7. The Virtual Machine II: Flow Control[1]

It's like building something where you don't have to order the cement.
You can create a world of your own, your own environment,
and never leave this room.

(Ken Thompson, 1983 Turing Award lecture)

Chapter 6 introduced the notion of a virtual machine (VM), and ended with the construction of a basic VM implementation over the Hack platform. In this chapter we continue to describe the virtual machine abstraction, language, and implementation. In particular, we focus on a variety of *stack-based* mechanisms designed to handle nested subroutine calls (procedures, methods, functions) of procedural or object-oriented languages. As the chapter progresses, we extend the previously built basic VM implementation, ending with a full-scale VM translator. This translator will serve as the backend of the compiler that we will build in chapters 9 and 10, following the introduction of a high-level object-based language in chapter 8.

In any "Great Gems in Computer Science" contest, *stack processing* will be a strong finalist. The previous chapter showed how arithmetic and Boolean expressions could be calculated by elementary stack operations. In this chapter we show how this remarkably simple data structure can further support remarkably complex tasks like dynamic memory management, nested subroutine processing, and recursion. Stack processing is one of the secret weapons that make the implementation of modern programming languages less formidable than seen at first sight.

7.1 Background

The last chapter described the data manipulation commands of a stack-based virtual machine: arithmetic operations performed on stack elements as well as commands for moving data between the stack and the memory. In addition to these data-oriented commands, every virtual machine must have some form of control-oriented commands, allowing the construction of loops, conditional execution, and subroutine calls. While the implementation of the flow control commands needed to support loops and conditional execution is rather simple, the implementation of subroutine calls is rather challenging. Hence, virtual machines that implement subroutine calls as a primitive feature deliver a significant and useful abstraction. The Java VM as well as the VM we describe in this book both provide such a primitive. As will be seen throughout the chapter, the implementation of subroutine calls can be elegantly modeled on a *stack* structure.

Program flow

The default execution of computer programs is linear, one command after the other. In some cases this sequential flow needs to be broken, e.g. in order to embark on another iteration of a loop. In low-level programming, this is done by instructing the computer to continue the program execution at some specified part of the program other than in the next instruction. The re-direction directive is implemented by a "*goto destination*" command, also called "*jump*" or

[1] From *The Elements of Computing Systems*, Nisan & Schocken, MIT Press, forthcoming in 2003, www.idc.ac.il/csd

"*branch*". The destination specification can take several forms, the most primitive being the physical address of the instruction that should be executed next. A slightly more abstract redirection mechanism is established by assigning a symbolic label to the destination command and specifying this label (rather than the physical address) as the jump target. This variation requires that the language will also have some sort of a labeling command.

This basic *goto* mechanism described above can be easily modified to affect *conditional branching* as well. Instead of instructing the computer to jump to some destination unconditionally, a *conditional branch* command instructs to take the jump only if a certain condition is true; if the condition is false, the regular flow of control should continue. Different virtual machines provide different ways to specify the jump condition. In stack-based machines, the simplest and most natural approach is to condition the jump on the value of the stack's top element: if it's not zero, jump to the specified destination; otherwise execute the next command in the program.

The resulting set of low-level program flow commands (*labeling*, *goto label*, and *if condition then goto label*) can be used by compilers to translate all the conditional and repetition constructs found in high-level programming languages. Fig. 7-1 gives two typical examples.

High-level source code	***Compiled low-level code***

`if (cond)`
`  s1`
`else`
`  s2`
`…`
	`  code for computing cond`
	`  if-false-goto L1`
	`  code for executing s1`
	`  goto L2`
	`label L1`
	`  code for executing s2`
	`label L2`
	`  …`

`while (cond)`
`  s1`
`…`
	`label L1`
	`  code for computing cond`
	`  if-false-goto L2`
	`  code for executing s1`
	`  goto L1`
	`label L2`
	`  …`

Figure 7-1: VM programming: From high-level code to pseudo VM code.

Subroutine Calls

Any programming language is characterized by a fixed repertoire of elementary commands. The key abstraction mechanism provided by modern languages is the freedom to extend this repertoire with high-level operations, designed to meet various programming needs. Each high-level operation has an *interface* specifying how it can be used, and an *implementation* consisting

of elementary commands and previously defined high-level operations. In procedural languages, the high-level operations are called *subroutines*, *procedures*, or *functions*. In object-oriented languages they are usually called *methods,* and are typically grouped into *classes*. In this chapter we will use the term *subroutine* to refer to all of these high-level programming constructs.

