The Virtual Machine: Stack Arithmetic

Prof. Naga Kandasamy
ECE Department, Drexel University

Introduction

A compiler is a large, complex system software. It must understand the source program it takes as input and map the program's functionality to the target hardware. The distinct nature of these two tasks suggests a division of labor leading to a design that decomposes the compilation process into two major pieces: a *front end* and a *back end* as shown in Fig. 1. The front end focuses on understanding the source-language program whereas the back end focuses on mapping programs to the target hardware.

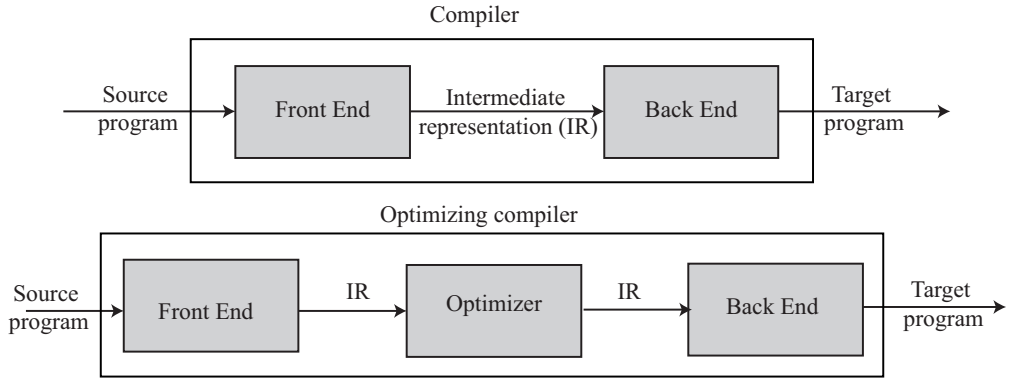

Fig. 1: Structure of a compiler.

The front end must encode its knowledge of the source program in some structure for later use by the back end. This *intermediate representation* (*IR*) becomes the compiler's definitive representation for the code it is translating. In a two-phase compiler, the front end must ensure that the source program is well formed — free of syntax errors — and it must map that code into the IR. The back end must map the IR program into the instruction set and the finite resources of the target machine. Since the back end only processes IR created by the front end, it can assume that the IR contains no syntactic or semantic errors.

The two-phase structure can simplify the process of retargeting the compiler:

- Multiple back ends can be constructed for a single front end to produce compilers that accept the same language but target different hardware platforms.
- Front ends for different languages can be developed that produce the same IR for a common back end.

Both scenarios assume that one IR can serve several combinations of source and target programs. Such an IR, therefore, must be designed to run on a *Virtual Machine* (*VM*) — an abstract computer that does not really exist but rather can be realized on other computer hardware. A key advantage of compiling to a VM is code transportability; since the VM may be implemented for multiple hardware platforms, the VM-based software can run on many processors and operating systems without having to modify the original source code — as long as the appropriate back end for the target machine is available.

Introducing an IR makes it possible to add an *optimization phase* to compilation, again as shown in Fig. 1. The optimizer takes an IR program as its input and produces a semantically equivalent IR program as its output. The optimizer is an IR-to-IR transformer that tries to improve the IR program in some way; it may rewrite the original IR such that the back end is able to generate a faster (or smaller) target program. Multiple optimization stages may be involved; for example, the `gcc` compiler has various optimization levels — `O0`, `O1`, `O2`, and `O3` — that optimize for compilation time, code size, and execution time.

We will focus on developing the back end of the compiler — specifically, taking an IR based on the stack machine model as input and producing machine code targeting the Hack computer.

The Stack Machine Model

The textbook uses a VM based on a stack machine, modeled after the Java Virtual Machine (JVM).[1] A program written as per the Java specification can be compiled by the front end into Java bytecode — the instruction set of the JVM — that can be executed by the JVM.

A stack is an abstract data structure that implements the last-in first-out (LIFO) storage model. It supports two basic operations:

- The *push* operation adds an element to the top of the stack; the element that was previously on top is pushed below the newly added element.
- The *pop* operation removes the top element from the stack; the element below it moves up to the top position.

