# The SMite Virtual Machine

Reuben Thomas

12th January 2019

### Typographical conventions

Actions and registers are shown in `Typewriter` font; interface calls are shown in **Bold** type, and followed by empty parentheses.

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, but certain implementation-dependent features, such as the size of the stacks, are purposely left unspecified, and the exact method of implementation is left to the implementor in many particulars.

SMite is self-contained. Machine code routines on the host computer may be accessed using the `CALL_NATIVE` action, which can also be used to implement I/O (see 3.10). SMite supports a simple object module format.

SMite is conceptually (and usually in fact) a library, embedded in other programs. A small interface is provided for other programs to control SMite.

## 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 3.7); not all registers are accessible, and only a few are writable.

| Register | Function |
|---|---|
| PC | The Program Counter. Points to the next byte from which an instruction may be loaded. |
| 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. |
| MEMORY | The size in bytes of SMite's memory, which must be a whole number of words. |
| SDEPTH | The data Stack DEPTH (number of items on the stack). |
| RDEPTH | The Return stack DEPTH (number of items on the return stack. |
| SSIZE | The number of words allocated for the data stack. |
| RSIZE | The number of words allocated for the return stack. |
| ENDISM | The endianness of SMite: 0 = Little-endian, 1 = Big-endian. |
| WORD_SIZE | The number of bytes in a word. Must be a power of 2, and in the range 2 to 32 inclusive. |
| NATIVE_POINTER_SIZE | The number of bytes in a host machine pointer (for CALL_NATIVE; see section 3.10). |

Table 1: Registers

## 2.2 Memory

SMite's memory is a contiguous sequence of bytes with addresses in the range 0 to MEMORY − 1.

## 2.3 Stacks

The data and return stacks are LIFO stacks of words. To **push** an item on to a 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 a stack implicitly pop their arguments and push their results.

The data stack is used for passing values to actions and routines and the return stack for holding subroutine return addresses. The return stack may be used for other operations subject to the restrictions placed on it by its normal usage: it must be returned before a RET action to the state it was in directly after the corresponding CALL.

In what follows, for "the stack" read "the data stack"; the return stack is always mentioned explicitly.

## 2.4 Operation

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

ENDISM, WORD_SIZE and NATIVE_POINTER_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 next instruction into I from the bytes pointed to by PC
    set PC to point to the next byte after the end of the 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 Termination

When SMite encounters a `HALT` action (see section 3.9), execution terminates.

Reason codes which are also valid error codes (either reserved (see section 2.6) or user error codes) should not normally be used. This allows error codes to be passed back to the calling program, so that the calling program can handle certain errors without confusing error codes and reason codes.

## 2.6 Errors

Error conditions are dealt with by **raising** an **error**: the action of `HALT` is performed as if the error code had been given as its argument. The instruction that caused the error to be raised is then considered to have finished executing. Error codes are signed numbers. $-1$ to $-511$ are reserved for SMite's own error codes; the meanings of those that may be raised by SMite are shown in table 2.

An error can be raised explicitly by `HALT` (see section 3.9); errors are also raised by various conditions, such as an attempt to access an invalid address, or divide by zero.

| Code | Meaning |
|---|---|
| $-9$ | Invalid address (see below). |
| $-10$ | Division by zero attempted (see section 3.4). |
| $-23$ | Address alignment error (see below). |
| $-256$ | Invalid opcode (see section 3.12). |

Table 2: Errors raised by SMite

Error $-9$ is raised whenever an attempt is made to access an invalid address, either by an instruction, or during an instruction fetch (because `PC` contains an invalid address). Error $-23$ is raised when an action expecting an address of type *`a-addr`* (word-aligned) is given a non-aligned address.

# 3 Instruction set

The instruction set is listed in sections 3.1 to 3.10, with the instructions grouped according to function. The instructions are given in the following format:

```
NAME                                                ( before - after )
                                                  R: ( before - after )
```

Description.

The first line consists of the name of the instruction. On the right are the stack effect or effects, which show the effect of the instruction on the data and return (R) stacks. 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.

| 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 |
| *n-addr* | native 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.

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*. Native addresses may occupy more than one word, according to the values of `WORD_SIZE` and `NATIVE_POINTER_SIZE`; the precise format of this address is implementation-dependent.

| |
|---|
| $u \Rightarrow x$ |
| $n \Rightarrow x$ |
| $flag \Rightarrow u$ |
| $byte \Rightarrow u$ |
| $a\text{-}addr \Rightarrow addr \Rightarrow u$ |

Table 4: The subtype relation

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.

If an instruction does not modify the return stack, the corresponding stack picture is omitted.

## 3.1 Numbers

number ( - *n* )

The number is pushed on to the stack.

## 3.2 Stack manipulation

These actions manage the data stack and move values between stacks.

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 *u*th. If *u* is zero, do nothing.

POP2R ( *x* - )
R: ( - *x* )

Move *x* to the return stack.

RPOP ( - *x* )
R: ( *x* - )

Move *x* from the return stack to the data stack.

RPUSH ( *u* - $x_u$ )
R: ( $x_u$ $x_{u-1} \ldots x_0$ - $x_u$ $x_{u-1} \ldots x_0$ )

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

## 3.3 Comparison

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

`LT` ( $n_1$ $n_2$ - *flag* )

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

`EQ` ( $x_1$ $x_2$ - *flag* )

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

`ULT` ( $u_1$ $u_2$ - *flag* )

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

## 3.4 Arithmetic

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

Addition and negation:

`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 $n_3|u_3$.

