Mite

a basis for ubiquitous virtual machines

Reuben Rhys Thomas
St John's College

A dissertation submitted for the Ph.D. degree

November 2000

Ars est celare artem

To Basil Rose, who did it first, and Tony Thomas, who hooked me.

Preface

Mite is a virtual machine intended to provide fast language and machine-neutral just-in-time translation of binary-portable object code into high quality native code, with a formal foundation.

Chapter 1 discusses the need for fast high-quality run-time translation of portable code, considers what functionality is needed, and sets out a list of goals that Mite should reach. The chapter ends by stating the contribution of the thesis. Chapter 2 discusses related work, concentrating on the extent to which the various systems examined fulfil Mite's goals. Chapter 3 elaborates Mite's design, and chapter 4 analyses the choices that it makes. Chapter 5 examines the implementation, which consists of a C compiler back end targeting Mite and a virtual code translator for the ARM processor, and shows how it could be extended to other languages and processors. Chapter 6 describes and analyses a series of tests performed on the implementation, then assesses both the design and implementation in their light. Chapter 7 describes future work; finally, chapter 8 concludes the thesis with an appraisal of how well Mite meets its goals, and a final perspective.

Appendices A–C give Mite's definition, appendix D lists two of the benchmark programs used in chapter 6, and appendix E gives the data collected from the tests.

This dissertation is my own work and includes nothing resulting from collaboration.

I owe many thanks. Alistair Turnbull's keen insight, blunt criticism and imaginative advice elaborated in many hours of enjoyable discussion have cheered and smoothed my path; he also read and criticized the text. Simon Peyton Jones has been a smiling source of balanced encouragement and criticism. Martin Richards supervised me with a light hand, and lit the murk of bureaucracy. Rosamunde Almond, Julian Seward, Simon Marlow and Eugenia Cheng pointed out errors logical and literary. Several kindly souls gave other assistance; in particular, Dave Hanson and Chris Fraser helped me far beyond the call of duty in my abuse of their LCC compiler, and Arthur Norman gave useful advice on the shape a thesis should have.

Contents

1 Introduction

This thesis describes Mite, a general-purpose low-level virtual machine (VM) with a semi-formal definition (see appendices A and B) and binary-portable code format (see appendix C). It is a good target for compiled languages, and allows compilers to perform many optimizations on the virtual code, so that its just-in-time (JIT) translator can be simple and fast, while still producing good native code.

The rest of this chapter motivates the design of Mite, sets out a list of goals that it should reach, and states the contribution of the thesis. Chapter 2 discusses related work. Chapter 3 elaborates Mite's design, and chapter 4 analyses the choices that it makes. Chapter 5 starts by outlining the structure of the implementation, which consists of a C compiler back end targeting Mite, and a virtual code translator for the ARM processor. A sample translation is followed from C to Mite code, and then to ARM assembler, and optimizing compilation is discussed. The chapter ends by introducing methods for dealing with other languages and processors. Chapter 6 describes and analyses a series of tests performed on the implementation, then assesses both the design and implementation in their light. Chapter 7 suggests future work; finally, chapter 8 concludes the thesis with an appraisal of how well Mite meets its goals, and ends with a final perspective. Appendices A–C give Mite's definition, appendix D lists two of the benchmark programs used in chapter 6, and appendix E gives the data collected from the tests.

1.1 Motivation

Computers are becoming increasingly diverse in form and function, and ever more connected, above all via the internet. At the same time, the tasks we use them for are becoming more distributed: we can send and receive email from our television and mobile phone as well as our PC, and keep our diary and address book synchronized between our desktop, laptop and PDA. This environment encourages the creation of software which is not only portable but mobile: sending code across the network gives much more flexibility and power than communicating only data.

Of course, many tools and languages for creating portable and mobile code already exist. Unfortunately, they rarely consist of reusable components. Code generators tend to be tied to a single compiler, which often means a single language; compilers usually produce code that runs only on one machine; VMs are often integrated with a particular language or run-time environment (or both). Moreover, tradeoffs such as those between portability and loading time, or speed of JIT translation and speed of execution, or compilation time and memory usage, tend either to be beyond the programmer's control, as in VM systems such as Java [43] or Inferno [26], or require a lot of work to alter, as in a compiler system such as GNU C [35].

It would be vainglory in the spirit of UNCOL [116] to attempt to design a single system that solves all these problems; not even Java claims to be that. The current diversity is a testament to the need for a range of solutions. At the same time, the increasing complexity of software systems underlines the necessity of an approach to programming that is at the same time more modular (building components rather than monoliths) and higher-level (programming with components rather than lines of code) than a single system can easily provide. What is really needed is a common basis from which to attack the problems outlined above.

Hence, it would be nice to have a system that provided:

A single target for code generation

Portable binaries and hence, potentially, mobile code

Support for a wide range of compiled languages

Fine control over the tradeoffs involved in compilation

An open and extensible basis for more complex systems

This is what Mite attempts to do.

1.2 Scope and goals

In the light of the desiderata listed above, the functionality required of Mite is outlined below. This leads to a list of goals that the system should meet in order to provide such functionality; afterwards, some non-goals are disposed of.

1.2.1 Required functionality

Given the list above of benefits that Mite would like to provide, what features must it have? To act as a single target for code generation, Mite must provide a machine-independent model of computation. Together with the requirements that it support most compiled languages and give a high degree of control over compilation tradeoffs, this seems to indicate a low-level execution model. To allow portable code, a standard binary format must be provided. To form an open basis for building more complex systems, it should be possible to interwork with native code. If Mite is to be trusted, its definition should be as precise as possible, preferably formal. Finally, if Mite is to give fine control over the code generation process and act as a basis for more complex systems, then it must be just as flexible as the underlying machines; it should not preclude alternative implementations of features not directly implemented by the processors themselves, such as memory management and concurrency. This, taken together with the indication of a low-level execution model, suggests that Mite should be little more than a processor abstraction.

1.2.2 Goals

The features that Mite must provide are now listed; for each, references to the points at which it is introduced and discussed are given.

A low-level processor-based VM model (chapter 3 and section 4.1) The VM model should be little more than a processor abstraction, and have a similar instruction set and execution model to those of conventional processors.

Architecture neutrality (sections 4.1, 5.3 and 5.5) Most of Mite's features should be common to all processors; for example, most instructions should map directly to native instructions. At the same time, the translator should be able to take advantage of processor features that are not modelled.

Language neutrality (sections 4.1 and 5.4) Mite should constrain a compiler no more than the underlying machine, so that language compilation techniques used by native code generators are applicable to Mite, and languages that are compiled directly to native code can be similarly compiled for Mite.

Portable virtual code (sections 4.1.2, 4.1.3 and 4.2) It should be possible (though not mandatory) to compile virtual code that can run unaltered on any target system.

Fast JIT translation (sections 4.1, 5.2.2 and 6.1.5 The virtual code should be translatable in a single pass, and most of Mite's instructions should map directly to machine instructions.

High-quality native code (sections 4.1, 5.3 and 6.1.2) With the combination of optimizable virtual code and annotations, Mite should be capable of generating excellent native code. More importantly, the responsibility for code quality should lie squarely on the compiler; generating good code should not slow the JIT translator down. To make significant further improvements should require the translator to make detailed machine-specific analysis.

