The Mit virtual machine

Reuben Thomas

2nd May 2020

Typographical conventions

Instructions and registers are shown in `typewriter` font.

Addresses are given in bytes. Addresses are written in hexadecimal; hex numbers are prefixed with "0x".

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 on the stack.

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 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 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 ir
    shift ir arithmetically 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. 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 performing a `halt` instruction (see section 3.1), which raises 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 raised by Mit are shown in table 2.

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 3.12).
-2	Stack overflow.
-3	Invalid stack read.
-4	Invalid stack write.
-5	Invalid memory read.
-6	Invalid memory write.
-7	Address alignment error: raised when an instruction is given a valid address, but insufficiently aligned.
-8	Division by zero attempted (see section 3.10).
-127	An instruction has completed without error.

Table 2: Errors raised 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*.

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
n	signed number
u	unsigned number
n|*u*	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

Extra instructions are introduced by the instruction `extra`:

`extra` (see below)

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

The following extra instructions are defined:

`next` (-)

Load the word pointed to by `pc` into `ir`, then add `word_bytes` to `pc`. If an invalid or unaligned address is accessed when loading `ir`, the appropriate error is raised (see section 2.5).

`halt` (*n* -)

Raise error n.

3.2 Control

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

`jump` (*a-addr* -)

If `ir` is not 0, raise error -1. Set `pc` to *a-addr*.

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

If *flag* is false then set `pc` to *a-addr* and set `ir` to 0.

`call` ($a\text{-}addr_1$ - $a\text{-}addr_2$)

If `ir` is not 0, raise error -1. Exchange `pc` with the top stack value.

There are also variants `jumpi`, `jumpzi` and `calli`, which take the address encoded as a `pc`-relative number of words in the remainder of `ir`.

3.3 Trap

An implementation may implement extra functionality using the `trap` instruction:

`trap` (undefined)

Perform arbitrary actions. Note in particular that `trap` may change the stack.

3.4 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.

3.5 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* - *x*)

Load the word *x* stored at *addr*, which must be a multiple of `word_bytes`.

`store` (*x addr* -)

Store *x* at *addr*, which must be a multiple of `word_bytes`.

`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.

3.6 Immediate 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.

3.7 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.

3.8 Comparison

These words compare two numbers on the stack, returning a flag (for equality, use `xor`; see section 3.7):

`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.

3.9 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.

3.10 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 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 division instructions raise error -8 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.

3.11 Instruction encoding

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

3.12 Instruction opcodes

Table 4 lists the instruction opcodes, and table 5 the extra instruction opcodes (see section 3.1). Other instruction opcodes are invalid.

Opcode	Instruction	Opcode	Instruction
0x0	`extra`	0x10	`push`
0x1	`jump`	0x11	`pushrel`
0x2	`jumpz`	0x12	`not`
0x3	`call`	0x13	`and`
0x4	`pop`	0x14	`or`
0x5	`dup`	0x15	`xor`
0x6	`swap`	0x16	`lt`
0x7	`trap`	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

Opcode	Instruction
0x0	`next`
0x1	`halt`

Table 5: 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 **traps** (see section 3.3) to provide extra computational primitives, and to offer access to system facilities, previously written code, native libraries and so on.

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 fount of criticism 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/`.