The SMite Virtual Machine

Reuben Thomas

13th September 2018

Abstract

The design of the SMite virtual machine is described. SMite's is a deliberately simple and highly portable low-level VM intended for study and experiment. SMite is able to call machine code and access the operating system.

Typographical notes

Instructions and SMite's 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 SMite's address space except where stated. Addresses are written in hexadecimal; hex numbers are prefixed with "$".

1 Introduction

SMite is a simple virtual machine for study and experiment. It is a stack machine, based on the more complex register machine [2]. 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. I/O can be implemented via extra instructions (see 3.12). Machine code routines on the host computer may be accessed using the `LINK` instruction. 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 wide. Words are four bytes wide. The word is the size of the numbers and addresses on which SMite operates, and of the items placed on the stacks. The word size is fixed to ensure compatibility of object code between implementations on different machines; the size of the byte and word have been chosen with a view to making efficient implementation of SMite 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.

Register	Function
`PC`	The `Program Counter`. Points to the next word from which an instruction word may be loaded.
`I`	The `Instruction`. Holds the opcode of an instruction to be executed.
`A`	The instruction `Accumulator`. Holds the opcodes of instructions to be executed.
`MEMORY`	The size in bytes of SMite's main memory, which must be a multiple of four.
`SP`	The data `Stack Pointer`.
`RP`	The `Return` stack `Pointer`.
`S0`	The data `Stack` base.
`R0`	The `Return` stack base.
`SSIZE`	The number of words allocated for the data stack.
`RSIZE`	The number of words allocated for the return stack.
`HANDLER`	The address placed in `PC` by a `THROW` instruction.
`ENDISM`	The endianness of SMite: 0 = Little-endian, 1 = Big-endian.
`BADPC`	The contents of `PC` when the last exception was raised.
`INVALID`	The last address which caused an address exception.
`PSIZE`	The number of words in a host machine pointer (for `LINK`).

Table 1: SMite's registers

All of the registers are word-wide quantities except for `I` and `ENDISM`, which are one byte wide.

To ease efficient implementation, the registers may only be accessed by instructions (see section 3.8); not all registers are accessible, and only a few are writable.

2.2 Memory

SMite's memory is a discontiguous sequence of bytes with addresses in the range 0 to $2^{32} - 1$. Some locations may be read-only. The memory is contiguous in the range 0 to `MEMORY` $- 1$; this part is referred to as "main memory".

2.3 Stacks

The data and return stacks are word-aligned LIFO stacks of words. The stack pointers point to the top stack item on each stack. To **push** an item on to a stack means to store the item in the word beyond the stack pointer and then adjust the pointer to point to it; to **pop** an item means to make the pointer point to the second item on the stack. Instructions 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 instructions 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` instruction 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 `PSIZE` to the appropriate value. The other registers should be initialised as shown in table 2, except for `I`, which need not be initialised.

Register	Initial value
`PC`	$0
`A`	$0
`THROW`	$0
`BADPC`	$FFFFFFFF
`INVALID`	$FFFFFFFF