The use of a subroutine is typically referred to as a *call* operation. Ideally, the part of the program that calls the subroutine -- the *caller* -- should treat the subroutine like any other basic operation in the language. To illustrate, the caller typically contains a sequence of commands like <*c1*, *c2*, `call` *s1*, *c3*, `call` *s2*, *c4*, …>, where the *c*'s are elementary commands and the *s*'s are subroutine names. In other words, the caller assumes that the code of the called subroutine will get executed -- somehow -- and that following the subroutine's termination the flow of control will return -- somehow -- to the next instruction in the caller's code. The freedom to ignore these implementation details enables the programmer to write programs in abstract terms, using high-level operations closer to the world of algorithmic thought than to the world of machine execution. Of course the more abstract is the high level, the more work we have to do at the low level. In particular, in order to support subroutine calls, VM implementations must handle several issues:

- Passing parameters to the called subroutine, and optionally returning a value from the subroutine back to the caller;
- Jumping to execute the subroutine's code;
- Allocating memory space for the local variables of the called subroutine, and freeing the memory when it is no longer needed;
- When the called subroutine terminates, returning (jumping back) to the command following the call operation in the caller's code.

These issues must be handled in a way that takes into account that subroutine calls can be arbitrarily nested, i.e. one subroutine may call another subroutine, which may then call another subroutine, and so on and so forth. To add to the complexity, we also need to support *recursion*, meaning that subroutines are allowed to call themselves, and each recursion level must be executed independently of the other calls. To sum up, the low-level handling of subroutine calls is an intricate task.

The property that makes this task tractable is the inherently hierarchical nature of any multi-subroutine program: the called subroutine must complete its execution before the caller can resume its own execution. This protocol implies a *Last-In-First-Out* (LIFO) structure, resembling (conceptually) a *stack* of active subroutines. All the layers in the stack are waiting for the top layer to complete its execution, at which point the stack become shorter and execution resumes at the level just below the previous top layer. Indeed, users of high-level programming languages often encounter terms like "function-call-stack," "stack overflow," and so on.

Fig. 7.2 illustrates a method calls pattern in some high-level program, along with some run-time checkpoints and the states of the abstract method-call stack associated with them.

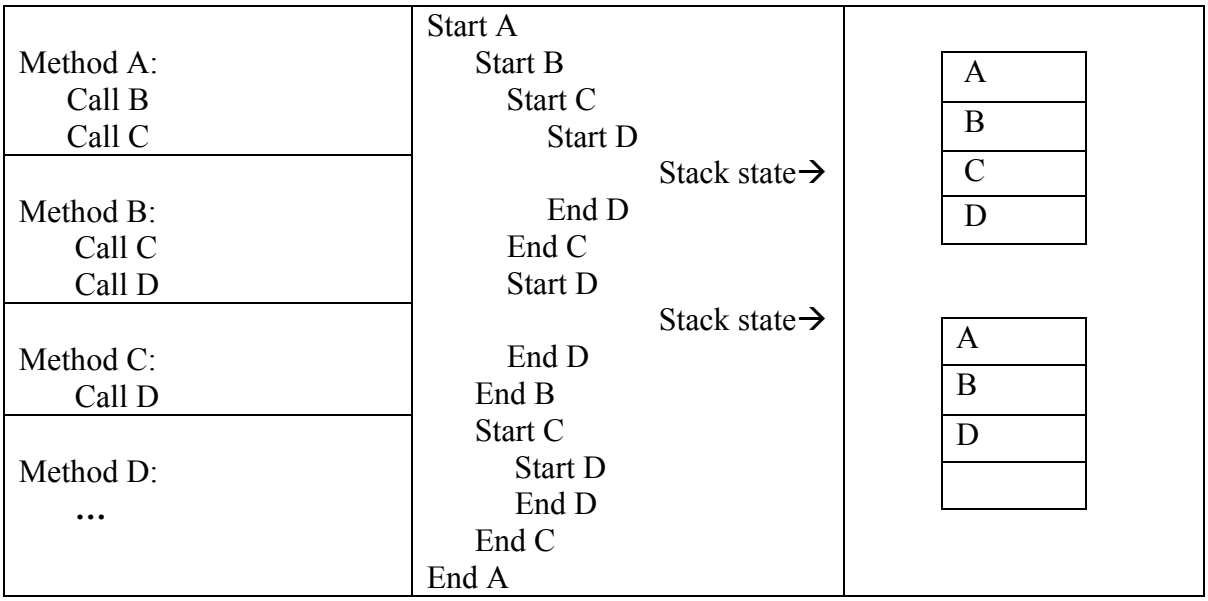

Figure 7-2: Subroutine calls and the abstract call-stack states generated by their execution.

