Mite: a tutorial introduction

Reuben Thomas

24th October 2006

1 Architecture

Mite has a load-store architecture. The main difference from conventional processors is its treatment of memory and registers, which must be dealt with carefully in order to cope with differences between machines: the number and size of physical registers, and alignment restrictions on accessing memory.

1.1 Quantities

The basic types with which Mite deals are strings of bits, called **quantities**. The length of the string is the quantity's **width**. A denotes the width of an address, and may be either 32 or 64. Quantities may be one, two, four or $A/8$ bytes wide. An address-wide quantity is called a **word**. The constant `ashift` has the value $\log_2 A/8$ (the number of bit positions that a quantity must be left-shifted to multiply it by $A/8$), and `a` denotes $A/8$ wherever a width is required.

1.1.1 Offsets and sizes

Since words may be 32 or 64 bits wide, and words must always be stored at word-aligned addresses (see section 1.2), data structures may be laid out in different ways on different machines. For example, a record consisting of two addresses separated by a four-byte integer would require 12 bytes of storage if A is 32, and 24 if A is 64, as shown in figure 1. In the latter case the integer has to be padded to the next eight-byte boundary so that the second address is correctly aligned.

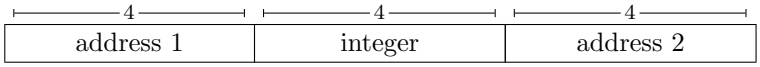
In order to describe the sizes of such structures, and offsets within them, Mite has **two-component numbers**. A two-component number is written as b`@`w, which has the value $b + Aw/8$. The second component may be omitted if it is zero. b is a number of bytes, and w a number of words.

The size of the record described above is `0@3`: the addresses are a word each, and the integer has to be rounded up to a full word so that the second address is properly aligned. The offset to the second address (third field) is `0@2`. If the integer were stored at the end, the size would be only `4@2`, and take only 20 bytes on a 64-bit machine.

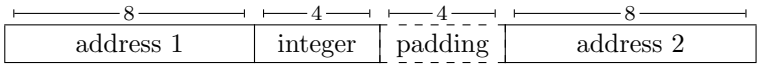
1.2 Memory

The memory is a word-indexed array of bytes; an index into the memory is called an **address**. The memory is effectively the data address space; not all addresses are necessarily valid. The stack (see the next section) is held in memory; code may also reside in memory, but it need not, and the effects of manipulating it if it does are undefined.

Memory may be accessed at any width; the address must be aligned to the given width. The effects of overlapping accesses at different widths are undefined;



(a) 32-bit machine; no padding needed



(b) 64-bit machine; padding required

Figure 1: Layout of a record at different word widths

for example, if the two-byte quantity ABCDh is stored at address 14, and the single byte at 15 is then examined, its value may be either ABh or CDh.

1.3 Stack

Mite uses a stack for data processing, which resides in memory. There are two types of stack item: registers, which are the same size as a machine address, and chunks, which can be any size. A register's value can be manipulated directly, and may be cached in physical registers; chunks reside permanently on the system stack, and can only have their address taken. Stack items are created on top of the stack, and only the top-most item may be destroyed. The items are referred to by their position in the stack, one being the bottom-most. All stack items occupy a whole number of words.

The current configuration of stack items is called the stack's **state**; the state is statically determined at each point in the program (see section 2.1).

Each register has a different rank between one and the number of registers. This is used for register allocation: typically, if the host machine has N physical registers, the virtual registers with ranks 1 to N will be assigned to physical registers. When a register's rank changes, it may be spilled from or loaded into a physical register.

Chunks are used for three main purposes: stack-allocated scratch space, passing structures by value to functions and subroutines, and for callee-saved information within subroutines and functions (the "return chunk"). Sections 2.6 and 2.7 explain the uses of chunks in functions.

1.4 Flags

There are four flags: zero (f_Z), negative (f_N), carry (f_C), and overflow (f_V). They are set by each data processing instruction (see section 2.3). The flags may only be tested by conditional branches, and must be tested immediately after they are set; see section 2.5 for examples.

2 Instruction set

The instruction set is RISC-like, and generally three-operand.

2.1 Creating and destroying stack items

A register is created by the instruction `NEW`. A newly created register is given rank 1. A chunk is created by `NEW_`n, where n is the size of the chunk, given as a two-component number, as in section 1.1.1. Since all stack items occupy a whole number of words, a chunk created with `NEW_3@0` will occupy either 4 or 8 bytes, depending on the value of A. `KILL` destroys the top-most stack item.

