Technical Report

UCAM-CL-TR-529
ISSN 1476-2986

Number 529

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

The triVM intermediate language reference manual

Neil Johnson

February 2002

This research was sponsored by a grant from ARM Limited.

JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

Series editor: Markus Kuhn

ISSN 1476-2986

Contents

Chapter 1

Introduction

This technical reference manual is the definitive description of the *tri*VM intermediate language. It is written for people interested in *tri*VM, and intended primarily for code compaction research.

1.1 Background

Program space compaction has become an important and active area of research, primarily in the field of embedded systems. A typical deeply-embedded system will consist of a compact code processor (e.g. ARM Thumb [7]), limited runtime storage (RAM) and limited code storage (EPROM, Flash, etc.), together with I/O interfaces specified for a particular application. In the majority of cases, such systems are mass produced in large volumes. Clearly, at these quantities material costs account for a major part of the total product design cost. As the size of ROM required to store the program is proportional to the size of the program, a worthwhile cost reduction can be achieved through reducing the size of the program code.

The *tri*VM intermediate language has been developed as part of a research programme concentrating on code space optimization. It is intended to be an experimental platform to support investigations into code analyses and transformations, providing a high-degree of flexibility and capability.

1.2 Aims

The primary aim in developing the *tri*VM intermediate language is to provide a language that removes the complexity of high-level languages, such as C or ML, while maintaining sufficient detail, at as simple a level as possible, to support research and experimentation into code size optimization.

A secondary aim is to develop an intermediate language that supports graph-based translation systems, using graph rewrite rules, in a textual, human-readable format. Experience has shown that text-format intermediate files are much easier to use for experimentation, while the penalty in translating this human-readable form to the more compact binary data structure format used by the software is negligible.

Another secondary aim is to provide a flexible language in which features and innovations can be evaluated. For example, this is one of the first intermediate languages (as opposed to data structures) to directly support the Static Single Assignment property (see Chapter 4). Another feature is the exposing of condition codes as one of the results obtained from arithmetic operations—an example is given in Chapter 7.

The basic structure of *tri*VM is a notional three-address machine, wherein the majority of arithmetic instructions take three registers—two supply the arguments and the result is placed in the third. This instruction format is somewhat similar to that of the ARM Thumb [7].

1.3 Related Work

While this paper is concerned solely with the description of the *tri*VM intermediate language, we present a brief summary of research on other intermediate languages. Broadly speaking, the area can be split into two sub-areas: *typed* intermediate languages, primarily used in functional language compilers (ML, Haskell, etc), and *untyped* intermediate languages which are primarily aimed at the translation and optimization of imperative languages (C, Pascal, etc).

Starting with the former, one of the most widely known works is Tarditi *et al*'s Typed Intermediate Language *TIL* [14], which paved the way for the development of Morrisett's Typed Assembly Language *TAL* [11], which is really another form of typed intermediate language, albeit based on a generic RISC-like architecture. *tri*VM initially looks similar to TAL in syntax and mnemonics. However, TAL has additional operations for allocating blocks of memory on the heap and type packing and unpacking. Target code generators have been developed for the Intel x86 architecture [6]. The main focus of this work has been to provide a language and tools to automatically produce and certify code for safety before being executed.

Another typed intermediate language is Henk [8]. This language is very closely tied to the Glasgow Haskell Compiler (GHC) [5], the original aim of the authors to incorporate Henk in the GHC. Henk's four main benefits are: it is directly based on typed lambda calculi, it is a small language, it uses a single syntax for terms, types and kinds, and it has an explicit concrete syntax. Again, the main focus of attention in on type-safety and type-checking programs as a means of producing correct programs.

One final example of typed intermediate languages is λ^{CIL} [15]. Again, this research is focused on compiling functional, polymorphically-typed languages, such as ML, but now with a bias towards optimization through flow-directed compilation.

In all these three instances, the goal has been compiling functional languages in a type-safe environment. The second, and arguably more relevant, selection of intermediate languages are focused on traditional imperative language models.

At IBM, the TOBEY compiler utilizes two intermediate languages: XIL and YIL [12]. The original intermediate language, XIL, was developed to facilitate the production of highly optimal scalar code. To support higher-level transformations, XIL was later extended to form YIL, whose primary function was to provide an abstraction layer above XIL to support optimizations of nests of loops in the presence of caches.

The Sun Microsystems' Clarity C++ Compiler was built on the MCode machine-independent intermediate language [10]. The significant difference with this intermediate language is that it was designed to be compiled into native code at runtime, the main advantages being that MCode can more compactly represent a program, and a runtime code generator can be finely tuned for each platform that it is running on.

Finally, one of the main works in this field is the Stanford SUIF compiler suite [4]. Here, the intermediate representation is constructed from an object-oriented extensible intermediate representation, encapsulating considerable high-level program details (e.g loop structures, array indices, field accesses). The overall structure of the system is of a front-end compiler, a number of plug-in modules that apply transformations to the intermediate representation, and a back-end target code generator. This structure is similar to *tri*VM; however, the intermediate file format is binary rather than the more easily readable textual format of *tri*VM. Also, the basic premise appears to be program representation supporting block-based control-flow analysis, rather than the combined control/data-flow information present in *tri*VM intermediate code.

1.4 Context

The complete *tri*VM compilation flow is illustrated in Figure 1.1. This paper is concerned with the definition of the *tri*VM language, which sits within the *tri*VM virtual environment.

1.5 Structure of this report

The remainder of this report is structured as follows. Chapter 2 describes the *tri*VM virtual machine, covering the machine registers, the memory hierarchy, addressing modes and the procedure calling mechanism. Chapter 3 then describes the number formats interpreted by the virtual machine and how they are stored in memory. Chapter 4 looks at the implementation of static single assignment in *tri*VM, together with a simple example. Chapter 5 details the non-executable *tri*VM directives necessary to define the overall structure of a *tri*VM program, while chapter 6 describes each *tri*VM instruction in detail. Finally, chapter 7 illustrates a number of common program structures, in both C and *tri*VM form, as an aid to helping the reader understand the application of the *tri*VM intermediate language.

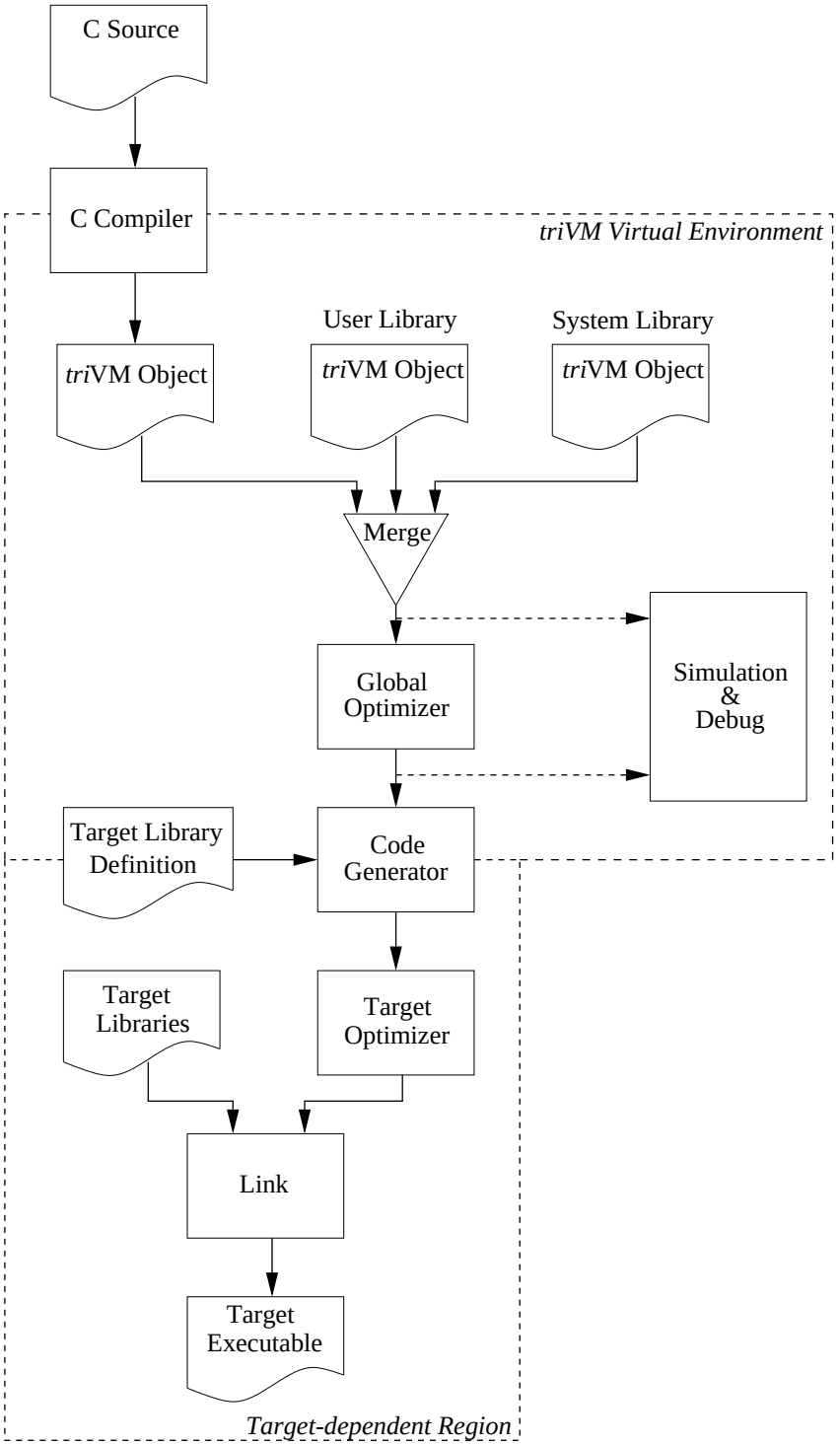


Figure 1.1: The *tri*VM compilation flow. Whole-program linking is performed at the *tri*VM source level, prior to optimization and target-dependent resource allocation. Target linking is performed as the final step to producing the executable.

Chapter 2

The *tri*VM Virtual Machine

The *tri*VM virtual machine defines the behaviour of *tri*VM programs. It specifies the width of the virtual registers, the operation of condition flags, the structure of the memory hierarchy, and the procedure calling mechanism.

This chapter describes each of these in detail, including examples where they help illustrate a particular point.

2.1 Virtual Registers

An *instance* of a *tri*VM procedure describes the dynamic existence of a procedure as opposed to its static existence: there is only one static body of a procedure's instructions, while there can be many dynamic instances of a procedure.

A procedure instance includes not only the body of code that forms the procedure, but also the set of registers that describes that particular instance of the procedure. One can draw a comparison between a *tri*VM procedure instance and the *activation record* [1] of traditional compilers, in that they both capture the dynamic behaviour of a procedure. The *tri*VM instance abstracts away the notion of a stack frame, replacing it with a set of virtual registers that are unique to each instance of a given procedure.

Each instance of a register within an instance of a procedure is unique — register `r4` in procedure P is distinct from register `r4` in procedure P', even if both procedures are instances of the same procedure body.

All of the *tri*VM registers adhere to the single-assignment property of SSA form (Chapter 4). Each register is assigned by only one instruction (the *definition node*) and used one or more times (the *use nodes*).

2.2 Condition Flags

The *tri*VM virtual machine consists of memory (discussed in the following section) and a virtual processor that executes the virtual instructions of the program.

Information within the virtual processor is of two types: numeric values and condition flags. Either type of data may be stored in a virtual register. The difference between the two types is defined by the meaning of the information: numeric values are the values directly determined and interpreted by the intention of the programmer (e.g. the sum of two integers), whereas the condition flags are solely for the purposes of the virtual processor in determining the status of the result of an arithmetic operation with respect to zero.