Virtual code annotation (sections 3.2.10, 6.1.2 and 6.1.3) It should be possible for compilers to help the JIT translator with machine-dependent aspects of code generation (which they cannot perform themselves), such as register allocation, by annotating the virtual code.

Interworking with native code (sections 3.2.7 and 5.1.5) Mite should reside in the same address space or spaces as native code on the host system, and be able to call and be called like normally compiled code. This will allow it to integrate with native libraries and object modules without glue code.

Portable object format (appendix C, and sections 3.4 and 4.4) A simple binary-portable object file format should be provided that is endianness-independent and quick to read, write, and traverse.

Precise definition (appendices A and B, and section 4.5) Mite's definition should be given in formal terms, so that implementors can be sure of its meaning, and allowing proofs about Mite programs to be made and automatically checked.

Section 8.1 discusses how well Mite meets these goals.

1.2.3 Non-goals

Mite does not aim to provide:

Write once, run anywhere This requires portable libraries as well as portable code, and leads towards a closed system. On the other hand, it would certainly be possible to develop a set of libraries that can be widely implemented and used by binary-portable programs, and use them with Mite to make such a system.

High-level mechanisms Mechanisms such as security, concurrency, exceptions and garbage collection should be implemented orthogonally to the main execution model as far as possible. [93] and [103] show how exceptions and accurate garbage collection can be added to a code generation system with only minor modifications to the core design, and section 5.4.3 considers how to support garbage collection in Mite.

The last word in execution speed This is unrealistic given an architecture-neutral VM model and a fast translator (although performance can still be near-optimal: see section 6.1.2). In any case, run-time optimization of the native code can probably produce better results than any static optimizations [66].

1.3 Contribution of this thesis

Mite makes the following contribution:

Novel mechanisms for portable just-in-time optimization It provides novel mechanisms for compilers to indicate optimizations in a machine-independent way that can be used rapidly by a just-in-time translator to produce good native code: the ranked register stack (section 4.2.2) and constant registers (section 4.2.5). In particular, register ranking can be used to encode any register allocation algorithm, with some constraints, so that it can be performed in linear time by the just-in-time translator (section 4.3.1.2).

Compiler can ensure good native code Mite's design allows a good compiler to ensure that good native code is produced, even by a simple JIT translator. Rather than the usual tradeoff between the time taken to perform the JIT translation and the speed of the resultant native code, Mite ensures that there is little a translator has to do to turn good virtual code into good native code. A simpler compiler's output can be improved by a language and machine-independent virtual code optimizer.

Flexible non-local return mechanism Mite provides a simple non-local return mechanism (section 4.3.6.1) that can be used to implement a wide range of control flow mechanisms such as exceptions and continuations, while remaining compatible with the system calling convention. The only system to offer a similar mechanism is `C--` [103], and though more flexible, its is also much more complex than Mite's.

Low-overhead 32/64-bit portability Three-component numbers (section 4.1.3) allow compilers to generate code that runs on both 32 and 64-bit machines, without having to defer calculations based on the word size to run time, or forcing the compiler to use a complex algebra to make such calculations at compile time.

Unique combination of features Mite uniquely occupies a point in the design space of VM-based systems. No other system offers a small, stand-alone, language-neutral VM capable of producing optimized native code. Useful in its own right as a compiler target and back end, Mite would make a good starting point for a wide range of VM-based applications, including dynamic code generation, portable execution environments along the lines of Java, and distributed operating systems. Its definition (appendices A and B) could be used as the basis for theoretical investigations of code generation issues. In addition, Mite is fully documented, and its JIT translator is free software (see http://rrt.sc3d.org/).

2 Context

A major reason for designing Mite was that no system known to me met all the requirements identified in section 1.2.1. However, a variety of systems share some of them, and meet others to varying degrees. Several are examined below. Each system's key features are described with the reasons for their introduction and the advantages and disadvantages that they entail. Direct comparisons with Mite are postponed to chapter 4, after Mite's design has been elaborated.

2.1 Java VM

Java is the most high-profile VM-based system currently in use. It shares with Mite the goal of providing an architecture-neutral platform for the execution of binary-portable object code, but otherwise the two are dissimilar. The Java VM [67] (JVM) is much more than a hardware abstraction layer, and is tightly coupled to the Java system [43], with direct support for language and system-specific features such as objects, monitors and byte-code verification.

Before discussing specific features of Java, it is worth noting one guiding principle, that of familiarity. Although Java as a whole was a novelty when it was introduced, it was built from tried-and-tested components: from the VM model to concurrency, the language to the I/O model, it used familiar, well understood and thoroughly tested concepts. Thus, it is hardly surprising that some of its design choices have obvious disadvantages.

The JVM is a stack-based architecture, with a zero-operand byte-coded instruction set. The project that produced Java was originally aimed at embedded devices, where low memory consumption was crucial, and the lack of processing power favoured an interpreter-based system. The byte-coded stack machine is a classic implementation technique, exemplified by the p-code system [85], and gives good code density and performance on smaller machines, especially 8-bit microprocessors. However, general-purpose interpreters of this sort tend to be an order of magnitude slower than optimized native code for compute-bound tasks [24].[1] Hence, JIT translators are generally used to execute Java. To produce good native code, register allocation must be performed for stack locations, and this is almost as much work as compiling Java source to native code; thus, JIT translators cannot easily be fast and produce fast code. Also,

[1]In more specialized systems the reverse can be the case. For example, APL interpreters spend most of their time executing the opcodes [4], so the overhead of instruction fetch and decode is negligible. In addition, since the VM instructions are heavily tuned for the language, and are implemented in carefully hand-crafted code, programs generally run faster than if they were compiled conventionally into native code.

though the byte-code is fairly compact, as zero-operand code tends to be, Java class files are usually bulky compared with equivalent C or C++ executables, because of all the type information they contain.

The JVM has direct support for Java language types, both primitive integral, floating point and character types, and object types. This is primarily to ensure safe execution of Java programs: while they can have run-time errors, including type errors, they cannot, in theory, gain unauthorized access to the host system. The JVM's type system allows many of the necessary checks to be performed at load time by the byte code verifier, which increases execution speed. The main disadvantage is that the object types are tied closely to the Java language, and even the primitive types are not general; for example, there are no unsigned integral types. This makes it hard to compile languages other than Java for the JVM: untyped and weakly typed languages must either adapt to it or simulate memory access, and strongly typed languages must contrive a mapping to Java types. Nevertheless, Forth [14, 110] and ML [118] compilers exist for the JVM.

Other aspects of the JVM's language support, such as monitors and the class file format, have similar tradeoffs: they improve efficiency and safety for running Java programs, but are at best tricky and at worst a hindrance when implementing other languages with different semantics for the same constructs.

2.2 Dis

As the JVM is to Java, so Dis [141] is to Inferno [26]. While the JVM is heavily specialized for the Java language, Dis is much more language-neutral, as Inferno supports several languages. Dis is however tied to Inferno.