2.2 Register assignment

The instruction `MOV r,v` assigns v to register r. v can be either a register or an immediate constant. An immediate constant is either a two-component number or a label, optionally with a two-component offset added to it.

Registers can be either constant or variable. A constant register may only have its value changed by a later `DEF` or `MOV`, while variable registers may be updated by ordinary instructions such as `ADD`, `MUL`, and so on.

The instruction `DEF r,v` makes r a constant register with value v; `UNDEF r` makes r a variable. Using `MOV` has the same effect as `UNDEF`, and makes the register variable from then on.

`DEF` and `UNDEF` are static declarations, and apply from the point in the program at which they occur to the next `MOV` or `DEF` acting on the same register. `MOV` is an ordinary dynamic instruction, and loads the register with the specified value only when it is executed (though it makes the register variable statically, just like `UNDEF`). Sections 2.5 and 5 contain examples illustrating the difference between constant and variable registers.

2.3 Data processing

Most data processing instructions take two operands and one result. Destination registers are given before operand registers. All operations are performed to word precision. Every data processing instruction except `MUL` and `DIV` sets f_Z if its result is zero and f_N if the most significant bit of the result is one.

2.3.1 Simple arithmetic

For addition (`ADD`), subtraction (`SUB`) and multiplication (`MUL`), each instruction takes one destination and two operands. The result of `MUL` is the least-significant word of the product of its operands.

As an example, the following code computes the discriminant of a quadratic equation $ax^2 + bx + c = 0$ where a is in register 1, b in register 2 and c in register 3:

`MUL`	`2, 2, 2`	Compute b^2 in register 2
`MUL`	`1, 1, 3`	Compute ac in register 1
`KILL`		c (register 3) is no longer needed
`NEW`		Create a register to hold the constant 4
`DEF`	`3, #4`	Define register 3 to be constant with value 4
`MUL`	`1, 1, 3`	Compute $4ac$ into register 1
`KILL`		4 (in register 3) is no longer needed
`SUB`	`1, 2, 1`	Compute $b^2 - 4ac$ into register 1
`KILL`		b^2 (register 2) is no longer needed

The result of the calculation is placed in register 1. Note how the top-most stack item is killed as soon as it is finished with, possibly freeing a physical register or stack slot. Declaring the register created to hold 4 as constant enables the translator to treat it as an immediate quantity, and it may never need to be loaded into a physical register.

The `NEG` instruction takes a single operand, which it negates and stores in the destination.

`ADD`, `SUB` and `NEG` set f_C to the carry out from the most significant bit of the result, and f_V if signed overflow occurred.

2.3.2 Division

The `DIV` instruction takes two destinations and two operands: `DIV q,r,x,y` computes $x \div y$ and $x \bmod y$, placing the quotient in q and the remainder in r. The operands are treated as unsigned. Either destination may be omitted if the corresponding result is unwanted, but not both.

`DIVS` performs signed division, rounding the quotient towards minus infinity ($17 \div -7 = -3$ rem. -4); `DIVSZ` rounds towards zero ($17 \div -7 = -2$ rem. 3). In all cases the identity $qy + r = x$ holds, where x is the dividend, y the divisor, q the quotient and r the remainder.

2.3.3 Logic and shifts

`AND`, `OR`, `XOR` and `NOT` perform the corresponding bitwise logical operations. `SL` performs a left shift, `SRA` an arithmetic right shift, and `SRL` a logical right shift. The first operand to a shift is the quantity to be shifted and the second is the number of places to shift, which must be between 0 and A inclusive. Shift instructions set f_C to the carry out of the shift.

The following code sets register 1 to zero if it was previously negative:

```
NEW                     temporary register
NEW
DEF     3, #-1@8        number of bits in a word minus 1
SRA     2, 1, 3         make mask out of sign bit
KILL                    constant no longer needed
NOT     2, 2            invert mask
AND     1, 1, 2         clear if it was negative; otherwise leave it
                        alone
KILL                    mask no longer needed
```

2.3.4 Comparisons

The instructions `SUB`, `AND` and `XOR` may also be used to make comparisons by omitting the destination, so that they affect only the flags. Examples of this use are given in section 2.5.

2.4 Addressing memory

`LD_`w x`,[`a`]` loads the w-wide quantity at the address in register a, which must be a multiple of w, into register x. The quantity is zero-extended if necessary. `ST_`w x`,[`a`]` stores the least significant w bytes of x at the address in a.

