The Mite VM: bridging the complexity gulf

Reuben Thomas
Department of Computing Science, University of Glasgow*

31st October 2001

Stand-alone VMs are increasingly used as platforms for code generation and execution. The most popular, such as the JVM, are functionally complex and rich, and tend to force their world view on the code they execute. At the other end of the scale is Machine Forth, whose design is so simple as to be impoverished, and which lacks several features necessary for efficient portable code. Mite attempts to bridge this gulf, being simple yet sufficient for the task of portable code generation and execution. It also attempts to bridge the cultural divide which separates the above-mentioned systems, by being on the one hand familiar in design and compatible with conventional ideas on code generation and execution, and on the other hand giving users a considerable degree of freedom in its use and integration with other systems.

1 Introduction

VMs are increasingly recognised as useful platforms in their own right, rather than just a convenient way of writing language interpreters. The VM *du jour* is the JVM [3], a complex beast that encompasses memory management, concurrency, security and an object model. Programs that run on it must either fit its rules and assumptions, or lose efficiency and functionality. The Forth world's nearest relative is PRACTICAL [2], which, though rather more flexible in terms of data representation and code structure, shares the JVM's unwillingness to be integrated into larger systems (unsurprisingly, as it is designed to be used as part of the kernel of secure embedded systems). At the other end of the scale is Machine Forth [5], a VM which has been implemented in both hardware and software, and intended to be used as a basis for Forth programming, as its instruction set closely resembles low-level Forth, though with one or two innovations. However, a literal reading of its very concrete specification, which fixes the width of stack items and the size of the data stack, which is defined as a circular buffer, makes it unsuited to software implementation on other processors (but see [8] for a rather more generous discussion). Similarly, it is hard to see how Machine Forth code could be generated so that it is both portable and efficient.

*rrt@sc3d.org

In tandem with this technical range comes a cultural range. The JVM's design assumes that its users are happy for it to perform basic resource management, including the memory allocation and scheduling strategies. On the other hand, Machine Forth's aggressively simple design is hard even to generate code for with conventional compiler techniques (again, to be fair, this was not one of its design goals).

The central assumption of this paper is that the most useful features of virtual machines *per se* are to provide a single target for code generation, and the ability to run code portably on multiple platforms. Mite attempts to provide these features, and these only, while remaining compatible with existing tools and techniques, and allowing integration with non VM-based systems.

2 History

Mite originally appeared in a more complex design [9] that aimed to allow native code quality competitive with that of native optimising compilers. This is now referred to as Mite0. While Mite0 performed well, its implementation was complex, and in particular, implementing a back end for an optimising compiler was a daunting task. Also, its complexity reduced the speed of code generation. It therefore seemed reasonable to investigate how well a simpler design could perform. Since most of Mite0's complexity centred on its register model, this was discarded in favour of a fixed register set. This simple change made a whole set of simplifications of the design and implementation possible, whose results are presented here.

3 Using Mite

Mite's design is set out in the appendix. Here, we merely list some of the most important high-level features.

Simplicity The design is simple. It is easy to implement, target, and even hand-code for, with few pitfalls for the unwary.

Registers The fixed orthogonal register set, which makes it a simple target for conventional compilers. On the other hand, the number of registers can be varied for different uses, for example according to the number of registers available on different ranges of target architectures.

Stack pointer The provision of a stack pointer, push and pop instructions, and a "stack direction" constant abstracts the stack just enough to allow Mite to use the native stack on most systems, including non-contiguous stacks. The ability to read and set the stack pointer directly means that non-local returns can be performed without extra instructions, as was necessary in Mite0.

Instruction set Most Mite instructions correspond directly to single native instructions on most architectures. The use of a three-operand format eases code generation and improves efficiency on three-operand machines, while being easy to convert into multiple instructions where necessary for two, one or zero operand machines. The simple correspondence also

makes it possible to write in Mite code what would otherwise need to be written in native code, for example, some parts of device drivers, and make it straightforward to interwork between Mite and native code.

Definition Mite's definition is brief but comprehensive: all possible instruction behaviours are either defined or given as being undefined. It is straightforward to write a "sandboxed" interpreter that checks all error conditions and does not allow unsafe operations on the host system.