Table 2: Registers with prescribed initial values

`MEMORY`, `ENDISM` and `PSIZE` 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
    copy the least-significant byte of A to I
    shift A arithmetically 8 bits to the right
    execute the instruction in I
repeat
```

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

The execution loop need not be implemented as a single loop; it is designed to be short enough that the contents of the loop can be appended to the code implementing each instruction.

Note that the calls **run()** and **single_step()** do not perform the initialisation specified above; that must be performed before calling them.

2.5 Termination

When SMite encounters a `HALT` instruction (see section 3.11), it returns the top data stack item as the reason code, unless `SP` does not point to a valid word, in which case reason code -257 is returned (see section 2.6).

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

2.6 Exceptions

When a `THROW` instruction (see section 3.11) is executed, an **exception** is said to have been **raised**. The exception code is the number on top of the stack at the time the exception is raised. Some exceptions are raised by other instructions, for example by `DIVMOD` when division by zero is attempted; these push the exception code on to the stack and then execute a `THROW`.

Exception codes are signed numbers. -1 to -511 are reserved for SMite's own exception codes; the meanings of those that may be raised by SMite are shown in table 3.

Code	Meaning
-9	Invalid address (see below).
-10	Division by zero attempted (see section 3.5).
-20	Attempt to write to a read-only memory location.
-23	Address alignment exception (see below).
-256	Illegal opcode (see section 3.13).

Table 3: Exceptions raised by SMite

Exception -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). Exception -23 is raised when an instruction expecting an address of type *`a-addr`* (word-aligned) is given a non-aligned address. When SMite raises an address exception (-9 or -23), the offending address is placed in `INVALID`.

The initial values of `BADPC` and `INVALID` are unlikely to be generated by an exception, so it may be assumed that if the initial values still hold no exception has yet occurred.

If `SP` is unaligned when an exception is raised, or putting the code on the stack would cause `SP` to be out of range, the effect of a `HALT` with code -257 is performed (although the actual mechanics are not, as that too would involve putting a number on the stack). Similarly, if `HANDLER` contains an invalid address, the effect of `HALT` with code -258 is performed.

3 Instruction set

The instruction set is listed in sections 3.3 to 3.12, 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 4.

Symbol	Data type
flag	flag
true	true flag
false	false flag
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 4: 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 5 shows the subtype relationships. The subtype relation is transitive.

u $\Rightarrow$ *x*
n $\Rightarrow$ *x*
byte $\Rightarrow$ *u*
a-addr $\Rightarrow$ *addr* $\Rightarrow$ *u*
flag $\Rightarrow$ *x*

Table 5: The subtype relation

Numbers are represented in twos complement form. *addr* consists of all valid 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*, *j*x* and *k*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. Some instructions have two forms, the latter ending in "`I`". This denotes `I`mmediate addressing: the instruction's argument is included in the instruction word (see section 3.1), rather than being placed separately in the next available word.

3.1 Programming conventions

Since branch destinations must be word-aligned, some instruction sequences may contain gaps. These must be padded with `NEXT` (opcode $00).

Literals and branch addresses should be placed in the word after the instruction. Further instructions may still be stored in the current word. If more than one literal or branch instruction is encoded in one instruction word, the literal values follow each other in successive words.

3.2 Execution cycle

`NEXT` performs an instruction fetch when SMite runs out of instructions in the `A` register.

`NEXT` (--)

Load the word pointed to by `PC` into `A` then add four to `PC`.

3.3 Stack manipulation

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

`POP` ($x_u \ldots x_1$ u --)

Remove u items from the stack.

`PUSH` ($x_u \ldots x_0$ u -- $x_u \ldots x_0$ x_u)

Remove u. Copy x_u to the top of the stack. If there are fewer than $u + 2$ items on the stack before `PUSH` is executed, the memory word which would have been x_u were there $u + 2$ items is copied 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.

`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)
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. If there are fewer than $u + 2$ items on the stack before `RPUSH` is executed, the memory word which would have been x_u were there $u + 2$ items is copied to the top of the data stack.

3.4 Comparison

These words compare two numbers (or, for equality tests, any two words) on the stack, returning a flag, true with all bits set if the test succeeds and false otherwise.

`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.

3.5 Arithmetic

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

Addition:

`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 exception -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.

Sign function:

`NEGATE` (n_1 -- n_2)
Negate n_1, giving its arithmetic inverse n_2.

3.6 Logic and shifts

