8. 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 7 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 develop 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 10 and 11, following the introduction of a high-level object-based language in chapter 9.

In any "Great Gems in Computer Science" contest, *stack processing* will be a strong finalist. The previous chapter showed how any arithmetic and Boolean expression could be calculated by elementary stack operations. This chapter goes on to show how this remarkably simple data structure can also support remarkably complex tasks like dynamic memory management, nested subroutine calling, parameter passing, and recursion. Most people tend to take these programming capabilities for granted, expecting modern programming languages to deliver them, one way or another. We are now in a position to open this black box, and see how these fundamental programming mechanisms can be supported and implemented using a relatively simple stack-processing model.

1. Background

The previous chapter focused on the arithmetic, logical, and data management operations of a typical stack-based, virtual machine. This, of course, was just the beginning. If we want our VM to become the backend of present and future compilers, we obviously need to support program flow and subroutine handling capabilities as well. We will do this by equipping the basic VM with two additional and final sets of commands: *program flow* commands for handling conditional and unconditional branching, and *function commands* for handling subroutine calls.

The remainder of this section gives an informal introduction to both subjects. This sets the stage for section 2, which rounds up the VM specification started in Chapter 7. Sections 3 and 4 discuss how to actually complete the VM implementation, leading to a full-scale VM-to-Hack translator.

1.1 Program flow

The default execution of computer programs is linear, one command after the other. This sequential flow is occasionally broken, e.g. to embark on another iteration of a loop. In low-level

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

programming, this branching logic is accomplished by instructing to continue execution at some specified part of the program other than in the next instruction, using a *"goto destination"* command. 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 is established by describing the jump destination using a symbolic label rather than a physical address. This variation requires that the language be equipped with some *labeling command*, designed to assign symbolic labels to selected points in the program.

The basic *goto* mechanism just described can be easily altered to affect *conditional branching* as well. Instead of jumping to some destination unconditionally, an "*if-goto destination*" command instructs to take the jump only if a certain Boolean condition is true; if the condition is false, the regular program flow should continues, executing the next command in the code. How should we introduce the Boolean condition specification into the *if-goto* mechanism? 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. Since the topmost stack value can be computed using any series of arithmetic and logical VM operations, one can condition the jump operation on arbitrarily complex Boolean expressions.

As is often the case in computer science, humble appearance often belies a great power of expression. In this case, the simple *goto* and *if-goto* commands can be used to express all the conditional and repetition constructs found in any high-level programming language. Figure 1 gives two typical examples.

High-level source code	***Compiled low-level pseudo 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 1: Implementing flow of control using *goto* and *if-goto* commands.

1.2 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 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 us to write programs in abstract terms, using high-level operations that are 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 the low level must do. In particular, in order to support subroutine calls, VM implementations must handle several issues:

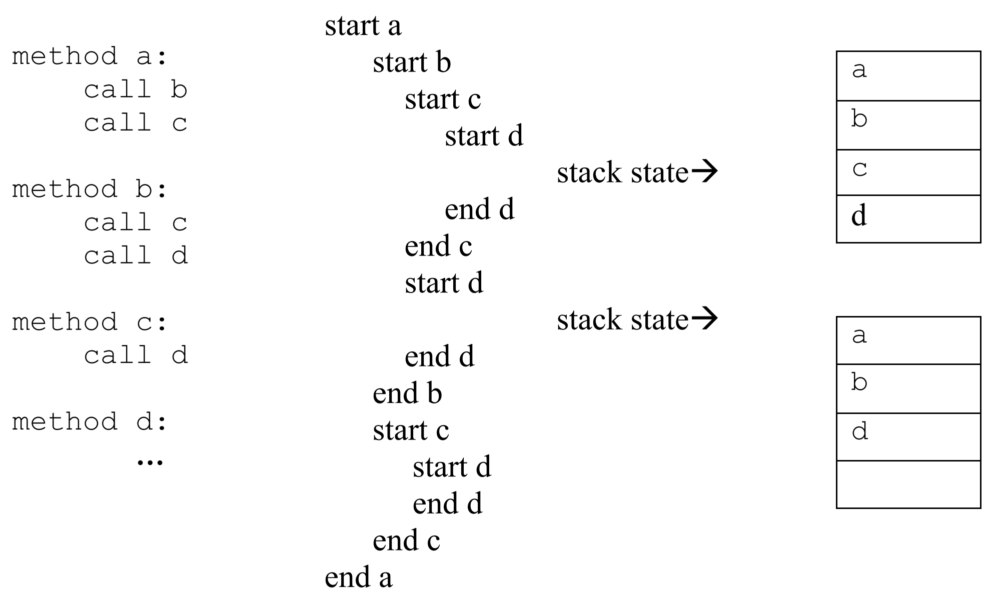
- Passing parameters to the called subroutine, and optionally returning a value from the called subroutine back to the caller;
- Allocating memory space for the local variables of the called subroutine, and freeing the memory when it is no longer needed;
- Jumping to execute the called subroutine's code;
- 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 any desired depth. To add to the complexity, we also need to support *recursion*. This means that subroutines should be allowed to call themselves, and each recursion level must be executed independently of the other calls.