Dis has a three-operand memory-to-memory instruction set. This was designed to allow a range of compilation techniques, from naïve to flow-analysis-based register allocation, while retaining a simple JIT translator; Dis instructions map more directly to machine instructions than JVM instructions. Pike [97] asserts that "a load-store model requires . . . register allocation in the compiler. Memory-to-memory doesn't, but with careful design doesn't preclude effective *native* register allocation." While this is true, mapping memory locations to registers is hampered by aliasing (see section 4.1.1). In addition, this statement suggests an interesting intention: to reduce the effort needed by the compiler (for register allocation) and instead to make the JIT translator do the work. From Mite's perspective, this is bizarre: the compiler has the time and information to perform good register allocation which the translator lacks. On the other hand, perhaps the designers of Inferno did not think that compilers could perform effective register allocation for virtual code; in this case, the service might as well be centralized in the JIT translator.

The memory-to-memory architecture also makes for faster interpretation, because fewer VM instructions are required, so the fetch-decode overhead of the interpreter is reduced. For example, to implement the assignment `a := x + y` may take two pushes, an add, and a pop on a stack machine, whereas it is a single Dis instruction.[2]

[2]This example, taken from [141], is a little unfair: a VM whose stack items can be permuted avoids many

Dis has a binary-portable object module format, which is structured into code, data, symbol, and type information. As well as allowing portability, the module design is aimed at aiding system security and run-time memory management. Because Dis has reference-counting garbage collection built in, the VM run-time system has to know the structure of every type used, so that pointers can be tracked. Security features include support for the cryptographic signature of type information. The result is that the module format, though much simpler than Java's, is still more than a plain object file format. Not only is it specific to Inferno, but the type information uses the type structure of Limbo, Inferno's main language. Limbo's type system is amenable to use for other languages, as it contains records and pointers, unlike Java's object-oriented type system; nonetheless there are one or two awkwardnesses, such as the lack of 16-bit integral types.

There are other examples of both OS support, such as threading primitives, and language support, such as instructions for managing lists and arrays. However, while it is hard to see how the OS support could be used with systems other than Inferno, the language support could easily be used with other languages. While Limbo is definitely favoured by Dis, other languages are not excluded.

In conclusion, Dis is closer to Mite's ideals than Java in most respects, but is still rather higher level.

2.3 VCODE

VCODE [29] is a dynamic code generation system that provides a machine-independent one-pass code generation interface. It is lightweight, typically executing fewer than 10 instructions per machine instruction generated.

VCODE uses a procedural interface that presents an idealized RISC-like machine. There is a code generation function for each sort of instruction that can be generated, plus functions to deal with register allocation and general housekeeping. This makes dynamic code generation convenient, as the client can call the code generation routines directly, and fast, as no intermediate data structure need be built or consumed. On the other hand, because the virtual code has no representation, it must be generated by the program that wants to use it, and cannot be stored or transmitted. It is also harder to specify the semantics of an application programming interface (API) than of a language or VM, although since VCODE has little global state, the code generation functions are largely independent. Since VCODE is implemented as a set of C macros, it is tricky to use with languages other than C.

VCODE's machine model is very low level. This allows it to offer direct access to machine registers and machine-independent delay slot instruction scheduling, and contributes to VCODE's rapid code generation. On the down side, it complicates code generation: rather than providing an infinite number of virtual registers, VCODE provides

pushes and pops by keeping frequently-used quantities on top of the stack, where they can be accessed directly; also, many stack machines' instructions implicitly pop their parameters and push their results.

calls to claim and release registers, and the client must deal with spilling when necessary. Also, the machine model is so close to that of processors that some awkwardnesses are exposed. For example, constants may either be loaded into virtual registers or placed in immediate operands. This is a natural choice to offer on a particular processor, but it is less sensible for a machine-independent system where the range of immediate constants is unknown (see section 4.2.5).

VCODE has little system or language support other than a way of calling C functions; this makes it extremely flexible. While it is obvious that to build a system such as Inferno or Java on top of VCODE would require many features to be added, this is only a disadvantage of VCODE insofar as the current design makes their addition difficult. In fact, VCODE is designed to be extensible, and adding new instructions, at least, is straightforward.

An automatic back end generator makes porting VCODE easy: back ends are generated from patterns that match virtual instructions to native instructions. Though it has several RISC implementations, VCODE has notably not been implemented on the Intel IA-32 architecture, which raises questions as to the universality of its machine model (but see the description of PASM in section 2.7).

An extension to VCODE has been written, called ICODE [99], which provides an infinite number of registers, and performs global optimization on an intermediate representation of the code. This seems to confirm VCODE's suitability as a basis for more ambitious systems, while allowing it to remain more flexible than a monolithic approach such as the JVM. However, to be useful for more than run-time code generation, VCODE needs a portable binary form. This involves more than just designing a virtual object format: at the moment, because VCODE exposes the host's register set, virtual code is not portable.

2.4 ANDF

ANDF [87] (Architecture-Neutral Distribution Format) is the multi-language binary-portable object encoding adopted by The Open Group.

ANDF is a distribution format rather than an execution format; programs are intended to be compiled once on each machine or network by an "installer"; thereafter, the compiled binary is used. To this end, ANDF models language features rather than machine features, and encodes program in a high-level tree structure that resembles an abstract syntax tree. This means that the installer can be rather like a compiler, and can perform all the usual optimizations. Performance is about the same as that of code generated by conventional optimizing compilers [89]. The tree encoding is extensible, and since many optimizations can be performed on it directly, about 70% of the installer code is machine-independent.

By giving freedom for installers to optimize it, the high level encoding is not so good for JIT translation, and dynamic code generation would be slow. In order to be able to use standard system libraries, it contains language-specific constructs, and therefore needs to be extended for each new source language. Also, the installers are rather more complex than most JIT translators, and maintaining them for a wide range of platforms is expensive [89]. The problem is exacerbated by the languages that ANDF supports, C, C++, FORTRAN, and Ada, most of which are not designed for binary portability. For example, the C installer requires special platform-neutral header files, which are then mapped to the target platform's headers; sometimes, further work is required to use native headers with ANDF. Supporting a wide range of targets thus requires much the same effort as supporting a multiple-language native code compiler. Some burden is also placed on the application writer: C applications must define application-specific APIs rather than using conditional compilation for different targets. This does however have the advantage of providing semantic checks that C normally omits.

In summary, ANDF is specialized both with respect to the languages it supports, and in the functions it performs. It is interesting to note that although ANDF is well documented and freely available for many systems, it is not widely used.

2.5 `C--`

`C--` [91] is a portable assembly language based on C, which aims to be a good target for compilers, particularly of garbage-collected languages.

`C--` is a language rather than a VM or a code generation API. This has allowed it to be defined in a familiar manner, and makes it naturally extensible; extensions have been designed for exception handling [103] and accurate garbage collection [93], both of which require intimate interaction between the `C--` and high-level language's run-time systems. `C--` is also more human-readable than the other formats described in this chapter. Its C-like design makes it straightforward to compile with existing compiler technology, and to achieve the same or better code quality (`C--` has special support for constructs such as tail-calls which other languages tend to lack). Being a fully-fledged language, not just an assembler, `C--` is rather more expensive to translate than a low-level VM assembly language, and correspondingly harder to implement, as a full front and back end are required.

