The SMite Virtual Machine

Reuben Thomas

26th February 2019

Typographical conventions

Actions 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_SIZE` bytes. The word is the size of the numbers and addresses on which SMite operates, and of the items placed on the stacks. 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 may have the bytes stored in big-endian or little-endian order. The address of a word is that of the byte in it with the lowest address.

2.1 Registers

The registers, each with its function, are set out in table 1.

The registers are word quantities.

To ease efficient implementation, the registers may only be accessed by actions (see section 3.9); not all registers are accessible, and only a few are writable.

2.2 Memory

SMite's memory is a contiguous sequence of bytes with addresses starting at 0.

Register	Function
PC	The Program Counter. Points to the next byte from which an instruction may be loaded.
BAD_PC	The value of PC when an error was last raised.
BAD_ADDRESS	The invalid stack position, or invalid or unaligned memory address, that caused the last memory error.
ITYPE	The type of an instruction to be executed: 0 for a number, and 1 for an action.
I	The Instruction. Holds the opcode of an instruction to be executed.
STACK_DEPTH	The number of items on the stack.
ENDISM	The endianness of SMite: 0 = Little-endian, 1 = Big-endian.
WORD_SIZE	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 stacks of words used for passing values to actions 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. Actions 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_SIZE to the appropriate value. The other registers should be initialised to 0.

ENDISM and WORD_SIZE must not change while SMite is executing.

SMite is started by a call to the interface calls **run()** or **single_step()** (see section 4.2). In the former case, the execution cycle is entered:

```
begin
    decode the instruction pointed to by PC into I
    set PC to point to the next instruction
    execute the instruction in I
repeat
```

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

Note that the calls **run()** and **single_step()** do not perform the initialisation specified above.

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 that caused the error to be raised is considered not to have executed, and BAD_PC is set to point to it. If the error is a stack or memory access error, `BAD_ADDRESS` is set to the stack position or address that caused the error.

Execution can also be terminated explicitly by performing a `HALT` action (see section 3.2), which also returns a code. The `HALT` action *is* considered to have completed.

Error codes are signed numbers. Conventionally, negative numbers are considered to represent error conditions, and non-negative numbers (in particular, zero) successful termination. -1 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
-1	Invalid opcode (see section 3.12).
-2	Attempt to set `STACK_DEPTH` greater than `STACK_SIZE`. `BAD_ADDRESS` is set to the extra number of words of stack space required.
-3	Invalid stack read. `BAD_ADDRESS` is set to the invalid stack position.
-4	Invalid stack write. `BAD_ADDRESS` is set to the invalid stack position.
-5	Invalid memory read. `BAD_ADDRESS` is set to the invalid address.
-6	Invalid memory write. `BAD_ADDRESS` is set to the invalid address.
-7	Address alignment error: raised when an instruction expecting a word-aligned address is given a valid but non-aligned address.
-8	Division by zero attempted (see section 3.5).
-128	**single_step()** has terminated without error.

Table 2: Errors raised by SMite

3 Instruction set

The instruction set is listed in sections 3.1 to **??**, 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

$$\texttt{i}_1\ \texttt{i}_2 \texttt{...} \texttt{i}_{n-1}\ \texttt{i}_n$$

where the $\texttt{i}_k$ are stack items, each of which occupies a whole number of words, with $\texttt{i}_n$ being on top of the stack. The symbols denoting different types of stack item are shown in table 3.

Symbol	Data type
`flag`	a Boolean flag, 1 for true and 0 for false
`byte`	byte
`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

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 "$i \Rightarrow j$" 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`
`byte` ⇒ `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.

The symbols `i*x` and `j*x` are used to denote different collections of zero or more words of any data type. Ellipsis is used for indeterminate numbers of specified types of word.

3.1 Numbers

`number` (– n)

The number is pushed on to the stack.

3.2 Errors

This action raises an error (see section 2.5):

`HALT` (x –)

Stop SMite, returning reason code x to the calling program (see section 4.2).

3.3 Stack manipulation

These actions manage the stack:

`POP` ($x_u \ldots x_1$ u –)

Remove u items 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` ($x_u \ldots x_0$ u – $x_0 x_{u-1} \ldots x_1$ x_u)

Exchange the top stack item with the uth. If u is zero, do nothing.

`ROTATE` ($x_n \ldots x_0$ n – $x_{n-1} \ldots x_1$ x_{+n} | x_0 $x_n \ldots x_1$)

If n is non-negative, rotate the nth stack item to the top; otherwise, rotate the top stack item to position n.