We now describe some of the points that need to be addressed when providing such a call-and-return subroutine mechanism. Any particular VM implementation will have to handle all of them, as well as take care of coordinating between these tasks as well as with the low level details of implementation. The complete set of data that is kept for each subroutine invocation is usually called a "*frame*", and it is these frames that are actually kept in the call-stack.

Return Address

The VM implementation of the "`call subName`" command is straightforward. Since the name of the target subroutine is specified in the command, the VM implementation simply has to resolve the name to an address in memory, and then jump to execute the instruction stored in that address.

Returning from the called subroutine via a "`return`" command is trickier, as the command specifies no return address. Indeed, the caller's anonymity is inherent in the very notion of a subroutine. Subroutines like `sqrt(x)` or `modulu(x,y)` are designed to serve many unknown callers, implying that the return address cannot be part of their code. Thus a "`return`" command should be interpreted as follows: re-direct the program's execution to the command following the command that called the current subroutine, wherever this command may be in the program's code. The memory location to which we have to return is called *return address*.

One way to implement this calling protocol is to have the VM implementation save the return address in advance. This piece of information must be saved just before the subroutine is called, and retrieved just after the subroutine exits. This store-and-recall setting lends itself perfectly to a *stack* storage paradigm: the VM implementation can push the return address onto the stack when a subroutine is called, and pop it from the stack when the subroutine returns.

Parameter Passing

An important requirement in well-designed languages is that the high-level operations defined by the programmer will have the same "look and feel" as that of elementary commands. Consider for example the operations *add* and of *raise to a power*. VM implementations will typically feature the former as an elementary operation, while the latter may be written as a subroutine. In spite of their different implementations, we would like to use both operations in the same way. Thus, assuming that we have already written a `Power(x,y)` subroutine that computes x to the y'th power, we would like to be able to write VM code segments like those depicted in Prog. 7-3.

```
// x+3
push x
push 3
add
```

```
// x^3
push x
push 3
call power
```

```
// (x^3+2)^y
push x
push 3
call power
push 2
add
push y
call power
```

PROGRAM 7-3: VM elementary commands (left) and high-level operations (middle) have the same look-and-feel in terms of arguments usage and return values. Thus they can be easily mixed together (right).

Note that from the caller's perspective, any subroutine -- no matter how complex -- is viewed and treated as a black box operation. In particular, just like with primitive arithmetic VM commands, the caller expects the subroutine to remove its arguments from the stack and replace them with a return value (which may be ignored). Thus, the caller can pass the arguments to the subroutine by pushing them onto the stack; the called subroutine pops the arguments from the stack, carries out its computation, and then pushes a return value onto the stack. The result is a simple and natural parameter passing protocol requiring no memory beyond the already available stack structure.

Allocation of Local Variables

Subroutines usually use local variables for temporary storage. These local variables must be stored in memory only during the subroutine call's lifetime, i.e. from the point the subroutine starts executing until a return command is encountered, at which point the memory space allocated to the local variables can be freed. Further, when the subroutine is used recursively, each recursion level must have its own set of local variables. How can the VM implementation effect this dynamic memory management?

Once again, the stack comes to the rescue. Although the subroutines calling chain may be arbitrarily deep as well as recursive, only one subroutine (or subroutine instance in case of recursion) executes at the end of the chain, while all the other subroutines up the calling chain are waiting. The VM implementation can exploit this last-in-first-out processing model by storing the local variables of all the waiting subroutines on the stack, and reinstating them when control returns to the subroutine to which they belong.

.

7.2 VM Specification, Part II

This section extends the basic VM Specification (section 6.2) with *program flow* and *function calling* commands. This completes the overall VM speciation.