As it is a language, code generators need not be written in a language that can call the `C--` implementation directly. This makes it more accessible than competitors such as ML RISC [42] or GNU C's RTL [35], which provide an API instead, which in GNU C's case is not separable from the rest of the compiler. Most of the systems discussed in this chapter have a similar property, though with binary rather than textual formats.

`C--` uses entirely concrete types: its variables, which are mapped to registers and memory locations, are of fixed size. This means that offsets and space requirements are simple to calculate, but does mean that though code generators that emit `C--` can be highly portable, `C--` programs themselves are not.

C, C++ and Microsoft COM object calling conventions are all supported by `C--`, which makes it highly interoperable.

`C--` seems a promising approach for those used to the horrors of writing code generators that emit C, but it is not clear whether it will succeed. The alternative approach of adding a few judiciously chosen structures to a C compiler, and perhaps using a

C preprocessor to ensure that undesirable features of C are not used in `C--` programs might have yielded a solution that was easier to implement. Most worryingly, unlike the other systems described in this chapter, `C--` is still largely vapourware: an implementation has been in progress for some time, but is still not usable.

2.6 TAL and TALx86

TAL [83] is a typed assembly language, a small RISC-like assembly language annotated with static type information that makes it possible to prove statically that TAL programs are type-safe. An implementation for Intel IA-32 processors exists, called TALx86 [82], along with compilers for C and Scheme-like languages. TAL does not aim to provide portable code generation in any sense: the TALx86 compilers generate ordinary IA-32 code. However, TAL's theory is certainly architecture-neutral, and its implications for portable code generation merit its place in this chapter. As will be seen below, TAL can also be considered as a useful adjunct to portable code generation.

TAL was designed to provide a flexible route to safe and certified code. TALx86, which is just a subset of IA-32 assembly language, is trivially language-neutral, and can be optimised conventionally, subject to the constraints placed upon it by the requirements of type safety (TAL's theoretical framework is largely type-based). TAL's type checker is fast, and typically invoked by the compiler on its assembler output, which contains type annotations as comments, before the code is passed to the assembler.

Hence, TALx86 can be used at a very low level: for example, it can be used to verify the output of a JIT translator, which therefore need not itself be trusted. This gives better safety than the JVM, which verifies its byte-code input, but might not translate it correctly into native code. TALx86's type system includes support for type-checking of stack frames, and allows object modules to be type-checked with respect to each other; inter-module references can then be checked at link time. This allows it to be used straightforwardly with conventional separate compilation.

In its present state, TAL is limited in scope: it does not support partial safety in languages that are not type-safe, and has important features missing, such as floating point arithmetic. In addition, certain classes of optimization, mainly high-level code transformations, are forbidden. Most importantly, it does not support portable code, though it could be used in conjunction with a JIT translator to guarantee the type safety of JIT translation. Nevertheless, since TAL's machine model is similar to Mite's, it ought to be possible to define a portable system formally. If the type checks were more flexible, the system designer would also have better control over the compromises between safety, complexity, and speed of verification.

2.7 Other systems

Several other systems are also used as points of comparison with Mite in the rest of the thesis. Since they overlap in aims and functionality with systems described above, it would be battological to describe them in detail; instead, their principal features are sketched, and their interesting points of difference from the systems already discussed are highlighted.

Microsoft .NET VM [77] is similar in scope to the JVM, but aims for broader applicability, in two ways. First, its instruction set is much more language neutral, as it aims to support a wide range of languages. Secondly, its security is more flexible: it allows code either to be verified, in a similar manner to JVM code, or merely validated, which means that it is simply structurally correct, but may not be type-safe. This makes it easier for languages such as C and C++ to target it.

PRACTICAL [28] is a VM that was designed for embedded systems in financial networks. It is designed to be run on small microprocessors, and hence uses threaded code to improve code density; the interpretive overhead of this technique is much less than on more powerful microprocessors. It is not language neutral, but supports C and Forth. Because it is designed for high security networks, it has a very simple and concrete execution model, though it is not formally specified.

Cintcode [57] is a VM designed to execute BCPL, for which it is highly specialized. It has a simple, concrete design, and runs successfully on a wide range of machines, from 8-bit microprocessors to 64-bit workstations. Its emphasis is on simplicity and ease of porting. It has a portable C interpreter, as well as a collection of hand-written assembly language interpreters, each only a few kilobytes in size. The BCPL compiler which runs on it is also simple, performing few optimisations; nevertheless, it performs reasonably well, largely because of its high code density and the small interpreter, which fits in the instruction cache of most modern processors.

Juice [37] is similar to Java, though much less well developed. Its language is Oberon [142], and it aims to provide both smaller binaries and better native code than the JVM. To do this, it uses a binary format similar to ANDF's; the tree-based code structure is both more compressible than a typical virtual code and, since it is a higher-level representation of the program, contains more useful information that the JIT translator can use to optimize the native code that it produces.

PASM [21] is a dynamic code generation system very similar to VCODE, but less well developed; there are no publications about it, and little documentation. It has two notable features: it allows multiple functions to be created in parallel in a thread-safe manner (VCODE allows only one), and it provides an infinite number of virtual registers without the other overheads of ICODE. It has been implemented for the Intel IA-32, which is about as different from its machine model as any workstation microprocessor: it is CISC rather than RISC, allows direct memory operands in most instructions, and has few general-purpose registers. This suggests that PASM's machine model, and hence VCODE's, is applicable to most real machines.

***lightning* [16]** is a dynamic code generation system inspired by VCODE. It aims to be even faster at code generation, largely through having a simpler machine model, in order to support languages such as Smalltalk that rely on frequent incremental recompilation. It is entirely written in C macros, so generates code extremely quickly. Its VM model is brutally simple: it has just six fixed virtual registers, and does not allow functions of more than six arguments.

There are many other systems that provide portable code generation in some form, notably the many VMs in the run-time systems of languages such as Perl [134] and Python [75]. These VMs are in wide use, but not as systems in their own right; they occupy positions near the JVM in the design space. On a different tack, several commercial systems claim to fulfil goals similar to those of Mite, including Elate [117], Omniware [2, 70] and ORIGIN [69]. Unfortunately, information about them seems to be commercially restricted.