1.3 Stack-Based Implementation

We see that the low-level handling of subroutine calls is rather delicate. The property that makes this task tractable is the hierarchical structure of the call-and-return logic: 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 "call-stack," "stack overflow," and so on. To illustrate, figure 2 shows a method calling pattern in a high-level program, along with some run-time checkpoints and the states of the abstract call-stack associated with them.



```
method a:
    call b
    call c

method b:
    call c
    call d

method c:
    call d

method d:
       ...
```

```
start a
   start b
      start c
         start d
                  stack state→
         end d
      end c
      start d
                  stack state→
      end d
   end b
   start c
      start d
      end d
   end c
end a
```

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

It is perhaps useful to note that from this point onward, the term *stack* will be used rather freely, and the reader should be able to tell from the sentence context the which stack we are taking about. For example, the *call stack* in Figure 2 is merely a conceptual notion, listing the names of all the active subroutine that are presently running. The *global stack*, on the other hand, is a real object. In particular, note that each subroutine that has not yet returned must maintain somehow its private set of local variables, argument values, pointers, and so on. Taken together, these data items are called the *method's frame*. Where should we keep all these frames? As Figure 2 shows, we can put them on the *global stack*. The reader may wander where the *working stack* from the previous chapter fits in -- the stack that supports the VM's push, pop, and arithmetic operations. Well, this stack can be maintained at the very top of the global stack, as we will see later.

The agent responsible for maintaining the global stack and implementing the call-and-return mechanism is the VM implementation. In order to carry out this stack, the VM implementation must handle such issues as return addresses, local variables allocation and de-allocation, and parameter passing.

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 has to resolve the name to a memory address -- a rather simple task -- and then jump to execute the code starting 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 call. For example, 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. Instead, 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 the return logic is to have the VM implementation save the return address just before the subroutine is called, and have it retrieved just after the subroutine exits. Conveniently, this store-and-recall setting lends itself perfectly to *stack* storage: 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. In terms of Figure 2, the return address can be kept in the method's frame.