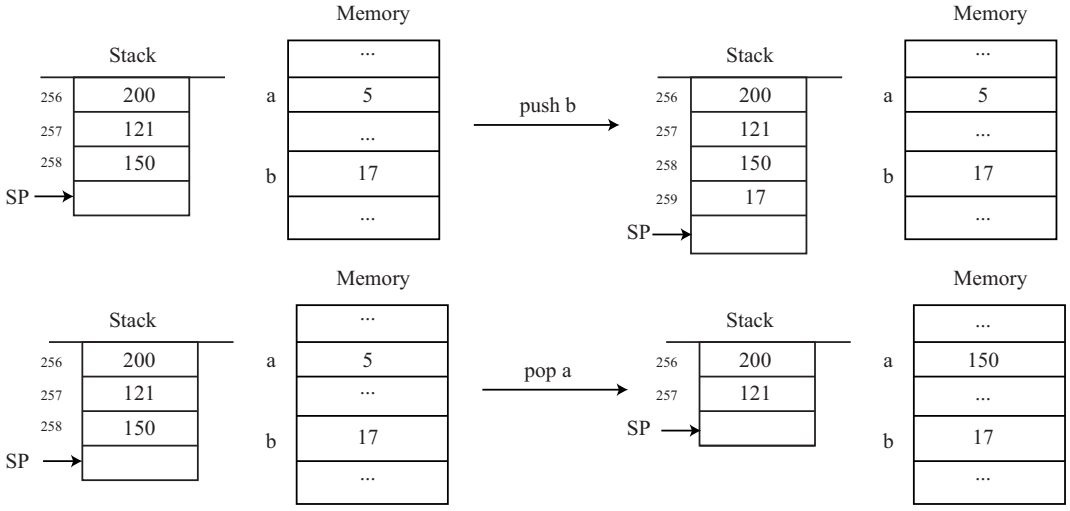

Fig. 2: Basic stack operations: post-increment push and pre-decrement pop.

Figure 2 illustrates the push and pop operations. The stack starts at memory location 256 and grows from low to high addresses. The *SP* variable maintains the memory address corresponding to the top of the stack at any given time — top of the stack being the empty slot just above the top-most element. Consider the *push* operation in which we wish to store the value located in some memory address, labeled as *b*, to the stack. Figure 2 shows an instance of a "post-increment" push in that we first store *b* at the array entry pointed to by *SP* and then increment *SP*. The *pop* operation is implemented by first decrementing *SP*, then returning the value stored in the top position and overwriting the memory location labeled *a*. This is a "pre-decrement" pop operation.

[1] There are examples of register-based VMs as well; for example, the Dalvik VM for the Android Operating System.

Stack access differs from conventional memory access in the following aspects:

- The stack is accessible only from the top, one item at a time.
- Reading the stack is a lossy operation — the only way to retrieve the top value is to remove it from the stack. In contrast, reading from a regular memory location has no impact on the memory's state.
- Writing an item onto the stack adds it to the stack's top, without changing the rest of the stack. In contrast, writing an item into a regular memory location is a lossy operation, since it overwrites the location's previous value.

Consider the VM-language code

```
set sp 256      // Initialize SP to 256
...
push b
...
pop a
...
```

to be translated into the Hack assembly language. The `set` command loads the value of 256 into the `SP` register — also register `R0` or `RAM[0]` — in the Hack computer. This VM-language command can easily be replaced with the appropriate collection of Hack assembly-language commands:

```
@256            // Load constant 256 into A register
D = A
@SP
M = D           // SP <-- 256
```

The push command is translated as follows:

```
1  @b
2  D = M           // D <-- b
3  @SP
4  A = M           // A <-- address of top of stack
5  M = D           // Memory[A] <-- D
6  @SP
7  M = M + 1       // Adjust SP <-- SP + 1
```

Lines 1 and 2 load the value stored in memory location labeled *b* into the `D` register. Lines 3 through 5 implement indirect addressing by loading contents of the `SP` register — which stores the address of the top of the stack — into the `A` register, and then storing `D` into that address location. Finally, lines 6 and 7 increment the stack pointer to complete the push operation.

