Mit virtual machine specification

Reuben Thomas

27th June 2020

1 Introduction

Mit is a simple virtual machine, designed to be both easy to implement efficiently on most widely-used hardware, and to compile for. It aims to be formally tractable. This specification is intended for those who wish to implement or program Mit.

2 Parameters

The virtual machine has the following parameters:

Endianness	Memory can be either big- or little-endian.
word_bytes	The number of bytes in a word, 4 or 8.

3 Memory

The flat linear address space contains word_bytes-byte words of 8-bit bytes. Addresses range from 0 to $2^{8 \times \text{word\_bytes}} - 1$ inclusive and identify a byte; the address of a quantity is that of the byte in it with the lowest address. Whether a given word may be read or written can change during execution.

4 Registers

The registers are word quantities:

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.

5 Computation stack

Computation is performed with a last in, first out stack of words. The computation stack is usually referred to simply as "the stack". To **push** a word on to the

stack means to add a new word to the top; to **pop** a word means to discard the top item. Instructions implicitly pop their arguments and push their results.

Stack effects are written

$$before \rightarrow after$$

where `before` and `after` are stack pictures showing the items on top of the 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

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

where the i_k are stack items, with i_1 being on top of the stack.

6 Call stack and catch stack

Subroutines are implemented with the call stack, a last in, first out stack of computation stacks. The top-most is the current computation stack, which is used by instructions.

When a subroutine call is performed, a new computation stack is pushed on to the call stack. This is shown in a stack effect as:

$$caller \ ; \ callee$$

where `caller` and `callee` are stack pictures for the caller and callee respectively.

Error handling is implemented with the catch stack, a last in, first out stack of call stacks. The top-most is the current call stack. This is shown in a stack effect as:

$$handler \ | \ handlee$$

where `handler` and `handlee` are stack pictures for the top-most computation stacks in adjacent call stacks.

7 Execution

The registers (see section 4) are initialised to desired 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 8.9) and trap -1 (see section 8.10).

7.1 Errors and termination

In exceptional situations, such as an invalid memory access, or division by zero, an **error** may be **thrown**; see section 8.8. An **error code** is returned to the handler.

Execution can be terminated explicitly by a `throw` instruction (see section 8.8), which throws an error.

Error codes are signed numbers. 0 to -127 are reserved for the specification. The specified error codes are:

Code	Meaning
0	Execution has terminated without error.
-1	Invalid opcode (see section 8.11).
-2	Stack overflow.
-3	Invalid stack read.
-4	Invalid stack write.
-5	Invalid memory read.
-6	Invalid memory write.
-7	Address is valid but insufficiently aligned.
-8	Division by zero attempted (see section 8.4).
-9	Division overflow (see section 8.4).

Errors -2 to -9 inclusive are optional: an implementation may choose to raise them, or not.

8 Instructions

The instructions are listed below, grouped according to function, in the following format:

`NAME`   *before* $\rightarrow$ *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.

The symbols denoting different types of stack item are shown in table 1.

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 1: 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.

8.1   Stack manipulation

These instructions manage the stack:

`pop` $x \rightarrow$

Remove *x* from the stack.

`dup` $x_u \ldots x_0 \ u \rightarrow x_u \ldots x_0 \ x_u$

Remove *u*. Copy x_u to the top of the stack.

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

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

`swap` $x_{u+1} \ldots x_0 \ u \rightarrow x_0 \ x_u \ldots x_1 \ x_{u+1}$

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

8.2   Memory

These instructions fetch and store quantities to and from memory.

`load`   *a-addr* $\rightarrow x$

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

`store`   *x a-addr* $\rightarrow$

Store *x* at *a-addr*.

`load1`   *addr* $\rightarrow x$

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

`store1`   *x addr* $\rightarrow$

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

`load2`   *addr* $\rightarrow x$

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* $\rightarrow$

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

`load4`   *addr* $\rightarrow x$

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* $\rightarrow$

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

8.3 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` $\times$ *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 8.11.

8.4 Arithmetic

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

```
negate                                                   s₁  → s₂
```
Negate s_1, giving s_2.

```
add                                                  n₁  n₂  → n₃
```
Add n_2 to n_1, giving the sum n_3.

```
mul                                                  n₁  n₂  → n₃
```
Multiply n_1 by n_2 giving the product n_3.

```
divmod                                           s₁  s₂  → s₃  s₄
```
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                                          u₁  u₂  → u₃  u₄
```
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.

8.5 Logic

Logic functions:

```
not                                                      x₁  → x₂
```
Invert all bits of x_1, giving its logical inverse x_2.