The associated bit patterns for numeric values can be directly related to information relevant to the application (e.g. the ASCII value for the character 'A'). The bit pattern meaning for the condition flags however, has no meaning to any software other than to the virtual processor. All we can say is that for a given condition the relevant conditional instructions will behave as expected; *how* this behaviour is achieved is not important.

All the arithmetic and logical operations generate numeric and, optionally, condition flag values. Where the status of the result is not required the condition code register can be omitted.

For integer comparisons, *exactly* one of either `blt` or `bge` (or `bbl` or `bae` for unsigned comparisons) will be taken, while for floating-point comparisons *at most* one of either `blt` or `bge` will be taken[1].

For example, the follow instruction calculates the sum of registers `r2` and `r3`, storing the numeric result in register `r4`

```
add r4, r2, r3
```

Whereas the following instruction repeats the same operation but in addition stores the status of the result in register `r5` (e.g. whether the result was zero, positive, etc)

```
add (r4, r5), r2, r3
```

We can then use the contents of `r5` to determine the properties of the numeric result with respect to zero (i.e. if it was equal or not equal to zero, greater than zero, or less than zero)[2]. This compound instruction is semantically equivalent to computing the numerical result, then comparing that result with zero and storing the condition codes in the second result register. Figure 2.1 illustrates two semantically equivalent instruction sequences.

```
add   r3, r1, r2
ldi   r4, 0
cmp   r5, r3, r4
```

(a)

```
add   (r3, r5), r1, r2
```

(b)

Figure 2.1: Semantically equivalent instruction sequences. In (a) we show the original, explicit instruction sequence. However, we do not make any use of the condition codes implicitly generated by the `add` instruction. In (b) we explicitly identify these condition codes, saving two instructions.

[1]The unsigned branches `bbl` and `bae` are always untaken for floating-point comparisons.

[2]For floating-point operations, a result that generates a non-number (e.g. NaN) will generate a condition code that causes no conditional branch to take place.

Condition	Branch	Alternative Interpretation
==	`beq`	zero
! =	`bne`	non-zero
<	`blt`	negative (less than zero)
≥	`bge`	positive (greater than or equal to zero)
< (unsigned)	`bbl`	
≥ (unsigned)	`bae`	

Figure 2.2: The six main condition codes in the *tri*VM machine. The bottom two tests (`bbl` and `bae`) only apply to unsigned integer comparisons of two numbers; the other four apply to both signed integer and floating point operations.

2.2.1 Status Information

The contents of the condition register is opaque to user code. This accomplishes two things: firstly, it forces the programmer to not rely on any particular encoding of the condition flags, thus providing a high degree of cross-platform portability [3], and secondly, the only instructions that use the contents of condition code registers are the conditional branch instructions. This tight coupling between condition code registers and conditional instructions helps separate the two distinct types of information within a *tri*VM program, which will aid in subsequent target code resource allocation.

There are six properties of a numeric result that can be tested, given in Table 2.2.

As a means of comparison, Table 2.3 summarises the condition code flags for four popular microprocessor architectures.

Condition	ARM	x86	SPARC v9	68000
== ! =	Z $\overline{Z}$	Z $\overline{Z}$	Z $\overline{Z}$	Z $\overline{Z}$
Signed < ≥	 $N! = V$ $N = V$	 $N! = V$ $N = V$	 $N! = V$ $N = V$	 $N! = V$ $N = V$
Unsigned < ≥	 $\overline{C}$ C	 C $\overline{C}$	 C $\overline{C}$	 C $\overline{C}$

Figure 2.3: Condition codes for four popular microprocessors. The four flags are: *C*arry, o*V*erflow, *N*egative and *Z*ero. Note that the interpretation of the Carry flag in the ARM processor is the opposite of the other three processors. This is due to the ARM treating the Carry flag as an *inverted-borrow* flag.

[3] Not all processors implement the same flags, or indeed interpret them in the same way.

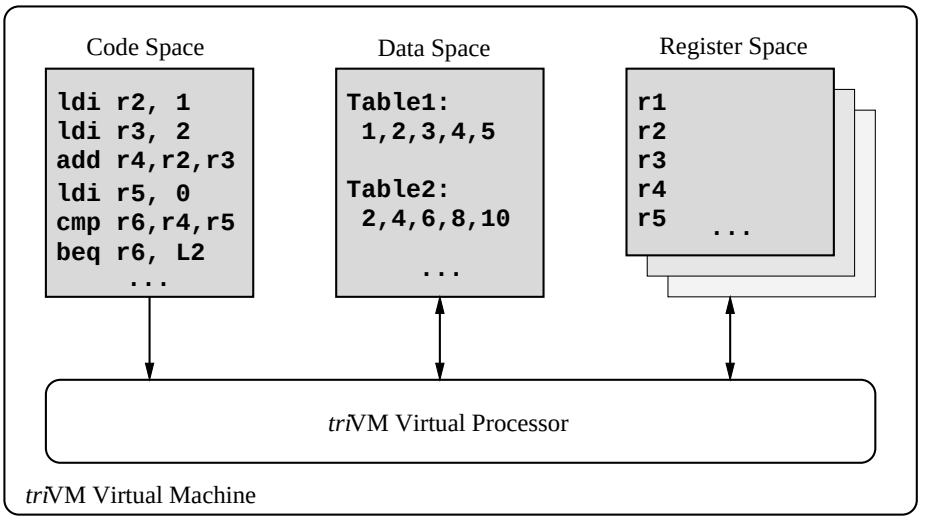


Figure 2.4: The *tri*VM virtual machine components. Note the code space is considered read-only, while both data- and register-spaces are readable and modifiable. Each dynamic instance of a procedure has its own register set, indicated by the stack of register blocks.

2.3 Memory Hierarchy

The memory of the *tri*VM virtual machine is based on the Harvard architecture — both code ("X-space") and data ("M-space") are placed in separate memory spaces. In addition, registers occupy a third region ("R-space"). Figure 2.4 illustrates the three main memory units and their relation to the virtual processor.

The data memory directives (`data` and `const`) split the data memory into two further subcategories. The `data` directive specifies modifiable memory that would typically be held in RAM, while `const` specifies immutable memory such as might be held in ROM.

Data is stored in either R-space or M-space. All local automatic variables (i.e. those declared within a procedure block without static storage) whose address is not computed are located in R-space. Global, static and data objects whose address is taken are stored in M-space.

Some data memory has the special property of being *volatile* — accessing that memory may change some other aspect of the system, and conversely the contents of that memory may change due to some (external) influence. An example of this behaviour is a memory-mapped timer: its count increases with every tick of the timer, and the action of reading the timer resets the count. To interface the *tri*VM virtual machine to this special type of memory a refined set of memory access instructions are employed — `vld` loads from a volatile pointer[4] and `vst` stores through a volatile pointer.

Traditional processor architectures use the notion of a stack to locate local automatic variables. Each instantiation of a local variable is unique to each call

[4]We define a *volatile pointer* as a memory address pointing to a volatile memory cell.

of the procedure, and is typically implemented as pre-computed storage within the activation record of the procedure. The same process is used for thread-local storage, where each thread has its own stack, on which the activation records for the procedures in that thread are stored.

This mechanism is supported in *tri*VM by the scoping rules of the `data` directive. The two locations of the `data` directive specify the storage locality of the data—at module-scope the data is static (and may also be global), while at procedure-scope the data is automatic. Note that procedure-scope automatic data may not have global visibility.

2.4 Addressing Modes

The *tri*VM virtual machine has three addressing modes: register, immediate and register indirect, defined below.

Register

Both source and destination operands are located in R-space. All computation operations are register-addressing only. For example,

```
add  r12, r3, r9     ; r12 = r3 + r9
```

Immediate

Immediate addressing implements the loading of immediate data to R-space—the data for the operation is encoded within the instruction itself. It is used exclusively for loading compile-time constants into registers. For example,

```
ldi  r4, 12          ; r4 = 12
```

Register Indirect

The address of the source or target, in M-space or X-space, is stored in a register. Six instructions allow for indirectly accessing M-space to implement indirection, and two instructions allow for computed branches and calls. For example,

```
bra  [r23]           ; branch to the address stored in r23
```

2.5 Procedure Calling Mechanism

Procedures split a large program up into a number of smaller parts, joined together through procedure calls and returns. The *tri*VM instructions `call` and `ret` provide this facility, allowing a specified number of values to be passed back and forth between procedures.

The `call` instruction uses register-indirect addressing to refer to the procedure to be called (the *callee*). The arguments are supplied in registers, and any return values are assigned to a matching number of result registers. The `ret` instruction returns control to the calling procedure (the *caller*) with the return values supplied in registers. The example in Figure 2.5 illustrates all of these points.

```
proc foo (0), 1
    ldi   r1, bar
    ldi   r2, 'a'
    ldi   r3, 5
    call  [r1](r2, r3), r4
    ret   r4
end
```

(a)

```
proc bar (2), 1
    add   r3, r1, r2
    ret   r3
end
```

(b)

Figure 2.5: Procedure ***foo*** (a) calls procedure ***bar*** with two arguments and expects one return value, stored in `r4`, which is returned to ***foo***'s caller. Procedure ***bar*** (b) takes two arguments, which are initially assigned to registers `r1` and `r2`, and returns their sum (from `r3`).

There are no restrictions on the number of arguments or return values a procedure can handle. The precise details of the `call` instruction are to be found in section 6.12. However, the number of arguments supplied and return values requested must match the numbers given in the procedure definition; any mismatch is a syntax error.

All procedures that terminate must do so through the `ret` instruction—it is not possible to "fall-through" to the next procedure. This is partly due to the interpretation of the phrase "next procedure". Certainly, in a target architecture it is possible to fall-through to the next procedure. However, in the virtual context of *tri*VM there is no such concept of a next procedure. Indeed, because there is no locating done on *tri*VM code it is undetermined which procedure, if any, follows any given procedure. Given the wide variety of memory architectures it would be an artificial constraint to provide a mechanism to allow fall-through.

The effect of fall-through can be easily obtained through a chain-call: the last action of a procedure (prior to the terminating `ret`) is to call the next "fall-through" procedure. Subsequent target-specific optimization may eliminate the now-redundant `ret`, and may even translate the `call` into a fall-through if it can guarantee that the fall-through procedure will be placed *immediately* after the procedure.

2.5.1 Variadic Functions

Variadic functions are functions that take (a) a fixed number of defined arguments, and (b) a set of zero or more optional arguments. The hard constraint in *tri*VM on the enumeration of the procedure arguments precludes the direct implementation of variadic functions.

A proposed solution to the implementation of variadic functions is as follows. The callee procedure is defined as taking the required number of fixed arguments and one additional argument: a pointer to an array in which the additional arguments are placed by the caller prior to the call. The callee can thus maintain an index into the array through which the additional arguments can be accessed[5].

[5]This mechanism is similar to the indirect method used by some compilers for passing structures to functions: a temporary memory block is generated for the structure, the structure is copied into this block, a pointer to the structure is passed to the callee, and the compiler

automatically generates an additional level of indirection to "hide" the real location of the structure. After the call the temporary block is discarded. For returning structures a similar mechanism is employed, except the contents of the temporary block is copied into the target structure before it is freed.

Chapter 3

Number Formats and Data Storage

The previous chapter introduced the *tri*VM virtual machine from an execution perspective. In this chapter we describe the format of numbers and their storage in the memory space, M-Space. We look at both integer and floating point number representations, how they are laid out in the byte-aligned memory, and illustrate data accesses using the M-space interfacing instructions.

3.1 Integer Numbers

The *tri*VM machine is a little-endian machine, whereby the lowest byte of a multi-byte value is stored at the lowest address. For example, for a two-byte integer `0x1234`, the lowest byte (`0x34`) is stored at the first byte in memory, and the highest byte (`0x12`) is stored at the next byte in memory.