These instructions consist of bitwise logical operators and bitwise shifts. The result of performing the specified operation on the argument or arguments is left on the stack.

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 32, 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 32, x_2 is zero.

3.7 Memory

These instructions fetch and store words and bytes to and from memory; there is also an instruction 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*)

If `ENDISM` is 1, exclusive-or *addr* with 3. Fetch the byte stored at *addr*. The unused high-order bits are all zeroes.

`STOREB` (*byte* *addr* --)

If `ENDISM` is 1, exclusive-or *addr* with 3. Store *byte* at *addr*. Only one byte is transferred.

3.8 Registers

As mentioned in section 2.1, the stack pointers `SP` and `RP` may only be accessed through special instructions:

`PUSH_SP` (-- a-*addr*)

a-*addr* is the value of `SP`.

`STORE_SP` (a-*addr* --)

Set SP to a-*addr*.

`PUSH_RP` (-- a-*addr*)

a-*addr* is the value of `RP`.

`STORE_RP` (*a-addr* --)

Set `RP` to *a-addr*.

`PUSH_PC` (-- *a-addr*)

Push `PC` on to the stack.

`PUSH_S0` (-- *a-addr*)

Push `S0` on to the stack.

`PUSH_SSIZE` (-- *u*)

Push `SSIZE` on to the stack.

`PUSH_R0` (-- *a-addr*)

Push `R0` on to the stack.

`PUSH_RSIZE` (-- *u*)

Push `RSIZE` on to the stack.

`PUSH_HANDLER` (-- *a-addr*)

Push `HANDLER` on to the stack.

`STORE_HANDLER` (*a-addr* --)

Set `HANDLER` to *a-addr*.

`PUSH_MEMORY` (-- *a-addr*)

Push `MEMORY` on to the stack.

`PUSH_BADPC` (-- *a-addr*)

Push `BADPC` on to the stack.

`PUSH_INVALID` (-- *a-addr*)

Push `INVALID` on to the stack.

`PUSH_PSIZE` (-- *u*)

u is the value of `PSIZE`.

3.9 Control structures

These instructions implement unconditional and conditional branches, and subroutine call and return.

Branches:

`BRANCH` (*a-addr* --)

Set `PC` to *a-addr*, then 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`.

Subroutine call and return:

`CALL` ($a\text{-}addr_1$ --)
R: (-- $a\text{-}addr_2$)

Push `PC` on to the return stack, put $a\text{-}addr_1$ into `PC`, then perform the action of `NEXT`.

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

Put *a-addr* into `PC`, then perform the action of `NEXT`.

3.10 Literals

This instructions encodes literal values which are placed on the stack.

`LITERAL` (-- *x*)

Push the word pointed to by `PC` on to the stack, then add four to `PC`.

3.11 Exceptions

These instructions give access to SMite's exception mechanisms.

`THROW` (--)

Put the contents of `PC` into `BADPC`, then load `PC` from `HANDLER`. Perform the action of `NEXT`. If `HANDLER` contains an out of range or unaligned address stop SMite, returning reason code -258 to the calling program (see section 4.2).

`HALT` (*x* --)

Stop SMite, returning reason code *x* to the calling program (see section 4.2). If `SP` is out of range or unaligned, -257 is returned as the reason code.

3.12 External access

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

`LINK` (*i*x* --)

Make a subroutine call to the routine at the address given (in the host machine's format, padded out to a number of words) on the data stack. The size and format of this address are machine-dependent. If the address given is 0, perform implementation-dependent actions; for example, this can be used to implement system-dependent functionality such as I/O.

3.13 Opcodes

Table 6 lists the opcodes in numerical order. Undefined opcodes raise exception -256.

4 External interface

SMite's external interface comes in three parts. The calling interface allows SMite to be controlled by other programs. The `LINK` instruction and extra instructions mechanism allow 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.

Opcode	Instruction	Opcode	Instruction	Opcode	Instruction
$00/FF	`NEXT`	$10	`AND`	$20	`PUSH_S0`
$01	`POP`	$11	`OR`	$21	`PUSH_SSIZE`
$02	`PUSH`	$12	`XOR`	$22	`PUSH_R0`
$03	`SWAP`	$13	`LSHIFT`	$23	`PUSH_RSIZE`
$04	`RPUSH`	$14	`RSHIFT`	$24	`PUSH_HANDLER`
$05	`POP2R`	$15	`LOAD`	$25	`STORE_HANDLER`
$06	`RPOP`	$16	`STORE`	$26	`PUSH_MEMORY`
$07	`LT`	$17	`LOADB`	$27	`PUSH_BADPC`
$08	`EQ`	$18	`STOREB`	$28	`PUSH_INVALID`
$09	`ULT`	$19	`PUSH_SP`	$29	`CALL`
$0A	`ADD`	$1A	`STORE_SP`	$2A	`RET`
$0B	`MUL`	$1B	`PUSH_RP`	$2B	`THROW`
$0C	`UDIVMOD`	$1C	`STORE_RP`	$2C	`HALT`
$0D	`DIVMOD`	$1D	`PUSH_PC`	$2D	`LINK`
$0E	`NEGATE`	$1E	`BRANCH`	$2E	`LITERAL`
$0F	`INVERT`	$1F	`BRANCHZ`	$2F	`PUSH_PSIZE`

Table 6: SMite's opcodes

4.1 Object module format

The first seven bytes of an object module should be the ASCII codes of the letters "smite" padded with ASCII NULs ($00), then the one-byte contents of the `ENDISM` register of the system which saved the module. The next four bytes should contain the number of words the code occupies. The number must have the same endianness as that indicated in the previous byte. Then follows the code, which must fill a whole number of words.

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. A SMite must provide the following calls:

native_address (*integer, boolean*) : pointer

Return a native pointer corresponding to the given SMite address. If the SMite 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` instruction (see section 3.11), the reason code is returned as the result.

single_step () : integer

Execute a single pass of the execution cycle, and return reason code -259, unless a `HALT` instruction was obeyed (see section 3.11), 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 SMite 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 0 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, on which SMite is based.

References

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

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