CMSC 22610 Winter 2015 | Implementation of Computer Languages I | Handout 2 March 1, 2015

The Virtual Machine Reference

1 Introduction

Project 4 involves generating code targeted to a stack-based virtual machine (VM). This document is a reference for the VM and includes a description of the machine model and instruction set. The Project 4 description discussed the code generation interface.

2 The Flang virtual machine

In this section, we describe the **Flang** virtual machine (VM). The virtual machine is a stand-alone program that takes an executable file and runs it. An VM executable consists of a code sequence, a literal table that contains string literals, and a C function table that contains runtime system functions used to implement services such as I/O.

2.1 Values

The VM supports three types of values: 31-bit tagged integers, 32-bit pointers to heap-allocated records of values, and 32-bit pointers to strings. A integer value n is represented by $2n + 1$ in the VM (this tagging is required for the garbage collector). The VM takes care of tagging/untagging, so the only impact of this representation on your code generator is that integer literals must be in the range -2^{30} to $2^{30} - 1$. We use word address for values (but byte addressing for instructions).

2.2 Registers

The **Flang** VM has four special registers: the stack pointer (SP), which points to the current top of the stack; the frame pointer (FP), which points to the base of the current stack frame and is used to access local variables; the environment pointer (EP), which points to the current closure object and is used to access global variables; and the program counter (PC), which points to the next instruction to execute.

3 Instructions

We define the semantics of the instructions using the following notation

$$\cdots \alpha \quad \texttt{instr} \quad \Longrightarrow \cdots \beta$$

which means that the instruction **`instr`** takes a stack configuration with α on the top and maps it to a stack with β on the top. The instructions are organized by kind in the following description.

3.1 Arithmetic instructions

These instructions operate on 31-bit tagged integer values. The untagging and tagging is handled automatically.

$\cdots\, i_1\, i_2$ **`add`** $\Longrightarrow \cdots (i_1 + i_2)$
pops the top two integers, adds them and pushes the result.

$\cdots\, i_1\, i_2$ **`sub`** $\Longrightarrow \cdots (i_1 - i_2)$
pops the top two integers, subtracts them and pushes the result.

$\cdots\, i_1\, i_2$ **`mul`** $\Longrightarrow \cdots (i_1 \times i_2)$
pops the top two integers, multiplies them and pushes the result.

$\cdots\, i_1\, i_2$ **`div`** $\Longrightarrow \cdots (i_1 / i_2)$
pops the top two integers, divides them, and pushes the result. The VM halts with an error if i_2 is zero.

$\cdots\, i_1\, i_2$ **`mod`** $\Longrightarrow \cdots\, i_1 \bmod i_2$
pops the top two integers, divides them, and pushes the remainder. The VM halts with an error if i_2 is zero.

$\cdots\, i$ **`neg`** $\Longrightarrow \cdots - i$
pops the integer on the top of the stack and pushes its negation.

3.2 Comparison instructions

$\cdots\, v_1\, v_2$ **`equ`** $\Longrightarrow \cdots\, b$
pops and compares the two values on top of the stack. If they are equal, then it pushes 1, otherwise it pushes 0. Note that if the values are pointers, then the comparison pushes true if the pointers are equal

$\cdots\, i_1\, i_2$ **`less`** $\Longrightarrow \cdots\, b$
pops and compares the two integers on top of the stack. If $i_1 < i_2$, then it pushes 1, otherwise it pushes 0.

$\cdots\, i_1\, i_2$ **`lesseq`** $\Longrightarrow \cdots\, b$
pops and compares the two integers on top of the stack. If $i_1 \leq i_2$, then it pushes 1, otherwise it pushes 0.

$\cdots\, v$ **`not`** $\Longrightarrow \cdots\, b$
pops v and pushes 1 if $v = 0$; otherwise it pushes 0.

$\cdots\, v$ **`boxed`** $\Longrightarrow \cdots\, b$
pops v and pushes 1 if v is boxed; otherwise it pushes 0.

3.3 String instructions

$\cdots s$ **`size`** $\Longrightarrow \cdots i$
pops the string s off the stack and pushes its length.

$\cdots s\, i$ **`subscript`** $\Longrightarrow \cdots c$
pops the integer i and the string s off the stack and pushes the character at position i in the string. The VM halts with an error if i is out of bounds.

3.4 Heap instructions

$\cdots v_0 \cdots v_{n-1}$ **`alloc(`n`)`** $\Longrightarrow \cdots \langle v_0, \ldots, v_{n-1}\rangle$
allocates an n element record, which is initialized from the top n stack values.

$\cdots \langle v_0, \ldots, v_{n-1}\rangle$ **`explode`** $\Longrightarrow \cdots v_0 \cdots v_{n-1}$
pops a tuple off the stack and pushes its elements.