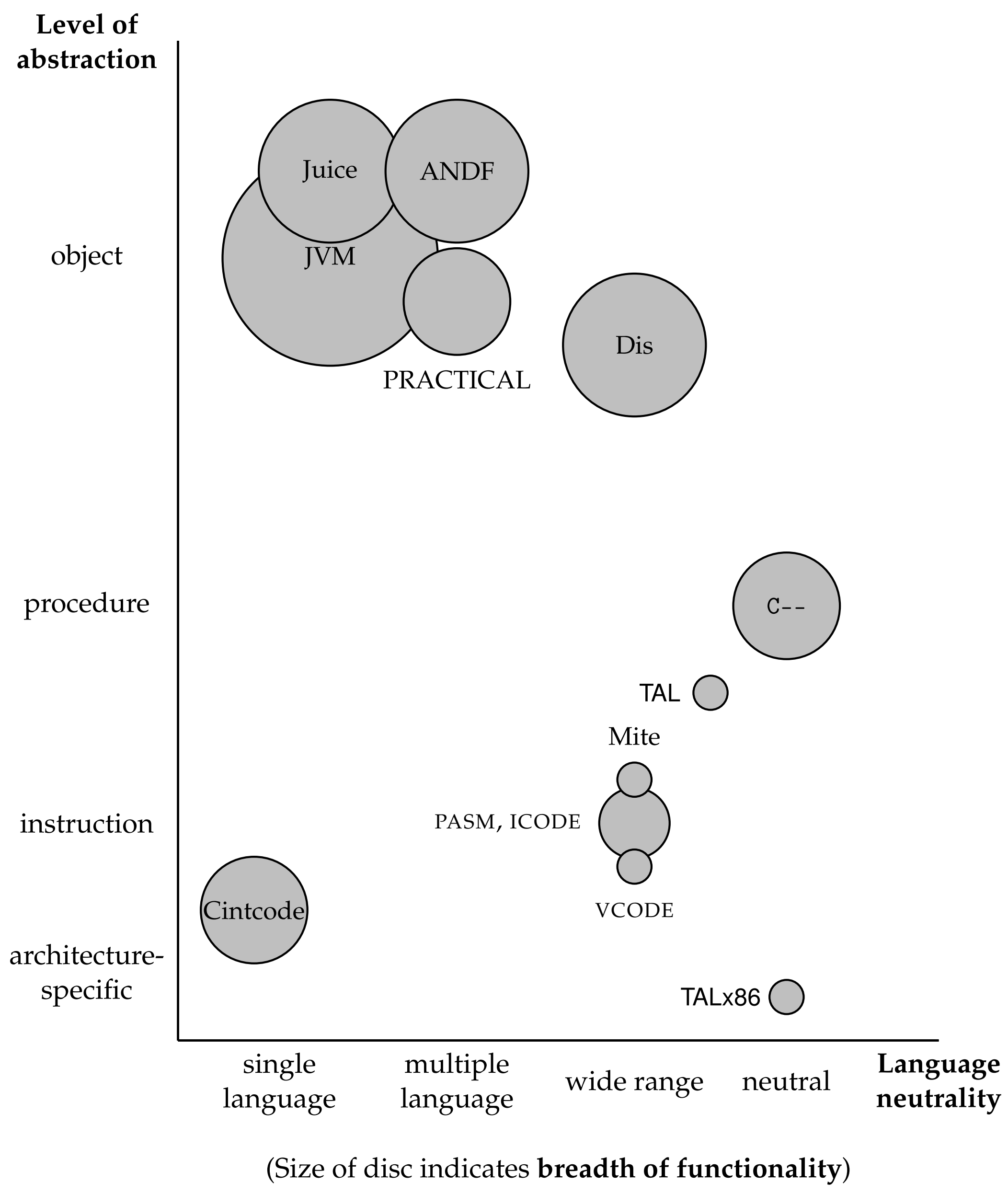


Figure 2.1: Comparison of code generation systems

		JVM	Dis	Juice	PRACTICAL	Cintcode	VCODE	ICODE	PASM	ANDF	C--	TAL	TALx86	Mite
Neutrality	Machine-neutral	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓
	Language-neutral						✓	✓	✓		✓	✓	✓	✓
Closeness to machine	Optimizations in virtual code			✓			✓	✓	✓	✓		✓	✓	✓
	CPU-like VM model					✓	✓	✓	✓			✓	✓	✓
	Direct machine access						✓	✓	✓		✓		✓	✓
Binary representation	Portable binary format	✓	✓	✓	✓	✓				✓				✓
	Intermediate representation	✓	✓	✓	✓	✓				✓	✓	✓	✓	✓
Run-time translation	Dynamic code generation						✓	✓	✓					
	JIT translation	✓	✓	✓		✓							✓	✓
Safety	Sandbox execution	✓	✓	✓	✓	✓				✓		✓	✓	
	Verifiable virtual code	✓	✓	✓						✓		✓	✓	
Additional functionality	Garbage collection	✓	✓	✓										
	Threads	✓	✓	✓	✓									

Table 2.1: Comparison of code generation systems

2.8 Comparison

Figure 2.1 and table 2.1 give some at-a-glance comparisons between Mite and the systems described in this chapter. They necessarily gloss over some of the differences and subtleties discussed above. For example, the JVM and Dis are identical according to the table, and the wide range of VM models is reduced to whether or not each resembles a real machine. In the figure, ANDF and PRACTICAL are shown as being equally language-neutral, though their approaches to language neutrality differ substantially and have quite different tradeoffs.

The figure gives an idea of the relationships between the various systems. It plots level of abstraction on the vertical axis against language neutrality on the horizontal axis. The size of each circle indicates the system's breadth of functionality, which means the level of support for non-computational activities such as multi-threading, garbage collection and security. There seems to be a negative correlation between level of abstraction and the other two qualities.

The table gives an overview of the functionality of each system, according to thirteen features divided into six categories. Some of the headings are worth elucidating:

Optimizations in virtual code Compilers can express optimizations in the virtual code, whether high-level, as in ANDF, or low-level, as in VCODE.

Portable binary format A format for virtual binaries that can be used on any host platform.

Intermediate representation A concrete intermediate representation, whether textual, such as `C--`'s, or binary, such as the JVM's, as opposed to an API such as VCODE's.

Dynamic code generation Run-time code generation by an application, typically of application-specific code.

JIT translation Load-time translation of virtual code into native code.

Sandbox execution The ability to execute virtual code safely, without its being able to attack the host machine, according to some well-defined security policy.

Verifiable virtual code Virtual code whose safety can automatically be verified statically with reference to some security policy.

Most of the systems discussed in this chapter offer portable code generation. Three sorts of portability are involved: that of the code generation interface, that of the generated code, and that of the system itself. Each system strikes a different balance, and differs in its degree of specialization and in the leeway for tradeoff between compiler and run-time translator.

Mite aims to be portable in all three senses: the code generation interface is a machine-independent assembly language, the generated code is stored in a standard binary format, and the machine-dependent part of the system is small. Specialized systems such as the Java VM could be built on top of Mite, though machine-dependent features that it lacks, such as concurrency, would obviously require extra work. Mite makes code quality almost entirely the compiler's responsibility, but simple compilers can easily generate naïve code; higher-level VMs which generate native code directly do not offer this tradeoff. It would also be possible to write a language-independent optimiser for Mite virtual code.

More Mite-centric comparisons are made in chapter 4, but first the next chapter introduces Mite's design.

3 Design

Mite's definition is given in appendices A to C. It consists of three layers: a semantic definition, a concrete syntax, and an object module encoding. Each layer introduces additional constraints; the concrete syntax also adds several features not present in the semantic model of the VM, most of which relate to optimization. This chapter introduces the design from a programmer's point of view, giving example code to illustrate Mite's features.

In this chapter, hexadecimal numbers are written followed by "h"; for example, 100 decimal is 64h in hexadecimal.