Little-endian behaviour has several advantages over its opposite (big-endian) when considering numeric compatibility between different-sized number formats. Consider a half-word integer of value `0x0012`. Clearly, if we read this as a single byte value, we maintain the numeric value of the original number.

The three *tri*VM integer number formats are: byte, half-word and word. Figure 3.1 shows how these three data formats are aligned with respect to each other.

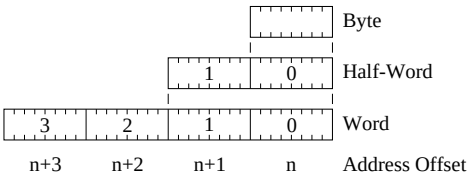


Figure 3.1: The three integer data formats in the *tri*VM virtual machine. The numbers represent the byte-offset within the multi-byte formats. For example, for the word format, the lowest byte (the right-hand-most byte) has an offset of 0, while the highest byte has an offset of 3.

3.2 Floating Point Numbers

The previous section described the integer number formats, where each bit has a direct mapping to the numeric value. In contrast, floating point numbers uses a compact multi-format representation, where several fields of different meaning fit into the single word-length format.

The *tri*VM virtual machine uses the IEEE-754 standard format for 32-bit single-precision floating point numbers [3]. The real numeric value is stored in three fields: sign (1 bit), exponent (8 bits) and significand (23 bits), shown in Figure 3.2.

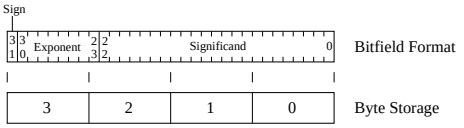


Figure 3.2: The IEEE-754 compatible single-precision floating point data format of *tri*VM.

The three fields of the data format are interpreted as follows. Starting from the left, the sign bit is 0 for positive values and 1 for negative values. The exponent is stored as an unsigned integer value with a bias of 127. This limits the range of the exponent to $[-127 \ldots 128]$[1]. Finally, the significand is an unsigned 24-bit value, covering the range $[1.0 \ldots 2.0)$. To save storage the most significant '1' is hidden, and is not stored in memory.

Floating point numbers are stored in memory as if they were 4-byte integer values. The first byte contains the first eight bits of the significand, the second byte contains bits 9 to 15, the third byte contains the remaining bits of the significand and the first bit of the exponent, while the last byte contains the remaining seven bits of the exponent and the sign bit.

3.3 Accessing Multi-byte Data

All three data sizes (byte, half-word and word) are accessed through size-specific load/store instructions. For example, in the case of byte-sized data, one would use the `ldb` and `stb` instructions to load and store bytes respectively. The load instruction also performs a sign-extend, replicating the topmost bit of the source byte into the rest of the target register. If a zero-extended load is required then the topmost bits of the register must be explicitly cleared. In the case of a word load no sign-extend is performed.

Since floating-point values are the same size as word integers, the word load and store operations are also used for floating-point loads and stores.

There is no requirement to ensure that multi-byte values are aligned to some boundary, as is often the case in target processors.

The following programs (Figure 3.3) illustrate the use of the load/store instructions.

[1] We use '[' or ']' to denote inclusive range, and '(' or ')' to denote exclusive range.

```
data giVar[4]

proc foo:
    ...
    ldi  r1, 1
    ldi  r2, giVar
    stb  [r2],r1
    ldw  r3, [r2]
    ...
end
```

(a)

```
data giVar[4]

proc bar:
    ...
    ldi  r1, 1
    ldi  r2, giVar
    add  r3, r2, r1
    ldh  r4, [r3]
...
end
```

(b)

```
const Pi = 3.141593

proc area (1), 1
    ldi  r2, Pi
    ldw  r3, [r2]
    fmul r4, r1, r1
    fmul r5, r4, r3
    ret  r5
end
```

(c)

Figure 3.3: In (a) we store a byte in the first element of global integer variable `giVar`, then load the entire contents of `giVar` into register `r3`. Example (b) illustrates an unaligned half-word load from the middle of a word variable. Finally (c) shows how to calculate the circumference of a circle, with the radius in `r1` and the constant `Pi` held in data ROM, as defined by the `const` directive.

Chapter 4

Single Assignment Behaviour

Static Single Assignment (SSA) [2] is a representation of a program such that for each variable there is only one assignment statement. A program in this form has many nice features, especially concerning data-flow-based analysis and optimization. Typically, an existing program not in SSA form is translated into SSA form prior to optimization.

4.1 Conversion into SSA Form

Converting non-SSA form programs into SSA form generally consists of two steps: (i) renaming all variables to impose the static single assignment property, and (ii) inserting ϕ-functionswhere appropriate to merge two or more reaching definitions of a variable at a merge point within the control-flow graph.

4.2 Variable Renaming

Renaming variables is mostly a matter of appending a subscript of some form (typically a numeric integer) to a variable name, replacing each instance of variable access with the name of the reaching definition of that variable, and incrementing the subscript on every new definition of that variable. Figure 4.1 shows how a basic block of statements can be converted into SSA form.

The renaming approach is sufficient for simple scalar variables but does not apply to array variables. The approach generally taken for arrays is to apply two pseudo-operations *Access* and *Update*. The simpler of the two, *Access*, takes both the name of the array variable and the index into the array, returning the contents of the indexed array cell. The second operator, *Update*, takes the names of the array variable, the index into the array, and the new value for the indexed array cell, and returns a new variable whose contents is that of the old array variable but includes the change to the indexed cell. Figure 4.2 illustrates this.

1:	$x = a + b;$	$\mapsto$	$x_1 = a_1 + b_1;$
2:	$y = c + d;$		$y_1 = c_1 + d_1;$
3:	$x = e + y;$		$x_2 = e_1 + y_1;$
4:	$z = f(x);$		$z_1 = f(x_2);$
	(a) Original		(b) SSA Form

Figure 4.1: SSA conversion for a basic block. The subscripts identify each new variable name, while keeping the name of the original variable for exposition. For example, consider variable x in (a) which has two assignments (lines 1 and 3). In (b) x is split into two distinct variables x_1 and x_2 in order to impose the single assignment property.

1:	$x = A[5];$	$\mapsto$	$x_1 = Access(A_1, 5);$
2:	$x = x + 2;$		$x_2 = x_1 + 2;$
3:	$A[5] = x;$		$A_2 = Update(A_1, 5, x_2);$
4:	$y = A[6];$		$y_1 = Access(A_2, 6);$
	(a) Original		(b) SSA Form

Figure 4.2: SSA conversion for arrays. The pseudo-operation *Update* modifies the array, which is treated as a single-assignment entity. A new copy of the array is thus created on every *Update* operation.

4.3 Control Flow and Phi-Functions

Implementing single assignment in basic blocks is a trivial exercise, as shown above, since at any point in the block there can only be one valid definition for each variable. This is clearly shown in Figure 4.1 at line 3: for the new definition of x there is only one reaching definition of y, from line 2. Thus we use the current definition of y in the expression.

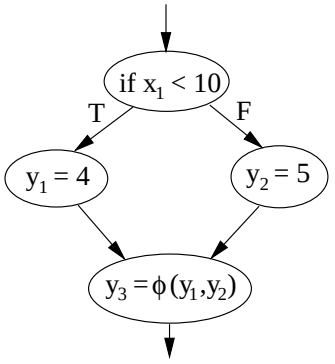
Where there are several possible paths of execution to a use of a variable we have several potentially distinct definitions of that variable, and we require a mechanism to choose one of these definitions. We cannot just implement multiple stores to a new variable as this would violate the single assignment rule of SSA form. The solution to this problem is the ϕ-function.

The ϕ-function takes as its arguments all the reaching definitions of a variable that reach a merge point in a control-flow graph, and returns the value of whichever definition reaches that point in the program. Generally, if flow reaches the ϕ-function on the first edge, it returns the first argument, if on the second edge then the second argument, and so on. Figure 4.3 graphically illustrates the operation of the ϕ-function in a simple `if...then...else` case.

The method of determining where to place ϕ-functions has been the subject of much research. The interested reader is referred to [2] for an introduction to the subject.

In *tri*VM ϕ-functions are implemented with the `phi` instruction. It is the *only* instruction in the entire instruction set that has the property of multiple assignment. Its purpose is to implement ϕ-functions in a distributed fashion—the ϕ-function is split into a set of `phi` instructions, each of which is placed at the root of each edge ending in the target merge node.

```
if (x < 10)
    y = 4;
else
    y = 5;
```




```
    ; x in r1
    ldi r2, 10
    cmp r3, r1, r2
    blt r3, L1
    ldi r4, 5
    phi r6, r4
    bra L2
L1:
    ldi r5, 4
    phi r6, r5
L2:
    ; y in r6
```

(a) C source (b) Control-flow graph (c) *tri*VM code

Figure 4.3: SSA ϕ-function example. In (a) is the original C source code. The control-flow graph (b) shows the merge node at the bottom. The ϕ-function merges the two reaching definitions of variable y (y_1 and y_2) into y_3. Finally, the *tri*VM `phi` instruction can be seen in the intermediate code (c).

Chapter 5

Intermediate Language Directives

The *tri*VM intermediate language also contains a small number of directives for specifying non-executable content. As for the instruction set, there are three classes of directives: labels, structural directives, and data directives. They are summarised in Table 5.1.

Syntax	**Description**
LabelName:	Defines a label at the specified point in the program. Note that this directive is only valid within procedure bodies.
`module` *ModuleName*	Defines the start of a module, typically a source or library module.
`endmod`	Marks the end of a module.
`proc` *ProcName* `(` N_{in} `),` N_{out}	Identifies the start of procedure *ProcName* and specifies the number of argument registers N_{in} and result registers N_{out}.
`end`	Marks the end of a procedure.
`data` *DataName*$\langle$ `[` n `]` $\rangle\langle$ `=` *initlist*$\rangle$	Defines static storage for one or n bytes of variable memory, with optional initializer list (where $\lvert initlist \rvert \leq n$).
`const` *ConstName* `=` *initlist*	Defines constant byte-wide data and the values to be stored at that location.

Table 5.1: The *tri*VM intermediate language directives. $\langle \ldots \rangle$ denotes optional elements.

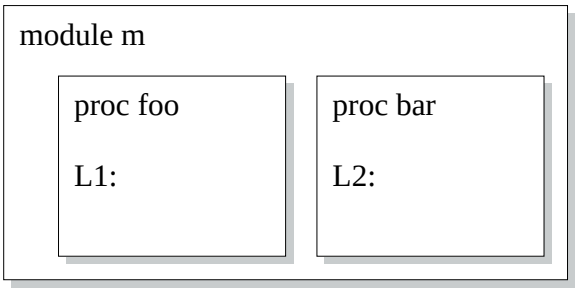


Figure 5.1: A simple illustration of the *tri*VM scoping rules. Label `L1` is only visible within procedure ***foo***. Likewise, `L2` is only visible within procedure ***bar***. Because both ***foo*** and ***bar*** are defined in the same module both names are visible to each other (***foo*** can call ***bar*** and *vice versa*).

5.1 Identifiers

There are two classes of identifiers in *tri*VM: scope identifiers and branch target identifiers. Module and procedure names explicitly implement the *tri*VM scoping rules. Procedure names also identify the targets for `call`s. Labels identify the target points for intra-procedural branches and conditional branches. Figure 5.1 indicates the scoping rules for identifiers.

Labels are the targets of branch instructions, and as such are only valid within the confines of a procedure body. Label names may use the characters a-z, A-Z, 0-9, _ (underscore), . (period), $, !, / (forward slash) and &. Label names are, as for all names in *tri*VM, case-sensitive (e.g. `LABEL` and `label` are distinct). Label and procedure names must start with a non-numeric character.