Though nothing may be assumed about the ordering of the bytes in multi-byte quantities, it is possible to write some machine-dependent operations in a machine independent way; for example, the following code reverses the order of the bytes in the two-byte quantity at the address in register 1:

```
NEW
NEW
DEF     3, #1
ADD     2, 1, 3         copy address and add 1
KILL
NEW                     registers to hold bytes
NEW
LD_1    3, [1]          load bytes
LD_1    4, [2]
ST_1    3, [2]          store bytes the other way around
```

```
ST_1    4, [1]
KILL                    kill all registers used
KILL
KILL
KILL
```

2.5 Branching

`B`*c* *a* causes a branch to address *a* if condition *c* is true. There are fourteen conditions such as `EQ` ("equals"), `MI` ("minus") and `CS` ("carry set"), plus `AL` ("always"). The address *a* is a label, or a register holding the value of a label. Arbitrary addresses may not be used as branch targets; as mentioned in section 1.2, code need not even reside in addressable memory. Stack items which are live at the source and destination of a branch must match, in the sense that they must have the same type, and constants must have the same value.

The following code uses conditional branches and the comparison instructions introduced in section 2.3.4 to count the number of ones in register 1.[1] Note that register 2 is used as a counter, so its initial value is `MOV`ed into it, as it is a variable register, whereas registers 4 and 5 are constant, so their values are `DEF`ed.

```
NEW                     ones counter
MOV     2, #0
NEW                     temporary register
NEW                     constant needed in loop
DEF     4, #1
SUB     , 1, 2          test special case where input is 0
BEQ     .exit           finish if so
.loop                   label marking the start of the loop
ADD     2, 2, 4         increment counter
SUB     3, 1, 4         word−1
AND     1, 3, 1         knock off least-significant one in word
BNE     .loop           go back for next one if non-zero
.exit
KILL
KILL
```

2.6 Subroutines

The instruction `CALL` *a*`,`*n*`,[`t_1`,...,`t_n`]` calls the subroutine at address *a*. The topmost *n* items on the stack are passed as arguments to the subroutine. $t_1 \ldots t_n$ describe the return values as follows: t_1 gives a number of registers, t_2 the size of a chunk, t_3 a number of registers and so on. The effect of a `CALL` on the stack is as if the following code were executed:

```
KILL                    (n times)
NEW                     (t1 times)
NEW_t2
  ⋮
```

The arguments passed to the subroutine are killed, and the return values are created in their place.

A subroutine entry point is indicated by a label preceded by an `s`. The `s` may be followed by `l` if the subroutine is a leaf routine, that is, it contains no `CALL`

[1] There are much more efficient ways of doing this, but they are useless for the present purpose, as they involve neither comparisons nor loops.

instructions; this may allow the translator to optimize it. On entry to a subroutine the stack holds the arguments passed to it, with the return address and any other callee-saved information in a chunk on top of the stack.

`RET` c`,[`i_1`,...,`i_n`]` returns from the current subroutine. c is the chunk holding the return address, which is the stack item directly above the last argument. The return values $i_1 \ldots i_n$ are copied into the caller's stack.

The following code shows the use of a subroutine which computes the sum and difference of its arguments.

```
NEW                         declare arguments
NEW
sl.sumdif                   subroutine entry point
NEW                         register to hold difference
SUB     4, 1, 2             calculate difference
ADD     1, 1, 2             calculate sum
RET     3, [1, 4]           return results
KILL                        kill the live registers
KILL
KILL
KILL
.main                       entry point of program
NEW                         set up arguments
MOV     1, #5
NEW
MOV     2, #7
CALL    .sumdif, 2,         call subroutine
[2]
```

The type and number of the subroutine's arguments are given by the state of the stack at the subroutine label. In this case, two registers are declared directly before the label. The label itself effectively declares the return chunk. The calculation is performed, and the `RET` instruction specifies which stack items should be returned as results. Finally, all four stack items live at the end of the subroutine must be killed.

Mite's specification does not define how execution commences. The convention used here is to start execution at the label `.main`.

2.7 Functions

Mite supports the host environment's calling convention with functions, allowing Mite code to call and be called by native code. Function labels start with `f`, and `CALLF` and `RETF` call and return from functions. (Subroutines need not obey the system calling convention, which allows them to return multiple results, and have a more lightweight implementation.)

Function labels have up to three modifiers after the initial `f`: a leaf function is marked `l`, a function whose return value is a chunk is marked `c`, and a variadic function is marked `v`. The modifiers must occur in that order.

There are two forms of `CALL` for functions: if the function returns a register or has no result, `CALLF` is used, which has the same syntax as `CALL`, but allows a maximum of one return value. For functions returning a chunk, `CALLFC` is used; instead of a return list, a single item is given, which is either the chunk into which the result should be copied, or a register containing the address at which it should be stored. When a variadic function is called, `V` must be appended to the instruction name.