3.1 Architecture

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

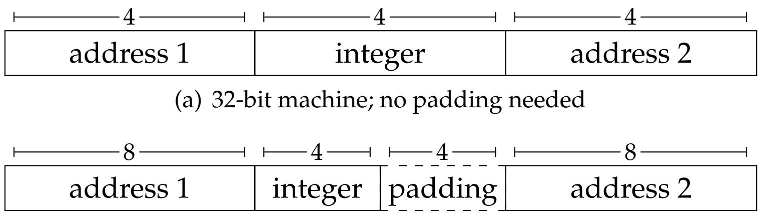
3.1.1 Quantities

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

3.1.1.1 Offsets and sizes

Since words may be 32 or 64 bits wide, and words must always be stored at word-aligned addresses (see section 3.1.2), data structures may be laid out in different ways on different machines. For example, a record consisting of two addresses separated by a four-byte integer would require 12 bytes of storage if A is 32, and 24 if A is 64, as shown in figure 3.1. In the latter case the integer has to be padded to the next eight-byte boundary so that the second address is correctly aligned.

In order to describe the sizes of such structures, and offsets within them, Mite has **three-component numbers**. A three-component number is written as b`@`w`@`r, which has the value $b + Aw/8 + 4\lfloor A/64 \rfloor r$. The second and third components may be omitted if they are zero. b is a number of bytes, and w a number of words. r deals with the fact that



(a) 32-bit machine; no padding needed

(b) 64-bit machine; padding required

Figure 3.1: Layout of a record at different word widths

words may need to be aligned to 4 or 8-byte boundaries. It gives the number of words in the structure being described that would be on 4-byte boundaries if no padding, or "roundings up", were inserted. In the example above, *r* is 1. The value of the third component is 0 on 32-bit machines, and 4*r* on 64-bit machines.

The size of the record described above is `4@2@1`, made up of 4 bytes for the integer, 2 words for the addresses, and 1 rounding up from the integer to the second address. The offset to the second address (third field) is `4@1@1`. If the integer were stored at the end, the size would be only `4@2@0`, and take only 20 bytes on a 64-bit machine.

3.1.2 Memory

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

Memory may be accessed at any width; the address must be aligned to the given width. The effects of overlapping accesses at different widths are undefined; for example, if the two-byte quantity ABCDh is stored at address 14, and the single byte at 15 is then examined, its value may be either ABh or CDh.

3.1.3 Stack

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

The current configuration of stack items is called the stack's **state**; the state is statically determined at each point in the program (see section 3.2.1).

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

Chunks are used for three main purposes: stack-allocated scratch space, passing structures by value to functions and subroutines, and for callee-saved information within subroutines and functions (the "return chunk"). Sections 3.2.6 and 3.2.7 explain the uses of chunks in functions.

3.1.4 Flags

There are four flags: zero (f_Z), negative (f_N), carry (f_C), and overflow (f_V). They are set by each data processing instruction (see section 3.2.3). The flags may only be tested by conditional branches, and must be tested immediately after they are set; see section 3.2.5 for examples.

3.2 Instruction set

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

3.2.1 Creating and destroying stack items

A register is created by the instruction $\boxed{\texttt{NEW}}$. A newly created register is given rank 1. A chunk is created by $\boxed{\texttt{NEW\_}n}$, where n is the size of the chunk, given as a three-component number, as in section 3.1.1.1. Since all stack items occupy a whole number of words, a chunk created with $\boxed{\texttt{NEW\_3@0@0}}$ will occupy either 4 or 8 bytes, depending on the value of A. $\boxed{\texttt{KILL}}$ destroys the top-most stack item.

3.2.2 Register assignment

The instruction $\boxed{\texttt{MOV}\ r\texttt{,}v}$ assigns v to register r. v can be either a register or an immediate constant. An immediate constant is either a three-component number or a label, optionally with a three-component offset added to it.

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

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

`DEF` and `UNDEF` are static declarations, and apply from the point in the program at which they occur to the next `MOV` or `DEF` acting on the same register. `MOV` is an ordinary dynamic instruction, and loads the register with the specified value only when it is executed (though it makes the register variable statically, just like `UNDEF`). Sections 3.2.5 and 3.5 contain examples illustrating the difference between constant and variable registers.

3.2.3 Data processing

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

3.2.3.1 Simple arithmetic

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

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

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

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

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

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

3.2.3.2 Division

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

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

3.2.3.3 Logic and shifts

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

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

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

3.2.3.4 Comparisons

The instructions `SUB`, `AND` and `XOR` may also be used to make comparisons by omitting the destination, so that they affect only the flags. Examples of this use are given in section 3.2.5.

3.2.4 Addressing memory

$\boxed{\texttt{LD\_}w\ x\texttt{,}\ \texttt{[}a\texttt{]}}$ loads the w-wide quantity at the address in register a, which must be a multiple of w, into register x. The quantity is zero-extended if necessary. $\boxed{\texttt{ST\_}w\ x\texttt{,}\ \texttt{[}a\texttt{]}}$ stores the least significant w bytes of x at the address in a.

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

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

```
        KILL
        KILL
        KILL
```

3.2.5 Branching

$\boxed{\texttt{B}c\ a}$ causes a branch to address *a* if condition *c* is true. There are fourteen conditions such as `EQ` ("equals"), `MI` ("minus") and `CS` ("carry set"), plus `AL` ("always"). The address *a* is a label, or a register holding the value of a label. Arbitrary addresses may not be used as branch targets; as mentioned in section 3.1.2, code need not even reside in addressable memory. Stack items which are live at the source and destination of a branch must match, in the sense that they must have the same type, and constants must have the same value.

The following code uses conditional branches and the comparison instructions introduced in section 3.2.3.4 to count the number of ones in register 1.[1] Note that register 2 is used as a counter, so its initial value is `MOV`ed into it, as it is a variable register, whereas registers 4 and 5 are constant, so their values are `DEF`ed.

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

3.2.6 Subroutines

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

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

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

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

A subroutine entry point is indicated by a label preceded by an `s`. The `s` may be followed by `l` if the subroutine is a leaf routine, that is, it contains no `CALL` instructions; this may allow the translator to optimize it. On entry to a subroutine the stack holds the arguments passed to it, with the return address and any other callee-saved information in a chunk on top of the stack.

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

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

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

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

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

3.2.7 Functions

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

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

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

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

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

```
NEW_0@2                        chunk to hold result
NEW                            the variadic arguments
DEF     2, #2
NEW
DEF     3, #4
NEW
DEF     4, #7
NEW
DEF     5, #3                  the number of variadic arguments
CALLFCV .f, 4, 2               pass 4 arguments and receive result in chunk 2
  ⋮
NEW_0                          the variadic argument chunk
NEW                            the argument count
flcv.f                         a variadic leaf function returning a chunk
NEW                            address increment
DEF     4, #0@1
NEW                            accumulator for sum
NEW                            accumulator for product
MOV     5, #0                  set sum to zero
MOV     6, #1                  set product to one
NEW                            pointer to variadic arguments
MOV     7, 1                   address of first argument
NEW
NEW
DEF     9, ashift
SL      8, 2, 9                get address of end of variadic arguments
KILL
ADD     8, 8, 7
```

```
NEW                          temporary
.accumulate
LD_a    9, [7]               load variadic argument
ADD     5, 5, 9              add to sum
MUL     6, 6, 9              multiply into product
ADD     7, 7, 4              increment the address
SUB     , 7, 8               repeat until all arguments read
BNE     .accumulate
KILL                         get rid of temporary
NEW_0@2                      create chunk to hold results
MOV     8, 9                 get address of return chunk
ST_a    5, [8]               store sum
ST_a    6, [8, 4]            store product
RETF    3, [9]               return result
KILL                         kill all items
KILL
KILL
KILL
KILL
KILL
KILL
KILL
KILL
```

As for subroutines in the previous section, the state of the stack at the function label is taken to declare the function's arguments, and the function label itself implicitly declares the return chunk. The variadic arguments to a function come below the normal arguments on the stack. They are stored in chunk 1, which must be declared to have size 0; the format in which they are stored is system-dependent. In this example it does not matter provided that each argument occupies a single word.

