An implementation of the SMite virtual machine for POSIX version 0.1

Reuben Thomas

23rd December 2018

1 Introduction

The SMite virtual machine [4] provides a portable virtual machine environment for study and experiment. To this end, SMite is itself written in ISO C, since almost all systems have an ISO C compiler available for them.

As well as the virtual machine, C SMite provides a debugger, which is described in [3].

The SMite virtual machine is described in [4]. This paper only describes the features specific to this implementation.

2 Using C SMite

This section describes how to compile C SMite, and the exact manner in which the interface calls and SMite's memory and registers should be accessed.

2.1 Configuration

SMite is written in ISO C99 using POSIX-1.2001 APIs.

The SMite virtual machine is inherently 32-bit, but will run happily on systems with larger (or smaller) addresses.

2.2 Compilation

SMite's build system is written with GNU autotools, and the user needs only standard POSIX utilities to run it. Installation instructions are provided in the top-level file `README.md`.

2.3 Registers and memory

SMite's registers are declared in `smite.h`. Their names correspond to those given in [4, section 2.1], although some have been changed to meet the rules for C identifiers. C SMite does not allocate any memory for SMite, nor does it initialise any of the registers. C SMite provides the interface call **smite_init()** to do this (see section 2.5).

The variables `EP`, `I`, `A`, `MEMORY`, `SP`, `RP`, `HANDLER`, `BADPC` and `INVALID` correspond exactly with the SMite registers they represent, and may be read and assigned to

accordingly, bearing in mind the restrictions on their use given in [4]. `HANDLER`, `BADPC` and `INVALID` are mapped into SMite's memory, so they are automatically updated when the corresponding memory locations are written to, and vice versa.

C SMite provides the ability to map native memory blocks into SMite's address space; see below.

2.4 Extra instructions

C SMite provides some libraries of `EXTRA` instruction functions, which are called in general as:

Stack effect	Description
*i*x u1 u2* -- *j*x*	Call the *u1*th function of library *u2*, passing arguments i*x, with results *j*x*.

Exception -259 is raised if an invalid library is accessed, and -260 for an invalid function in a known library.

2.4.1 SMite

SMite provides access to its own API as library -1. This allows the current state to be controlled, or fresh states to be created and controlled.

Code	Name	Stack effect	Description
0x0	CURRENT_STATE	-- *n-addr*	a pointer to the current `state`
0x1	LOAD_WORD	*n-addr a-addr* -- *x n*	
0x2	STORE_WORD	*n-addr a-addr x* -- *n*	
0x3	LOAD_BYTE	*n-addr addr x n*	
0x4	STORE_BYTE	*n-addr addr x* -- *n*	
0x5	PRE_DMA	*n-addr* $a\text{-}addr_1$ $a\text{-}addr_2$ *f* -- *n*	
0x6	POST_DMA	*n-addr* $a\text{-}addr_1$ $a\text{-}addr_2$ -- *n*	
0x7	MEM_HERE	*n-addr* -- *addr*	
0x8	MEM_ALLOT	$n\text{-}addr_1$ $n\text{-}addr_2$ *u f* -- *addr*	
0x9	MEM_ALIGN	*n-addr* -- *addr*	
0xa	NATIVE_ADDRESS	$n\text{-}addr_1$ *addr f* -- $n\text{-}addr_2$	
0xb	RUN	*n-addr* -- *n*	
0xc	SINGLE_STEP	*n-addr* -- *n*	
0xd	LOAD_OBJECT	*n-addr fid addr* -- *n*	
0xe	INIT	*a-addr* u_1 u_2 u_3 -- *n-addr*	
0xf	DESTROY	*n-addr*	
0x10	REGISTER_ARGS	*n-addr n-addr n* -- *n*	

2.4.2 Standard library

Standard C runtime and library functionality is accessed via library 0.

2.4.2.1 Command-line arguments

Two calls are provided to access command-line arguments passed to C SMite (excluding any that it interprets itself). They are copied from Gforth ([2]).

Code	Name	Stack effect	Description
0x0	ARGC	-- *u*	the number of arguments
0x1	ARG	*u1* -- *c-addr u2*	the *u1*th argument

2.4.2.2 Standard I/O streams

These extra instructions provide access to POSIX standard input, output and error. Each call returns a corresponding file identifier.

Opcode	POSIX file descriptor
0x2	`STDIN_FILENO`
0x3	`STDOUT_FILENO`
0x4	`STDERR_FILENO`

2.4.2.3 File system

The file system extra instructions correspond directly to ANS Forth words, as defined in [1].

Opcode	Forth word
0x5	`OPEN-FILE`
0x6	`CLOSE-FILE`
0x7	`READ-FILE`
0x8	`WRITE-FILE`
0x9	`FILE-POSITION`
0xa	`REPOSITION-FILE`
0xb	`FLUSH-FILE`
0xc	`RENAME-FILE`
0xd	`DELETE-FILE`
0xe	`FILE-SIZE`
0xf	`RESIZE-FILE`
0x10	`FILE-STATUS`

The implementation-dependent word returned by `FILE-STATUS` contains the POSIX protection bits, given by the `st_mode` member of the `struct stat` returned for the given file descriptor.

File access methods are bit-masks, composed as follows:

Bit value	Meaning
1	read
2	write
4	binary mode

To create a file, set both read and write bits to zero when calling `OPEN-FILE`.

