Mit virtual machine specification

Reuben Thomas

21st July 2020

1 Introduction

Mit is a simple virtual machine, designed to be easy both to implement efficiently on most widely-used hardware, and to compile for. It aims to be formally specified. 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 are unsigned words and identify a byte; the address of a quantity larger than a byte 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 of code.
`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 remove the top item. Instructions implicitly pop their arguments and push their results.

Stack effects are written

$$\textit{before} \rightarrow \textit{after}$$

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

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

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 as:

$$\textit{caller} \ ; \ \textit{callee}$$

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 as:

$$\textit{handler} \mid \textit{handlee}$$

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

7 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.

Instruction fetch is the action of setting `ir` to the word pointed to by `pc` and making `pc` point to the next word. This is done whenever extra instruction 0 (see section 8.9) or trap -1 (see section 8.10) is executed.

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. A **status code** is returned to the handler.

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

Status codes are signed numbers. 0 to -127 are reserved for the specification. The defined status codes are given in table 1.
Errors -2 to -9 inclusive are optional: an implementation choose not to detect these conditions.

Code	Meaning
0	Normal termination.
−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).

Table 1: Status codes

8 Instructions

The instructions are listed below, grouped according to function, 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.

Numbers are represented in two's complement form. Where a stack item's name (including any numerical 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 items. An item called *count* is interpreted as an unsigned number.

8.1 Stack manipulation

These instructions manage the stack.

`pop` $x \rightarrow$

Pop x from the stack.

`dup` $x_{count} \ldots x_0\ count \rightarrow x_{count} \ldots x_0\ x_{count}$

Pop *count*. Copy x_{count} to the top of the stack.

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

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

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

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

8.2 Memory

These instructions fetch and store quantities to and from memory.

`load` *addr* → *x*

Push the word stored at *addr*, which must be word-aligned.

`store` *x addr* →

Store *x* at *addr*, which must be word-aligned.

`load1` *addr* → *x*

Push the byte stored at *addr*, setting unused high-order bits to zero, giving *x*.

`store1` *x addr* →

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

`load2` *addr* → *x*

Push the 2-byte quantity stored at *addr*, setting unused high-order bits to zero, giving *x*. *addr* must be a multiple of 2.

`store2` *x addr* →

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

`load4` *addr* → *x*

Push the 4-byte quantity stored at *addr*, setting any unused high-order bits to zero, giving *x*. *addr* must be a multiple of 4.

`store4` *x addr* →

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

8.3 Constants

`push` → *x*

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

`pushrel` → *addr*

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* → *addr*

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

The operand of `pushi` and of `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 below.

`neg` *a* → *b*

Negate *a*, giving *b*.

`add` *a b* → *c*

Add *a* to *b*, giving the sum *c*.

`mul` *a b* → *c*

Multiply *a* by *b*, giving the product *c*.

`divmod` *a b* → *q r*

Divide *a* by *b*, giving the quotient *q* and remainder *r*. The quotient is rounded towards zero. If *b* is zero, throw error -8. If *a* is $-2^{(8\times \texttt{word\_bytes}-1)}$ and *b* is -1, throw error -9.

`udivmod` *a b* → *q r*

Divide *a* by *b*, giving the quotient *q* and remainder *r*. If *b* is zero, throw error -8.

8.5 Logic

Logic functions:

`not` *a* → *b*

b is the bitwise complement of *a*.

`and` *a b* → *c*

c is the bitwise "and" of *a* with *b*.

`or` *a b* → *c*

c is the bitwise inclusive-or of *a* with *b*.

`xor` *a b* → *c*

c is the bitwise exclusive-or of *a* with *b*.

8.6 Shifts

`lshift` *a count* → *b*

Shift *a* left by *count* bits, filling vacated bits with zero, giving *b*.

`rshift` *a count* → *b*

Shift *a* right by *count* bits, filling vacated bits with zero, giving *b*.

`arshift` *a count* → *b*

Shift *a* right by *count* bits, filling vacated bits with the most significant bit of *a*, giving *b*.