$\cdots \langle v_0, \ldots, v_{n-1}\rangle$ **`select(`i`)`** $\Longrightarrow \cdots v_i$
pops a record off the stack and pushes the record's ith component.

3.5 Stack instructions

$\cdots$ **`int(`n`)`** $\Longrightarrow \cdots n$
pushes the integer n onto the stack.

$\cdots$ **`literal(`i`)`** $\Longrightarrow \cdots s_i$
pushes a reference to the ith string literal (s_i) onto the stack.

$\cdots$ **`label(`l`)`** $\Longrightarrow \cdots \mathit{addr}$
pushes the code address named by the label. Note that in the encoding of this instruction, the code address is specified as an offset from the **`label`** instruction.

$\cdots v_1\, v_2$ **`swap`** $\Longrightarrow \cdots v_2\, v_1$
swaps the top two stack elements (equivalent to `swap(1)`).

$\cdots v_0\, v_1 \cdots v_{n-1}\, v_n$ **`swap(`n`)`** $\Longrightarrow \cdots v_n\, v_1 \cdots v_{n-1}\, v_0$
swaps the top stack element with the n'th from the top. All other stack elements are unchanged.

$\cdots v$ **`pop`** $\Longrightarrow \cdots$
pops and discards the top stack element.

$\cdots v_1 \cdots v_n$ **`pop(`n`)`** $\Longrightarrow \cdots$
pops and discards the top n stack elements.

$\cdots v$ **`dup`** $\Longrightarrow \cdots v\, v$
duplicate the top of the stack (equivalent to `swap(0)`).

$\cdots v_n \cdots v_0$ **`push(`n`)`** $\Longrightarrow \cdots v_n \cdots v_0\, v_n$
pushes the nth element from the top of the stack.

$\cdots$ **`loadlocal(`**n**`)`** $\Longrightarrow \cdots v$
fetches the value (v) in the word addressed by FP $+ n$ and pushes it on the stack. Note that a function's argument will be at offset 2, while the local variables start at offset -1.

$\cdots v$ **`storelocal(`**n**`)`** $\Longrightarrow \cdots$
pops v off the stack and stores it in the word addressed by FP $+ n$.

$\cdots$ **`loadglobal(`**n**`)`** $\Longrightarrow \cdots v$
fetches the value (v) in the word addressed by EP $+ n$ and pushes it on the stack.

$\cdots$ **`pushep`** $\Longrightarrow \cdots ep$
push the current contents of the EP on the stack.

$\cdots ep$ **`popep`** $\Longrightarrow \cdots$
pop a value from the stack and store it in the EP.

3.6 Control-flow instructions

$\cdots$ **`jmp(`**n**`)`** $\Longrightarrow \cdots$
transfer control to instruction PC $+ n$.

$\cdots b$ **`jmpif(`**n**`)`** $\Longrightarrow \cdots$
pops b off the stack and if $b \neq 0$ it transfers control to instruction PC $+ n$.

$\cdots addr$ **`call`** $\Longrightarrow \cdots pc$
pop the destination address ($addr$), push the current PC value (which will be the address of the next instruction), and transfers control to $addr$.

$\cdots$ **`entry(`**n**`)`** $\Longrightarrow \cdots fp\ w_1 \cdots w_n$
pushes the current value of the FP register and sets FP to SP. Then it allocates n uninitialized words on the stack.

$\cdots v\ pc\ fp \cdots$ **`ret`** $\Longrightarrow \cdots v$
resets the stack pointer to the frame-pointer, pops the saved FP into the FP register, pops the return PC, and then jumps to the return address.

$\cdots v\ pc\ fp \cdots addr$ **`tailcall`** $\Longrightarrow \cdots v\ pc$
pops the code address $addr$, resets the stack pointer to the frame-pointer, pops the saved FP into the FP register,then transfers control to $addr$. This operation can be used to implement a tail call from a function f to a function g, where f and g have the same number of arguments.

$\cdots v_1 \cdots v_m$ **`ccall(`**n**`)`** $\Longrightarrow \cdots v$
Calls the nth C function. The C function will pop its arguments (v_i) from the stack and push its result.

3.7 Miscellaneous instructions

$\cdots$ **`nop`** $\Longrightarrow \cdots$
no operation.

$\cdots$ **`halt`** $\Longrightarrow \cdots$
halts the program.

4 Runtime functions

The VM provides the **`ccall`** instruction to invoke C functions. C functions expect their arguments on the stack and return their result on the stack.[1] C functions are specified by an index into the C function table.

The VM provides the following runtime system functions. We present them using the same convention that we used to present the semantics of the bytecode instruction set.