The pop operation is translated as follows:

```
@SP
M = M - 1       // Adjust SP <-- SP - 1
A = M           // A <-- address of top of stack
D = M           // D <-- Memory[A]
@a
M = D           // a <-- D
```

Figure 3 illustrates pre-increment push and post-decrement pop operations. In pre-increment push, we first increment *SP* and then store *b* at the array entry pointed to by *SP*. The post-decrement pop is implemented by returning the value stored in the top position and then decrementing *SP*.

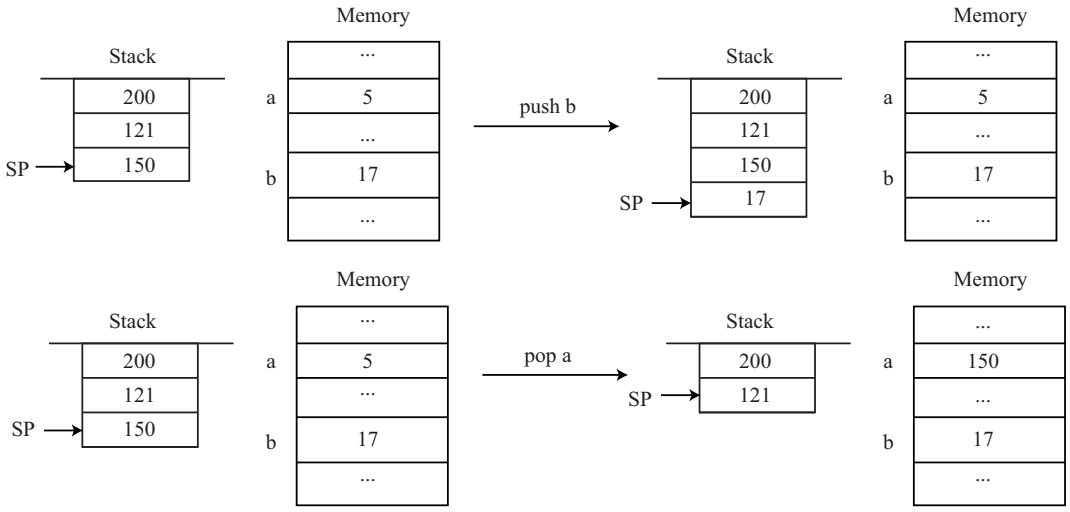

Fig. 3: Basic stack operations: pre-increment push and post-decrement pop.

The Hack assembly code for pre-increment push and post-decrement pop is developed along similar lines:

```
// push b
@b
D = M            // D <-- b
@SP
M = M + 1        // Adjust SP <-- SP + 1
A = M            // A <-- address of top of stack
M = D            // Memory[A] <-- D

// pop a
@SP
A = M            // A <-- address of top of stack
D = M            // D <-- Memory[A]
@a
M = D            // a <-- D
@SP
M = M - 1        // Adjust SP <-- SP - 1
```

To ensure correctness, post-increment push must have the pre-decrement pop at its counterpart, and pre-increment push must have post-decrement pop as its counterpart. The choice between various implementation options is usually left to the system software developer. The textbook uses post-increment push and pre-decrement pop operations, and so we will use the same in the remainder of this document.