The modelling of functions, in particular the treatment of variadic arguments, structure return values and the return chunk, is discussed further in section 4.3.3.

3.2.8 Non-local exit

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

When `THROW` is executed, the stack's state is changed to that of the handler label. Since this may not be the same as at the corresponding `CATCH` instruction, only those stack items which are live at both the `CATCH` and the handler label have a defined value. Moreover, since the values of registers that are cached in physical registers may be lost when a THROW is executed, all registers are assumed to be held in memory at a handler, and the SYNC modifier is provided to save all registers to memory. It has one operand, a handler label, which is used to decide which registers need to be saved. SYNC may be attached to a CALL or THROW instruction, and is only needed when the handler is in the same subroutine or function as the instruction being SYNCed. The workings of SYNC are explained further in section 4.3.6.1.

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

```
.main
NEW
DEF      2, .store                     address of results
NEW
MOV      3, #1                         first value to store
h.handler
NEW                                    address offset
NEW                                    shift
DEF      5, ashift
SL       4, 3, 5
KILL
ST_a     3, [2, 4]
DEF      4, #1
ADD      3, 3, 4                       next value to throw
NEW
CATCH    5, .handler                   get catch value
NEW
MOV      6, .handler                   address to throw to
DEF      4, #2                         second value to store
SUB      , 3, 4                        decide next destination
BEQ      .same                         based on previous result
DEF      4, #3                         third value to store
SUB      , 3, 4
BEQ      .sub
RETF     1, []
.same
DEF      4, #2
THROW    6, 5, 4 SYNC .handler         throw to same subroutine
.sub
DEF      4, #3
CALL     .sub, 3, [] SYNC .handler     in fact, this call won't return
KILL                                   kill remaining items
KILL
```

```
KILL
NEW                                 create parameters
NEW
NEW
sl.sub
THROW    3, 2, 1
KILL
KILL
KILL
d.store
SPACEZ_a 4                          words to hold results
```

3.2.9 Escape

There is a general-purpose escape mechanism: $\boxed{\texttt{ESC}\ \#n}$ performs implementation-dependent function n.

3.2.10 Optimization directives

The instruction $\boxed{\texttt{RANK}\ r, n}$ changes the rank of register r to n. The ranks of the other registers are altered accordingly so that all the ranks are still distinct and in the range 1–N, where N is the number of registers. The mechanics of ranking are explained further in section 4.3.1.2.

$\boxed{\texttt{REBIND}}$ causes the binding of virtual to physical registers to be updated to reflect the current ranking. It is intended for use just before a loop, to minimize spills within. The thinking behind `REBIND` is explained in section 4.3.1.3.

3.3 Data

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

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

3.4 Object format

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

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

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

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

3.5 Pitfalls

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

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

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

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

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

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

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

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

Live ranges may not contain holes Since registers must be created and destroyed in stack order, live ranges may not contain holes. This places restrictions on the order in which compilers can linearize flow graphs; or alternatively, on how tightly they can define live ranges. For example, if a register is live in the test of an `if` statement, and continues to be live in the `then` branch, but not in the `else` branch, then the compiler must either compile the `else` branch second, and kill the register at the end of the `then` branch, or it must allow the register to be live in the `else` branch, even though the quantity it holds is not used. This is the converse of the nesting problem discussed in section 4.3.1.2.

Code sharing is clumsy Since `RET` may only be used to return from the textually current subroutine or function, using a portion of code as part of two or more subroutines or functions is only possible using continuation addresses, which is awkward and inefficient. Therefore, shared code should normally either be encapsulated in a subroutine or inlined. Code sharing is discussed further in section 4.6.

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

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

4 Rationale

The details of Mite's design are now examined to show the reasoning behind them. First, section 4.1 discusses Mite's overall architecture; section 4.2 concentrates on the treatment of registers, its most important single feature. Next, section 4.3 scrutinizes the instruction set. The object format and semantics are examined in sections 4.4 and 4.5. Finally, section 4.6 discusses some shortcomings of the design. Throughout, comparisons are drawn with alternative design decisions in other systems.

4.1 Architecture

Mite's architecture is very like that of a conventional RISC processor. The most important difference from that of a real processor, and those of most other VMs, is that its definition leaves a number of behaviours undefined for the sake of implementation efficiency. This is done in two ways. First, the behaviour of instructions when they are misused (such as attempting to divide by zero) is not specified, which removes the need for costly translate-time and run-time checks to detect incorrect usage. Secondly, when some behaviour of an instruction differs between target machines, it is usually omitted, to keep Mite architecture-neutral. Hence, `MUL` does not affect the condition flags (see section 4.3.2).

This is similar to the way that the ANSI C standard [6], for similar reasons, makes many behaviours "implementation-dependent", meaning that they are not necessarily defined.[1]

Dynamic code generation systems such as VCODE and PASM take a similar approach; VMs that support sandbox execution or verification rely on run-time checks or verification to catch illegal usages (they may also allow the execution of unverifiable code, like Microsoft's .NET common runtime [77]). Cintcode is rare in being almost completely specified; this reflects its heritage of running on 8-bit systems where optimizations of this sort are less beneficial.

4.1.1 Load-store three-operand register model

One of the most important characteristics of a virtual or real machine is the way it addresses data. Mite has three key features in its addressing mechanism: first, its use of registers; secondly its load-store architecture; and thirdly, three-operand instructions.

[1]As opposed to "implementation-defined", meaning that the behaviour must be defined, but may differ between implementations; this is not portable, and hence not useful for Mite.

Most VMs have a simple and restricted set of addressing modes. This is because translating a wide range of addressing modes is a lot of work for a translator, and can impose an unacceptably high cost on instruction decoding, particularly for intepreters. Like VCODE, the fastest translating system discussed here, Mite uses a load-store model, which restricts memory access to load and store instructions, and provides only two memory addressing modes, register indirect and register indirect plus offset. Similarly, Mite also restricts other addressing modes: most instructions only take register operands, and immediate constants are confined to two instructions.

This simplicity speeds up code generation on RISC machines, because it uses only addressing modes that most of them directly support; however, it makes it harder to take advantage of the richer addressing modes found on CISC machines, which Dis and the JVM can better exploit, with their use respectively of memory and stack operands.