$\cdots i$ **`ccall`**`("VM_arg")` $\Longrightarrow \cdots s$
: pops the integer i and pushes the ith command-line argument (argument 0 is the name of the object file). The VM halts with an error if i is out of bounds.

$\cdots$ **`ccall`**`("VM_argc")` $\Longrightarrow \cdots i$
: pushes the number of command-line arguments (plus one) on the stack.

$\cdots s_1\ s_2$ **`ccall`**`("VM_concat")` $\Longrightarrow \cdots (s_1\hat{}\,s_2)$
: pops the strings s_1 and s_2 and pushes their concatenation.

$\cdots s$ **`ccall`**`("VM_fail")` $\Longrightarrow \cdots$
: pops the string s, prints it to standard error, and then halt the VM with an error.

$\cdots s$ **`ccall`**`("VM_print")` $\Longrightarrow \cdots 0$
: prints the string s to standard output and then pushes the `Unit` value on the stack.

5 Instruction encodings

Most instructions in the VM are either one, two, or three bytes long.[2] The first byte is consists of a two-bit length field (bits 6 and 7), and a six-bit opcode field (bits 0-5). The length field encodes the number of extra instruction bytes (*i.e.*, zero for one-byte instructions, one for two-byte instructions, and two for three-byte instructions). In the case of the two and three byte instructions, the extra bytes contain immediate data (*e.g.*, the offset of a `load` instruction), which is stored in 2's complement big-endian format.[3] Figure 1 gives a list of the instructions and their lengths; note that some instructions have both one and two or two and three-byte forms. The actual opcodes for the VM instructions are given in the `opcode.sml` file, which is part of the sample code.

6 Running programs using the VM

To run a **Flang** program using the virtual machine, you first compile it to produce a VM object file (say *foo.vmo*) and then execute the following shell command:

vm [*options*] *foo.vmo* $arg_1\ \cdots\ arg_n$

[1]The project handout states that "It is the responsibility of the caller to remove the arguments from the stack," but I have decided that it is easier to let the runtime functions pop their arguments.

[2]The one exception if the **`int`** instruction, which has a five byte form.

[3]The term "big-endian" means that the most significant byte comes first. For example, the number $513 = 2 * 256 + 1$ is represented as the byte sequence $2, 1$.

Instruction	Length	Comment
add, **sub**, **mul**, **div**, **mod**, **neq**, **equ**, **less**, **lesseq**, **not**, **boxed**, **size**, **subscript**	1	
alloc(n)	2	if $0 \leq n < 256$
alloc(n)	3	if $256 \leq n < 2^16$
select(i)	2	if $0 \leq i < 256$
select(i)	3	if $256 \leq i < 2^16$
explode	1	
int(n)	2	if $-128 \leq n < 128$
int(n)	3	if $n < -128$ or $128 \leq n$
int(n)	5	if $n < -2^15$ or $2^15 \leq n$
literal(n)	2	if $-128 \leq n < 128$
literal(n)	3	if $n < -128$ or $128 \leq n$
label(n)	2	if $-128 \leq n < 128$
label(n)	3	if $n < -128$ or $128 \leq n$
swap	1	
swap(n)	2	$0 \leq n < 256$
push(n)	2	$0 \leq n < 256$
pop	1	
pop(n)	2	$0 \leq n < 256$
loadlocal(n)	2	$-128 \leq n < 128$
loadlocal(n)	3	if $n < -128$ or $128 \leq n$
storelocal(n)	2	$-128 \leq n < 128$
storelocal(n)	3	if $n < -128$ or $128 \leq n$
loadglobal(n)	2	$n < 256$
loadglobal(n)	3	if $256 \leq n < 2^{16}$
pushep, **popep**	1	
jmp(n)	2	if $-128 \leq n < 128$
jmp(n)	3	if $n < -128$ or $128 \leq n$
jmpif(n)	2	if $-128 \leq n < 128$
jmpif(n)	3	if $n < -128$ or $128 \leq n$
call(n)	2	if $-128 \leq n < 128$
call(n)	3	if $n < -128$ or $128 \leq n$
entry(n)	2	$0 \leq n < 256$
entry(n)	3	if $256 \leq n < 2^{16}$
ret, **tailcall**	1	
ccall(i)	2	$0 \leq i < 256$
nop, **halt**	1	

Figure 1: VM instruction lengths

The virtual machine understands three options:

-h print a help message describing the options.

-t trace the program's execution

-g display garbage collector messages

The name of the object file and the command-line arguments following it ($arg_1 \quad \cdots \quad arg_n$) are available to the **Flang** program using the `arg` built-in function.

7 Document history

March 11 updated description of `VM_print` built-in function to note that it pushes the `Unit` value on the stack.

March 11 added **`size`** and **`subscript`** to Figure 1.

March 1 Original version.