5.2 Structural Directives

Modules in *tri*VM use a two-level hierarchy to support module-level scoping required by high-level languages such as C[1]. The module level is supported by the `module` directive, which optionally names the module. If a name is not provided, the source file name will be used.

The procedure level is supported by the `proc` directive, which identifies the procedure by name and indicates the *single* entry point for that procedure. It also specified the number of argument and result registers for that procedure. This information is used during analysis to (a) allocate the first N_{in} registers to procedure arguments, and (b) check that each `ret` instruction returns exactly N_{out} result registers.

The boundary between the two levels can only be crossed from inner to outer. The normal scope for procedure names is within the module in which they are declared; a $-prefix places the procedure name within the global name-space, allowing calls to be made from procedures in other modules.

[1]The Pascal nested-procedure hierarchy is not directly supported, and will require some extra effort of the part of the high-level language compiler, e.g. name-mangling.

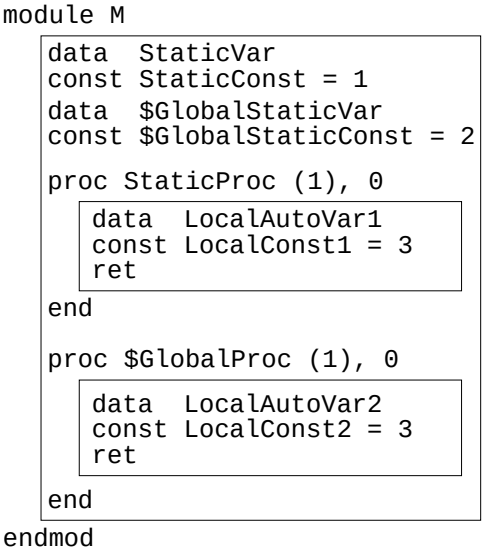


Figure 5.2: Complete example of *tri*VM scoping rules. The boxes highlight the scoping regions.

5.3 Data Directives

The two data directives relate to the two types of data that are available to the system—variable data (or local data whose address is evaluated) and constant data.

The scoping rules are similar to those for procedure names with one exception. Data and constant names with a $-prefix are placed in the global name-space, *except* for `data` directives placed within a procedure body which are always local to that procedure, and are unique to each instance of that procedure. For `const` directives placed within a procedure body there is only one instance of the constant data, but its visibility is restricted to that procedure.

5.4 Complete Scoping Example

To illustrate the above scoping rules in action, Figure 5.2 shows all the scoping rules in a hypothetical example. The variable, constant and procedure names illustrate both the use of the $-prefix, and describe their visibility within the context of the whole program.

5.5 Constants

As well as labels and procedure names, there are four other types of constants in *tri*VM assembly language: integer constants, real constants, characters and strings.

5.5.1 Labels and References

All visible labels and procedure names may be used as constants. Within a procedure this includes: all local labels, the procedure's own name, all constant, data and procedure names within the parent module, and all global constant, data and procedure names.

5.5.2 Integer Constants

All integer constants are processed as signed numbers. The maximum range of integers is defined by the width of the target processor. They can be written in decimal or hexadecimal notation (beginning with '0x' or '0X'). All integer constants are stored as 32-bit signed numbers. Large unsigned constants (e.g. `0xDEADBEEF`) are written as is, and will be converted into the correct bit pattern.

Additional suffixes are added to the number to indicate the size of constant integers for `const` declarations. The suffix begins with a ':' and one of 'B', 'H' or 'W' for byte, half-word and word sizes respectively. All immediate loads via `ldi` are treated as word-loads, irrespective of the actual width of the constant argument: numbers without a unary minus are zero-extended, while those with a leading minus sign are sign-extended. For example, to load the above large constant into a register, we would write:

```
ldi  r12, 0xDEADBEEF
```

whereas to load the signed number `-4` into a register would be written as:

```
ldi  r13, -4
```

5.5.3 Real Constants

Real constants are written with a decimal point and optional exponent, for example `1.23e12` to represent 1.23×10^{12}. Valid real constants lie in the range 1.175494×10^{-38} to $3.402823 \times 10^{+38}$.

5.5.4 Character Constants

All character constants are written as a single character, or standard C escape sequence [9], enclosed between single quotes.

5.5.5 String Literals

Strings are enclosed between double quotes, and consist of one or more characters or escape codes. Strings in *tri*VM are *not* null-terminated, as in C. To null-terminate a string append a null value, either as a separate constant, or with the escape code `'\0'`.

5.5.6 Examples

Some examples of the four constants are shown in Figure 5.3 below:

```
 12        ; signed integer word constant
-12        ; another signed integer word constant

 12:B      ; byte-wide integer constant
 384:H     ; half-word-wide integer constant

 1.23e2    ; real constant, equivalent to 123.0

 'c'       ; character constant
 '\x41'    ; another character constant, ASCII character 'A'

 "hello"   ; string literal, consisting of five characters.
```

Figure 5.3: Examples of *tri*VM constants.

Chapter 6

The Virtual Instruction Set

The *tri*VM processor is a 3-address machine. The majority of the instructions take three arguments, typically three registers. The procedure `call` and `ret` instructions take a variable number of register arguments.

The *tri*VM instruction set can be categorized into three main classes:

Load/store — for transferring data between R-space and M-space.

Computation — for performing calculations on data in R-space.

Flow control — for transferring control to a different part of the program.

In this section we describe each *tri*VM instruction in alphabetical order. For each instruction we describe:

Syntax of the instruction

Operation of the instruction in pseudo-code

Description of the instruction

Notes additional information, usage, caveats, etc

Example to illustrate the use of the instruction

6.1 Notation

The following symbols are used in this chapter are shown in Table 6.1.

All the numeric operations can optionally return a condition code, generated by the *condition(operation)* expression. This is equivalent to comparing the result of the *operation* expression with zero.

For example, consider the `add` instruction with two source registers whose contents are 2 and 3. The numeric result of this operation is 5. The condition code result is made by comparing 5 with 0, producing a code that represents *not-equal*, *greater-than-or-equal* and *above-or-equal*.

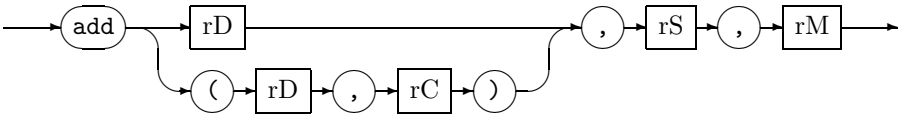
The *fcondition* provides the equivalent functionality for floating-point operations.

Symbol	Meaning
rA	Argument register
rC	Comparand register or Condition Code register
rD	Destination register
rM	Modifier register
rR	Result register
rS	Source register
nConst	Numeric constant ‘Const’
M[ ]:B	Byte-wide memory contents
M[ ]:H	Half-word-wide memory contents
M[ ]:W	Word-wide memory contents

Figure 6.1: Symbols used in the instruction descriptions.

6.2 `add` — integer add

6.2.1 Syntax



6.2.2 Operation

$rD := rS + rM$
$rC := condition(rS + rM)$

6.2.3 Description

The `add` instruction adds together the contents of `rS` and `rM`, placing the result in `rD`. Optionally, the condition codes of the result can be placed in `rC`.

6.2.4 Notes

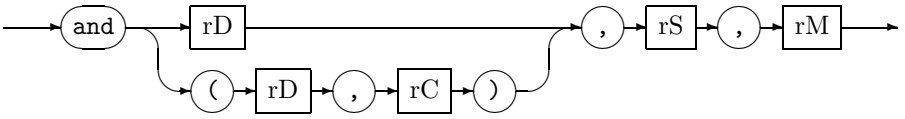
This instructions applied to both signed and unsigned values.

6.2.5 Example

```
ldi    r1, 1
ldi    r2, 2
add    r3, r1, r2         ; r3 = r1 + r2 = 3
add    (r4, r5), r1, r2   ; r4 =    ...
                          ; r5 = condition(r1 + r2)
```

6.3 and — bitwise AND

6.3.1 Syntax



6.3.2 Operation

$rD := rS \wedge rM$
$rC := condition(rS \wedge rM)$

6.3.3 Description

The `and` instruction bitwise ANDs together the contents of `rS` and `rM`, placing the result in `rD`. Optionally, the condition codes of the result can be placed in `rC`.

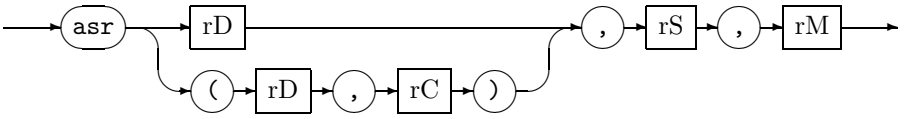
6.3.4 Notes

6.3.5 Example

```
ldi    r1, 1
ldi    r2, 2
and    r3, r1, r2        ; r3 = r1 AND r2 = 0
```

6.4 asr — arithmetic shift right

6.4.1 Syntax



6.4.2 Operation

$rD := rS >> rM$
$rC := condition(rS >> rM)$

6.4.3 Description

The `asr` instruction shifts the contents of `rS` by `rM` places to the right, preserving the sign bit, and placing the result in `rD`. Optionally, the condition codes of the result can be placed in `rC`.

6.4.4 Notes

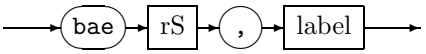
This right-shift operation maintains the sign of the argument, unlike `lsr` which is safe only for unsigned arguments. Shift values less than or equal to zero have no effect on the result — `rS` is copied directly into `rD`.

6.4.5 Example

```
ldi    r1, -12
ldi    r2, 1
asr    r3, r1, r2        ; r3 = r1 >> r2 = -6
```

6.5 `bae` — branch if above-or-equal (unsigned)

6.5.1 Syntax



6.5.2 Operation

if *rS = ABOVE-OR-EQUAL* **then**
    goto *label*
fi

6.5.3 Description

The `bae` instruction branches to the given label if the condition in `rS` is *above-or-equal*.

6.5.4 Notes

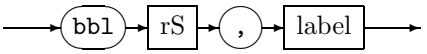
The equivalent signed branch is `bge`.

6.5.5 Example

```
    bae   r2, L
    ...
L:
    ...
```

6.6 `bbl` — branch if below (unsigned)

6.6.1 Syntax



6.6.2 Operation

if $rS = BELOW$ **then**
goto *label*
fi

6.6.3 Description

The `bbl` instruction branches to the given label if the condition in `rS` is *below*.

6.6.4 Notes

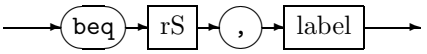
The equivalent signed branch is `blt`.

6.6.5 Example

```
    bbl   r2, L
    ...
L:
    ...
```

6.7 `beq` — branch if equal

6.7.1 Syntax



6.7.2 Operation

if $rS = EQUAL$ **then**
    goto *label*
fi

6.7.3 Description

The `beq` instruction branches to the given label if the condition in `rS` is *equal*.

6.7.4 Notes

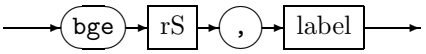
This instruction is applicable to both integer and floating-point comparisons.

6.7.5 Example

```
    beq   r2, L
    ...
L:
    ...
```

6.8 bge — branch if greater-than-or-equal (signed)

6.8.1 Syntax



6.8.2 Operation

if *rS = GREATER-THAN-OR-EQUAL* **then**
goto *label*
fi

6.8.3 Description

The `bge` instruction branches to the given label if the condition in `rS` is *greater-than-or-equal*.

6.8.4 Notes

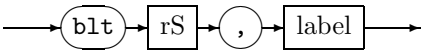
The equivalent unsigned branch is `bae`. This instruction is also applicable to floating-point comparisons.

6.8.5 Example

```
    bge   r2, L
    ...
L:
    ...
```

6.9 `blt` — branch if less-than (signed)

6.9.1 Syntax



6.9.2 Operation

if *rS = LESS-THAN* **then**
goto *label*
fi

6.9.3 Description

The `blt` instruction branches to the given label if the condition in `rS` is *less-than*.