8.7 Comparison

These instructions compare two numbers on the stack:

`eq` *a b* → *flag*

flag is 1 if *a* is equal to *b*, otherwise 0.

`lt` *a b* → *flag*

flag is 1 if *a* is less than *b* when considered as signed numbers, otherwise 0.

`ult` *a b* → *flag*

flag is 1 if *a* is less than *b* when considered as unsigned numbers, otherwise 0.

8.8 Control

Unconditional and conditional branches:

`jump` *[addr]* →

If `ir` is zero, set `pc` to *addr*, which must be aligned, otherwise to `pc` + `ir` × `word_bytes`. Set `ir` to 0.

`jumpz` *flag [addr]* →

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

Subroutine call and return; the quantities called *args* and *rets* are treated as unsigned numbers:

`call` $x_{args} \ldots x_1$ *args rets [addr]* → *rets ret-addr* ; $x_{args} \ldots x_1$

Let *ret-addr* be the value of `pc`. Perform the action of `jump`. Move $x_{args} \ldots x_1$ from the caller's stack to the callee's.

`ret` *rets ret-addr* ; $x_{rets} \ldots x_1$ → $x_{rets} \ldots x_1$

or *rets ret-addr* | $x_{rets} \ldots x_1$ → $x_{rets} \ldots x_1$ 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 *ret-addr*, and move $x_{rets} \ldots x_1$ to the new computation stack. Set `ir` to 0. If a call stack was popped, push 0 on to the computation stack.

Error handlers and raising errors:

`catch` $x_{args} \ldots x_1$ *args rets addr* → *rets ret-addr* | $x_{args} \ldots x_1$

Set `ir` to 0. Perform the action of `call`, 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.

`throw` *rets ret-addr* | *n* → *n*

Pop the current call stack from the catch stack.

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. The stack effect depends on the extra instruction. Extra instruction 0 performs instruction fetch (see section 7).

8.10 Traps

Traps, using the `trap` instruction, are similar to extra instructions, but are 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 performs 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:

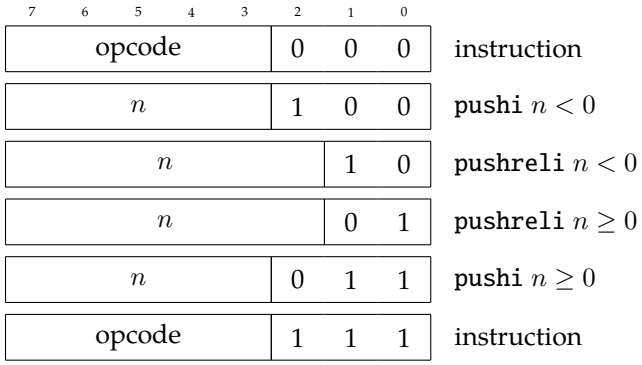

Table 2 lists the opcodes for instructions whose least-significant 3 bits are 000, and table 3 for 111. Other instruction opcodes with those endings are invalid. Table 4 lists the valid extra instruction opcodes.

Opcode	Instruction	Opcode	Instruction
0x00	`extra`	0x10	`load`
0x01	`not`	0x11	`store`
0x02	`and`	0x12	`load1`
0x03	`or`	0x13	`store1`
0x04	`xor`	0x14	`load2`
0x05	`lshift`	0x15	`store2`
0x06	`rshift`	0x16	`load4`
0x07	`arshift`	0x17	`store4`
0x08	`pop`	0x18	`push`
0x09	`dup`	0x19	`pushrel`
0x0a	`set`	0x1a	`negate`
0x0b	`swap`	0x1b	`add`
0x0c	`jump`	0x1c	`mul`
0x0d	`jumpz`	0x1d	`eq`
0x0e	`call`	0x1e	`lt`
0x0f	`ret`	0x1f	`ult`

Table 2: Instruction opcodes for "000" instructions

Opcode	Instruction
0xff	`trap`

Table 3: Instruction opcodes for "111" instructions

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

Table 4: Extra instruction opcodes

9 External interface

- 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.
- Implementations should provide an API to create and run virtual machine code, and to add more traps.

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. `https://www.gnu.org/software/lightning/`.

[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/`.