7.2.1 Program Flow Commands

The VM language features three program flow commands:

`label c`	This command labels the current location in the function's code. Only labeled locations can be jumped to from other parts of the program. The label `c` is an arbitrary string composed of letters, numbers, and the special characters "`_`", "`:`", and "`.`". The scope of this label is the current function.
`goto c`	This command effects a "goto" operation, causing the program to continue execution from the location labeled by c. The jump destination must be in the same function.
`if-goto c`	This command effects a "conditional goto" operation. The command pops the topmost value from the stack and goes to label c if the value is non-zero. The destination must be in the same function.

TABLE 7-4: Program flow commands.

7.2.2 Function-related Commands

Each function has a symbolic name that is used globally to call it. The function name is an arbitrary string composed of letters, numbers, and the special characters "`_`" and "`.`". (We expect that a method `m` in class `C` in some high-level language will be translated by the language compiler to a VM function named `C.m`).

The VM language features three function-related commands:

`function f n`	Here starts the code of a function named `f`, which has `n` local variables;
`call f m`	Call function `f`, stating that `m` arguments have already been pushed unto the stack;
`return`	Return to the calling function.

TABLE 7-5: Function calling commands.

7.2.2.1 The Calling Protocol

The events of calling a function and returning from a function can be viewed from three different perspectives, as follows.

The *calling function* view:

1. Before calling the function, the caller must push all the arguments unto the stack;
2. The caller invokes the function *f* using the command "`call` *f*";
3. After the function returns, all arguments have disappeared from the stack and the function's *return value* (that always exists) appears at the top of the stack;
4. All the memory segments (e.g. `arguments` and `locals`) of the caller are the same as before the call, except for the `Temp` segment that is now undefined.

The *called function* view:

1. Upon getting called, the working stack is empty, the `local` variables segment has been allocated and initialized to zero, the `argument` segment is initialized with the arguments passed by the caller, the `static` segment is set to the `static` segment of the file to which the called function belongs, and the T-segments are the same as they were in the calling function;
2. Just before returning, a return value must be pushed onto the stack.

The *VM implementation* view: The function call protocol described above is implemented by the VM implementation, as follows.

Upon ***calling*** a function, the VM implementation:

- Saves the return address and the segment pointers of the calling function, except for `temp`, which is not saved;
- Allocates, and initializes to zero, as many local variables as needed by the called function;
- Sets the `local` and `argument` segments of the called function;
- Transfers control to the called function.

Upon ***returning*** from a function, the VM implementation:

- Clears the arguments and other junk from the stack;
- Restores the `local`, `argument`, `this` and `that` segments of the calling function;
- Transfers control back to the calling function, by jumping to the saved return address.

Initialization

When the VM starts running (or is reset) the VM function named "`Sys.init`" gets executed.

7.3 Implementation

This section describes how to complete the VM implementation that we started to build in Chapter 6. We begin by describing the intricate stack structure that must be maintained by the implementation, continue to describe its standard mapping over the Hack platform, and end with design suggestions and a proposed API for the VM translator. This completes the implementation notes from Section 6.3, leading to the construction of a full-scale VM implementation.

The Global Stack

The "internal memory" of the VM is implemented by maintaining a global stack. Each time a function is called, a new block is added to the global stack. The block consists of the *arguments* that were set for the called function, a set of *pointers* used to save the state of the calling function, the *local variables* of the called function (initialized to 0), and an empty *working stack* for the called function. Importantly, the called function sees only the tip of this iceberg, , i.e. the working stack. The rest of the global stack is used only by the VM implementation.