6.9.4 Notes

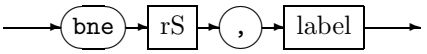
The unsigned equivalent is `bbl`. This instruction is also applicable to floating-point comparisons.

6.9.5 Example

```
    blt   r2, L
    ...
L:
    ...
```

6.10 `bne` — branch if not-equal

6.10.1 Syntax



6.10.2 Operation

if *rS = NOT-EQUAL* **then**
goto *label*
fi

6.10.3 Description

The `bne` instruction branches to the given label if the given condition in `rS` is *not-equal*.

6.10.4 Notes

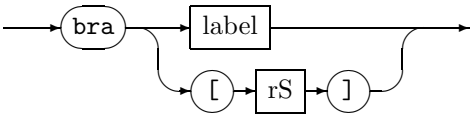
This instruction is applicable to both integer and floating-point comparisons.

6.10.5 Example

```
    bne   r2, L
    ...
L:
    ...
```

6.11 `bra` — branch to label (direct and indirect)

6.11.1 Syntax



6.11.2 Operation

goto *label*
or
goto [*rS*]

6.11.3 Description

The `bra` instruction branches to the label identified either by the given label name, or by the target-dependent label address in `rS`.

6.11.4 Notes

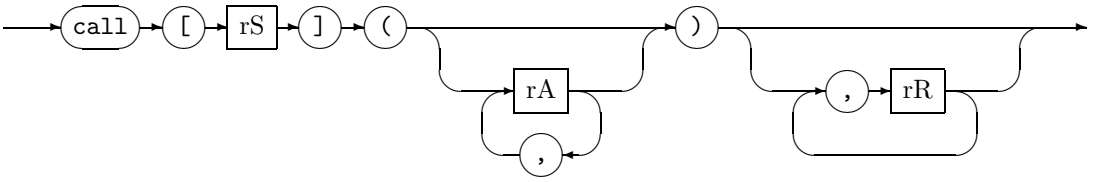
Indirect branch addresses must be direct references — address calculation is not permitted.

6.11.5 Example

```
    ldi   r1, L
    bra   [r1]
    ...
    bra   L        ; the same as above
    ...
L:
    ...
```

6.12 `call` — call a procedure

6.12.1 Syntax



6.12.2 Operation

if rA **then**
$rA \mapsto r_1^{callee}, r_2^{callee}, \ldots, r_N^{callee}$
fi
call rS

6.12.3 Description

The `call` instruction calls the procedure identified by `rS`. Arguments, `rA`, are passed to the callee. On return (see the `ret` instruction) results are assigned to the `rR` registers.

6.12.4 Notes

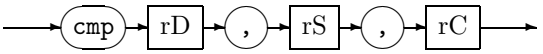
The target address pointed to by `rS` must be the beginning of a procedure identified by a visible procedure identifier.

6.12.5 Example

```
ldi   r1, putchar
ldi   r2, 'a'
call  [r1](r2), r3        ; r3 = putchar('a')
```

6.13 `cmp` — integer compare

6.13.1 Syntax



6.13.2 Operation

$rD := condition(rS - rC)$

6.13.3 Description

The `cmp` instruction compares two integer values in `rS` and `rC`. The resulting condition codes are stored in `rD` for subsequent analysis by a conditional branch instruction.

This instruction is equivalent to subtracting `rC` from `rS` and comparing the result with zero. The numeric result is discarded. Any subsequent conditional branch test is based on the comparison of `rS` with `rC`:

Relation	Branches taken
`rS` < `rC`	`bne blt bbl`
`rS` = `rC`	`beq bge bae`
`rS` > `rC`	`bne bge bae`

6.13.4 Notes

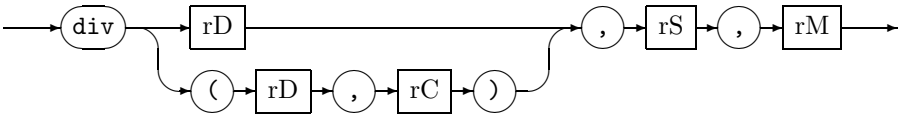
The value stored in `rD` is target-dependent. The only guarantee is that for a given target the result of the comparison will be correctly interpreted by the conditional branch instructions.

6.13.5 Example

```
cmp   r1, r2, r3     ; r1 = comparison of r2 with r3
```

6.14 `div` — integer divide (signed)

6.14.1 Syntax



6.14.2 Operation

$rD := rS \div rM$
$rC := condition(rS \div rM)$

6.14.3 Description

The `div` instruction divides the signed contents of `rS` by `rM`, placing the result in `rD`. Optionally, the condition codes of the result can be placed in `rC`.

6.14.4 Notes

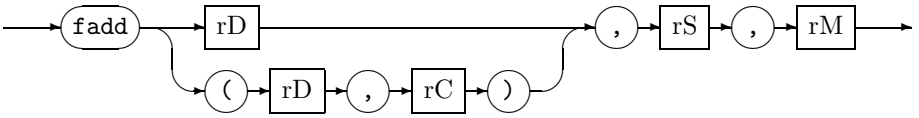
Division by zero is handled in a target-dependent manner.

6.14.5 Example

```
ldi    r1, -4
ldi    r2, 2
div    r3, r1, r2        ; r3 = r1 / r2 = -2
```

6.15 `fadd` — floating-point add

6.15.1 Syntax



6.15.2 Operation

$rD := rS + rM$
$rC := fcondition(rS + rM)$

6.15.3 Description

The `fadd` instruction adds together the floating-point contents of `rS` and `rM`, placing the result in `rD`. Optionally, the condition codes of the result can be placed in `rC`.

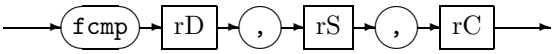
6.15.4 Notes

6.15.5 Example

```
ldi    r1, 1.5
ldi    r2, 2.7
fadd   r3, r1, r2        ; r3 = r1 + r2 = 4.2
```

6.16 `fcmp` — floating-point compare

6.16.1 Syntax



6.16.2 Operation

$rD := fcondition(rS - rC)$

6.16.3 Description

The `fcmp` instruction compares two floating-point values in `rS` and `rC`. The resulting condition codes are stored in `rD` for subsequent analysis by a conditional branch instruction.

This instruction is equivalent to subtracting `rC` from `rS` and comparing the result with zero. The numeric result is discarded. Any subsequent conditional branch test is based on the comparison of `rS` with `rC`:

Relation	Branches taken
`rS` $<$ `rC`	`bne blt`
`rS` $=$ `rC`	`beq bge`
`rS` $>$ `rC`	`bne bge`

Neither `bbl` nor `bae` will be taken on a floating-point comparison result.

6.16.4 Notes

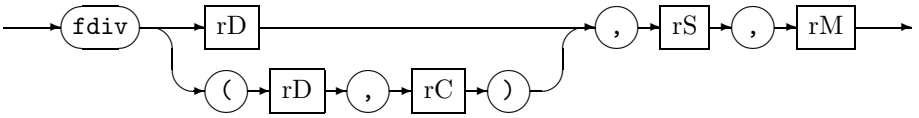
The value stored in `rD` is target-dependent. The only guarantee is that for a given target the result of the comparison will be correctly interpreted by the conditional branch instructions.

6.16.5 Example

```
fcmp  r1, r2, r3     ; r1 = comparison of r2 with r3
```

6.17 `fdiv` — floating-point divide

6.17.1 Syntax



6.17.2 Operation

$rD := rS \div rM$
$rC := fcondition(rS \div rM)$

6.17.3 Description

The `fdiv` instruction divides the floating-point contents of `rS` and `rM`, placing the result in `rD`. Optionally, the condition codes of the result can be placed in `rC`.

6.17.4 Notes

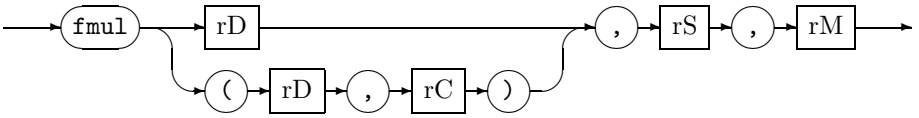
Division by zero is handled in a target-dependent manner.

6.17.5 Example

```
ldi    r1, 1.5
ldi    r2, 2.7
fdiv   r3, r1, r2        ; r3 = r1 / r2 = 0.55555556
```

6.18 `fmul` — floating-point multiply

6.18.1 Syntax



6.18.2 Operation

$rD := rS * rM$
$rC := fcondition(rS * rM)$

6.18.3 Description

The `fmul` instruction multiplies together the floating-point contents of `rS` and `rM`, placing the result in `rD`. Optionally, the condition codes of the result can be placed in `rC`.

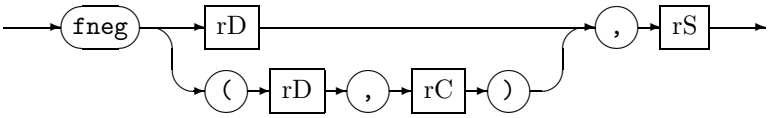
6.18.4 Notes

6.18.5 Example

```
ldi    r1, 1.5
ldi    r2, 2.7
fmul   r3, r1, r2        ; r3 = r1 - r2 = 4.05
```

6.19 `fneg` — floating-point negate

6.19.1 Syntax



6.19.2 Operation

$rD := -rS$
$rC := fcondition(-rS)$

6.19.3 Description

The `fneg` instruction negates the floating-point contents of `rS`, placing the result in `rD`. Optionally, the condition codes of the result can be placed in `rC`.

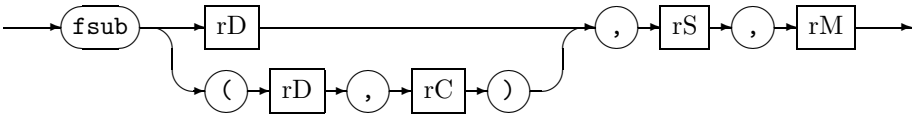
6.19.4 Notes

6.19.5 Example

```
ldi    r1, 1.5
fneg   r2, r1           ; r2 = -r1 = -1.5
```

6.20 `fsub` — floating-point subtract

6.20.1 Syntax



6.20.2 Operation

$rD := rS - rM$
$rC := fcondition(rS - rM)$

6.20.3 Description

The `fsub` instruction subtracts the floating-point contents of `rM` from `rS`, placing the result in `rD`. Optionally, the condition codes of the result can be placed in `rC`.

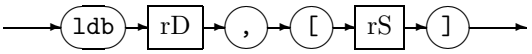
6.20.4 Notes

6.20.5 Example

```
ldi    r1, 1.5
ldi    r2, 2.7
fsub   r3, r1, r2        ; r3 = r1 - r2 = -1.2
```

6.21 `ldb` — load sign-extended byte

6.21.1 Syntax



6.21.2 Operation

$rD := SignExtend(M[rS] : B)$

6.21.3 Description

The `ldb` instruction loads the byte (8-bit) value addressed by the contents of `rS` from memory and sign-extends it to fill `rD`.

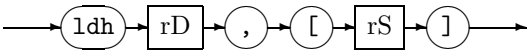
6.21.4 Notes

6.21.5 Example

```
ldi    r1, ByteVar
ldb    r2, [r1]      ; r2 = sign-extended contents of ByteVar
```

6.22 `ldh` — load sign-extended half-word

6.22.1 Syntax



6.22.2 Operation

$rD := SignExtend(M[rS] : H)$

6.22.3 Description

The `ldh` instruction loads the half-word (16-bit) value addressed by the contents of `rS` from memory and sign-extends it to fill `rD`.

