>
<td>GP register</td>
</tr>
<tr>
<td>16-255</td>
<td>N/A</td>
<td>Static</td>
<td>static vars.</td>
</tr>
<tr>
<td>256-2047</td>
<td>N/A</td>
<td>Stack</td>
<td>Stack (including LCL and ARG)</td>
</tr>
<tr>
<td>2048-16383</td>
<td>N/A</td>
<td>Heap</td>
<td>Heap (including THIS and THAT)</td>
</tr>
<tr>
<td>16384-24575</td>
<td>N/A</td>
<td>Scrn. &amp; Kbd.</td>
<td>Memory-Mapped I/O</td>
</tr>
</tbody>
</table>

*The memory addresses in purple can be changed if there is a reason to do so.
Bootstrapping

- A high-level jack program (application) is a set of class files.
- By convention, one jack class must be called Main, and this class must have at least one function, called main.
- The VM Implementation must therefore call Main.main when Jack program begins to execute

**Implementation:**

- After the program is compiled, each class file is translated into a .vm file
- The operating system is also implemented as a set of .vm files (“libraries”) that co-exist alongside the program’s .vm files
- One of the OS libraries, called Sys.vm, includes a method called init. The Sys.init function starts with some OS initialization code (we’ll deal with this later, when we discuss the OS), then calls Main.main
- Thus, to bootstrap, the VM implementation has to effect (in Hack assembly), the following operations:

```
SP = 256       // initialize the stack pointer to 0x0100
call Sys.init  // call the function that calls Main.main
```
VM implementation over the Hack Platform

- Extends the VM implementation described in Chapter 7.
- The result: a single assembly program file with a number of agreed-upon symbols:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SP, LCL, ARG, THIS, THAT</strong></td>
<td>These predefined symbols point, respectively, to the stack top and to the base addresses of the virtual segments local, argument, this, and that.</td>
</tr>
<tr>
<td><strong>R13 - R15</strong></td>
<td>These predefined symbols can be used for any purpose.</td>
</tr>
<tr>
<td><strong>Xxx.vm</strong></td>
<td>Each static variable $j$ in a VM file <em>Xxx.vm</em> is translated into the assembly symbol <strong>Xxx.$j</strong>. In the subsequent assembly process, these symbolic variables will be allocated RAM space by the Hack assembler.</td>
</tr>
<tr>
<td><strong>functionName$label</strong></td>
<td>Each label $b$ command in a VM function $f$ should generate a globally unique symbol “$f$b” where “$f$” is the function name and “$b$” is the label symbol within the VM function’s code. When translating goto $b$ and if-goto $b$ VM commands into the target language, the full label specification “$f$b” must be used instead of “$b$”.</td>
</tr>
<tr>
<td><strong>and...</strong></td>
<td></td>
</tr>
<tr>
<td><strong>filename$$label</strong></td>
<td></td>
</tr>
<tr>
<td><strong>(FunctionName)</strong></td>
<td>Each VM function $f$ should generates a symbol “$f$” that refers to its entry point in the instruction memory of the target computer.</td>
</tr>
<tr>
<td><strong>(return-address)</strong></td>
<td>Each VM function call should generate and insert into the translated code a unique symbol that serves as a return address, namely the memory location (in the target platform’s memory) of the command following the function call.</td>
</tr>
</tbody>
</table>
**Parser**: Handles the parsing of a single .vm file, and encapsulates access to the input code. It reads VM commands, parses them, and provides convenient access to their components. In addition, it removes all white space and comments.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Arguments</th>
<th>Returns</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor</td>
<td>Input file / stream</td>
<td>--</td>
<td>Opens the input file/stream and gets ready to parse it.</td>
</tr>
<tr>
<td>hasMoreCommands</td>
<td>--</td>
<td>boolean</td>
<td>Are there more commands in the input?</td>
</tr>
<tr>
<td>advance</td>
<td>--</td>
<td>--</td>
<td>Reads the next command from the input and makes it the current command. Should be called only if hasMoreCommands is true. Initially there is no current command.</td>
</tr>
<tr>
<td>commandType</td>
<td>--</td>
<td>C_ARITHMETIC, C_PUSH, C_POP, C_LABEL, C_GOTO, C_IF, C_FUNCTION, C_RETURN, C_CALL</td>
<td>Returns the type of the current VM command. C_ARITHMETIC is returned for all the arithmetic commands.</td>
</tr>
<tr>
<td>arg1</td>
<td>--</td>
<td>string</td>
<td>Returns the first arg. of the current command. In the case of C_ARITHMETIC, the command itself (add, sub, etc.) is returned. Should not be called if the current command is C_RETURN.</td>
</tr>
<tr>
<td>arg2</td>
<td>--</td>
<td>int</td>
<td>Returns the second argument of the current command. Should be called only if the current command is C_PUSH, C_POP, C_FUNCTION, or C_CALL.</td>
</tr>
</tbody>
</table>
**Book’s VM Translator Implementation: CodeWriter (Ch. 7)**

**CodeWriter: Translates VM commands into Hack assembly code.**

<table>
<thead>
<tr>
<th>Routine</th>
<th>Arguments</th>
<th>Returns</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor</td>
<td>Output file / stream</td>
<td>--</td>
<td>Opens the output file/stream and gets ready to write into it.</td>
</tr>
<tr>
<td>setFileName</td>
<td>fileName (string)</td>
<td>--</td>
<td>Informs the code writer that the translation of a new VM file is started.</td>
</tr>
<tr>
<td>writeArithmetic</td>
<td>command (string)</td>
<td>--</td>
<td>Writes the assembly code that is the translation of the given arithmetic command.</td>
</tr>
<tr>
<td>WritePushPop</td>
<td>command (C_PUSH or C_POP), segment (string), index (int)</td>
<td>--</td>
<td>Writes the assembly code that is the translation of the given command, where command is either C_PUSH or C_POP.</td>
</tr>
<tr>
<td>Close</td>
<td>--</td>
<td>--</td>
<td>Closes the output file.</td>
</tr>
</tbody>
</table>