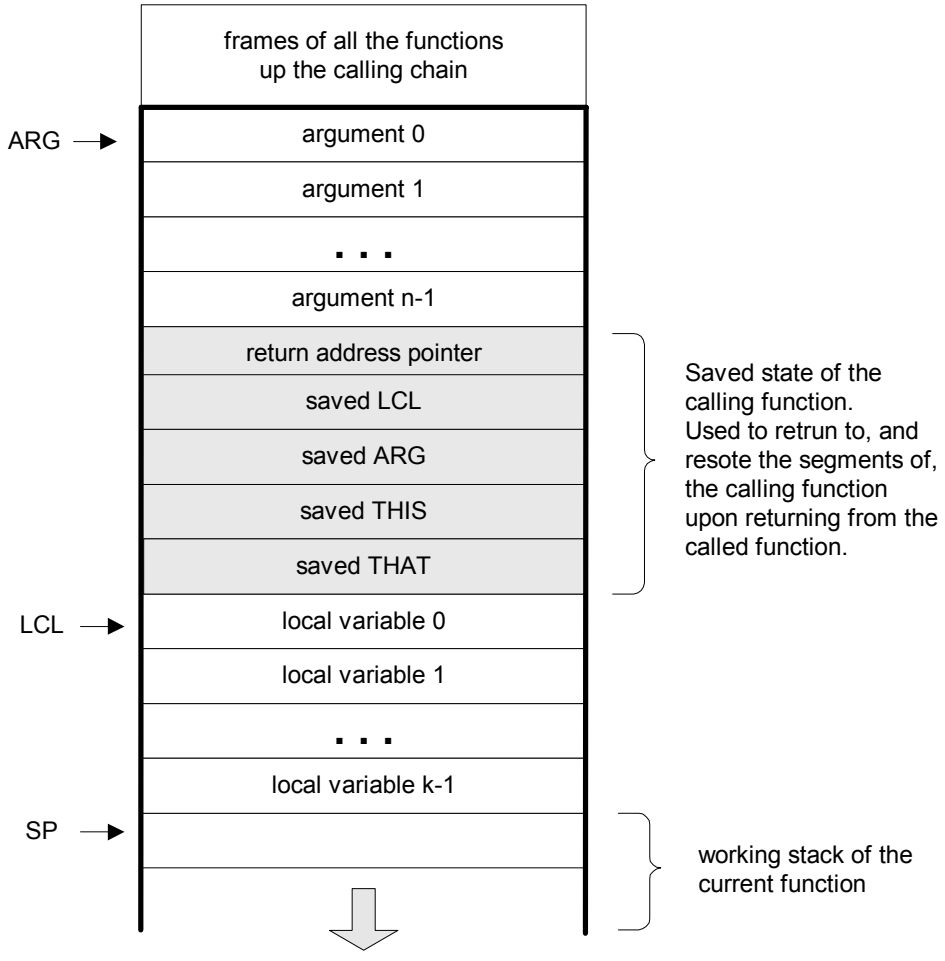

FIGURE 7-6: The global stack: the currently executing function sees only the working stack. The rest of the global stack is managed by the VM implementation behind the scene.

Example: The factorial ($n!$) of a given number n can be computed by the bottom-up iterative formula $n! = 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n$. A VM program that implements this calculation is given at the top of Fig. 7-7.

```
function: fact
    args: n
    vars: result,j
    result=1
    j=1
loop:
    j=j+1
    if j>n goto end
    result=mult(result,j)
    goto loop
end:
    return result
```

```
function: mult
  args: x,y
  vars: sum,j
  sum=0
  j=y
loop:
  if j==0 goto end
  sum=sum+x
  j=j-1
  goto loop
end:
  return sum
```