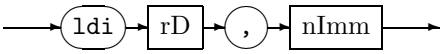
6.22.4 Notes

6.22.5 Example

```
ldi    r1, ShortVar
ldh    r2, [r1]      ; r2 = sign-extended contents of ShortVar
```

6.23 `ldi` — load immediate

6.23.1 Syntax



6.23.2 Operation

$rD := nImm$

6.23.3 Description

The `ldi` instruction loads the immediate constant `nImm` into `rD`. Immediate constants include procedure names and local labels, which are to be computed by a target linker/loader during the later code generation phases.

6.23.4 Notes

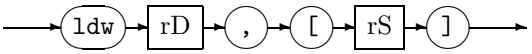
Only procedure names and local labels visible to an `ldi` instruction may be referenced. Any use of a non-visible name is an error and will be trapped by the *tri*VM linker.

6.23.5 Example

```
ldi    r1, 12       ; r1 = 12
```

6.24 `ldw` — load word

6.24.1 Syntax



6.24.2 Operation

$rD := M[rS] : W$

6.24.3 Description

The `ldw` instruction loads the word value addressed by the contents of `rS` from memory into `rD`.

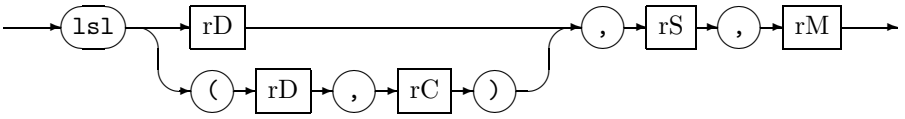
6.24.4 Notes

6.24.5 Example

```
ldi    r1, WordVar
ldw    r2, [r1]      ; r2 = contents of WordVar
```

6.25 `lsl` — logical shift left

6.25.1 Syntax



6.25.2 Operation

$rD := rS << rM$
$rC := condition(rS << rM)$

6.25.3 Description

The `lsl` instruction shifts the contents of `rS` by `rM` places to the left, inserting 0's into the rightmost bits, and placing the result in `rD`. Optionally, the condition codes of the result can be placed in `rC`.

6.25.4 Notes

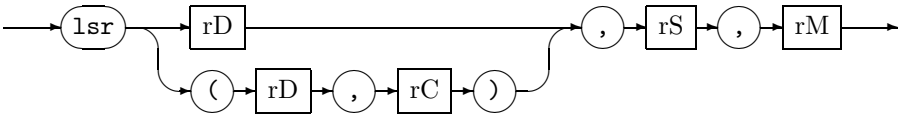
Shift values less than or equal to zero have no effect on the result — `rS` is copied into `rD`.

6.25.5 Example

```
ldi    r1, 3
ldi    r2, 2
lsl    r3, r1, r2        ; r3 = r1 << r2 = 12
```

6.26 `lsr` — logical shift right

6.26.1 Syntax



6.26.2 Operation

$rD := rS >> rM$
$rC := condition(rS >> rM)$

6.26.3 Description

The `lsr` instruction shifts the contents of `rS` by `rM` places to the right, inserting 0's into the leftmost bits, and placing the result in `rD`. Optionally, the condition codes of the result can be placed in `rC`.

6.26.4 Notes

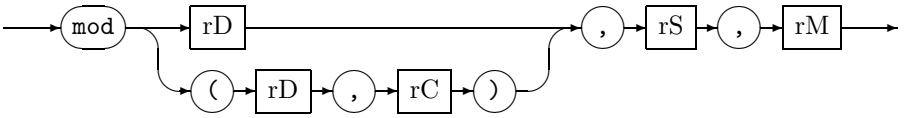
Shift values less than or equal to zero have no effect on the result — `rS` is copied into `rD`.

6.26.5 Example

```
ldi    r1, 12
ldi    r2, 1
lsr    r3, r1, r2        ; r3 = r1 >> r2 = 6
```

6.27 `mod` — integer modulus (signed)

6.27.1 Syntax



6.27.2 Operation

$rD := rS \ mod \ rM$
$rC := condition(rS \ mod \ rM)$

6.27.3 Description

The `mod` instruction calculates the remainder from the signed division of `rS` by `rM`, placing the result in `rD`. Optionally, the condition codes of the result can be placed in `rC`.

6.27.4 Notes

The absolute value of the result of the `mod` instruction is guaranteed only to be smaller than the absolute value of the divisor. It always holds that

$$rS \ mod \ rM \ = \ rS \ - \ rM * \lfloor rS \ / rM \rfloor$$

If both operands happen to be non-negative, then the remainder will also be non-negative.

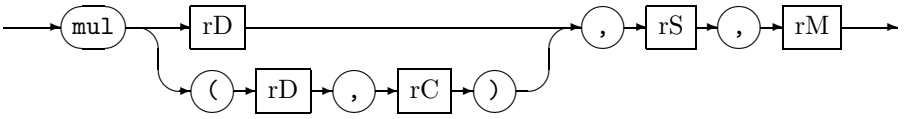
Modulus by zero is handled in a target-dependent manner.

6.27.5 Example

```
ldi    r1, -3
ldi    r2, 2
mod    r3, r1, r2        ; r3 = r1 mod r2 = -1
```

6.28 `mul` — integer multiply

6.28.1 Syntax



6.28.2 Operation

$rD := rS * rM$
$rC := condition(rS * rM)$

6.28.3 Description

The `mul` instruction multiplies the contents of `rM` with `rS`, placing the result in `rD`. Optionally, the condition codes of the result can be placed in `rC`.

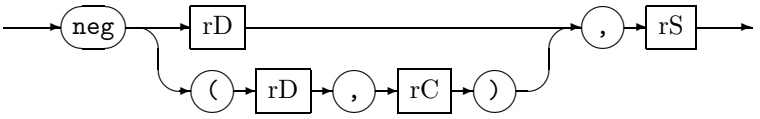
6.28.4 Notes

6.28.5 Example

```
ldi    r1, 3
ldi    r2, -2
mul    r3, r1, r2        ; r3 = r1 * r2 = -6
```

6.29 neg — integer negate

6.29.1 Syntax



6.29.2 Operation

$rD := -rS$
$rC := condition(-rS)$

6.29.3 Description

The `neg` instruction calculates the signed 2's complement value of `rS`, placing the result in `rD`. Optionally, the condition codes of the result can be placed in `rC`.

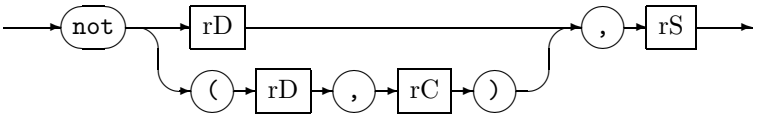
6.29.4 Notes

6.29.5 Example

```
ldi    r1, 3
neg    r2, r1          ; r2 = -r1 = -3
```

6.30 `not` — bitwise complement

6.30.1 Syntax



6.30.2 Operation

$rD := \neg rS$
$rC := condition(\neg rS)$

6.30.3 Description

The `not` instruction calculates the 1's complement value of `rS`, placing the result in `rD`. Optionally, the condition codes of the result can be placed in `rC`.

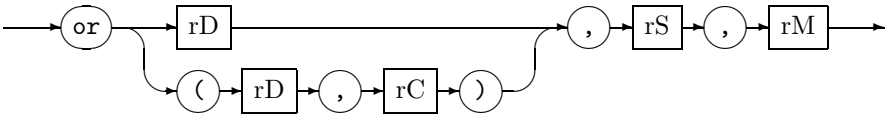
6.30.4 Notes

6.30.5 Example

```
ldi    r1, 1
not    r2, r1          ; r2 = ~r1 = 0xFFFFFFFE
```

6.31 or — bitwise OR

6.31.1 Syntax



6.31.2 Operation

$rD := rS \vee rM$
$rC := condition(rS \vee rM)$

6.31.3 Description

The `or` instruction bitwise ORs together the contents of `rS` and `rM`, placing the result in `rD`. Optionally, the condition codes of the result can be placed in `rC`.

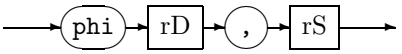
6.31.4 Notes

6.31.5 Example

```
ldi    r1, 1
ldi    r2, 2
or     r3, r1, r2        ; r3 = r1 OR r2 = 3
```

6.32 `phi` — phi-merge

6.32.1 Syntax



6.32.2 Operation

$rD := rS$

6.32.3 Description

The `phi` instruction merges the contents of `rS` into `rD`. It directly implements the ϕ-function of SSA-form.

6.32.4 Notes

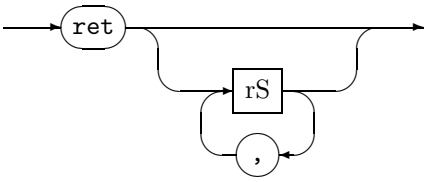
This is the *only* legal multiple-assignment instruction in *tri*VM.

6.32.5 Example

```
ldi    r1, 0x5678
ldi    r2, 0x1234
phi    r1, r2      ; r1 = 0x1234
```

6.33 `ret` — return from a procedure

6.33.1 Syntax



6.33.2 Operation

if rS **then**
$rS \mapsto rR^{caller}$
fi
return

6.33.3 Description

The `ret` instruction returns to the caller. Return values, `rS`, are mapped to the caller's result registers, `rR`.

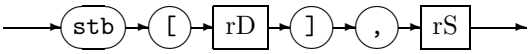
6.33.4 Notes

6.33.5 Example

```
ldi   r1, 'a'
ret   r1              ; return 'a'
```

6.34 `stb` — store byte

6.34.1 Syntax



6.34.2 Operation

$M[rD] : B := rS$

6.34.3 Description

The `stb` instruction stores the bottom eight bits of `rS` in the byte addressed by the contents of `rD`.

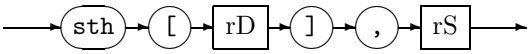
6.34.4 Notes

6.34.5 Example

```
ldi    r1, ByteVar
ldi    r2, 0x12345678
stb    [r1], r2      ; M[ByteVar] = 0x78
```

6.35 `sth` — store half-word

6.35.1 Syntax



6.35.2 Operation

$M[rD] : H := rS$

6.35.3 Description

The `sth` instruction stores the bottom sixteen bits of `rS` in the half-word addressed by the contents of `rD`.

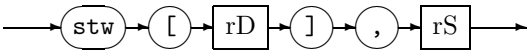
6.35.4 Notes

6.35.5 Example

```
ldi    r1, ShortVar
ldi    r2, 0x12345678
sth    [r1], r2      ; M[ShortVar] = 0x5678
```

6.36 `stw` — store word

6.36.1 Syntax



6.36.2 Operation

$M[rD] : W := rS$

6.36.3 Description

The `stw` instruction stores `rS` in the word addressed by the contents of `rD`.

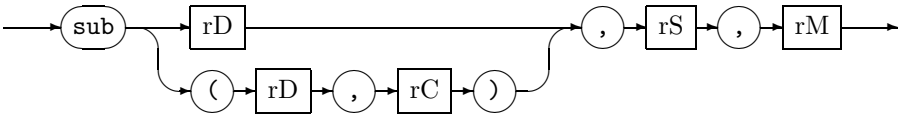
6.36.4 Notes

6.36.5 Example

```
ldi    r1, WordVar
ldi    r2, 0x12345678
stw    [r1], r2      ; M[WordVar] = 0x12345678
```

6.37 sub — integer subtract

6.37.1 Syntax



6.37.2 Operation

$rD := rS - rM$
$rC := condition(rS - rM)$

6.37.3 Description

The `sub` instruction subtracts the contents of `rM` from `rS`, placing the result in `rD`. Optionally, the condition codes of the result can be placed in `rC`.