Parameter passing: An important characteristic of well-designed languages is that 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 *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 Program 3.

```
// x+3
push x
push 2
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 3: VM elementary commands and high-level operations have the same look-and-feel in terms of arguments usage and return values. Thus they can be easily mixed together, yielding well-designed and readable code.

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 elementary 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 by the caller). Thus, the contract is as follows: the caller passes the arguments to the subroutine by pushing them onto the stack; the called subroutine pops the arguments from the stack, as needed, 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.

Local variables: Subroutines rely on local variables for temporary storage. And when a subroutine is used recursively, each recursion level must maintain its own set of local variables. Note however that these variables must be stored in memory only during the subroutine call's lifetime, i.e. from the point the subroutine starts executing until it returns. At this point, the memory space allocated to the local variables can be freed. How can the VM implementation effect this dynamic memory allocation?

Once again, the hard-working stack comes to the rescue. Although the subroutine calling chain may be arbitrarily deep as well as recursive, only one subroutine 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 (LIFO) processing model by storing the local variables of all the waiting subroutines on the stack, and reinstate them when control returns to the subroutine to which they belong. Revisiting Figure 2, we see that local variables can be saved in, and indeed they are part of, the method's frame.
.

2. VM Specification, Part II

This section extends the basic VM Specification from Chapter 7 with *program flow* and *function* commands. This completes the overall VM speciation.

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 the label is the current function.
`goto c`	This command effects a "goto" operation, causing execution to continue from the location marked by the `c` label. The jump destination must be located in the same function.
`if-goto c`	This command effects a "conditional goto" operation. The stack's topmost value is popped; if the value is not zero, execution continues from the location marked by the `c` label; otherwise, execution continues from the next command in the program. The jump destination must be located in the same function.

TABLE 4: Program flow commands.

2.2 Function 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 `bar` in class `Foo` in some high-level language will be translated by the language compiler to a VM function named `Foo.bar`).