```
and                                                  x₁  x₂  → x₃
```
x_3 is the bit-by-bit logical "and" of x_1 with x_2.

```
or                                                   x₁  x₂  → x₃
```
x_3 is the bit-by-bit inclusive-or of x_1 with x_2.

```
xor                                                  x₁  x₂  → x₃
```
x_3 is the bit-by-bit exclusive-or of x_1 with x_2.

8.6 Shifts

`lshift` $\hspace{6cm} x_1\ u\ \rightarrow\ x_2$

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` $\hspace{6cm} x_1\ u\ \rightarrow\ x_2$

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` $\hspace{6cm} x_1\ u\ \rightarrow\ x_2$

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.

8.7 Comparison

These instructions compare two numbers on the stack, returning a flag; see section 8.5):

`eq` $\hspace{6cm} s_1\ s_2\ \rightarrow\ flag$

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

`lt` $\hspace{6cm} s_1\ s_2\ \rightarrow\ flag$

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

`ult` $\hspace{6cm} u_1\ u_2\ \rightarrow\ flag$

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

8.8 Control

These instructions implement unconditional and conditional branches, and subroutine call and return, with and without error handling.

`jump` $\hspace{6cm} [a\text{-}addr]\ \rightarrow$

If `ir` is 0, the target address is *a-addr*, otherwise it is $\texttt{ir} \times \texttt{word\_bytes} + \texttt{pc}$. Set *pc* to the target address, and `ir` to 0.

`jumpz` $\hspace{6cm} flag\ [a\text{-}addr]\ \rightarrow$

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

`call` $\hspace{2.5cm} x_{u_1} \ldots x_0\ u_1\ u_2\ [a\text{-}addr_1]\ \rightarrow\ u_2\ a\text{-}addr_2\ ;\ x_{u_1} \ldots x_0$

Let the initial value of `pc` be *a-addr$_2$*. Perform the action of `jump`. Move $x_{u_1} \ldots x_0$ from the caller's stack to the callee's.

`ret` $\hspace{3cm} u\ a\text{-}addr\ ;\ x_u \ldots x_0\ \rightarrow\ x_u \ldots x_0$

$\hspace{3.5cm}$ or $\hspace{0.6cm} u\ a\text{-}addr\ |\ x_u \ldots x_0\ \rightarrow\ x_u \ldots x_0\ 0$

Pop the current computation stack from the call stack; if the call stack is now empty, pop the current call stack. Set `pc` to *a-addr*, and move $x_u \ldots x_0$ to the new stack. Set `ir` to 0. If a call stack was popped, push 0 on top of the stack.

```
throw                                                u a-addr | n → n
```
Throw error *n*. Pop the current call stack from the catch stack. Pop the return address and number of return items, and push the error code.

```
catch            x_u...x_0 u_1 u_2 a-addr_1 → u_2 a-addr_2 | x_u...x_0
```
Perform the action of `call` as if `ir` were 0, then push a new call stack on to the catch stack and move the top-most computation stack from the previous call stack to the new one.

8.9 Extra instructions

Extra instructions, using the `extra` instruction, offer necessary functionality too rare or slow to deserve a core instruction.

```
extra
```
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 7). The stack effect depends on the extra instruction.

8.10 Traps

Traps, using the `trap` instruction, are similar to extra instructions, 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.

```
trap
```
Perform trap `ir`; if `ir` is not the code of a valid trap, throw error -1. The stack effect depends on the trap. Trap code -1 is used for instruction fetch (see section 7).

8.11 Instruction encoding

Instructions are encoded as bytes, packed into words, which are executed as described in section 7. 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 2 lists the instructions whose least-significant 3 bits are 000 or 111. Other instruction opcodes with those endings are invalid. Table 3 lists the extra instruction opcodes.

7

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 2: Instruction opcodes

Opcode	Instruction
0x1	divmod
0x2	udivmod
0x3	catch
0x4	throw

Table 3: Extra instruction opcodes

9   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 8.9.

Acknowledgements

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/`.