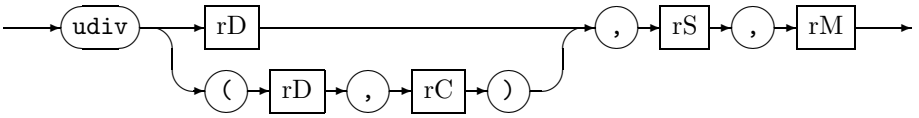
6.37.4 Notes

6.37.5 Example

```
ldi    r1, 1
ldi    r2, 2
sub    r3, r1, r2         ; r3 = r1 - r2 = -1
add    (r4, r5), r1, r2   ; r4 = r1 - r2 = -1
                          ; r5 = condition(r1 - r2)
```

6.38 `udiv` — integer divide (unsigned)

6.38.1 Syntax



6.38.2 Operation

$rD := rS \div rM$
$rC := condition(rS \div rM)$

6.38.3 Description

The `udiv` instruction divides the unsigned contents of `rS` by `rM`, placing the result in `rD`. Optionally, the condition codes of the result can be placed in `rC`.

6.38.4 Notes

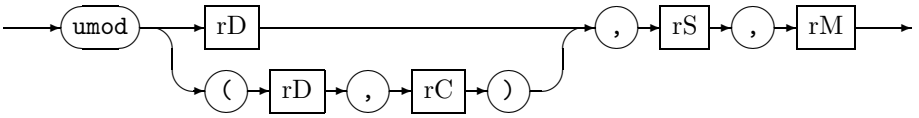
Division by zero is handled in a target-dependent manner.

6.38.5 Example

```
ldi    r1, 4
ldi    r2, 2
mul    r3, r1, r2        ; r3 = r1 / r2 = 2
```

6.39 `umod` — integer modulus (unsigned)

6.39.1 Syntax



6.39.2 Operation

$rD := rS \ umod \ rM$
$rC := condition(rS \ umod \ rM)$

6.39.3 Description

The `umod` instruction calculates the unsigned remainder from the division of `rS` by `rM`, placing the result in `rD`. Optionally, the condition codes of the result can be placed in `rC`.

6.39.4 Notes

It always holds that

$$rS \ umod \ rM \ = \ rS \ - \ rM * \lfloor rS \ / rM \rfloor$$

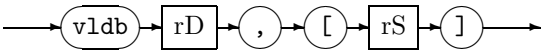
Modulus by zero is handled in a target-dependent manner.

6.39.5 Example

```
ldi    r1, 3
ldi    r2, 2
umod   r3, r1, r2        ; r3 = r1 mod r2 = 1
```

6.40 `vldb` — volatile load sign-extended byte

6.40.1 Syntax



6.40.2 Operation

$rD := SignExtend(M[rS] : B)$

6.40.3 Description

The `vldb` instruction loads the byte (8-bit) value addressed by the contents of `rS` from memory, sign-extends it to fill `rD`, and potentially changes some other aspect of the system.

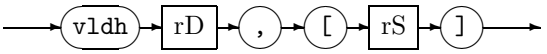
6.40.4 Notes

6.40.5 Example

```
ldi    r1, ByteVar
vldb   r2, [r1]      ; r2 = sign-extended contents of ByteVar
```

6.41 `vldh` — volatile load sign-extended half-word

6.41.1 Syntax



6.41.2 Operation

$rD := SignExtend(M[rS] : H)$

6.41.3 Description

The `vldh` instruction loads the half-word (16-bit) value addressed by the contents of `rS` from memory, sign-extends it to fill `rD`, and potentially changes some aspect of the system.

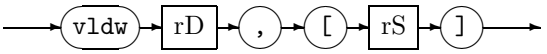
6.41.4 Notes

6.41.5 Example

```
ldi    r1, ShortVar
vldh   r2, [r1]      ; r2 = sign-extended contents of ShortVar
```

6.42 `vldw` — volatile load word

6.42.1 Syntax



6.42.2 Operation

$rD := M[rS] : W$

6.42.3 Description

The `vldw` instruction loads the word value addressed by the contents of `rS` from memory into `rD` and potentially changes some aspect of the system.

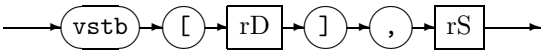
6.42.4 Notes

6.42.5 Example

```
ldi    r1, WordVar
vldw   r2, [r1]      ; r2 = contents of WordVar
```

6.43 `vstb` — volatile store byte

6.43.1 Syntax



6.43.2 Operation

$M[rD] : B := rS$

6.43.3 Description

The `vstb` instruction stores the bottom eight bits of `rS` in the byte addressed by the contents of `rD` and potentially changes some aspect of the system.

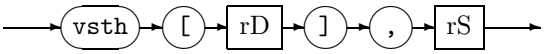
6.43.4 Notes

6.43.5 Example

```
ldi    r1, ByteVar
ldi    r2, 0x12345678
vstb   [r1], r2      ; M[ByteVar] = 0x78
```

6.44 `vsth` — volatile store half-word

6.44.1 Syntax



6.44.2 Operation

$M[rD] : H := rS$

6.44.3 Description

The `vsth` instruction stores the bottom sixteen bits of `rS` in the half-word addressed by the contents of `rD` and potentially changes some aspect of the system.

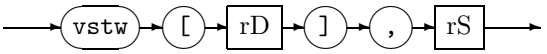
6.44.4 Notes

6.44.5 Example

```
ldi    r1, ShortVar
ldi    r2, 0x12345678
vsth   [r1], r2      ; M[ShortVar] = 0x5678
```

6.45 `vstw` — volatile store word

6.45.1 Syntax



6.45.2 Operation

$M[rD] : W := rS$

6.45.3 Description

The `vstw` instruction stores `rS` in the word addressed by the contents of `rD` and potentially changes some aspect of the system.

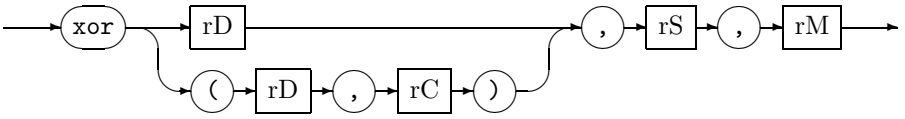
6.45.4 Notes

6.45.5 Example

```
ldi    r1, WordVar
ldi    r2, 0x12345678
vstw   [r1], r2      ; M[WordVar] = 0x12345678
```

6.46 `xor` — bitwise Exclusive-OR

6.46.1 Syntax



6.46.2 Operation

$rD := rS \oplus rM$
$rC := condition(rS \oplus rM)$

6.46.3 Description

The `xor` instruction bitwise Exclusive-ORs together the contents of `rS` and `rM`, placing the result in `rD`. Optionally, the condition codes of the result can be placed in `rC`.

6.46.4 Notes

6.46.5 Example

```
ldi    r1, 1
ldi    r2, 3
xor    r3, r1, r2        ; r3 = r1 XOR r2 = 1
```

Chapter 7

Common Program Structures

We demonstrate a number of typical program structures in C and their *tri*VM equivalents. The examples are designed to illustrate key features of the *tri*VM intermediate code for supporting high-level constructs.

We start with basic blocks, including expressions, and reviewing common sub-expressions. We then illustrate control-flow constructs—tests and loops. The following section describes the procedure structure and calling mechanism. We then continue with a look at data, both local and global. Finally, we bring all of these elements together with a larger example.

The *tri*VM programs presented here are also typical of source code layout. Each *tri*VM source line is either a `const` declaration, a `data` declaration, or a single line of code, consisting of an optional label followed by an optional instruction. All *tri*VM source lines free-format—white space (spaces and tabs) may be freely interspersed between the line components—and are terminated with a carriage return.

7.1 Basic Blocks

A basic block is defined as having one control-flow entry edge and one control-flow exit-edge. The example in Figure 7.1 shows an example of a basic block and its unoptimized *tri*VM translation.

A common sub-expression (CSE) is a sub-expression which is common to two or more enclosing expressions, and whose arguments are live between all uses of that sub-expression, i.e. they are not killed by any intervening assignment. Figure 7.2 shows how a common sub-expression can occur and the *tri*VM code so generated.

7.2 Tests

An important part of any language is decision-making. In *tri*VM we have the comparison operators (one for integer values and one for floating-point values) and a set of conditional branches. These work together to implement the

```
int foo( int x, int y )
{
    int a, b, c, d, e, f, g;

    a = x + y;
    b = x - y;
    c = x * y;
    d = x / y;
    e = a + b;
    f = c - d;
    g = e % f;

    return g;
}
```

(a)

```
proc foo (2), 1
    add r3, r1, r2
    sub r5, r1, r2
    mul r7, r1, r2
    div r9, r1, r2
    add r11, r3, r5
    sub r13, r7, r9
    mod r15, r11, r13
    ret r15
end
```

(b)

Figure 7.1: Example of a basic block. Execution starts at the first statement (`a = x + y`) and ends at the last statement (`g = e % f`), not counting the return statement. The *tri*VM version confirms this block structure, beginning with the first `add` and ending with the `mod`—there are no labels and no jumps into, out of, or within this block.

```
int foo( int x, int y )
{
    int a, b, c, d;

    a = x + y;
    b = x - y;
    c = x + y;
    d = a + b + c;

    return d;
}
```

(a)

```
proc foo (2), 1
    add r3, r1, r2   <-
    sub r5, r1, r2
    add r7, r1, r2   <-
    add r10, r3, r5
    add r9, r10, r7
    ret r9
end
```

(b)

Figure 7.2: In this contrived example, we spot the CSE "`x + y`" in the original C source (a). This is reflected in the *tri*VM code (b) as an instruction with both the same operation and the same arguments. In this case, the operation is `add`, and the two common arguments are `r1` and `r2`. The SSA property of *tri*VM ensures that we can easily and safely mark these two *tri*VM statements as equivalent, and thus candidates for subsequent optimization.

```
int foo( int x, int y )
{
    int a;

    if ( x < 10 )
        a = 1;
    else
        a = -1;

    return y * a;
}
```

(a)

```
proc foo (2), 1
    ldi r3, 10
    cmp r10, r1, r3
    bge r10, L2
    ldi r4, 1
    phi r7, r4
    bra L3
L2:
    ldi r9, -1
    phi r7, r9
L3:
    mul r8, r2, r7
    ret r8
end
```

(b)

Figure 7.3: A simple if-then-else program structure. The two branches (the conditional branch to `L2` and the direct branch to `L3`) implement the two-way branch, covering both the then- and else-clauses. We also identify the phi-loads, for `r7`, which combine the two definitions of variable a into one new *tri*VM register.

decision-making structures, illustrated in Figure 7.3 for an if-then-else case.

Figure 7.3 also illustrates the use of the `phi` instruction to handle multiple-assignment to a register. In this example there are two possible definitions of a that reach the end of the function, one each from the then- and else-clauses. In SSA-form this multiple-assignment requirement would be met with a ϕ-function; in *tri*VM we use the `phi` instruction to implement ϕ-function, placing a `phi` load at the root of each path to the merge point. In Figure 7.3 the merge point is at `L3`, so we place `phi` loads at each point that branches to `L3`: one immediately prior to the label and one just prior to the direct branch to `L3`.

An alternative means of generating condition values is through the second result register pairing for the arithmetic and logic instructions (Figure 7.4). These provide a means of exposing the behaviour of such operations on the condition registers found in the majority of modern microprocessor designs.

7.3 Loops

The previous example introduced conditional branches to support decisions. For selection structures (if-then-else or switch-case) the predicated jumps are forwards either over the false block (in a true condition) or to the end of the structure for false conditions. The other major structure involving non-linear control-flow is the loop, which relies on backwards control-flow.

In general, a loop structure consists of two parts—the loop body and the loop test[1]. The body may be executed zero, one or multiple times, while the test is executed at least once.