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 onto the stack;
`return`	Return to the calling function.

TABLE 5: Function calling commands.

The Calling Protocol

The events of calling a function and returning from a function can be viewed from three different perspectives: that of the calling function, the called function, and the VM implementation.

The *calling function* view:

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

The *called function* view:

1. Upon getting called, my `argument` segment has been initialized with values passed by the caller, my `local` variables segment has been allocated and initialized to zero, the working stack that I see is empty, and the `static` segment that I see has been set to the `static` segment of the file to which I belong. All the other memory segments are undefined and can be used as needed.
2. Just before returning, I must push a value onto the stack.

The *VM implementation* view:

When a function ***calls*** another function, I (the VM implementation) must:

- Save the return address and the segment pointers of the calling function (except for `temp` which is not saved);
- Allocate, and initialize to zero, as many local variables as needed by the called function;
- Set the `local` and `argument` segments of the called function;
- Transfer control to the called function.

When a function ***returns***, I (the VM implementation) must:

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

2.3 Initialization

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

3. Implementation

We are now ready to complete the VM implementation whose first part was specified in Chapter 7. We begin by laying out the full stack structure that must be maintained by the implementation, and how it can be mapped over the Hack platform. Next, we give design suggestions and a proposed API, leading to a full-scale virtual machine implementation, based on a VM-to-Hack translator. This program can be viewed as a stand-alone language translator, as well as the backend module of our future compiler.

3.1 The Global Stack

The “system 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 and is not visible to VM functions.

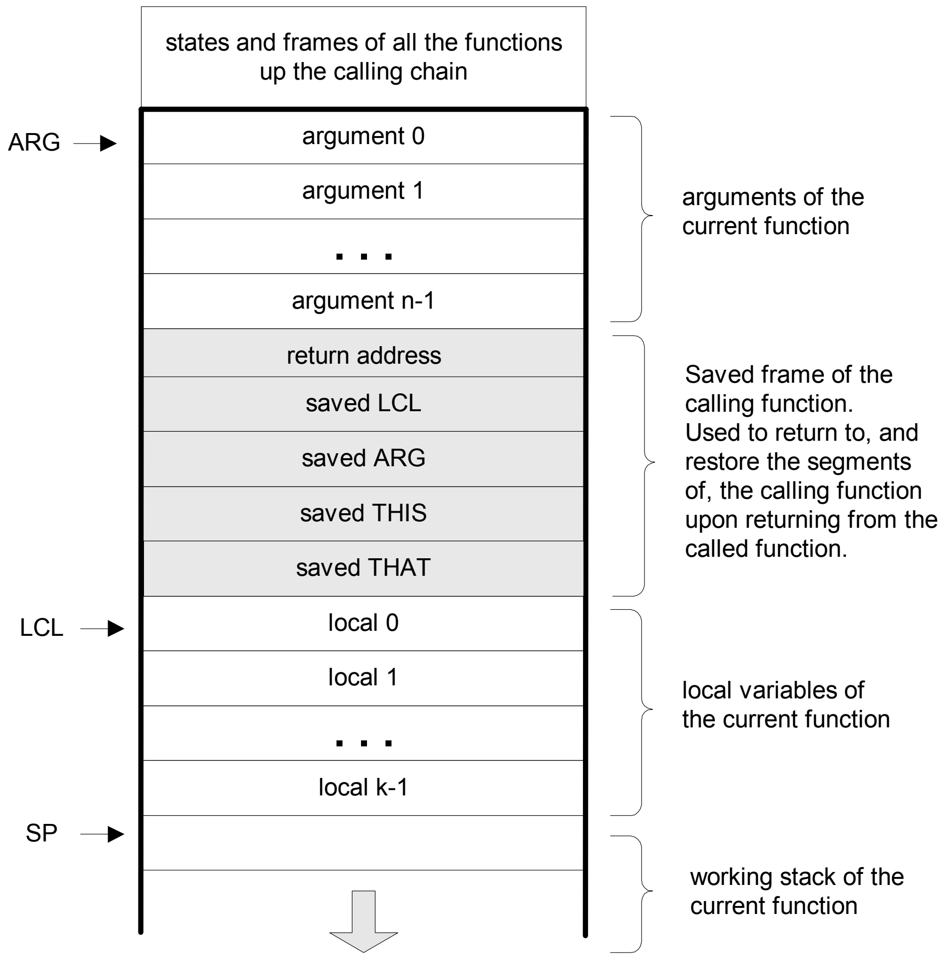

states and frames of all the functions up the calling chain
ARG
argument 0
argument 1
. . .
argument n-1
arguments of the current function
return address
saved LCL
saved ARG
saved THIS
saved THAT
Saved frame of the calling function. Used to return to, and restore the segments of, the calling function upon returning from the called function.
LCL
local 0
local 1
. . .
local k-1
local variables of the current function
SP
working stack of the current function

FIGURE 6: The global stack

Example: The factorial ($n!$) of a given number n can be computed by the bottom-up iterative formula $n!=1\cdot 2\cdot\ldots\cdot(n-1)\cdot n$. This algorithm is shown in Figure 7, along with a time-line of a typical run-time.