These features make Mite straightforward to implement, straightforward for compilers to target, and easy in general for programmers to think about. Its similarity to hardware processors make it easy to integrate Mite code with native code, and easier to take advantage of.

4 Integration

Unlike the JVM, which needs a special interface, the JNI (Java Native Interface), to interwork with native code, Mite can simply call native code directly using the `calln` instruction; in a native code implementation, this will usually be exactly the same as a `call`. In a particular implementation, further knowledge can be used, for example about the mapping between Mite registers and physical registers.

To allow fully portable code to integrate with native C libraries, however, it is still necessary to provide special instructions, like Mite0's (essentially duplicates of `call`, `ret` and subroutine labels). Alternatively, a higher-level solution that maps types dynamically at run-time could be used. Mite's openness and flexibility allows a range of solutions to be used.

Of course, it may also be desirable to integrate with libraries written in other languages; it is simply the case that inter-language working is usually done through the medium of C. More complex mechanisms, such as .NET's Common Language Subset [4], could be used just as they are from native code (again, Mite's closeness to real processors allows old solutions to be re-used).

5 Implementation

Although Mite was designed to be straightforward to implement, considerable attention has also been paid to the structure of the implementation. Mite is implemented in C, for speed and portability, but the C code is itself generated from a series of specifications, for the instructions and operand types, and one for each input type (assembly and object code) and output type (object code and each sort of native code. Even the interpreter is implemented by means of special output types.

A series of translators, for example, from assembly to object code, and from object code to each sort of native code, is then constructed, each one a combination of an input and an output module. Thus each translator is customised for the desired type of translation, and runs very fast; nonetheless, the simplicity of the code means that it is not too large (a typical translator takes around 6Kb of IA32 code when compiled by GCC `-O2`).

6 Future work

To make Mite useful, it needs compiler support. This is the current focus of development effort. First, explicit size suffixes will be added to existing instructions, and C call and return instructions adapted from Mite0. Then, the LCC back-end targeting Mite0 will be adapted for Mite. Dynamic linking support will be added, followed by a floating-point extension to Mite's design. At this point it will be possible to compile C programs that will then run on any system supported by Mite that uses the same type representations. In practice, most systems with a given data width should be compatible.

Next, native back-ends will be written for various processors, starting with the most difficult (and also most widespread), the Intel IA32 architecture. Load-time type mapping will be added to allow code to be compiled portably across all architectures; this however will require support for each library.

A back-end will then be written for GCC. This will enable Mite's performance to be accurately evaluated, and increase the number of languages that can be compiled to Mite. A GCC back-end opens other doors, too: a Linux port, perhaps?

Mite's development is taking place on SourceForge, at `mite.sourceforge.net`.

7 Conclusion

Mite is a general-purpose VM, intended to bridge the complexity gulf that exists between the all-embracing JVM and the ascetic Machine Forth, as well as the cultural gap between the philosophies that spawned them. By concentrating on code generation and execution, eschewing ultimate performance in favour of simplicity, and emphasising integration with existing systems and the use of conventional compilation techniques, Mite aims to be a compelling choice as a compiler target and execution engine for a wide range of systems.

References

[1] Paulo Bonzini. Using and porting GNU *lightning*, 2000. `ftp://alpha.gnu.org/gnu/`.

[2] MicroProcessor Engineering. The PRACTICAL virtual machine architecture, 1998.

[3] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[4] .NET framework SDK technology preview. `http://msdn.microsoft.com/downloads/`.

[5] Charles Moore and Jeff Fox. Preliminary specification of the F21, 1995. `http://pisa.rockefeller.edu:8080/MISC/F21.specs`.

[6] Martin Richards. Cintcode distribution, 2000. `http://www.cl.cam.ac.uk/~mr/BCPL.html`.

[7] Reuben Thomas. Beetle and pForth: a Forth virtual machine and compiler. BA dissertation, University of Cambridge, 1995. `http://sc3d.org/rrt/`.

[8] Reuben Thomas. Machine Forth for the ARM processor. In *EuroForth '99 conference proceedings*, 1999. `http://sc3d.org/rrt/`.

[9] Reuben Thomas. *Mite: a basis for ubiquitous virtual machines*. PhD thesis, University of Cambridge Computer Laboratory, November 2000. `http://sc3d.org/rrt/`.

