The Mit virtual machine

Reuben Thomas

10th May 2020

1 Introduction

Mit is a simple virtual machine. It is a stack machine, based on the more complex register machine Mite [4]. This paper gives a full description of Mit.

Mit is conceptually (and usually in fact) a library, embedded in other programs.

2 Architecture

The 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 number of bytes in a word is called `word_bytes`, and must be 4 or 8.

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. The registers are initialised to 0.

Register	Function
`pc`	The program counter. Points to the next word from which `i` may be loaded.
`ir`	The instruction register. Contains instructions to be executed.
`stack_depth`	The number of words in the current stack frame.

Table 1: Registers

2.2 Memory

Mit's memory consists of discontiguous words in a flat address space. The address of a word is that of the byte in it with the lowest address.

2.3 Stack

The stack is a last-in-first-out stack of call frames. Each frame is a stack of words. The phrase "the stack" usually refers to the current frame, and all references to stack items refer to the current frame. To **push** a word on to the stack means to add a new word to the top of the stack, increasing the stack 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:

```
repeat
    let opcode be the least significant byte of ir
    shift ir arithmetically one byte to the right
    execute the instruction given by opcode,
      or throw error −1 if the opcode is invalid
```

If an error occurs during execution (see section 2.5), stack frames are popped until reaching one which contains an error handler. The error handler, return address and number of return items are discarded, and the error code is pushed on to the stack.

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** may be **thrown**. If the error is non-zero, the effect of the current instruction is undone (see section 2.4). An **error code** is returned to the caller.

Execution can be terminated explicitly by a `throw` instruction (see section 3.1), which throws an error.

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

Errors −2 to −8 inclusive are optional: an implementation may choose to raise them, or not.

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.

Code	Meaning
0	Execution has terminated without error.
-1	Invalid opcode (see section 3.11).
-2	Stack overflow.
-3	Invalid stack read.
-4	Invalid stack write.
-5	Invalid memory read.
-6	Invalid memory write.
-7	Address alignment error: thrown when an instruction is given a valid address, but insufficiently aligned.
-8	Division by zero attempted (see section 3.9).

Table 2: Errors thrown by Mit

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*. An stack item *[x]* in square brackets is optional.

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 word, 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, 0 for false or non-zero for true
s	signed number
u	unsigned number
n	number (signed or unsigned)
x	unspecified word
addr	address
a-addr	word-aligned address

Table 3: Types used in stack effects

Numbers are represented in two's 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 Extra instructions and traps

Extra instructions offer functionality not available in the core instruction set, via the `extra` instruction. Traps, using the `trap` instruction, are similar, but intended to be implementable as add-ons to an implementation, rather than as an integrated part of it. Extra instructions and traps may modify the memory and stack, but may not directly change the values of registers.

`extra` see below

Perform extra instruction `ir`; if `ir` is not the code of a valid extra instruction, throw error -1.

`trap`

Perform trap `ir`; if `ir` is not the code of a valid trap, throw error -1.

The following extra instructions are defined:

`next` 0 -

Load the word pointed to by `pc` into `ir`, then add `word_bytes` to `pc`.

`stack_depth` 1 - *u*

Push the value of `stack_depth` on to the stack.

`throw` 2 *n* -

Throw error n.

`catch` 3 $x_u \ldots x_0$ u_1 u_2 *[a-addr*$_1$*]* - *h* u_2 *a-addr*$_2$ || $x_u \ldots x_0$

Add an error handler *h* to the current stack frame, then perform the action of `call` (see section 3.3).

The following trap is defined:

`nextff` -1 -

Perform the action of `next`.

3.2 Stack manipulation

These instructions manage the stack:

`pop` *x* -

Remove *x* 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` $x_{u+1} \ldots x_0$ *u* - x_0 $x_u \ldots x_1$ x_{u+1}

Exchange the top stack word with the *u+1*th.

3.3 Control

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

`jump` *[a-addr]* -

If `ir` is 0, set `pc` to *a-addr*; otherwise, add `ir` $\times$ `word_bytes` to `pc`, and set `ir` to 0.

`jumpz` *flag [a-addr]* -

If *flag* is false then perform the action of `jump`; otherwise, set `ir` to 0.

`call` $x_u \ldots x_0$ u_1 u_2 *[a-addr*$_1$*]* - u_2 *a-addr*$_2$ || $x_u \ldots x_0$

Perform the action of `jump`. Move $x_{u_1} \ldots x_0$ to a new call frame. Push the initial value of `pc` to the old frame.

`ret` *[h]* u_2 *a-addr* || $x_{u_2} \ldots x_0$ - $x_u \ldots x_0$ *[0]*

Pop the current stack frame, set `pc` to *a-addr*, and move $x_{u_2} \ldots x_0$ to it. Set `ir` to 0. If there is an error handler *h* in the inner stack frame, discard it and push 0 on top of the stack.

3.4 Memory

These instructions fetch and store quantities to and from memory.