`NEGATE` ( $n_1$ - $n_2$ )

Negate $n_1$, giving its arithmetic inverse $n_2$.

Multiplication and division (note that all division actions raise error $-10$ 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 $n_3|u_3$.

`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 $u_4$.

`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.

## 3.5 Logic and shifts

These actions consist of bitwise logical operators and bitwise shifts.

Logic functions:

`INVERT` ( $x_1$ - $x_2$ )

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.

## 3.6 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.

## 3.7 Registers

PUSH_SDEPTH ( - *u* )

*u* is the value of SDEPTH, the number of items on the stack.

STORE_SDEPTH ( *u* - )

Set SDEPTH to *u*.

PUSH_RP ( - *a-addr* )

*a-addr* is the value of RP.

STORE_RP ( *a-addr* - )

Set RP to *a-addr*.

PUSH_PC ( - *addr* )

Push PC on to the stack.

PUSH_SSIZE ( - *u* )

Push SSIZE on to the stack.

PUSH_RSIZE ( - *u* )

Push RSIZE on to the stack.

PUSH_MEMORY ( - *a-addr* )

Push MEMORY on to the stack.

`PUSH_WORD_SIZE` ( - *u* )

*u* is the value of `WORD_SIZE`.

`PUSH_NATIVE_POINTER_SIZE` ( - *u* )

*u* is the value of `NATIVE_POINTER_SIZE`.

## 3.8 Control structures

These actions implement unconditional and conditional branches, and subroutine call and return; there is also a no-op.

No-op:

`NOP` ( - )

Do nothing.

Branches:

`BRANCH` ( *addr* - )

Set `PC` to *addr*.

`BRANCHZ` ( *flag addr* - )

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

Subroutine call and return:

`CALL` ( *a-addr*$_1$ - )
R: ( - *a-addr*$_2$ )

Push `PC` on to the return stack, put *a-addr*$_1$ into `PC`.

`RET` ( - )
R: ( *addr* - )

Put *addr* into `PC`.

## 3.9 Errors

This action gives access to SMite's error mechanism (see section 2.6).

`HALT` ( *x* - )

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

## 3.10 External access

These actions allow access to SMite's libraries, the operating system and native machine code.

`CALL_NATIVE` ( *n-addr* - )

Make a subroutine call to the routine at address *n-addr*, passing the current state as an argument.

`EXTRA` ( *i*x* - *j*x* )

Perform implementation-dependent actions; for example, this can be used to implement system-dependent functionality such as I/O.

### 3.11 Instruction encoding

Instructions 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 least significant end. The chunks are stored in consecutive bytes. All but the last byte have the seventh bit set and eighth bit clear. If the instruction is a number the final byte has the top two bits either both set or both clear, to match the number's most significant bit; otherwise, the top bit is set and the second bit clear, to indicate an action.

### 3.12 Action opcodes

Table 5 lists the action opcodes in numerical order. Other action opcodes are undefined. Undefined action opcodes raise error $-256$.

| Opcode | Action | Opcode | Action | Opcode | Action |
|---|---|---|---|---|---|
| 0x00 | NOP | 0x10 | AND | 0x20 | PUSH_WORD_SIZE |
| 0x01 | POP | 0x11 | OR | 0x21 | PUSH_NATIVE_POINTER_SIZE |
| 0x02 | DUP | 0x12 | XOR | 0x22 | PUSH_SDEPTH |
| 0x03 | SWAP | 0x13 | LSHIFT | 0x23 | STORE_SDEPTH |
| 0x04 | RPUSH | 0x14 | RSHIFT | 0x24 | PUSH_RDEPTH |
| 0x05 | POP2R | 0x15 | LOAD | 0x25 | STORE_RDEPTH |
| 0x06 | RPOP | 0x16 | STORE | 0x26 | PUSH_PC |
| 0x07 | LT | 0x17 | LOADB | 0x27 | PUSH_SSIZE |
| 0x08 | EQ | 0x18 | STOREB | 0x28 | PUSH_RSIZE |
| 0x09 | ULT | 0x19 | BRANCH | 0x29 | PUSH_MEMORY |
| 0x0a | ADD | 0x1a | BRANCHZ | | |
| 0x0b | NEGATE | 0x1b | CALL | | |
| 0x0c | MUL | 0x1c | RET | | |
| 0x0d | UDIVMOD | 0x1d | HALT | | |
| 0x0e | DIVMOD | 0x1e | CALL_NATIVE | | |
| 0x0f | INVERT | 0x1f | EXTRA | | |

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 CALL_NATIVE action (see section 3.10) 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:

**native_address** (*integer, boolean*) : pointer
: Return a native pointer corresponding to the given VM address. If the address is invalid, or the Boolean flag is true and the address is read-only, then a distinguished invalid pointer is returned.

**run** () : integer
: Start SMite by entering the execution cycle as described in section 2.4. If SMite ever executes a `HALT` action (see section 3.9), the reason code is returned as the result.

**single_step** () : integer
: Execute a single pass of the execution cycle, and return reason code $-257$, unless a `HALT` action was obeyed (see section 3.9), in which case the reason code passed to it is returned.

**load_object** (*file, address*) : integer
: Load the object module specified by *file*, which may be a filename or some other specifier, to the VM address *address*. First the module's header is checked; if the first seven bytes are not as specified above in section 4.1, or the endianness value is not 0 or 1, then return $-2$. If the code will not fit into memory at the address given, or the address is out of range or unaligned, return $-1$. 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 some other implementation-defined value if there is a filing system or other error.

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/`.