A Introduction

Mite is a general-purpose virtual machine. It is designed to be capable of efficient implementation, by interpretation or compilation, from a binary-portable object format. It has a flat linear memory, and single-threaded, non-interruptable execution. The only external access provided is the ability to call machine code.

B Parameters

Mite is parametrized on the following quantities:

W number of bytes in a word

R the number of registers ($R = 2^n, 2 < n$, for some n)

S direction of stack growth (-1 for an ascending stack, 1 for a descending stack)

C Registers and memory

Registers are word-sized. Mite has R registers `1` to R, a program counter `P`, and a stack pointer `S`. The stack is a LIFO stack; `S` points to the top-most item.

The memory M is an array of bytes. The stack resides in memory. $M(a)$ denotes the word in memory starting at byte a; a must be a multiple of W. Within a word, bytes may be stored in big or little-endian order. Two's complement number representation is used.

D Execution

Mite repeatedly loads the instruction at `P`, makes `P` point to the next instruction, and executes the loaded instruction. Program addresses need not be addresses in M.

E Instructions

In the syntax, each operand is denoted by a letter subscripted by a number. The letter indicates the sort of operand required, as shown in table 1; the number shows the number of the operand.

$\{P\}$ has the value 1 if the predicate P is true, and 0 otherwise.

r	register
i	immediate constant
t	label type
l	label
n	name

Table 1: Operand types

lab t_1 n_2	define a type t_1 label named n_2

Table 2: Label definition

E.1 Labels

Labels are defined by the lab instruction, shown in table 2.

There are three types of label: branch labels, denoted b, subroutine labels, denoted s, and data labels, denoted d. A label is written as its name. A name is a string of letters, digits and underscores, and may not start with a digit; names must be unique within a translation unit.

The value of a code or subroutine label is the address of the instruction immediately following it. A data label's value is the address of the first datum stored after it by a data instruction.

Branch labels have no effect when they are executed; subroutine and data labels may not be executed.

E.2 Computation

The computational instructions are shown in table 3.

Immediate constants are of the form [e][s][w]*n*[>>*r*]. *n* (and *r*, if present) is an integer. If e (for "endianness") is present, *n* is subtracted from *W* on a big-endian Mite, or left unaltered otherwise; then if s is present, it is multiplied by *S*, and if w is present, it is multiplied by *W*. The final value is truncated to 8*W* bits, then if >>*r* is present, it is rotated by *r* places, to the right if positive, and to the left if negative.

E.3 Data

The data instructions, shown in table 4, allow literal data to be included in object code; they may not be executed. Data between two labels are stored contiguously in *M*.

lit causes its operand to be truncated to a word and stored in the next word of memory. litl causes the values of the given label to be stored in the next word of memory.

space causes the given number of zero words to be stored in consecutive locations.

mov r_1 r_2	$r_1 \leftarrow r_2$
movi r_1 i_2	$r_1 \leftarrow i_2$
ldl r_1 l_2	$r_1 \leftarrow synd l_2$
ld r_1 r_2	$r_1 \leftarrow M(r_2)$
st r_1 r_2	$M(r_2) \leftarrow r_1$
gets r_1	$r_1 \leftarrow S$
sets r_1	$S \leftarrow r_1$
pop r_1	$r_1 \leftarrow M(S)$; $S \leftarrow S - sw$
push r_1	$S \leftarrow S + sw$; $M(S) \leftarrow r_1$
add r_1 r_2 r_3	$r_1 \leftarrow r_2 + r_3$
sub r_1 r_2 r_3	$r_1 \leftarrow r_2 - r_3$
mul r_1 r_2 r_3	$r_1 \leftarrow r_2 \times r_3$
div r_1 r_2 r_3	$r_1 \leftarrow r_2 \div r_3$ (unsigned)
rem r_1 r_2 r_3	$r_1 \leftarrow r_2$ mod r_3 (unsigned)
and r_1 r_2 r_3	$r_1 \leftarrow r_2$ bitwise and r_3
or r_1 r_2 r_3	$r_1 \leftarrow r_2$ bitwise or r_3
xor r_1 r_2 r_3	$r_1 \leftarrow r_2$ bitwise xor r_3
sl r_1 r_2 r_3	$r_1 \leftarrow r_2 << r_3$ ($0 \leq r_3 \leq 8w$)
srl r_1 r_2 r_3	$r_1 \leftarrow r_2 >> r_3$ (logical, $0 \leq r_3 \leq 8w$)
sra r_1 r_2 r_3	$r_1 \leftarrow r_2 >> r_3$ (arithmetic, $0 \leq r_3 \leq 8w$)
teq r_1 r_2 r_3	$r_1 \leftarrow \{r_2 = r_3\}$
tlt r_1 r_2 r_3	$r_1 \leftarrow \{r_2 < r_3\}$
tltu r_1 r_2 r_3	$r_1 \leftarrow \{r_2 < r_3$ (unsigned)$\}$
b l_1	$P \leftarrow synbl_1$
br r_1	$P \leftarrow r_1$
bf r_1 l_2	if $r_1 = 0$, $P \leftarrow synbl_2$
bt r_1 l_2	if $r_1 \neq 0$, $P \leftarrow synbl_2$
call l_1	$S \leftarrow S + sw$; $M(S) \leftarrow P$; $P \leftarrow synsl_1$
callr r_1	$S \leftarrow S + sw$; $M(S) \leftarrow P$; $P \leftarrow r_1$
ret	$P \leftarrow M(S)$; $S \leftarrow S - sw$
calln r_1	call native code at r_1