3.4 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.

3.5 Arithmetic

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

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 actions 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.

3.6 Logic and shifts

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.

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.

3.7 Memory

These actions fetch and store words and bytes to and from memory; there is also an action to add a number to another stored in memory:

`LOAD` (*a-addr* – *x*)

x is the value stored at *a-addr*.

`STORE` (*x a-addr* –)

Store *x* at *a-addr*.

`LOADB` (*addr* – *byte*)

Fetch the byte stored at *addr*. The unused high-order bits are all zeroes.

`STOREB` (*byte addr* –)

Store *byte* at *addr*. Only one byte is transferred.

3.8 Control

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

`BRANCH` (*addr* –)

Set `PC` to *addr*.

`BRANCHZ` (*flag addr* –)

If *flag* is false then set `PC` to *addr*.

`CALL` ($addr_1$ – $addr_2$)

Exchange `PC` with the top stack value.

3.9 Registers

`GET_WORD_SIZE` (– *u*)

u is the value of `WORD_SIZE`.

`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*.

3.10 No-op

`NOP` (–)

Do nothing.

3.11 Instruction encoding

Actions are encoded as a single byte with the top two bits clear.

Numbers are words encoded by one or more bytes, as follows: the significant bits of the number are split into groups of six bits, starting at the most significant end, and then stored in little-endian order in consecutive bytes. All but the last byte have the seventh bit set and eighth bit clear; the final byte has the eighth bit set, and the seventh bit matches the sign bit.

3.12 Action opcodes

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

Opcode	Action	Opcode	Action
0x00	`HALT`	0x10	`NOT`
0x01	`POP`	0x11	`AND`
0x02	`DUP`	0x12	`OR`
0x03	`SWAP`	0x13	`XOR`
0x04	`ROTATE`	0x14	`LOAD`
0x05	`EQ`	0x15	`STORE`
0x06	`LT`	0x16	`LOADB`
0x07	`ULT`	0x17	`STOREB`
0x08	`NEGATE`	0x18	`BRANCH`
0x09	`ADD`	0x19	`BRANCHZ`
0x0a	`MUL`	0x1a	`CALL`
0x0b	`DIVMOD`	0x1b	`GET_WORD_SIZE`
0x0c	`UDIVMOD`	0x1c	`GET_STACK_DEPTH`
0x0d	`LSHIFT`	0x1d	`SET_STACK_DEPTH`
0x0e	`RSHIFT`	0x1e	`NOP`
0x0f	`ARSHIFT`		

Table 5: Action opcodes

4 External interface

SMite's external interface comes in three parts. The calling interface allows SMite to be controlled by other programs. The `EXTRA` action (see section **??**) allows implementations to provide access to system facilities, previously written code, code written in other languages, and the speed of machine code in time-critical situations. 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" padded to eight bytes by ASCII NULs (0x00), then values of the `ENDISM` and `WORD_SIZE` registers of the system which saved the module, then the number of bytes the code occupies. These values are all encoded as in section 3.11. Then follows the code.

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

4.2 Calling interface

The calling interface is difficult to specify with the same precision as the rest of SMite, as it may be implemented in any language. However, since only basic types are used, and the semantics are simple, it is expected that implementations in different language producing the same result will be easy to program. A Modula-like syntax is used to give the definitions here. Implementation-defined error codes must be documented, but are optional. All addresses passed as parameters must be word-aligned. An implementation of SMite must provide the following calls:

run () : integer
Start SMite by entering the execution cycle as described in section 2.4. If SMite ever executes a `HALT` action (see section 3.2), the reason code is returned as the result.

single_step () : integer
Execute a single pass of the execution cycle. If an error is raised, it is returned; otherwise, return -128.

load_object (*file, address*) : integer
Load the object module given by the file descriptor *file*, which may be a filename or some other specifier, to the VM address *address*. If the module is invalid, then return -2. If the module cannot be loaded by the current implementation (for example, the word size or `ENDISM` is incompatible), return -3. If the code will not fit into memory at the address given, or the address is out of range or unaligned, return -4. Otherwise load the code into memory, converting it if the endianness value is different from the current value of `ENDISM`. The result is the length of the code in bytes if successful, and -1 if there is a filing system or other error.

save_object (*file, address, length*) : integer
Save the area of memory specified by *address* and *length* to the file descriptor given by *file*. On success, 0 is returned. If the address or length are invalid, the return code is -2. If a file system error occurs, the return code is -1.

SMite must also provide access to its registers and address space through appropriate data objects.

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/`.