The Mit virtual machine

Reuben Thomas

23rd June 2020

1 Introduction

Mit is a simple virtual machine. It is a stack machine, based on the more complex register machine Mite [**?**]. 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 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 3.8).
−9	Division overflow (see section 3.8).

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}`...`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_u`...`x_0 `u -`
Remove x_u`...`x_0 from the stack.

```
dup                                                    xᵤ...x₀ u - xᵤ...x₀ xᵤ
```

`dup` $x_u \ldots x_0\ u\ \texttt{-}\ x_u \ldots x_0\ x_u$

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

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

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

3.2 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 $\texttt{ir} \times \texttt{word\_bytes} + \texttt{pc}$. 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` $x_{u_1} \ldots x_0\ u_1\ u_2\ \textit{[a-addr}_1\textit{]}\ \texttt{-}\ u_2\ \textit{a-addr}_2\ \textit{||}\ x_{u_1} \ldots x_0$

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

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.

3.3 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.

3.4 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 3.10.

3.5 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.

3.6 Comparison

These instructions compare two numbers on the stack, returning a flag (for equality, use `xor`; see section 3.5):

```
lt                                     s₁ s₂  - flag
```
$flag$ is 1 if and only if s_1 is less than s_2.

```
ult                                    u₁ u₂  - flag
```
$flag$ is 1 if and only if u_1 is less than u_2.

3.7 Shifts

```
lshift                                   x₁ u  - x₂
```
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                                   x₁ u  - x₂
```
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.

3.8   Arithmetic

All calculations are made modulo $2^{(8\times\texttt{word\_bytes})}$, 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. If s_1 is $-2^{(8\times\texttt{word\_bytes}-1)}$ and s_2 is -1, throw error -9.

```
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.

3.9   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:

```
throw        1                                                      n  -
```

Throw error n.

```
catch      2  x_u...x_0  u_1  u_2  [a-addr_1]  -  h  u_2  a-addr_2  ||  x_u...x_0
```

Add an error handler h to the current stack frame, then perform the action of `call` (see section 3.2).

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
`pushreli` $n \geq 0$						0	1
`pushi` $n \geq 0$					0	1	1
`pushi` $n < 0$					1	0	0
`pushreli` $n < 0$						1	0
instruction					1	1	1

3.11 Instruction opcodes

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

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

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