Table 3: Computational instructions

lit i_1	a literal word
litl t_1 l_2	a literal label
space i_1	i_1 zero words ($i_1 > 0$)

Table 4: Data instructions

Instruction	Opcode	Instruction	Opcode	Instruction	Opcode
lab	01h	mul	0dh	b	19h
mov	02h	div	0eh	br	1ah
movi	03h	rem	0fh	bf	1bh
ldl	04h	and	10h	bt	1ch
ld	05h	or	11h	call	1dh
st	06h	xor	12h	callr	1eh
gets	07h	sl	13h	ret	1fh
sets	08h	srl	14h	calln	20h
pop	09h	sra	15h	lit	21h
push	0ah	teq	16h	litl	22h
add	0bh	tlt	17h	space	23h
sub	0ch	tltu	18h		

Table 5: Instruction opcodes

F Object format

In the description below, hexadecimal numbers are indicated by a leading "0x".

Object code consists of a series of instructions.

F.1 Instructions

Instructions are encoded as the opcode followed by the operands, in numerical order. Instructions that do not end on a 4-byte boundary are padded to the next such boundary with zero bytes.

Opcodes, registers and label types are encoded as one byte. The instruction opcodes are shown in table 5. Opcodes 0x25–0x7f are reserved for future expansion. Registers are encoded by their number. Label types are encoded as 1 for b, 2 for s, and 3 for d.

Labels are encoded as a long number (see section F.2). Labels of each type are numbered consecutively from 1 from the start of the translation unit, according to the order in which they are declared. For a lab instruction, only the label type is encoded (the number is redundant).

Immediate constants are encoded as a byte encoding the modifiers followed by a byte encoding the rotation (if any), followed by the basic value encoded as a long number. The modifiers are encoded as binary flags, as shown in table 6; the unused bits are zeroed.

F.2 Long numbers

Long numbers are encoded as follows:

1. Left-truncate the number to the minimum number of bits in which it can be represented.
2. Remove an $8n-1$-bit word from the most significant end of the number, where n is the number of bytes left in the current instruction word. If there are fewer than $8n-1$ bits in

Modifier	Bit position
rotation	0
w	1
s	2
e	3

Table 6: Constant modifiers encoding

the number then sign-extend it to that length first.

3. Add a bit to the most significant end of the word, which should be zero if more bits remain in the number, and one otherwise.
4. Store the word, with the bytes in big-endian order.
5. While there are still bits in the number, repeat the following steps:
    a) Remove a 31-bit word from the most significant end of the number, padding with zeroes if there are fewer than 31 bits left.
    b) Add a bit to the most significant end of the word, which should be zero if more bits remain in the number, and one otherwise.
    c) Store the word, with the bytes in big-endian order.

G Acknowledgements

Martin Richards introduced me to Cintcode [6], which kindled my interest in virtual machines, and led to Beetle [7] and an earlier version of Mite [9], of which the current Mite is a sort of synthesis. GNU *lightning* [1] helped inspire me to greater simplicity, while still aiming for good performance. Alistair Turnbull has been a fount of criticism for all my work on virtual machines.