

# The Mit virtual machine

Reuben Thomas

9th June 2019

## Typographical conventions

Instructions 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

Mit is a simple virtual machine for study and experiment. It is a stack machine, based on the more complex register machine Mite [3]. This paper gives a full description of Mit.

Mit is conceptually (and usually in fact) a library, embedded in other programs; it supports a simple object file format.

## 2 Architecture

Mit’s address unit is the byte, which is eight bits. Most of the quantities on which Mit operates are fixed-size words, which are stored in memory in either big- or little-endian order. The choice of byte and word size enable efficient implementation on the vast majority of machine architectures.

### 2.1 Registers

The registers are word quantities; they are listed, with their functions, in table 1. Registers without fixed values are initialised to 0.

### 2.2 Memory

Mit’s memory is a contiguous sequence of bytes with addresses starting at 0. The address of a word is that of the byte in it with the lowest address.

### 2.3 Stack

The stack is a LIFO stack of words used for passing values to instructions and routines and for holding subroutine return addresses. To **push** a word on to the stack means to add a new word to the top of the stack, increasing the stack

Register	Function
pc	The program counter. Points to the next word from which i may be loaded.
instruction	Contains instructions to be executed.
bad	The invalid stack position, or invalid or unaligned memory address, that last caused an error.
stack_depth	The number of words on the stack.
endism	The endianness of Mit: 0 = Little-endian, 1 = Big-endian. Fixed for a particular instance of Mit.
word_bytes	The number of bytes in a word. Must be in the range 2 to 32 inclusive, and a power of 2. Fixed for a particular instance of Mit.

Table 1: Registers

depth by 1; to **pop** a word means to reduce the stack depth by 1. Instructions that change the number of words on the stack implicitly pop their arguments and push their results.

## 2.4 Execution

Execution proceeds as follows:

```

begin
    let opcode be the least significant 8 bits of instruction
    shift instruction logically 8 bits to the right
    execute the instruction given by opcode
repeat

```

If an error occurs during execution (see section 2.5), the state of the virtual machine is reset to its state at the start of the loop before the error is raised. This allows instructions to be restarted after handling the error, where desired.

## 2.5 Errors and termination

When Mit encounters certain abnormal situations, such as an attempt to access an invalid address, or divide by zero, an **error** is **raised**, and execution terminates. The effect of the current instruction is undone (see section 2.4). If the error is a stack or memory access error, **bad** is set to the stack position or address that caused the error. An **error code** is returned to the caller.

Execution can be terminated explicitly by performing a **halt** instruction (see section 3.9.2).

Error codes are unsigned numbers. 0 to 127 are reserved for the specification; other error codes may be used by implementations. The meanings of those that may be raised by Mit are shown in table 2.

Code	Meaning
0	<b>single_step()</b> has terminated without error.
1	Invalid opcode (see section 3.11).
2	Stack overflow. <b>bad</b> is set to the extra number of words of stack space required.
3	Invalid stack read. <b>bad</b> is set to the invalid stack position.
4	Invalid stack write. <b>bad</b> is set to the invalid stack position.
5	Invalid memory read. <b>bad</b> is set to the invalid address.
6	Invalid memory write. <b>bad</b> is set to the invalid address.
7	Address alignment error: raised when an instruction is given a valid address, but insufficiently aligned. <b>bad</b> is set to the invalid address.
8	Invalid size (greater than $\log_2 \text{word\_bytes}$ ).
9	Division by zero attempted (see section 3.7).
127	A <b>halt</b> instruction was executed.

Table 2: Errors raised by Mit

### 3 Instruction set

The instruction set is listed below, 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

$$i_1 \ i_2 \dots i_{n-1} \ i_n$$

where the  $i_k$  are stack items, each of which occupies a whole number of words,<sup>1</sup> with  $i_n$  being on top of the stack. The symbols denoting different types of stack item are shown in table 3.

Types are only used to indicate how instructions treat their arguments and results; Mit 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.

<sup>1</sup>In this specification, each stack item occupies *precisely* one word.

Symbol	Data type
<i>flag</i>	a Boolean flag, 0 for false or non-zero for true
<i>size</i>	an integer in the range 0 to $\log_2 \text{word\_bytes}$ inclusive
<i>n</i>	signed number
<i>u</i>	unsigned number
<i>n u</i>	number (signed or unsigned)
<i>x</i>	unspecified word
<i>addr</i>	address
<i>a-addr</i>	word-aligned address

Table 3: Types used in stack effects

$u \Rightarrow x$
$n \Rightarrow x$
$flag \Rightarrow u$
$size \Rightarrow u$
$a-addr \Rightarrow addr \Rightarrow u$

Table 4: The subtype relation

Numbers are represented in twos complement form. *addr* consists of all valid virtual machine addresses.

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 each time to the identical stack item.

Ellipsis is used for indeterminate numbers of specified types of item.

### 3.1 Instruction fetch

If an invalid or unaligned address is accessed when loading *instruction*, the appropriate error is raised (see section 2.5).

*next* ( - )

Load the word pointed to by *pc* into *instruction* then add *word\_bytes* to *pc*.

### 3.2 Control

These instructions implement unconditional and conditional branches, and subroutine call and return (subroutine return is *jump*):

*jump* ( *a-addr* - )

If *instruction* is not 0, raise error 1. Set *pc* to *a-addr*. Perform the action of *next*.

*jumpz* ( *flag* *a-addr* - )

If *flag* is false then set *pc* to *a-addr* and perform the action of *next*.

call (  $a\text{-}addr_1$  -  $a\text{-}addr_2$  )  
 If instruction is not 0, raise error 1. Exchange pc with the top stack value.  
 Perform the action of next.

