The SMite Virtual Machine

Reuben Thomas

7th April 2019

Typographical conventions

Instructions and registers are shown in `Typewriter` font; interface calls are shown in **Bold** type.

Addresses are given in bytes and refer to the VM address space except where stated. Addresses are written in hexadecimal; hex numbers are prefixed with "0x".

1 Introduction

SMite is a simple virtual machine for study and experiment. It is a stack machine, based on the more complex register machine [3]. This paper gives a full description of SMite.

SMite is conceptually (and usually in fact) a library, embedded in other programs; it supports a simple object module format.

2 Architecture

SMite's address unit is the byte, which is eight bits. Words are `WORD_BYTES` bytes. Most of the quantities on which SMite operates are words. The size of the byte and range of word sizes allowed have been chosen with a view to making efficient implementation possible on the vast majority of current machine architectures.

Words have the bytes stored in big-endian or little-endian order, according to `ENDISM`.

2.1 Registers

The registers are word quantities; they are listed, with their functions, in table 1.

2.2 Memory

SMite's memory is a contiguous sequence of bytes with addresses starting at 0. The address of a word is that of the byte in it with the lowest address.

Register	Function
PC	The Program Counter. Points to the next word from which I may be loaded.
I	The Instruction register. Contains instructions to be executed.
BAD	The invalid stack position, or invalid or unaligned memory address, that last caused an error.
STACK_DEPTH	The number of items on the stack.
ENDISM	The endianness of SMite: 0 = Little-endian, 1 = Big-endian.
WORD_BYTES	The number of bytes in a word. Must be in the range 2 to 32 inclusive, and a power of 2.

Table 1: Registers

2.3 Stack

The stack is a LIFO stack of words used for passing values to instructions and routines and for holding subroutine return addresses. To **push** an item on to the stack means to add a new item to the top of the stack, increasing the stack depth by 1; to **pop** an item means to reduce the stack depth by 1. Instructions that change the number of items on the stack implicitly pop their arguments and push their results.

2.4 Operation

Before SMite is started, ENDISM should be set to 0 or 1 according to the implementation, and WORD_BYTES to the appropriate value. The other registers should be initialised to 0.

ENDISM and WORD_BYTES must not change.

Execution proceeds as follows:

```
begin
    execute the instruction in the 8 least-significant bits of I
    shift I logically 8 bits to the right
repeat
```

In the latter case, the contents of the execution cycle is executed once, and control returns to the calling program.

2.5 Errors and termination

When SMite encounters certain abnormal situations, such as an attempt to access an invalid address, or divide by zero, an **error** is **raised**, and execution terminates; an **error code** is returned to the caller. The instruction has no effect, and PC is not advanced to the next instruction. If the error is a stack or memory access error, BAD is set to the stack position or address that caused the error.

Execution can also be terminated explicitly by performing a HALT instruction (see section 3.10).

Error codes are unsigned numbers. 0 to 128 are reserved for SMite's own error codes; the meanings of those that may be raised by SMite are shown in table 2.

Code	Meaning
0	**single_step()** has terminated without error.
1	Invalid opcode (see section 3.12).
2	Stack overflow. `BAD` is set to the extra number of words of stack space required.
3	Invalid stack read. `BAD` is set to the invalid stack position.
4	Invalid stack write. `BAD` is set to the invalid stack position.
5	Invalid memory read. `BAD` is set to the invalid address.
6	Invalid memory write. `BAD` is set to the invalid address.
7	Address alignment error: raised when an instruction is given a valid address, but insufficiently aligned.
8	Invalid size (greater than log_2`WORD_BYTES`).
9	Division by zero attempted (see section 3.7).
128	A `HALT` instruction was executed.

Table 2: Errors raised by SMite

3 Instruction set

The instruction set is listed below, with the instructions grouped according to function. The instructions are given in the following format:

`NAME` `( before - after )`

Description.

The first line consists of the name of the instruction. On the right is the stack effect, which shows the effect of the instruction on the stack. Underneath is the description.

Stack effects are written

`( before - after )`

where *`before`* and *`after`* are stack pictures showing the items on top of a stack before and after the instruction is executed. An instruction only affects the items shown in its stack effects. The brackets and dashes serve merely to delimit the stack effect and to separate *`before`* from *`after`*. **Stack pictures** are a representation of the top-most items on the stack, and are written

$$i_1 \ i_2 \ldots i_{n-1} \ i_n$$

where the i_k are stack items, each of which occupies a whole number of words, with i_n being on top of the stack. The symbols denoting different types of stack item are shown in table 3.