[1]Occasionally the loop test may be simplified to a branch in non-terminating loops, e.g. the classic "`while (TRUE) do ... ;`"

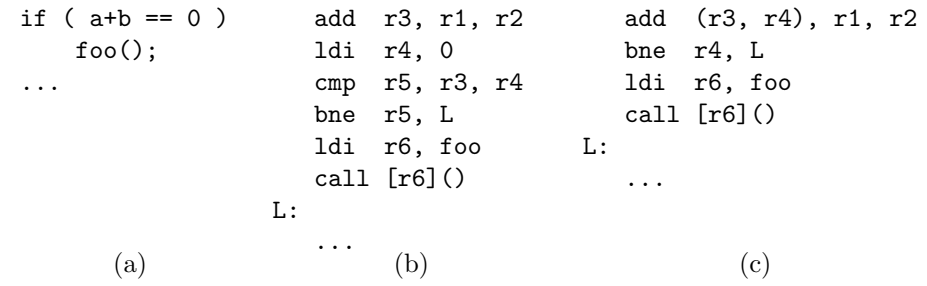


Figure 7.4: Example use of condition code registers to expose additional information available from arithmetic operations—in this instance `add`. The trivial example in (a) illustrates a common pattern, and (b) shows the naive translation to *tri*VM. In (b) we show that using the additional condition code result from the addition saves one constant load and one comparison operation.

As for the test code we have potentially multiple definitions of a given variable, in particular any induction variables, for which `phi` loads will be required.

Figure 7.5 shows a `for` loop that executes for a variable number of times, and whose induction variable is not used in the loop body.

7.4 Procedures

The previous sections have illustrated some examples of the basic procedure structure: the `proc` and `end` directives and the `ret` instruction. In this section we show, in Figure 7.6, the implementation of the call site from which such procedures are called, together with argument passing and result handling.

7.5 Global and Local Data

So far, all the previous examples have operated on *tri*VM virtual registers in R-space. Here we illustrate the use of data held in M-space, both at the local and global levels.

With reference to Figure 7.7, variable *globalVar* has global visibility within this source module, and in *tri*VM is defined with the `data` directive, together with a size of four bytes[2]. In this instance there is no initial value.

The second data declaration is for the local variable *localVar*. Its scope is restricted to within ***foo***, and is akin to memory storage on the stack of a target processor. Its address may be taken and operated on, unlike registers which may not. It is allocated on entry to the procedure, and destroyed on exit. If the storage were to be statically qualified, it would be placed outside of ***foo*** at module-level.

A further example, illustrating local data and procedure parameters, is shown in Figure 7.8.

Within the body of the procedure, accessing the variables is done through the `ldw` and `stw` instructions (load-word and store-word respectively). Support

[2]We assume that integers are four bytes wide.

```
int foo( int x, int y )
{
    int a, b;

    b = 0;
    for ( a = 0; a < x; a++ )
        b = b + y;

    return b;
}
```

(a)

```
proc foo (2), 1
    ldi r3, 0
    ldi r5, 0
    phi r9, r3
    phi r8, r5
    bra L5
L2:
    add r10, r9, r2
    ldi r12, 1
    add r11, r8, r12
    phi r9, r10
    phi r8, r11
L5:
    cmp r13, r8, r1
    blt r13, L2
    ret r9
end
```

(b)

Figure 7.5: Our example loop, in C (a) and *tri*VM (b). Note the position of the loop test below the loop body—fewer branches are executed in this form $(n + 2)$ than compared to the more obvious form $(2n + 1)$, gaining execution performance with no code size penalty. Also, if we can determine that the loop body executes at least once we can eliminate the initial branch, saving one instruction. Note also that registers `r3` and `r5` could be merged into one register, again saving one instruction.

for different data sizes (byte and half-word) is through sign-extend loads (`ldb` and `ldh` respectively) and part-word stores (`stb` and `sth` respectively).

7.6 Putting It All Together

We conclude with a large example (Figure 7.9) that draws together all of the elements of the *tri*VM language, including directives, data types and program structures.

The example chosen is an implementation of the combination operation, defined as

$$ {}_nC_r = \frac{n!}{(n-r)!r!} $$

Referring to the C source code in Figure 7.9(a), aside from the enumeration and manifest constant to aid readability, we have the global variable *errno*, which maintains a global error number value, used by the two functions[3].

The two procedures, ***fac*** and ***combination***, implement the calculation of the number of combinations of r items from a set of n items. Procedure ***fac*** calculates the factorial of its argument, validating the input prior to the calculation itself—if the argument is greater than some predetermined limit

[3]Borrowed from the Standard C Library[13].

```
int foo( int x )
{
    return x + 1;
}

int bar( int x, int y )
{
    int a;

    a = foo( x );
    return y + a;
}
```

(a)

```
proc foo (1), 1
    ldi r3, 1
    add r2, r1, r3
    ret r2
end

proc bar (2), 1
    ldi r4, foo
    call [r4](r1),r3
    add r7, r2, r3
    ret r7
end
```

(b)

Figure 7.6: A complete caller-callee example. Procedure ***bar*** calls ***foo*** with a single argument from `r1`. The single return value is placed in register `r3`, which is then added to the second argument of ***bar*** (in `r2`), with the final result passed back to ***bar***'s caller.

(*MAX_FAC_ARG*) the global error value is set and the procedure returns the numeric value 1, and likewise for negative arguments.

Procedure ***combination*** initially clears *errno* prior to calling ***fac*** twice to calculate the bottom half of the expression. This will trap incorrect values of both r and $n-r$. If there are any errors a result of 0 is returned to the caller, else the remaining call to ***fac*** is made, and the final stage in computation performed.

While this may not be a particularly rigorous example, it does illustrate all of the previous concepts: basic blocks, tests, loops, procedure bodies and calls, and global data.

With reference to Figure 7.9(b), the operation of the two procedures should be evident based on the previous examples. Of note here is that both procedures have been hand-optimized to minimise their size for this paper. In particular, all constant loads (`ldi rN, C`) have been moved to the top of each procedure, effectively parameterizing the main body of code. While not specifically addressed in this paper, we believe this format to be of benefit in code space optimization.

```
int globalVar;

int foo( int x )
{
    int localVar;

    localVar = x;
    globalVar = localVar + x;

    return localVar + globalVar;
}
```

(a)

```
data globalVar[4]

proc foo (1), 1
data localVar[4]
    ldi r2, localVar
    stw [r2], r1
    ldw r4, [r2]
    add r3, r4, r1
    ldi r6, globalVar
    stw [r6], r3
    ldw r8, [r2]
    ldw r10, [r6]
    add r7, r8, r10
    ret r7
end
```

(b)

Figure 7.7: Accessing M-space data through the `ldw` and `stw` instructions.

```
void bar( int *p )
{
    *p = 42;
}

int foo( int i )
{
    bar( &i );
    return i;
}
```

(a)

```
proc bar (1), 0
    ldi r2, 42
    stw [r1], r2
    ret
end

proc foo (1), 1
data !p!i
    ldi r2, !p!i
    stw [r2], r1
    ldi r4, bar
    call [r4](r2)
    ldw r5, [r2]
    ret r5
end
```

(b)

Figure 7.8: This example illustrates taking the address of a parameter. This forces it to be placed in M-space, whereupon its address can be computed into a register. In this instance the single argument is stored in local variable *!p!i*, whose address is then passed to ***bar***.

```
#define MAX_FAC_ARG ( 50 )

enum {
    E_NO_ERROR,
    E_FAC_ARG_TOO_BIG,
    E_FAC_ARG_NEGATIVE
};

int errno;

int fac( int n )
{
    int result = 1;

    if ( n > MAX_FAC_ARG )
        errno = E_FAC_ARG_TOO_BIG;
    else if ( n < 0 )
        errno = E_FAC_ARG_NEGATIVE;
    else
        for ( ; n; n-- )
            result *= n;

    return result;
}

int combination( int n, int r )
{
    int result, bottom;

    errno = E_NO_ERROR;
    bottom = fac(n - r)*fac(r);

    if ( errno != E_NO_ERROR )
        result = 0;
    else
        result = fac(n)/bottom;

    return result;
}
```

(a)

```
data errno[4]

proc fac (1), 1
    ldi r2,  1
    ldi r4, 50
    ldi r10, 0
    ldi r11, 2
    ldi r6, errno
    phi r9, r2
    cmp r22, r4, r1
    bge r22, L3
    stw [r6], r2
    bra L4
L3: phi r16, r1
    cmp r23, r1, r10
    bge r23, L10
    stw [r6], r11
    bra L4
L7: mul r19, r9, r16
    sub r20, r16, r2
    phi r9, r19
    phi r16, r20
L10:cmp r24, r16, r10
    bne r24, L7
L4: ret r9
end

proc combination (2), 1
    ldi r3, 0
    ldi r4, errno
    ldi r7, fac
    stw [r4], r3
    sub r5, r1, r2
    call [r7](r5),r6
    call [r7](r2),r9
    mul r12, r6, r9
    ldw r14, [r4]
    cmp r29, r14, r3
    beq r29, L1
    phi r21, r3
    bra L2
L1: call [r7](r1),r26
    div r28, r26, r12
    phi r21, r28
L2: ret r21
end
```

(b)

Figure 7.9: Complete *tri*VM code example. Refer to text for description.

Bibliography

[1] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[2] Cytron, R. K., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Programming Languages and Systems 13*, 4 (October 1991), 451–490.

[3] Goldberg, D. What every computer scientist should know about floating-point arithmetic. In *ACM Computing Surveys* (1991), ACM.

[4] Hall, M. W., Anderson, J. M., Amarasinghe, S. P., Murphy, B. R., Liao, S.-W., Bugnion, E., and Lam, M. S. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer 29*, 12 (December 1996), 84–89.

[5] imon L. Peyton Jones, Hall, C. V., Hammond, K., Partain, W., and Wadler, P. The Glasgow Haskell Compiler: a technical overview. In *Proc. Joint Framework for Information Technology (JFIT) Technical Conference* (March 1993), DTI/SERC.

[6] Intel. *i486 Processor Programmer's Reference Manual*. Intel Corp./Osborne McGraw-Hill, San Fransisco, 1990.

[7] Jaggar, D. *ARM Architecture Reference Manual*. Prentice Hall, Cambridge, UK, 1996.

[8] Jones, S. P., and Meijer, E. Henk: a typed intermediate language. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation (TIC'97)* (Amsterdam, The Netherlands, 1997), ACM Press.

[9] Kernighan, B. W., and Ritchie, D. M. *The C Programming Language*, 2nd ed. Prentice Hall, 1988.

[10] Lewis, B. T., Deutsch, L. P., and Goldstein, T. C. Clarity MCode: A retargetable intermediate representation for compilation. In *ACM SIGPLAN Workshop on Intermediate Representations* (San Francisco, CA USA, January 1995), ACM Press, pp. 119–128.

[11] Morrisett, G., Walker, D., Crary, K., and Glew, N. From System F to Typed Assembly Language. *ACM Trans. Programming Languages and Systems 21*, 3 (1999), 527–568.

[12] O'Brien, K., O'Brien, K. M., Hopkins, M., Shepherd, A., and Unrau, R. XIL and YIL: The intermediate languages of TOBEY. In *ACM SIGPLAN Workshop on Intermediate Representations* (San Francisco, CA USA, January 1995), ACM Press, pp. 71–82.

[13] Plauger, P. J. *The Standard C Library.* Prentice Hall, 1992.

[14] Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., and Lee, P. TIL: A type-directed compiler for ML. In *ACM SIGPLAN Conf. Programming Language Design and Implementation* (New York, 1996), ACM Press, pp. 181–192.

[15] Wells, J. B., Dimock, A., Muller, R., and Turbak, F. A. A typed intermediate language for flow-directed compilation. In *Proc. TAPSOFT'97, Theory and Practice of Software Development (LNCS 1214)* (Lille, France, April 1997), Springer-Verlag, pp. 757–771.