An implementation of the SMite virtual machine
for POSIX
version 0.1

Reuben Thomas

1st March 2019

1   Introduction

The SMite virtual machine [1] provides a portable virtual machine environment for study and experiment.

The SMite virtual machine is described in [1]. 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

C SMite is written in ISO C99 using POSIX-1.2001 APIs, and can be configured with a `WORD_SIZE` of 4 or 8. It defaults to the word size of the host system.

2.2   Installation

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 [1, section 2.1], although some have been changed to meet the rules for C identifiers. C SMite provides the interface call **smite_init()** to initialise a VM instance (see section 2.5).

2.4 Extra instructions

C SMite provides some extra action instructions, which give access to various library functions. They take the following form:

Stack effect	Description
i*x u - j*x	Call the uth function of the library, passing arguments i*x, with results j*x.

Error -15 is raised for an invalid function.

2.4.1 SMite

SMite provides access to its own API via the `LIB_SMITE` instruction, opcode 0x3f. This allows the current state to be controlled, or fresh states to be created and controlled. The API is mapped to the following functions:

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	REALLOC_MEMORY	n-addr$_1$ n-addr$_2$ u f - addr	
0x6	REALLOC_STACK	n-addr$_1$ n-addr$_2$ u f - addr	
0x7	NATIVE_ADDRESS_OF_RANGE	n-addr$_1$ addr f - n-addr$_2$	
0x8	RUN	n-addr - n	
0x9	SINGLE_STEP	n-addr - n	
0xa	LOAD_OBJECT	n-addr fid addr - n	
0xb	INIT	a-addr u$_1$ u$_2$ u$_3$ - n-addr	
0xc	DESTROY	n-addr	
0xd	REGISTER_ARGS	n-addr n-addr n - n	

2.4.2 Standard library

Standard C runtime and library functionality is accessed via the `LIB_C` instruction, opcode 0x3e.

2.4.2.1 Command-line arguments

Some functions are provided to access command-line arguments passed to C SMite (excluding any that it interprets itself).

Code	Name	Stack effect	Description
0x0	ARGC	- u	the number of arguments
0x1	ARG_LEN	u$_1$ - u$_2$	the length of the u1th argument
0x2	ARG_COPY	u$_1$ c-addr u$_2$ - u$_3$	copy argument u$_1$ to the buffer of length u$_2$ at c-addr, leaving t

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
0x3	STDIN_FILENO
0x4	STDOUT_FILENO
0x5	STDERR_FILENO

2.4.2.3 File system

Opcode	Forth word
0x6	OPEN-FILE
0x7	CLOSE-FILE
0x8	READ-FILE
0x9	WRITE-FILE
0xa	FILE-POSITION
0xb	REPOSITION-FILE
0xc	FLUSH-FILE
0xd	RENAME-FILE
0xe	DELETE-FILE
0xf	FILE-SIZE
0x10	RESIZE-FILE
0x11	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 [1]. Here, the C prototypes corresponding to the idealised prototypes used in [1] are given. The names are prefixed with **smite_**. The first argument to most routines is a @PACKAGE_state *, as returned by smite_init.

```
uint8_t *native_address_of_range(smite_state *state,
smite_UWORD address, smite_UWORD length)
```
   Returns NULL when the address range is not entirely valid.

```
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.

```
ptrdiff_t smite_load_object(smite_state *state, smite_UWORD
address, int fd)
```
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 stack_size)
```
memory_size is the size of b_array in *words* (not bytes); similarly,
stack_size gives the size of the stack in words; these are allocated by
smite_init. The return value is NULL if memory cannot be allocated,
or if any requested size is too large, and a pointer to a new state
otherwise. All the registers are initialised as per [1].

```
int smite_realloc_memory(smite_state *state, smite_UWORD size)
```
Resize the memory to the given size. Any new memory is zeroed.
Returns 0 on success or -1 if the requested size of memory cannot be
allocated.

```
int smite_realloc_stack(smite_state *state, smite_UWORD size)
```
Resize the stack to the given size. Any new memory is zeroed.
Returns 0 on success or -1 if the requested size of memory cannot be
allocated.

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
error 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 access to SMite's stacks. On success, they return 0, and on failure, the relevant error code.

```
int smite_load_stack(smite_state *state, smite_UWORD pos,
smite_WORD *value)
```
      Load the word at the given position of the data stack into the given `smite_WORD *`.

```
int smite_store_stack(smite_state *state, smite_UWORD pos,
smite_WORD value)
```
      Store the given `WORD` value at the given position in the data stack.

```
int smite_pop_stack(smite_state *state, smite_WORD *value)
```
      Pop the top word off the data stack, decrementing its depth, into the given `smite_WORD *`.

```
int smite_push_stack(smite_state *state, smite_WORD value)
```
      Push the given `smite_WORD` value on to the data stack of the given size, incrementing its depth.

The following routine allows the calling program to register command-line arguments that can be retrieved by the `ARGC` and `ARG_COPY` 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/smite.h` and be linked with the SMite library. `smite/opcodes.h`, which contains an enumeration type of SMite's instruction set, and `smite/smite_aux.h`, which contains other useful declarations, may also be useful; they are not documented here.

2.6   Other extras provided by C SMite

C SMite provides various useful extras in `smite/aux.h`. These are used internally, and are thought to be useful, but may change at any time, so their stability should not be relied on.

References

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