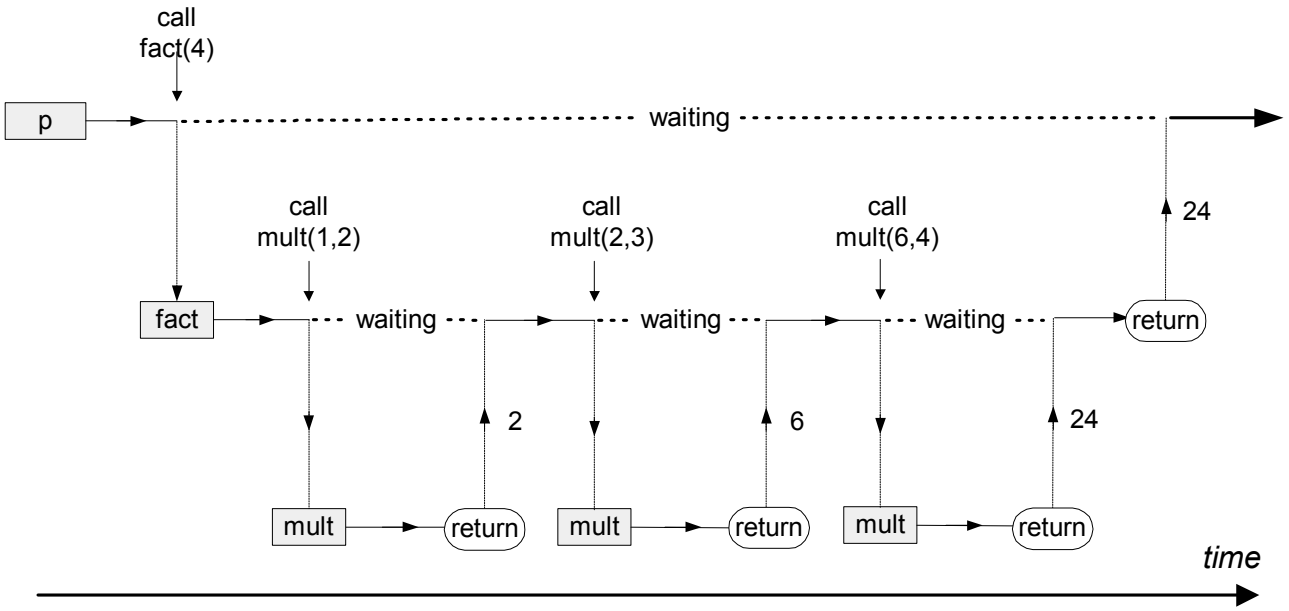

FIGURE 7-7: The life cycle of function calls: function `p` (an arbitrary function that needs factorial services) calls function `fact`, which then calls `mult` several times. Vertical arrows depict transfer of control from one function to another. Full horizontal lines depict the currently running function, whereas broken horizontal lines depict "waiting" states. At any given point of time, only one function is running, while all the functions up the calling chain are waiting for it to return. When a function returns, the function that called it resumes its execution (which typically does something useful with the value returned by the called function).

Fig. 7-8 shows some snapshots from the global stack that the VM implementation maintains while the program is running.