### 3.3 Stack manipulation

These instructions manage the stack (see also section 3.9.1):

pop (  $x$  - )  
 Remove  $x$  from the stack.  
 dup (  $x_u \dots x_0$   $u$  -  $x_u \dots x_0$   $x_u$  )  
 Remove  $u$ . Copy  $x_u$  to the top of the stack.  
 swap (  $x_{u+1} \dots x_0$   $u$  -  $x_0$   $x_u \dots x_1$   $x_{u+1}$  )  
 Exchange the top stack word with the  $u+1$ th.

### 3.4 Literals

lit ( -  $n$  )  
 The word pointed to by pc is pushed on to the stack, and pc is incremented to point to the following word.  
 lit\_pc\_rel ( -  $n$  )  
 Like lit, except that the initial value of pc is added to the value pushed on to the stack.  
 lit\_0 ( - 0 )  
 Push 0 on to the stack.  
 lit\_1 ( - 1 )  
 Push 1 on to the stack.  
 lit\_2 ( - 2 )  
 Push 2 on to the stack.  
 lit\_3 ( - 3 )  
 Push 3 on to the stack.

### 3.5 Logic and shifts

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  $x_2$ .

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.

**sign\_extend** (  $u$  *size* -  $n$  )  
 Sign extend the  $2^{size}$ -byte quantity  $u$  to  $n$ .

### 3.6 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 1 if and only if  $x_1$  is bit-for-bit the same as  $x_2$ .

**lt** (  $n_1$   $n_2$  - *flag* )  
*flag* is 1 if and only if  $n_1$  is less than  $n_2$ .

**ult** (  $u_1$   $u_2$  - *flag* )  
*flag* is 1 if and only if  $u_1$  is less than  $u_2$ .

### 3.7 Arithmetic

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

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 instructions 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  $u_4$ .

### 3.8 Memory

These instructions fetch and store quantities to and from memory. If an invalid or unaligned address is accessed, the appropriate error is raised (see section 2.5).

**load** (  $addr$   $size$  -  $x$  )  
 Load the  $2^{size}$ -byte quantity  $x$  stored at  $addr$ , which must be a multiple of  $2^{size}$ . Any unused high-order bits are set to zero.

**store** (  $x$   $addr$   $size$  - )  
 Store the  $2^{size}$  least-significant bytes of  $x$  at  $addr$ , which must be a multiple of  $2^{size}$ .

### 3.9 Extra instructions

Since `instruction` must be 0 when `next` is performed, the rest of an instruction word following `next`, `jump` and `call` must normally be all zero bits.

Non-zero values following `next` and `call` are reserved for the Mit specification; non-zero values following `jump` may be used by implementations to implement extra functionality.

When an extra instruction is performed, the original instruction is considered to have completed executing.

#### 3.9.1 Stack depth manipulation

These instructions read and write the `stack_depth` register:

**get\_stack\_depth** (  $u$  - )  
 Set `stack_depth`, the number of words on the stack, to  $u$ .

**push\_stack\_depth** ( -  $u$  )  
 $u$  is the value of `stack_depth`, the number of words on the stack. If there is not enough space for  $u$  stack items, raise error 2.

#### 3.9.2 Termination

This instruction terminates execution (see section 2.5):

**halt** ( - )  
 Raise error 127.

### 3.10 Instruction encoding

Instructions are encoded as 8-bit opcodes; opcodes are packed into words, which are executed starting at the least-significant bits.

### 3.11 Instruction opcodes

Table 5 lists the instruction opcodes in numerical order. Table 6 lists the extra instruction opcodes (following `call`; see section 3.9). Other instruction opcodes are undefined.

Opcode	Instruction	Opcode	Instruction
0x0	<code>next</code>	0x10	<code>eq</code>
0x1	<code>jump</code>	0x11	<code>lt</code>
0x2	<code>jumpz</code>	0x12	<code>ult</code>
0x3	<code>call</code>	0x13	<code>negate</code>
0x4	<code>pop</code>	0x14	<code>add</code>
0x5	<code>dup</code>	0x15	<code>mul</code>
0x6	<code>swap</code>	0x16	<code>divmod</code>
0x7	(undefined)	0x17	<code>udivmod</code>
0x8	<code>load</code>	0x18	<code>not</code>
0x9	<code>store</code>	0x19	<code>and</code>
0xa	<code>lit</code>	0x1a	<code>or</code>
0xb	<code>lit_pc_rel</code>	0x1b	<code>xor</code>
0xc	<code>lit_0</code>	0x1c	<code>lshift</code>
0xd	<code>lit_1</code>	0x1d	<code>rshift</code>
0xe	<code>lit_2</code>	0x1e	<code>arshift</code>
0xf	<code>lit_3</code>	0x1f	<code>sign_extend</code>

Table 5: Instruction opcodes

Opcode	Instruction
0x1	<code>halt</code>
0x2	<code>get_stack_depth</code>
0x3	<code>set_stack_depth</code>

Table 6: Extra instruction opcodes

## 4 External interface

- Implementations should provide an **API** to create and run virtual machine instances, and provide access to its registers, stack and memory.
- Implementations can add **extra instructions** to provide extra computational primitives, and to offer access to system facilities, previously written code, native libraries and so on.
- The **object file** format allows compiled code to be saved, reloaded and shared between systems.



## 4.1 Object file format

The object file starts with the ASCII codes of the letters “mit”, followed by three zero bytes, then the one-byte values of the `endism` and `word_bytes` registers of the system which saved the file, then a word (of the given endianness and size) containing the number of bytes the code occupies. Then follows the code.

## 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], and my PhD project, Mite [3], on which Mit 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/>.