Comment: More routines will be added to CodeWriter in Project 8 (next slide).
<table>
<thead>
<tr>
<th>Routine</th>
<th>Arguments</th>
<th>Returns</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>writeInit</td>
<td>--</td>
<td>--</td>
<td>Writes the bootstrap code at the start of the asm output file</td>
</tr>
<tr>
<td>writeLabel</td>
<td>label (string)</td>
<td>--</td>
<td>Writes the assembly code that is the translation of the <code>label</code> command.</td>
</tr>
<tr>
<td>writeGoto</td>
<td>label (string)</td>
<td>--</td>
<td>Writes the assembly code that is the translation of the <code>goto</code> command.</td>
</tr>
<tr>
<td>WriteIf</td>
<td>label (string)</td>
<td>--</td>
<td>Writes the assembly code that is the translation of the <code>if-goto</code> command.</td>
</tr>
<tr>
<td>WriteCall</td>
<td>functionName (string) numArgs (int)</td>
<td>--</td>
<td>Writes the assembly code that is the translation of the <code>call</code> command.</td>
</tr>
<tr>
<td>WriteReturn</td>
<td>--</td>
<td>--</td>
<td>Writes the assembly code that is the translation of the <code>return</code> command.</td>
</tr>
<tr>
<td>WriteFunction</td>
<td>functionName (string) numArgs (int)</td>
<td>--</td>
<td>Writes the assembly code that is the translation of the <code>function</code> command.</td>
</tr>
</tbody>
</table>
enum VM_CommandType { VM_NO_COMMAND = 0, C_ARITHMETIC, C_PUSH, C_POP, C_LABEL, C_GOTO, C_IF, C_FUNCTION, C_RETURN, C_CALL };

... parseLine(); // sets commandType, vm language command, arg1 and arg2 if valid ...
switch (commandType) {
    case VM_CommandType.C_ARITHMETIC:
        writeArithmetic();
        break;
    case VM_CommandType.C_PUSH:
        WritePushPop(shortFileName);
        break;
    case VM_CommandType.C_POP:
        WritePushPop(shortFileName);
        break;
    case VM_CommandType.C_LABEL:
        WriteLabel(shortFileName);
        break;
    case VM_CommandType.C_GOTO:
        WriteGoto(shortFileName);
        break;
    case VM_CommandType.C_IF:
        WriteIf(shortFileName);
        break;
    case VM_CommandType.C_FUNCTION:
        functionName = null; // make sure that we are starting fresh
        WriteFunction();
        break;
    case VM_CommandType.C_RETURN:
        WriteReturn();
        break;
    case VM_CommandType.C_CALL:
        WriteCall();
        break;
    case VM_CommandType.VM_NO_COMMAND:
        // Do nothing.
        break;
    default:
        errorFile.WriteLine("Line No: {0}, illegal command: {1}" , lineNo, command);
        break;
private void WriteLabel(string filename)
{
    // Labels only have scope within the function in which they are declared, except
    // there may be VM code outside of a function definition,
    // so a function actually might not be declared.

    // Make a function/file unique name for the label.
    // If no function has been declared, the name will be
    // FileName$$arg1; Otherwise it will be FunctionName$arg1.

    if (isValidLabel(arg1))
    {
        if (functionName == null)
            outFile.WriteLine("(" + filename + "$$" + arg1 + ")");
        else
            outFile.WriteLine("(" + functionName + "$" + arg1 + ")");
    }
    else
    {
        errorFile.WriteLine("Line No: {0}, illegal label: {1}", lineNo, arg1);
    }
}
private void WriteFunction()
{
    // arg1 is the function name; arg2 is the number of locals (number as a string)
    if (isValidLabel(arg1)) // function names have the same restrictions as labels
    {
        functionName = arg1;
        // write the function label
        outFile.WriteLine("@{0}", arg1);
        // now push arg2 zeros onto the stack
        int numLocals = Int32.Parse(arg2);
        if (numLocals != 0) // nothing to do if there are no args
        {
            // set things up
            string loop = NextLabel();
            outFile.WriteLine("@{0}", arg2);
            outFile.WriteLine("D=-A");
            outFile.WriteLine("({0})", loop);
            // Push zeroes onto stack; we save one instruction with a little cleverness here
            outFile.WriteLine("@SP");
            outFile.WriteLine("AM=M+1"); // A = SP + 1; SP = SP + 1
            outFile.WriteLine("A=A-1"); // A = orig SP
            outFile.WriteLine("M=0"); // M[orig SP] = 0
            // are we done?
            outFile.WriteLine("@{0}", loop);
            outFile.WriteLine("D=D+1;JLT");
        }
    }
    else
    {
        // bad function name
        errorFile.WriteLine("Line No: {0}, illegal function name: {1}", lineNo, arg1);
    }
}
Benefits of the VM approach:

- Code transportability: compiling for different platforms requires replacing only the VM implementation.
- Language inter-operability: code of multiple languages can be shared using the same VM.
- Common software libraries.
- Code mobility.
- Some virtues of the modularity implied by the VM approach to program translation:
  - Improvements in the VM implementation are shared by all compilers above it.
  - Every new digital device with a VM implementation gains immediate access to an existing software base.
  - New programming languages can be implemented more readily using simple compilers.

Benefits of Managed Code:

- Security.
- Array bounds, index checking, ...
- Add-on code.
- Portability.

VM Cons:

- Performance.
- Debugging complexity.
- Low-level device access.