```
function p(...) {
...
   ... fact(4) ...
}
```

```
function fact(n) {
    vars result,j;
    result=1; j=1;
    while j<=n {
      result=mult(result,j);
      j=j+1;
    }
     return result;
}
```

```
function mult(x,y) {
    vars sum,j;
    sum=0; j=y;
    while j>0 {
      sum=sum+x;
      j=j-1;
    }
    return sum;
}
```

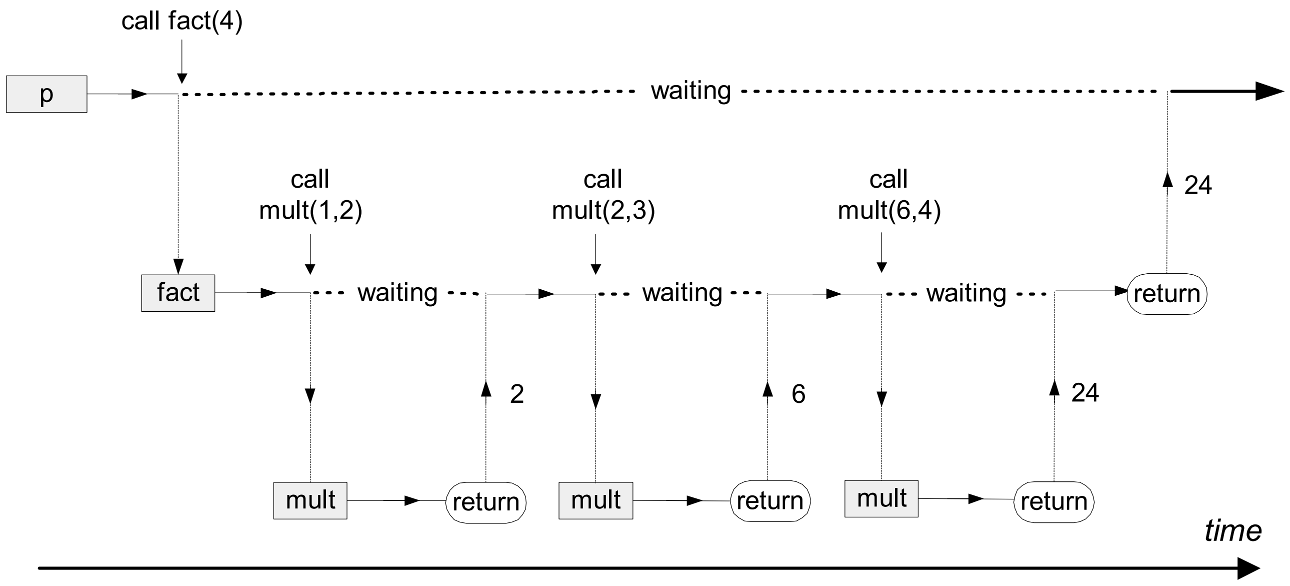


FIGURE 7: Function call-and-return routine: an arbitrary function `p` calls function `fact`, which then calls `mult` several times. Vertical arrows depict transfer of control from one function to another. At any given point of time, only one function is running, while all the functions up the calling 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).

Of course our concern here is neither the `fact` nor the `mult` functions, and that's why did not bother to write them in the VM language. Rather, we wish to shed light on the hidden infrastructure that enables these functions to interact with each other through parameter passing, return values, and control re-direction. The centerpiece of this infrastructure is the global stack, as seen in Figure 8.

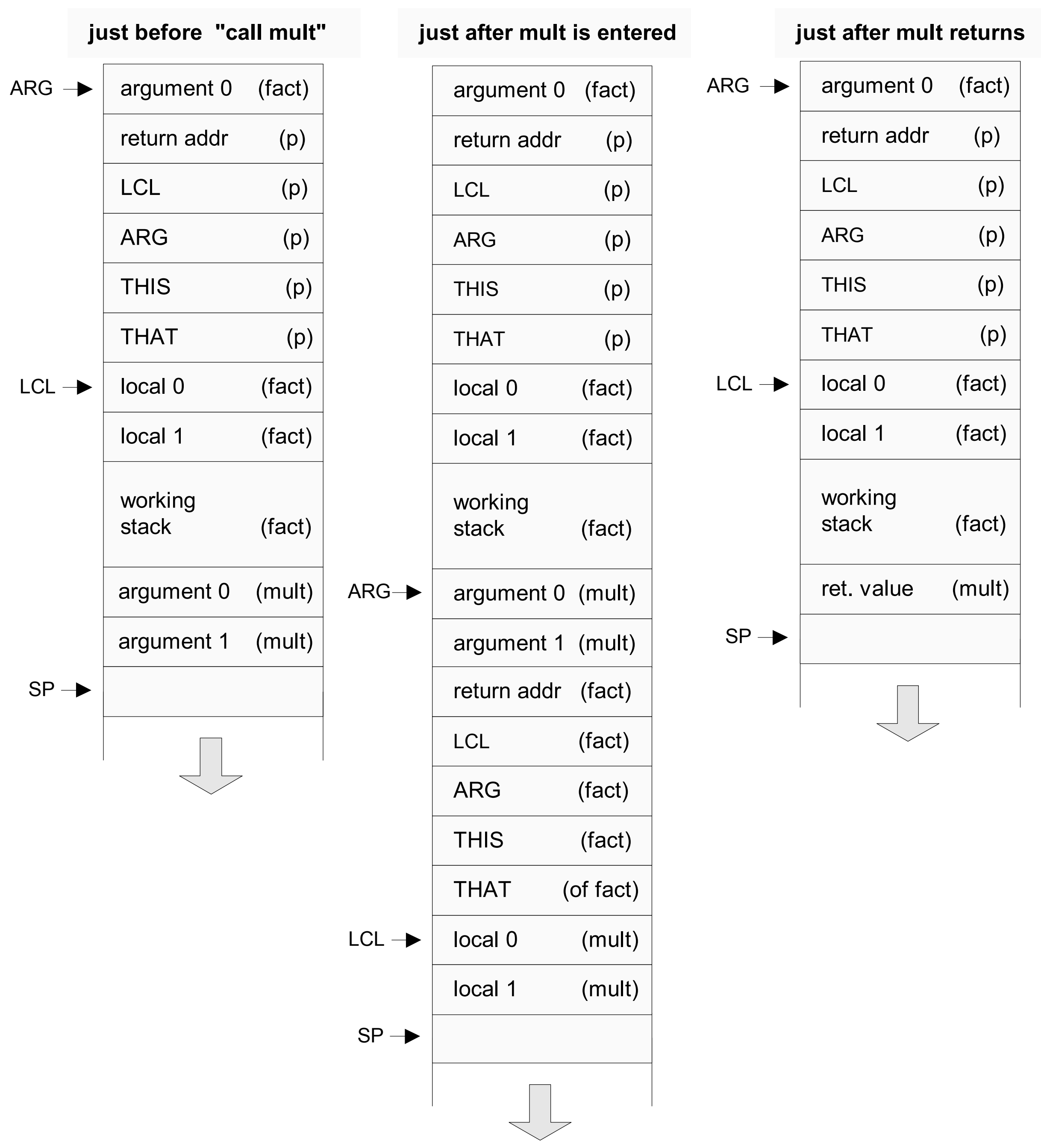


FIGURE 8: Global stack dynamics. We assume that function `p` (not seen in this figure) called `fact`, then `fact` called `mult`. If we ignore the middle stack instance, we observe that `fact` has set up some arguments and called `mult` to operate on them (left instance). When `mult` returns (right instance), 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.

3.2 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 VM-on-Hack mapping whose first part was given in Chapter 7.

Function Calling Protocol

The subroutine calling mechanisms of modern programming languages (e.g. Figures 6-7) can be implemented using stack operations. Table 9 gives the details.

VM command	*VM-on-Hack Implementation action (pseudo code)*

Calling a function:
 	
`call f n`
	`   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-n-5        // reposition ARG (n=number of args)`
	`   LCL = SP            // reposition LCL`
	`   goto f              // transfer control`
	`(return-address)       // label for the return address`

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

Returning from a function:
 	
`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 9: VM implementation of function commands (pseudo code).

Assembly Language Symbols

Symbol	*Usage*
"`functionName:label`" symbols	Each "`label` *b*" command in a 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*" commands into the target language, the full label specification *f*:*b* should be used instead of *b*.
"`functionName`" labels	Each function *f* should generate a symbol *f* that refers to its entry point in the instruction memory of the target architecture.
return address symbols	Each 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 10: Special assembly symbols prescribed by the standard mapping.

Bootstrap Code

Upon reset, the Hack hardware is wired to fetch and execute the word located in ROM address 0x0000. Thus, the code segment that starts at address 0x0000, called *bootstrap code*, is the first thing that gets executed when the computer "boots up". As a convention, we want this code to effect the following operations (in machine language):