2.5 Using the interface calls

The operation of the specified interface calls is given in [4]. Here, the C prototypes corresponding to the idealised prototypes used in [4] are given. The names are prefixed with **smite_**. The first argument to most routines is a `@PACKAGE_state *`, as returned by `smite_init`.

Files to be loaded and saved are passed as C file descriptors. Thus, the calling program must itself open and close the files.

```
uint8_t *native_address(smite_state *state, smite_UWORD address,
bool writable)
```

Returns `NULL` when the address is invalid, or the writable flag is true and the address is read-only.

`smite_state *state, smite_WORD run(smite_state *state)`

The reason code returned by **smite_run()** is a SMite word.

`smite_state *state, smite_WORD smite_single_step(smite_state *state)`

The reason code returned by **smite_single_step()** is a SMite word.

`int smite_load_object(smite_state *state, FILE *file, smite_UWORD address)`

If a file system error occurs, the return code is -3. If the endism of the object file does not match `ENDISM`, the return code is -4. If the word size of the object file is not `WORD_SIZE`, the return code is -5. As an extension to the specification, if an object file starts with the bytes 35, 33 (`#!`), then it is assumed to be the start of a UNIX-style "hash bang" line, and the file contents up to and including the first newline character (10) is ignored.

In addition to the required interface calls C SMite provides an initialisation routine **smite_init()** which, given a word array and its size, initialises SMite:

`smite_state smite_init(size_t memory_size, size_t data_stack_size, size_t return_stack_size)`

`memory_size` is the size of `b_array` in *words* (not bytes); similarly, `data_stack_size` and `return_stack_size` give the size of the stacks in words; these are allocated by `smite_init`. The return value is `NULL` if memory cannot be allocated, and a pointer to a new state otherwise. All the registers are initialised as per [4].

The following routines give easier access to SMite's address space at the byte and word level. On success, they return 0, and on failure, the relevant exception code.

`int smite_load_word(smite_state *state, smite_UWORD address, smite_WORD *value)`

Load the word at the given address into the given `smite_WORD *`.

`int smite_store_word(smite_state *state, smite_UWORD address, smite_WORD value)`

Store the given `WORD` value at the given address.

`int smite_load_byte(smite_state *state, smite_UWORD address, smite_BYTE *value)`

Load the byte at the given address into the given `smite_BYTE *`.

`int smite_store_byte(smite_state *state, smite_UWORD address, smite_BYTE value)`

Store the given `smite_BYTE` value at the given address.

The following routines give easier access to contiguous areas of SMite's address space. On success, 0 is returned; if `to` if not less than `from`, or the addresses are not contained in the same area, or not writable if desired, -1 is returned; if either address is unaligned, or some other error occurs, a memory exception code is returned.

`int smite_pre_dma(smite_state *state, smite_UWORD from, smite_UWORD to, bool write)`

Convert the given range to native byte order, so that it can be read (or written) directly.

`int smite_post_dma(smite_state *state, smite_UWORD from, smite_UWORD to)`
: Convert the given range to SMite byte order, so that it can be used by SMite after a direct access.

The following routines are provided to map system memory to SMite's address space and vice versa:

`smite_UWORD smite_mem_here(smite_state *state)`
: Returns the SMite address at which the next mapping will be made.

`smite_UWORD smite_mem_allot(smite_state *state, void *p, size_t n)`
: Map `n` bytes pointed to by `p` into SMite's address space, and return the address mapped. Addresses are mapped sequentially.

`smite_UWORD smite_mem_align(smite_state *state)`
: Rounds up the address at which the next memory allocation will occur to the next word.

The following routine allows the calling program to register command-line arguments that can be retrieved by the `ARG` and `ARGC` extra instructions.

`int smite_register_args(smite_state *state, int argc, char *argv[])`
: Maps the given arguments register, which has the same format as that supplied to **main()**, into SMite's memory. Returns 0 on success and -1 if memory could not be allocated, or -2 if an argument could not be mapped to SMite's address space.

Programs which use C SMite's interface must `#include` the header file `smite.h` and be linked with the SMite library. `smite_opcodes.h`, which contains an enumeration type of SMite's instruction set, and `smite_debug.h`, which contains useful debugging functions such as disassembly, may also be useful; they are not documented here.

2.6 Other extras provided by C SMite

C SMite provides the following extra type in `smite.h` which is useful for programming with SMite:

`smite_WORD_pointer`**:** a union with members `smite_WORD words[smite_NATIVE_POINTER_SIZE]` and `void (*pointer)(void)`, which allow a function pointer suitable for the `CALL_NATIVE` instruction to be easily stored and retrieved. It is assumed that the pointer is pushed on to the stack starting with `words[0]` and ending with `words[smite_NATIVE_POINTER_SIZE - 1]`.

References

[1] American National Standards Institute. *ANS X3.215-1994: Programming Languages—Forth*, 1994.

[2] M. Anton Ertl, Bernd Paysan, Jens Wilke, Neal Crook, David Kühling, et al. Gforth. `https://www.complang.tuwien.ac.at/forth/gforth/`, 1993.

[3] Reuben Thomas. A simple shell for the SMite Forth virtual machine, 2018. `https://rrt.sc3d.org/`.

[4] Reuben Thomas. The SMite Forth virtual machine, 2018. `https://rrt.sc3d.org/`.