Types are only used to indicate how instructions treat their arguments and results; SMite does not distinguish between stack items of different types. In stack pictures the most general argument types with which each instruction can be supplied are given; subtypes may be substituted. Using the phrase "$i \Rightarrow j$"

Symbol	Data type
flag	a Boolean flag, 1 for true and 0 for false
size	an integer in the range 0 to log_2WORD_BYTES inclusive.
n	signed number
u	unsigned number
n|*u*	number (signed or unsigned)
x	unspecified word
addr	address
a-addr	word-aligned address

Table 3: Types used in stack effects

to denote "i is a subtype of j", table 4 shows the subtype relationships. The subtype relation is transitive.

u ⇒ *x*
n ⇒ *x*
flag ⇒ *u*
size ⇒ *u*
a-addr ⇒ *addr* ⇒ *u*

Table 4: The subtype relation

Numbers are represented in twos complement form. *addr* consists of all valid virtual machine addresses. Numeric constants can be included in stack pictures, and are of type *n*|*u*.

Each type may be suffixed by a number in stack pictures; if the same combination of type and suffix appears more than once in a stack effect, it refers to identical stack items. Alternative *after* pictures are separated by "|", and the circumstances under which each occurs are detailed in the instruction description.

Ellipsis is used for indeterminate numbers of specified types of word.

3.1 Instruction fetch

NEXT (-)

If I is not 0, raise error 1. Load the word pointed to by PC into I then add WORD_BYTES to PC. If PC is not word-aligned, raise error 7, setting BAD to the contents of PC.

3.2 Control

These instructions implement unconditional and conditional branches, and subroutine call and return (subroutine return is BRANCH):

BRANCH (*a-addr* -)

Set PC to *a-addr*. Perform the action of NEXT.

BRANCHZ (*flag a-addr* -)

If *flag* is false then set PC to *a-addr* and perform the action of NEXT.

CALL (*a-addr*$_1$ - *a-addr*$_2$)

Exchange PC with the top stack value. Perform the action of NEXT.

3.3 Stack manipulation

These instructions manage the stack:

POP (*x* -)

Remove *x* from the stack.

DUP ($x_u \ldots x_0$ *u* - $x_u \ldots x_0$ x_u)

Remove *u*. Copy x_u to the top of the stack.

SWAP ($x_{u+1} \ldots x_0$ *u* - x_0 $x_u \ldots x_1$ x_{u+1})

Exchange the top stack item with the *u+1*th.

GET_STACK_DEPTH (- *u*)

u is the value of STACK_DEPTH, the number of items on the stack.

SET_STACK_DEPTH (*u* -)

Set STACK_DEPTH to *u*.

3.4 Literals

LIT (- *n*)

The word pointed to by PC is pushed on to the stack, and PC is incremented to point to the following word.

LIT_PC_REL (- *n*)

Like LIT, except that the initial value of PC is added to the value pushed on to the stack.

3.5 Logic and shifts

Logic functions:

NOT (x_1 - x_2)

Invert all bits of x_1, giving its logical inverse x_2.

AND (x_1 x_2 - x_3)

x_3 is the bit-by-bit logical "and" of x_1 with x_2.

OR (x_1 x_2 - x_3)

x_3 is the bit-by-bit inclusive-or of x_1 with x_2.

XOR (x_1 x_2 - x_3)

x_3 is the bit-by-bit exclusive-or of x_1 with x_2.

Shifts:

`LSHIFT` (x_1 u - x_2)

Perform a logical left shift of u bit-places on x_1, giving x_2. Put zero into the least significant bits vacated by the shift. If u is greater than or equal to the number of bits in a word, x_2 is zero.

`RSHIFT` (x_1 u - x_2)

Perform a logical right shift of u bit-places on x_1, giving x_2. Put zero into the most significant bits vacated by the shift. If u is greater than or equal to the number of bits in a word, x_2 is zero.

`ARSHIFT` (x_1 u - x_2)

Perform an arithmetic right shift of u bit-places on x_1, giving x_2. Copy the original most-significant bits into the most significant bits vacated by the shift. If u is greater than or equal to the number of bits in a word, all the bits of x_2 are the same as the original most-significant bit.

`SIGN_EXTEND` (*u size - n*)

Sign extend the 2^{size}-byte quantity u to n.

3.6 Comparison

These words compare two numbers (or, for equality tests, any two words) on the stack, returning a flag:

`EQ` (x_1 x_2 - *flag*)

flag is true if and only if x_1 is bit-for-bit the same as x_2.

`LT` (n_1 n_2 - *flag*)

flag is true if and only if n_1 is less than n_2.