```
SP=256          // initialize the stack pointer to 0x0100
call Sys.init   // invoke Sys.init
```

This code sets the stack pointer to its right value (as per the standard mapping) and then calls the `Sys.init` function. The contract is that `Sys.init` should then call the main function of the main program, and enter an infinite loop. Taken together, these operations should cause the translated VM program to start running.

The "main function" and the "main program" are compilation-specific and vary from one high level language to another. For example, in the Jack language, the default is that the first program unit that starts running automatically is the `main` method of a class named `Main`. In a similar fashion, when we tell the JVM to execute a given Java class, say `Foo`, it will look for, and execute, the `Foo.main` method. Such "automatic" start-up routines can be effected by the bootstrap logic described above.

3.3 Design Suggestions for the VM implementation

In chapter 7 we proposed implementing the VM translator as a main program consisting of two modules: *parser* and *code writer*. The basic translator built in Project 7 was based on basic versions of these modules. In order to turn the basic translator into a full-scale VM implementation, we have to extend the basic *parser* and *code writer* modules with the functionality described below.

Parser

If the basic parser that you built in Project 7 does not already parse the six commands specified in this chapter, then add their parsing now. Specifically, make sure that the `commandType` method developed in Project 7 also returns the constants corresponding to the six VM commands described in this chapter: `C_LABEL`, `C_GOTO`, `C_IF`, `C_FUNCTION`, `C_RETURN`, `C_CALL`.

Code Writer

The basic `CodeWriter` specified in Chapter 7 should be augmented with the following methods.

CodeWriter Module			
Translates VM commands into Hack assembly code.			
The routines listed below should be added to the `CodeWriter` module API given in Chapter 7.			
Routine	**Arguments**	**Returns**	**Function**
`writeInit`	--	--	Writes the assembly code that effects the VM initialization (also called *bootstrap code*). This code should be placed in the ROM beginning in address `0x0000`.
`writeLabel`	`label` (string)	--	Writes the assembly code that is the translation of the given `label` command.
`writeGoto`	`label` (string)	--	Writes the assembly code that is the translation of the given `goto` command.
`WriteIf`	`label` (string)	--	Writes the assembly code that is the translation of the given `if-goto` command.
`writeCall`	`functionName` (string) `numArgs` (int)	--	Writes the assembly code that is the translation of the given `Call` command.
`writeReturn`	--	--	Writes the assembly code that is the translation of the given `Return` command.
`writeFunction`	`functionName` (string) `numLocals` (int)	--	Writes the assembly code that is the trans. of the given `Function` command.

4. Perspective

Work in progress.

5. Build it

Objective: Extend the basic VM translator built in project 7 with the ability to handle the *program flow* and *function* commands specified in this chapter. The VM should be implemented on the Hack computer platform, conforming to the *standard mapping* described in this chapter.

Resources (same as in Project 7): You will need two tools: the programming language in which you will implement your VM Translator, and the *CPU Emulator* supplied with the book. This emulator allows executing the machine code generated by your VM Translator -- an indirect way to test the correctness of the latter. Another tool that may come handy in this project is the visual *VM Emulator* supplied with the book. This program allows experimenting with a working VM environment before you set out to implement it yourself. For more information about this tool, refer to the *VM Emulator Tutorial*.

Contract: Write a full-scale VM-to-Hack translator, extending the translator developed in Project 7. Use it to translate the test `.vm` programs supplied below, yielding corresponding `.asm` programs written in the Hack assembly language. When executed on the supplied CPU Emulator, the assembly programs generated by your translator should deliver the results mandated by the supplied test scripts and compare files.

Proposed Implementation Stages

We recommend implementing the translator in two stages.

- Stage I: Implementation of *program flow* commands
- Stage II: Implementation of *function* commands

This modularity will allow you to test your implementation incrementally, using the step-by-step test programs that we provide.

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 onto 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 function performs a simple calculation and returns the result.
- `FibonacciElement`: A full test of the function call commands, the bootstrap section and most of the other VM commands. The `FibonacciElement` directory 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*=4, and then loops infinitely.

  Since the overall program consists of two `.vm` files, the entire directory should be compiled in order to create a single `FibonacciElement.asm` file (compiling each .vm file separately will yield two separate `.asm` files, which is not desired here).
- `StaticTest`: A full test of static variables handling. Consists of two `.vm` files, each representing the compilation of a typical class file, and a `sys.vm` file, as usual. Once again, the entire directory should be compiled in order to create a single `StaticTest.asm` file.

As prescribed by the *VM Specification* (section 2), the bootstrap code must include a call to the `Sys.init` function.

Steps

1. Download `project8.zip` and extract its contents into a directory called `project8` on your computer, without changing the directories structure embedded in the zip file.

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