The Mit virtual machine

Reuben Thomas

26th June 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

2.1 Memory model

The **memory** consists of a possibly discontiguous set of **words**, which are composed of 8-bit **bytes**. The number of bytes in a word is called `word_bytes`, and must be 4 or 8. An **address** identifies a byte. The address space is flat and linear. Addresses are words. The bytes in a word have contiguous addresses, in big- or little-endian order. The address of a word is that of the byte in it with the lowest address, and is a multiple of `word_bytes`. An address is **valid** if it identifies a byte of a word in memory.

The choice of byte and word size enable efficient implementation on the vast majority of machine architectures.

2.2 Registers

The registers are word quantities; they are listed, with their functions, in table 1.

Register	Function
`pc`	The program counter. Points to the next word from which `ir` may be loaded.
`ir`	The instruction register. Contains instructions to be executed.

Table 1: Registers

2.3 Stack

The stack is a last-in-first-out stack of call frames. The **current frame** is the topmost. Each frame is a stack of words. To **push** a word on to the stack means

to add a new word to the top of the current frame, 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

The registers are initialised to appropriate values. 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
```

Instruction fetch means setting `ir` to the word pointed to by `pc` and making `pc` point to the next word. This is done whenever `ir` is 0 or −1. These are encoded respectively as extra instruction 0 (see section 3.9) and trap −1 (see section 3.9).

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 popped, and the error code is pushed.

2.5 Errors and termination

When Mit encounters certain exceptional situations, such as an attempt to access an invalid address, or divide by zero, an **error** may be **thrown**. An **error code** is returned to the caller.

Execution can be terminated explicitly by a `throw` instruction (see section 3.9), 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.

Code	Meaning
0	Execution has terminated without error.
−1	Invalid opcode (see section **??**).
−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 3.4).
−9	Division overflow (see section 3.4).

Table 2: Errors thrown by Mit

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 left, and the stack effect on the right. 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 effect. A stack item *[x]* in square brackets is optional.

Stack pictures represent the topmost stack items, and are written

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

where the i_k are stack items, each of which occupies a word, with i_1 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, zero 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 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.

`set` $x_{u+1} \ldots x_0$ *u* – x_0 $x_u \ldots x_1$

Set the *u+1*th stack word to x_0, then discard x_0.

`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.

3.2 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.

3.3 Constants

`push` – *n*

Push the word pointed to by `pc` on to the stack, and increment `pc` to point to the following word.

`pushrel` – *n*

Like `push` but add `pc` to the value pushed on to the stack.

`pushi_`*n* – *n*

Push *n* on to the stack. *n* ranges from -32 to 31 inclusive.

`pushreli_`*n* – *n*

Push `pc` + `word_bytes` × *n* on to the stack. *n* ranges from -64 to 63 inclusive.

The operand of `pushi` and `pushreli` is encoded in the instruction opcode; see section 3.10.

3.4 Arithmetic

All calculations are made modulo $2^{(8\times\texttt{word\_bytes})}$, except as detailed for certain instructions.

`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.

`mul` n_1 n_2 – n_3

Multiply n_1 by n_2 giving the product n_3.

3.5 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.

3.6 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 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 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 bit into the 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.

3.7 Comparison

These instructions compare two numbers on the stack, returning a flag; see section 3.5):

`eq` s_1 s_2 – *flag*

flag is 1 if and only if s_1 is equal to s_2.

`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 Control

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

`jump` *[a-addr]* -

If `ir` is 0, the target address is *a-addr*, otherwise it is `ir` × `word_bytes` + `pc`. Set *pc* to the target address, and `ir` to 0.

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

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

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

Call the initial value of `pc` *a-addr*$_2$. Perform the action of `jump`. Move $x_{u_1} \ldots x_0$ to a new call 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 the next stack frame. Set `ir` to 0. If there is an error handler *h* in the next stack frame, discard it and push 0 on top of the stack.

3.9 Extra instructions and traps

Extra instructions, using the `extra` instruction, offer necessary functionality too rare or slow to deserve a core 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. 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. Extra instruction code 0 is used for instruction fetch (see section 2.4).

`trap`

Perform trap `ir`; if `ir` is not the code of a valid trap, throw error -1. Trap code -1 is used for instruction fetch (see section 2.4).

The following extra instructions are defined:

`divmod` 1 s_1 s_2 - s_3 s_4

Divide s_1 by s_2, 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. If s_1 is $-2^{(8 \times \texttt{word\_bytes} - 1)}$ and s_2 is -1, throw error -9.

`udivmod` 2 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.

`throw` 3 *n* -
Throw error *n*.

`catch` 4 $x_u \ldots x_0$ 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 3.8).

3.10 Instruction encoding

Instructions are encoded as bytes, packed into words, which are executed as described in section 2.4. The bytes have the following internal structure:

7 6 5 4 3	2	1	0
instruction	0	0	0
`pushi` $n < 0$	1	0	0
`pushreli` $n < 0$		1	0
`pushreli` $n \geq 0$		0	1
`pushi` $n \geq 0$	0	1	1
instruction	1	1	1

Table 4 lists the instructions whose least-significant 3 bits are 000 or 111. Other instruction opcodes with those endings are invalid.

Opcode	Instruction	Opcode	Instruction
0x00	`extra`	0x80	`load`
0x08	`not`	0x88	`store`
0x10	`and`	0x90	`load1`
0x18	`or`	0x98	`store1`
0x20	`xor`	0xa0	`load2`
0x28	`lshift`	0xa8	`store2`
0x30	`rshift`	0xb0	`load4`
0x38	`arshift`	0xb8	`store4`
0x40	`pop`	0xc0	`push`
0x48	`dup`	0xc8	`pushrel`
0x50	`set`	0xd0	`negate`
0x58	`swap`	0xd8	`add`
0x60	`jump`	0xe0	`mul`
0x68	`jumpz`	0xe8	`eq`
0x70	`call`	0xf0	`lt`
0x78	`ret`	0xf8	`ult`

Table 4: Instruction opcodes

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 3.9.

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 for many years been a fount of ideas and criticism for all my work in computation, and lately a staunch collaborator on Mit.

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/`.