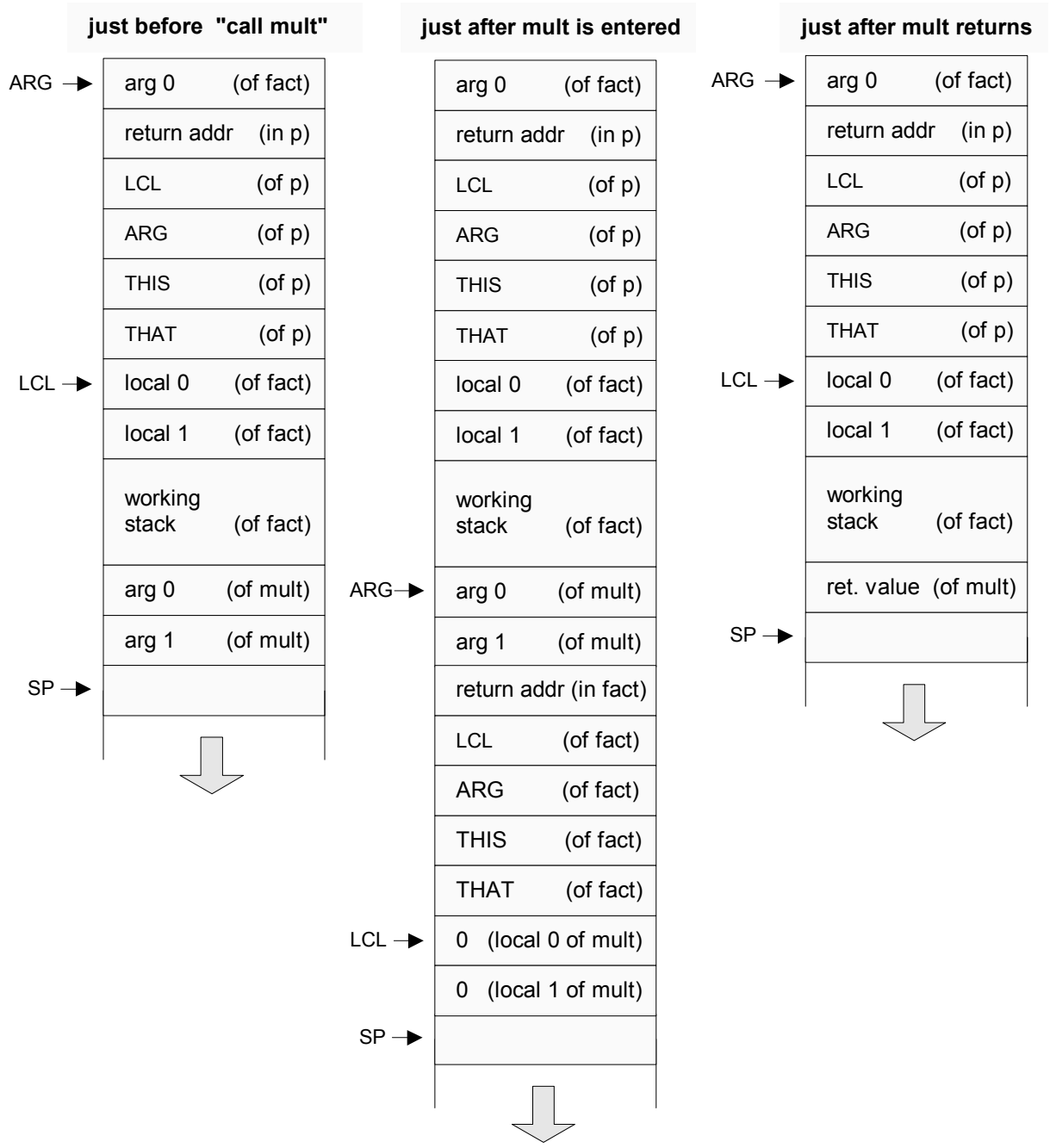

FIGURE 7-8: Dynamic global stack behavior. We assume that function `p` (not seen in this figure) called `fact`, then `fact` called `mult`. If we ignore the middle stack instance, we observe that `fact` has set up some arguments and called `mult` to operate on them (left instance). When `mult` returns (right instance), the arguments of the called function have been replaced with the function's return value. In other words, when the dust clears from the function call, the calling function has received the service that it has requested, and processing resumes as if nothing happened: the drama of `mult`'s processing (middle) has left no trace whatsoever on the stack, except for the return value.

7.3.1 Standard Mapping on the Hack Platform, Part II

By *standard mapping* we refer to a set of guidelines on how to map VM implementations on a specific target architecture. This section completes the standard mapping of the VM implementation over the Hack platform, with special emphasis on how to model and manage the *global stack.*

Function Calling Protocol

In order to implement the method calling protocol described in Section 7.2.2.1, the VM implementation should effect the following commands (pseudo-code):

VM command	*VM Implementation*

`call f c`	`push return-address // (using label below)`
	`push LCL // save LCL of calling function`
	`push ARG // save ARG of calling function`
	`push THIS // save THIS of calling function`
	`push THAT // save THAT of calling function`
	`ARG = SP-c-5 // reposition ARG (c=number of args)`
	`LCL = SP // reposition LCL`
	`goto f // transfer control`
	`(return-address) // label for the return address`

`function f c`	`(f) // label for function entry`
	`repeat c times: // c = number of local variables`
	`PUSH 0 // initialize all of them to 0`

`return`	`FRAME=LCL // FRAME is a temporary variable`
	`RET=*(FRAME-5) // Save return address in a temp. var`
	`*ARG=pop() // reposition return value for caller`
	`SP=ARG+1 // restore SP for caller`
	`THAT=*(FRAME-1) // restore THAT of calling function`
	`THIS=*(FRAME-2) // restore THIS of calling function`
	`ARG=*(FRAME-3) // restore ARG of calling function`
	`LCL=*(FRAME-4) // Restore LCL of calling function`
	`goto RET // GOTO the return-address`

TABLE 7-9: VM implementation of the function call commands (in pseudo code).

Assembly Language Symbols

Symbol	*Usage*
"`functionName:label`" symbols	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 function's code. When translating "`goto` *b*" and "`if-goto` *b*" VM commands into the target language, the full label specification *f*:*b* should be used instead of *b*.
"`functionName`" labels	Each VM function *f* should generates a symbol *f* that refers to its entry point in the instruction memory of the target architecture.
return address symbols	Each VM function call should generate a unique symbol that serves as a return address, i.e. the location of the command following the call command in the instruction memory of the target architecture.

TABLE 7-10: Special assembly symbols prescribed by the standard mapping.
This table completes Table 6-16.