`RETF` has the same syntax as `RET`, but functions may return at most one value.

The following example demonstrates a variadic function which returns a chunk:

```
NEW_0@2                 chunk to hold result
NEW                     the variadic arguments
DEF     2, #2
NEW
DEF     3, #4
NEW
DEF     4, #7
NEW
DEF     5, #3           the number of variadic arguments
CALLFCV .f, 4, 2        pass 4 arguments and receive result in
                        chunk 2
  ⋮
NEW_0                   the variadic argument chunk
NEW                     the argument count
flcv.f                  a variadic leaf function returning a chunk
NEW                     address increment
DEF     4, #0@1
NEW                     accumulator for sum
NEW                     accumulator for product
MOV     5, #0           set sum to zero
MOV     6, #1           set product to one
NEW                     pointer to variadic arguments
MOV     7, 1            address of first argument
NEW
NEW
DEF     9, ashift
SL      8, 2, 9         get address of end of variadic arguments
KILL
ADD     8, 8, 7
NEW                     temporary
.accumulate
LD_a    9, [7]          load variadic argument
ADD     5, 5, 9         add to sum
MUL     6, 6, 9         multiply into product
ADD     7, 7, 4         increment the address
SUB     , 7, 8          repeat until all arguments read
BNE     .accumulate
KILL                    get rid of temporary
NEW_0@2                 create chunk to hold results
MOV     8, 9            get address of return chunk
ST_a    5, [8]          store sum
ST_a    6, [8, 4]       store product
RETF    3, [9]          return result
KILL                    kill all items
KILL
KILL
KILL
KILL
KILL
KILL
KILL
KILL
```

As for subroutines in the previous section, the state of the stack at the function label is taken to declare the function's arguments, and the function label itself implicitly declares the return chunk. The variadic arguments to a function come

below the normal arguments on the stack. They are stored in chunk 1, which must be declared to have size 0; the format in which they are stored is system-dependent. In this example it does not matter provided that each argument occupies a single word.

2.8 Non-local exit

The `CATCH` and `THROW` instructions, together with handlers, allow non-local exit from a subroutine or function. A handler is a label with `h` prefixed. `CATCH` r`,`l saves the value of the stack pointer at handler label l in register r. Later, while the subroutine in which `CATCH` was executed is still live, `THROW` l`,`r`,`v returns control to the handler at l. The value v overwrites the top stack item live at the handler, which must be a register. The value of l in the `THROW` instruction must be the same as in the `CATCH` that yielded the value of r.

When `THROW` is executed, the stack's state is changed to that of the handler label. Since this may not be the same as at the corresponding `CATCH` instruction, only those stack items which are live at both the `CATCH` and the handler label have a defined value. Moreover, since the values of registers that are cached in physical registers may be lost when a `THROW` is executed, all registers are assumed to be held in memory at a handler, and the `SYNC` modifier is provided to save all registers to memory. It has one operand, a handler label, which is used to decide which registers need to be saved. `SYNC` may be attached to a `CALL` or `THROW` instruction, and is only needed when the handler is in the same subroutine or function as the instruction being `SYNC`ed.

The following code demonstrates the use of `CATCH` and `THROW`. The code from `h.handler` onwards is run three times: first on entry to `main`, then by the first `THROW`, which throws from one point in `main` to another, and finally by the second `THROW`, which causes a non-local exit from the subroutine at `.sub`. The result is that the four words at `.store` are changed from their initial contents of four zeros to the sequence 0, 1, 2, 3. Note that both `THROW` and `CALL` must be decorated with `SYNC`, so that the virtual registers live at the handler are sure to have the correct values when the handler is reached.

```
.main
NEW
DEF     2, .store               address of results

NEW
MOV     3, #1                   first value to store
h.handler
NEW                             address offset
NEW                             shift
DEF     5, ashift
SL      4, 3, 5
KILL
ST_a    3, [2, 4]
DEF     4, #1
ADD     3, 3, 4                 next value to throw

NEW
CATCH   5, .handler             get catch value

NEW
MOV     6, .handler             address to throw to

DEF     4, #2                   second value to store
```

```
SUB       , 3, 4                    decide next destination

BEQ       .same                     based on previous result

DEF       4, #3                     third value to store
SUB       , 3, 4
BEQ       .sub
RETF      1, []
.same
DEF       4, #2
THROW     6, 5, 4 SYNC .handler     throw to same subroutine

.sub
DEF       4, #3
CALL      .sub, 3, [] SYNC          in fact, this call won't re-
.handler                            turn
KILL                                kill remaining items
KILL
KILL
NEW                                 create parameters
NEW
NEW
sl.sub
THROW     3, 2, 1
KILL
KILL
KILL
d.store
SPACEZ_a 4                          words to hold results
```

2.9 Escape

There is a general-purpose escape mechanism: `ESC #`n performs implementation-dependent function n.