`ULT` (u_1 u_2 - *flag*)

flag is true if and only if u_1 is less than u_2.

3.7 Arithmetic

These instructions consist of monadic and dyadic operators. All calculations are made without bounds or overflow checking, except as detailed for certain instructions.

Negation and addition:

`NEGATE` (n_1 - n_2)

Negate n_1, giving its arithmetic inverse n_2.

`ADD` ($n_1|u_1$ $n_2|u_2$ - $n_3|u_3$)

Add $n_2|u_2$ to $n_1|u_1$, giving the sum $n_3|u_3$.

Multiplication and division (note that all division instructions raise error -4 if division by zero is attempted):

`MUL` ($n_1|u_1$ $n_2|u_2$ - $n_3|u_3$)

Multiply $n_1|u_1$ by $n_2|u_2$ giving the product $n_3|u_3$.

`DIVMOD` (n_1 n_2 - n_3 n_4)

Divide n_1 by n_2 using symmetric division, giving the single-word quotient n_3 and the single-word remainder n_4. The quotient is rounded towards zero.

`UDIVMOD` (u_1 u_2 - u_3 u_4)

Divide u_1 by u_2, giving the single-word quotient u_3 and the single-word remainder u_4.

3.8 Memory

These instructions fetch and store quantities to and from memory. If a given address is incorrectly aligned, raise error 7, setting `BAD` to the address.

`LOAD` (*addr size* - *x*)

Load the 2^{size}-byte quantity *x* stored at *addr*, which must be a multiple of *size*. Any unused high-order bits are set to zero.

`STORE` (*x addr size* -)

Store the 2^{size} least-significant bytes of *x* at *addr*, which must be a multiple of *size*.

3.9 External access

This instruction is a general-purpose escape. In contrast to extra instructions, which are intended to be frequently used and potentially optimized, `EXT`'s behaviour is not defined; in particular, it has an arbitrary stack effect.

`EXT` ( )

Perform arbitrary actions.

3.10 Termination

This instruction terminates execution (see section 2.5):

`HALT` (-)

Terminate execution with error code 128.

3.11 Instruction encoding

Instructions are encoded as 8-bit opcodes; opcodes are packed into words, which are executed starting at the least-significant bits.

3.12 Instruction opcodes

Table 5 lists the instruction opcodes in numerical order. Other instruction opcodes are undefined.

Opcode	Instruction	Opcode	Instruction
0x00	NEXT	0x10	EQ
0x01	BRANCH	0x11	LT
0x02	BRANCHZ	0x12	ULT
0x03	CALL	0x13	NEGATE
0x04	POP	0x14	ADD
0x05	DUP	0x15	MUL
0x06	SWAP	0x16	DIVMOD
0x07	undefined	0x17	UDIVMOD
0x08	NOT	0x18	LOAD
0x09	AND	0x19	STORE
0x0a	OR	0x1a	LIT
0x0b	XOR	0x1b	LIT_PC_REL
0x0c	LSHIFT		
1cundefined 0x0d	RSHIFT		
1dundefined 0x0e	ARSHIFT	0x1e	EXT
0x0f	SIGN_EXTEND	0x1f	HALT

Table 5: Instruction opcodes

4 External interface

- Implementations should provide an **API** to create and run virtual machine instances, and provide access to its registers, stack and memory.
- Implementations can add **extra instructions** to provide extra computational primitives.
- The EXT instruction can offer access to system facilities, previously written code, native libraries and so on.
- The **object module** format allows compiled code to be saved, reloaded and shared between systems.

4.1 Object module format

The object module starts with the ASCII codes of the letters "smite" followed by ASCII NUL (0x00), then the one-byte values of the ENDISM and WORD_BYTES registers of the system which saved the module, then a word (of the given endianness and size) containing the number of bytes the code occupies. Then follows the code.

Object modules have a simple structure, as they are only intended for loading an initial memory image into SMite.

Acknowledgements

Martin Richards's demonstration of his BCPL-oriented Cintcode virtual machine [1] convinced me it was going to be fun working on virtual machines. He

also supervised my BA dissertation project, Beetle [2], on which SMite is based.

References

[1] Martin Richards. Cintcode distribution, 2000. `https://www.cl.cam.ac.uk/~mr/BCPL.html`.

[2] Reuben Thomas. Beetle and pForth: a Forth virtual machine and compiler. BA dissertation, University of Cambridge, 1995. `https://rrt.sc3d.org/`.

[3] Reuben Thomas. *Mite: a basis for ubiquitous virtual machines*. PhD thesis, University of Cambridge Computer Laboratory, November 2000. `https://rrt.sc3d.org/`.