Bootstrap Code

When the Hack computer is reset, it is wired to fetch and execute the word located in ROM address 0x0000. Thus the code that starts at address 0x0000 is called "bootstrap code", as it is the first thing that gets executed when the computer is reset. With that in mind, VM implementations over the Hack platform should place the following code in that address:

```
SP = 0x0100     // initialize the stack pointer
call Sys.init   // invoke Sys.init
```

This bootstrap code sets the stack pointer to its right value and then calls the `Sys.init` function. The `Sys.init` function then calls the "main function" of the "main program" (compilation-specific concepts that vary from one high level language to another), and enters an infinite loop.

7.3.2 Design Suggestions for the VM implementation

In chapter 6 we proposed implementing the VM translator as a main program consisting of two modules: *parser* and *code writer*. The basic translator built in Project 6 was based on basic versions of these modules. In order to turn the basic translator into a full-scale VM implementation, you now have to extend the basic `Parser` and `CodeWriter` classes with the functionality described below.

Parser

If the basic parser that you built according to section 6.3.2 does not already parse the six commands specified in this chapter, then add their parsing now. Specifically, make sure that the method `getCommandType()` returns the constants that correspond to the six commands described in this chapter: `COMMAND_LABEL`, `COMMAND_GOTO`, `COMMAND_IF`, `COMMAND_FUNCTION`, `COMMAND_RETURN`, `COMMAND_CALL`.

Code Writer

The basic `CodeWriter` specified in section 6.3.2 should be augmented with the following methods.

`Void writeInit():` Writes the assembly code that effects the VM initialization (also called *bootstrap code*). This code should be placed in the ROM beginning in memory location `0x0000`.

`Void writeLabel(Sring label):` Writes the assembly code that is the translation of the given `label` command.

`Void writeGoto(String label):` Writes the assembly code that is the translation of the given `goto` command.

`Void writeIf(String label):` Writes the assembly code that is the translation of the given `if-goto` command.

`Void writeCall(String functionName, int numArgs):` Writes the assembly code that is the translation of the given `Call` command.

`Void writeReturn():` Writes the assembly code that is the translation of the given `Return` command.

`Void writeFunction(String functionName, int numLocals):` Writes the assembly code that is the translation of the given `Function` command.

7.4 Perspective

Work in progress.

7.5 Build it

Chapter 6 focused on building a basic VM translator, designed to implement the *arithmetic* and *memory access* commands of the VM language. This section describes how to complete this basic program into a full-scale VM implementation over the Hack platform.

Objective: Extend the basic VM translator built in project 6 with the ability to handle the *program flow* and *function call* commands specified in section 7.2. The VM should be implemented on the Hack computer platform, conforming to the *standard mapping* described in Section 7.3.1.

Contract: Write the full-scale VM translator program. Use it to compile all the test `.vm` programs supplied below, yielding corresponding `.asm` programs written in Hack assembly. When executed on the supplied CPU Emulator, the `.asm` programs generated by your translator should deliver the results mandated by the supplied test scripts and compare files.

Proposed Implementation Stages

We recommend carrying out the implementation in two stages:

- Implementation of *program flow* commands
- Implementation of *function-related* commands

This modularity will allow you to test your translator incrementally, using step-by-step test programs that we provide. The rest of the implementation tips are as in project 6.

Test Programs

The supplied test programs are designed to support the incremental development plan described above. We supply five test programs and test scripts, as follows.

Program Flow Test Programs

- `basicLoop`: simple test of `goto` and `if-goto` commands. Computes the sum $1+2+\cdots+n$ and pushes the result unto the stack;
- `fibonacci`: a more challenging test. Computes and stores in memory the first n elements of the fibonacci series.

Function Calling Test Programs

- `simpleFunction`: Simple test of the "`function`" and "`return`" commands. The called function performs a simple calculation and returns the result.
- `fullTest`: a full test of the function call commands, the bootstrap section and most of the other VM commands. Consists of two `.vm` files:
    - `Math.vm` contains one recursive function called `fibonacci()`. This function returns the n'th element of the Fibonacci series;
    - `Sys.vm` contains one function called `init()`. This function calls the `Math.fibonacci` function with n=5, and then loops infinitely.

> Since there are two `.vm` files, the whole directory should be compiled in order to create one `fullTest.asm` file (compiling each .vm file separately will result in 2 separate .asm files, which is not desired here).

As prescribed by the *VM Specification* (section 7.2), the bootstrap code must include a call to the `Sys.init` function.

Steps

1. Download the `project7.zip` file and extract its contents to a directory called `project7` on your computer.

2. Write and test the full-scale VM translator in stages, as described above.