`load` *a-addr* - *x*

Load the word *x* stored at *a-addr*.

`store` *x a-addr* -

Store *x* at *a-addr*.

`load1` *addr* - *x*

Load the byte *x* stored at *addr*. Unused high-order bits are set to zero.

`store1` *x addr* -

Store the least-significant byte of *x* at *addr*.

`load2` *addr* - *x*

Load the 2-byte quantity *x* stored at *addr*, which must be a multiple of 2. Unused high-order bits are set to zero.

`store2` *x addr* -

Store the 2 least-significant bytes of *x* at *addr*, which must be a multiple of 2.

`load4` *addr* - *x*

Load the 4-byte quantity *x* stored at *addr*, which must be a multiple of 4. Any unused high-order bits are set to zero.

`store4` *x addr* -

Store the 4 least-significant bytes of *x* at *addr*, which must be a multiple of 4.

3.5 Constants

`push` - *n*

The word pointed to by `pc` is pushed on to the stack, and `pc` is incremented to point to the following word.

`pushrel` - *n*

Like `push`, except that the initial value of `pc` is added to the value pushed on to the stack.

There are also variants `pushi` and `pushreli`, which encode a limited range of constants in the instruction opcode; see section 3.11.

3.6 Logic

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.

3.7 Comparison

These words compare two numbers on the stack, returning a flag (for equality, use `xor`; see section 3.6):

`lt` s_1 s_2 – *flag*

flag is 1 if and only if s_1 is less than s_2.

`ult` u_1 u_2 – *flag*

flag is 1 if and only if u_1 is less than u_2.

3.8 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.

3.9 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` s_1 – s_2

Negate s_1, giving s_2.

`add` $n_1\ n_2$ - n_3

Add n_2 to n_1, giving the sum n_3.

Multiplication and division:

`mul` $n_1\ n_2$ - n_3

Multiply n_1 by n_2 giving the product n_3.

`divmod` $s_1\ s_2$ - $s_3\ s_4$

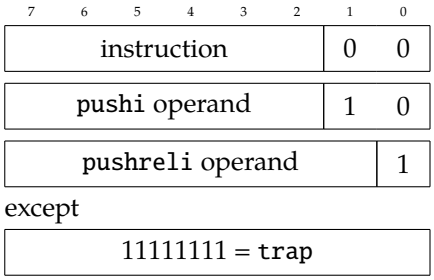
Divide s_1 by s_2 using symmetric division, giving the single-word quotient s_3 and the single-word remainder s_4. The quotient is rounded towards zero. If s_2 is zero, throw error -8.

`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. If u_2 is zero, throw error -8.

3.10 Instruction encoding

Instructions are encoded as byte opcodes; opcodes are packed into words, which are executed starting at the least-significant bits. The opcodes have the following structure:



The operands for `pushi` and `pushreli` are interpreted as signed numbers; the operand of `pushreli` is multiplied by `word_bytes`. Note that `pushreli`'s immediate constant cannot be -1, as the corresponding opcode is used for `trap`.

3.11 Instruction opcodes

Table 4 lists the instruction opcodes. Other instruction opcodes are invalid.

4 External interface

- Implementations should provide an API to create and run virtual machine code, and to add traps.
- Implementations can add extra instructions to provide extra computational primitives and other deeply-integrated facilities, and traps to offer access to system facilities, native libraries and so on; see section 3.1. Implementations may allow users to add their own traps.

Opcode	Instruction	Opcode	Instruction
0x0	`extra`	0x10	`push`
0x1	`pop`	0x11	`pushrel`
0x2	`dup`	0x12	`not`
0x3	`swap`	0x13	`and`
0x4	`jump`	0x14	`or`
0x5	`jumpz`	0x15	`xor`
0x6	`call`	0x16	`lt`
0x7	`ret`	0x17	`ult`
0x8	`load`	0x18	`lshift`
0x9	`store`	0x19	`rshift`
0xa	`load1`	0x1a	`arshift`
0xb	`store1`	0x1b	`negate`
0xc	`load2`	0x1c	`add`
0xd	`store2`	0x1d	`mul`
0xe	`load4`	0x1e	`divmod`
0xf	`store4`	0x1f	`udivmod`

Table 4: Instruction opcodes

Acknowledgements

Martin Richards introduced me to Cintcode [2], which kindled my interest in virtual machines, and led to Beetle [3] and Mite [4], of which Mit is a sort of synthesis. GNU *lightning* [1] helped inspire me to greater simplicity, while still aiming for speed. Alistair Turnbull has been a great collaborator for all my work on virtual machines.

References

[1] Paulo Bonzini. Using and porting GNU *lightning*, 2000. `ftp://alpha.gnu.org/gnu/`.

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

[3] Reuben Thomas. Beetle and pForth: a Forth virtual machine and compiler. BA dissertation, University of Cambridge, 1995. `https://rrt.sc3d.org/`.

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