2.10 Optimization directives

The instruction `RANK` r`,`n changes the rank of register r to n. The ranks of the other registers are altered accordingly so that all the ranks are still distinct and in the range 1–N, where N is the number of registers.

`REBIND` causes the binding of virtual to physical registers to be updated to reflect the current ranking. It is intended for use just before a loop, to minimize spills within.

3 Data

Constant data and static data areas are declared in data blocks. A data block starts with a label prefixed with `d`, or `dr` if the data is to be read-only, followed by a series of data directives enclosed in brackets.

The directives are `LIT_`w v_1`,`$\ldots$`,`v_n, which stores literal values $v_1 \ldots v_n$ of width w, and `SPACE_`w n, which reserves space for n w-wide quantities; `SPACEZ` causes the space reserved to be zero-initialized. All values stored and space reserved are aligned to the given width.

4 Object format

The object format is a simple byte-code. Multi-byte quantities are stored in little-endian order. The three main building blocks of the encoding are:

Numbers The number is divided into 7-bit words. Each is padded to a byte with a zero bit, except for the least-significant word, which is padded with a one. The bytes are stored most-significant first.

Header The object module header consists of a four-byte magic number, then a single-byte version number, then three bytes giving the length of the module in bytes, excluding the header. The number of labels follows, and finally the name of the module, stored as a counted string.

Instructions Each instruction consists of a one-byte opcode followed by a list of operands. When an operand consists of a list of values, the list is prefixed by its length. For example, the instruction `RET 4,[1,3,7]` is encoded as 86h 84h 83h 81h 83h 87h. The first byte is the opcode, the second is the encoding of the return chunk, number 4; the third byte encodes the length of the list of return items, which is 3. The return items follow.

5 Pitfalls

Programming Mite is deceptively similar to programming in a conventional assembly language, but there are fundamental differences that should be borne in mind. Almost all the pitfalls concern the use of registers.

Machine-independent coding is tricky It is easy to write Mite code that is machine-dependent. The biggest problem is assuming a particular width for the registers by mistake, just as in C one is tempted to assume particular sizes for types. A similar problem arises with optimization: it is tempting to try to second-guess the translator. Since Mite code may be run on a variety of machines, this is bootless; however, there is no clear dividing line between machine-independent and machine-dependent optimizations. The best weapon against both types of machine-dependence is never to think in terms of a particular host machine.

Registers are plentiful Although one should use as few registers as possible, quantities should always be held in a register where possible, rather than in memory or a chunk, so that the translator has a chance to allocate them to physical registers.

Registers are not concrete One of the hardest things to remember is that registers have limited scope, and that their use must be checked more carefully than in most assembly languages, especially around and across branches.

The stack must match across branches Bugs are easily introduced by branching to a place with a different stack state; the type of each item live at both source and destination of a branch must match. The next point is an example of this.

Constants are static `DEF` operates statically, not dynamically, which can create problems in loops. The following code:

```
DEF     1, #4
.loop
DEF     1, #3
BAL     .loop
```

is illegal, because the value of constant register 1 is not the same at the `BAL`, where its value is 3, as at `.loop`, where its value is 4.

Live ranges may not contain holes Since registers must be created and destroyed in stack order, live ranges may not contain holes. This places restrictions on the order in which compilers can linearize flow graphs; or alternatively, on how tightly they can define live ranges. For example, if a register is live in the test of an `if` statement, and continues to be live in the `then` branch, but not in the `else` branch, then the compiler must either compile the `else` branch second, and kill the register at the end of the `then` branch, or it must allow the register to be live in the `else` branch, even though the quantity it holds is not used.

Code sharing is clumsy Since `RET` may only be used to return from the textually current subroutine or function, using a portion of code as part of two or more subroutines or functions is only possible using continuation addresses, which is awkward and inefficient. Therefore, shared code should normally either be encapsulated in a subroutine or inlined.

Data may move Since there is no fixed relationship between the locations of code and data, there is little point storing data in places where it must be branched around simply in order to improve locality. On the other hand, storing data between functions or after unconditional branches may improve locality on machines that store data and code together,[2] and be harmless on those that do not.

[2] If they do not have separate instruction and data caches