VM Specification

A VM program is organized in units called *functions* written in the VM language. Each function has its own stand-alone code which is stored in a text file with `.vm` extension. A VM program is a collection of one or more such files. From a compilation standpoint, these constructs correspond, respectively, to the notions of program, class, and method in an object-oriented language. Within a `.vm` file, each VM command appears

in a separate line and in one of the following formats: *command* (for example, add or sub); *command arg* (for example, goto loop); and *command arg1 arg2* (for example push local 3). Arguments are separated from each other and from the command part by an arbitrary number of spaces. Comments beginning with "//" can appear at the end of any line and are ignored. Blank lines are permitted and ignored. Following is a VM program that adds two variables a and b and stores the result in variable c. Assume that the `SP` register has been initialized to 256 prior to executing the code.

```
// simple_add.vm
push constant 5
pop local 0         // a <-- 5 where a is stored in (local segment, 0)
push constant 10
pop local 1         // b <-- 10, where b is stored in (local segment, 1)
push local 0        // Push a into stack
push local 1        // Push b into stack
add
pop local 2         // c <-- 15, where c is stored in (local segment, 2)
```

The VM language has a single 16-bit data type that can be used as an integer, a Boolean, or a pointer. The language consists of four command types:

- *Arithmetic commands* that perform arithmetic and logical operations on the stack.
- *Memory access commands* that transfer data between the stack and various virtual memory segments.
- *Program flow commands* that facilitate conditional and unconditional branching operations.
- *Function calling commands* that call functions and return from them.

This lecture discusses arithmetic and memory access commands. Program flow and support for functions will be discussed in the next lecture.

Stack Arithmetic

The stack-based VM language developed in the textbook features nine arithmetic and logical commands — seven of which are binary in that they pop two items off the stack, compute a binary function on them, and push the result back onto the stack; and the remaining two commands are unary in that they pop a single item off the stack, compute a unary function on it, and push the result back onto the stack. Each command has the net impact of replacing its operand(s) with the command's result, without affecting the rest of the stack. The following table summarizes the various commands.

Command	**Return value** (after popping operands)	**Comment**
`add`	$x + y$	Integer addition (in two's complement)
`sub`	$x - y$	Integer subtraction (in two's complement)
`neg`	$-x$	Arithmetic negation (in two's complement)
`eq`	if $x = y$ then true else false	Test for equality
`gt`	if $x > y$ then true else false	Test for greater than
`lt`	if $x < y$ then true else false	Test for less than
`and`	$x \& y$	Bit-wise AND
`or`	$x \mid y$	Bit-wise OR
`not`	$\neg x$	Bit-wise NOT

The `eq`, `gt`, and `lt` commands return Boolean values — true is represented as `0xFFFF` (or -1 in two's complement) and false is represented as `0x0000` (or 0).

Stack machine models can evaluate arithmetic and logical expressions in natural manner. Consider the expression `z = (2 - x) * (y + 5)` written in the so-called *infix notation* since the operator appears in between the two operands that it is evaluating. This expression can be converted into *postfix notation* as `z = 2x-y5+*` in which operators come after the corresponding operands, while obeying operator precedence rules. (We will discuss how to perform this conversion in a later lecture.) The corresponding VM-language code can be easily generated:

```
// Infix: z = (2 - x) * (y + 5)
// Postfix: z = 2x-y5+*
push 2
push x
sub
push y
push 5
add
mult
pop z
```

In case of binary arithmetic operations such as `add`, `sub`, and `mult`, two operands are popped in succession from the top of the stack, the required operation is performed on them, and the result is pushed back to the stack. The collection of Hack assembly-language commands that realize the VM-language command `sub` are as follows:

```
// Perform operand1 - operand2
@SP
M = M - 1
A = M
D = M           // D <-- operand2
@SP
M = M - 1       // Adjust stack pointer
A = M
D = M - D       // D <-- operand1 - operand2
// Push result to stack and adjust SP
@SP
A = M
M = D
@SP
M = M + 1
```

The case of relational operators is handled similarly. For example, the following VM code evaluates the expression `if (x < 10) or (y == 5)`.

```
// (x < 10) or (y == 5)
push x
push 10
lt          // x < 10
push y
push 5
eq          // y == 5
or
```

Accessing Memory

So far, we have illustrated memory access commands using pseudo-commands such as *push a* or *pop b* where the symbols a and b refer to address locations in memory. From an actual implementation viewpoint, the stack machine loads from and stores into eight separate memory segments.

- The *push segment index* command pushes the value of *segment*[*index*] to the stack.
- The *pop segment index* command pops the top stack value and stores it in *segment*[*index*].

The above addressing scheme is an example of *base* + *index* addressing, in which *segment* denotes the base address of the segment and *index* is the offset within that segment.

The functionality of various memory segments are summarized as follows:

- The `argument` segment stores the function's arguments. It is allocated dynamically by the VM implementation when entering the function. Segment's base address is stored in the `ARG` register.
- The `local` segment stores the function's local variables declared within the scope of that function. The segment is also allocated dynamically and is initialized to 0 when entering the function. Segment's base address is stored in the `LCL` register.
- The `static` segment stores static variables shared by all functions located in the same `.vm` file.
- The `constant` pseudo-segment holds constants in the range 0 through $2^{16} - 1$. This segment is seen by all functions in the VM program.
- General-purpose segments, `this` and `that`, can be made to correspond to different areas in the heap, and any VM function can use these segments to manipulate the heap. The base addresses are stored in `THIS` and `THAT` registers.
- A two-entry segment, `pointer`, holds the base address of the `this` and `that` segments. Any VM function can `set pointer 0` or `set pointer 1` to some address — which has the effect of aligning the `this` (or `that`) segment to the heap area beginning at that address.
- The `temp` segment holds eight entries as temporary variables for general-purpose use, shared by all functions in the VM program.

The following table lists the use of various Hack computer registers in the context of VM programs.

Register	RAM location	Usage
`SP`	`RAM[0]`	Stack pointer; points to topmost location in the stack.
`LCL`	`RAM[1]`	Points to base of the current VM function's `local` segment.
`ARG`	`RAM[2]`	Points to base of the current VM function's `argument` segment.
`THIS`	`RAM[3]`	Points to base of the current VM function's `this` segment.
`THAT`	`RAM[4]`	Points to base of the current VM function's `that` segment.
	`RAM[5]`–`RAM[12]`	Holds contents of the `temp` segment.
	`RAM[13]`–`RAM[15]`	General-purpose registers for use by the VM implementation.

The `local`, `argument`, `this`, and `that` segments are mapped directly to RAM, and the corresponding locations are maintained by keeping the physical base addresses in a dedicated registers `LCL`, `ARG`, `THIS`, and `THAT`. Any access to the i^{th} entry of any one of these segments should be translated to assembly code that accesses address $(base + i)$ in RAM, where $base$ is the current value stored in the register dedicated to that particular segment.

The `pointer` segment is mapped to `RAM[3]` and `RAM[4]` (also called `THIS` and `THAT`), and the `temp` segment to RAM locations 5–12. Therefore, access to pointer i should be translated to assembly code that accesses RAM location $3+i$, and access to temp i should be translated to assembly code that accesses RAM location $5+i$.

The `constant` segment is virtual; it occupies no physical space on the hardware. Instead, the VM implementation handles any access to this segment by simply supplying the requested constant.

Consider the following VM code that initializes the variable stored in `(local, 2)` to the constant 5.

```
push constant 5     // Push 5 to top of stack
pop local 2         // Pop stack element and store in (local, 2)
```

Assuming that `SP` and `LCL` registers have been initialized, the corresponding Hack assembly code is:

```
1  // push constant 5
2  @5
3  D = A
4  @SP
5  A = M
6  M = D               // Store 5 into stack
7  @SP
8  M = M + 1           // SP <-- SP + 1
9
10 // pop local 2
11 @LCL
12 D = M               // D <-- Base address stored in LCL
13 @2
14 D = D + A
15 @R13
16 M = D               // Temp register R13 <-- base + index
17 @SP
18 M = M - 1
19 A = M
20 D = M               // D <-- Topmost element in stack
21 @R13
22 A = M               // A <-- base + index
23 M = D               // Memory[base + index] <-- D
```

Pay careful attention to the indirect addressing being performed as part of the push and pop operations. Also, given the scarcity of working registers in the Hack computer, the absolute address corresponding to `(local, 2)` is stored in a temporary location `RAM[13]` for subsequent use.