However, Mite's registers are easier to map efficiently on to the register set of most modern processors, especially of RISC machines that have a large orthogonal register set, than the JVM's stack items, or Dis's memory locations. Even though registers must sometimes be spilt, and hence mapped to memory or stack locations, it is easier to do this than vice versa. This is because stack and memory locations can normally be accessed indirectly, so when mapping them to registers, aliasing must be detected; since virtual registers cannot be accessed indirectly, they cannot alias one another. Similarly, information about the liveness and importance of quantities, which is vital for good register allocation, is harder to specify for memory and stack locations, because memory locations are too numerous and stack items can be permuted.[2] Mite's register declarations (see section 4.3.1.1) and ranking (see section 4.3.1.2) provide a simple way to give information about the liveness and relative importance of virtual registers. The JVM and Dis do not try to provide such information, and hence need much more complex JIT translators to obtain the same performance.

The use of three-operand instructions may seem to run counter to the simplicity of Mite's addressing modes, by adding an implicit register move to each operation. However, it can equally be seen as a simplification, as instructions are not forced to overwrite one of their operands. For the benchmark programs, the use of three-operand instructions improves the code density of virtual code, though only by 1.8%, and with a standard deviation of 3.4%. On 3-operand RISC machines, where the extra operand is free, there would be a larger saving in native code; while three-operand instructions could be generated from a two-operand virtual instruction set, it would complicate and slow down translation. On two-operand machines, each instruction whose first operand is different from its second generates an extra register copy; however, it is as easy to do this as to generate the register copy from an explicit copy instruction (see section 5.5.2.2).

[2]To avoid this difficulty, the .NET virtual machine [77] forbids the permutation of stack items.

4.1.2 Memory

Mite's memory model is just concrete enough to allow memory access at different widths, and to allow endianness-dependent data to be dealt with where necessary. Unlike Cintcode, Mite cannot be completely concrete, as it must cope gracefully with different word widths and endiannesses (Mite's permutation functions (see section A.3) even cope with machines that are neither big nor little-endian). On the other hand, Mite cannot completely abstract the structure of memory, like the JVM, as it must allow quantities occupying a specific number of bytes to be loaded and stored.

This insistence on laying out memory in bytes rather than allowing an arbitrary ordering of bits is compatible with the vast majority of processors, and allows binary data with a particular byte ordering, such as big-endian network packets, to be dealt with simply. At the same time, the load and store instructions can use the host's preferred byte-ordering. It is quite possible for a Mite program to discover the byte ordering of the machine on which it is running, and then use optimized routines that assume a particular ordering; it is almost always reasonable to assume that memory is big or little-endian. Once again, Mite puts the tradeoff between efficiency and portability in the hands of the programmer. Most other systems that aim strongly for efficiency, such as VCODE and `C--` also use the native bit ordering (though not necessarily specifying that it must be a byte ordering). The JVM and Dis say nothing about byte ordering; in order to remain flexible, the JVM therefore has a large range of primitive types, and dealing with anything else is problematic. Dis attempts to remain simple, and therefore lacks some commonly used types, such as 16-bit integers. Mite has neither problem.

The assumption of a linear address space whose addresses can be held in a machine register is of a similar quality: it matches most real machines, while keeping the programming model simple and efficient. If Mite were to allow 16-bit registers, or registers larger than 64 bits, this assumption might change, but 32-bit machines tend to have 32-bit address spaces, and 64-bit machines 64-bit address spaces. VCODE and PASM take the same approach, while the JVM, by avoiding explicit pointers, avoids the problem altogether.

Code memory is organized in a different way: as the length, format and meaning of code varies between machines, there is no compelling reason to make it addressable as data. Allowing separate code and data address spaces means that Mite can be implemented on Harvard architecture machines, and makes it easier to build more secure systems on top of Mite in which code cannot be read or written, without changing Mite's semantics. Nevertheless, code is addressable via locations (see section B.9), so it is possible to build branch tables; also, it is of course possible to write Mite code that directly manipulates program code in a known format, when it is held in data-addressable memory, as it usually will be.

Mite's memory model therefore has most of the advantages of a fixed memory model such as Cintcode's; indeed, at the cost of portability, a fixed memory model can be assumed in Mite code. At the same time, Mite remains architecture-neutral.

4.1.3 Three-component numbers

Three-component numbers were introduced to allow compilers to optimize accesses to fixed array elements or record fields, even when their size is not known at compile time. The example in section 3.1.1.1 shows how this can be achieved for a record consisting of two words (which could be 32 or 64 bits) and a 32-bit integer. In particular, it shows the use of the third component in rounding up from 4-byte to word boundaries.

For arrays, three-component numbers can be used to access an element whose index is known at compile time. Calculating the address of run-time determined indices is made easier by the `ashift` constant (see section 3.1.1); an index into an array of words can be turned into an offset by the following code:

```
                                 register 1 holds the index
NEW                              create a register to hold the shift
DEF 2, ashift                    set the shift
SL 1, 1, 2                       turn the index into a shift
```

Three-component numbers are just one point in a spectrum. At one end, Cintcode has a single fixed-size datatype, the 32-bit word, and therefore works less efficiently on machines with different natural sizes. In VCODE and `C--` the sizes of data structures are known (and in `C--`'s case, specified) at compile time, but the generated code is not binary portable. Next come systems that have mostly fixed-size datatypes, and pointers whose size cannot be explicitly mentioned in virtual code. Dis, the JVM, and ANDF all calculate the sizes of data structures at load time, and therefore offsets to known fields must be calculated at run time, or special array access instructions must be used. Mite is unique among the systems discussed here in allowing compile-time calculation with quantities whose actual value is not known until run-time.

Still, Mite is not at the end of the spectrum. Constants could be polynomials in the word size: this could be used to express as a manifest constant the size of a word array which had as many members as there were bits in a word. However, an additional component would then be needed for each possible rounding up; for example, if a 16, 32 or 64 bit word size was allowed, then a rounding up from 2 to 4 byte boundaries would be required. Finally, to allow all possible offsets in a language such as C to be calculated at compile time in a fully portable manner, an extra component would be required for each independently-sized type, such as `char`, `int`, `long`, `void *` and so on. A compiler could end up calculating with linear combinations of a dozen variables where originally it worked with constants. This is clearly unacceptable; Mite's scheme keeps the complexity to a minimum, while allowing the commonest optimizations to be expressed.

One point of particular interest to C compilers is that since three-component numbers are manifest constants, they can be returned as the value of `sizeof`, which can therefore be used as normal, even in constant expressions.

Finally, note that it is of course possible for Mite code generators to delay calculation of offsets to run time, or to assume a fixed word size, and thereby trade compiler simplicity for loss of portability or run-time speed. In any case, a minimal overhead is imposed on translators: since the translator knows the machine word size, it can turn three-component numbers into ordinary constants as they are decoded.

4.2 Registers

The treatment of registers is the linchpin of Mite's design: most of its goals and constraints converge on this one element. As noted in section 4.1.1, how a VM addresses memory is an important characteristic. It is also one of the ways in which VMs tend to differ most.[3] Hence the JVM may be described as a stack machine, Dis as a memory-to-memory machine, VCODE as a register machine, and Cintcode as an accumulator machine.[4] As discussed in section 4.1.1, Mite is register-based in order to map easily on to current machine architectures. This implies that Mite's registers should correspond as directly as possible to machine registers, and this is exactly what Mite's method of handling registers aims to ensure.

There are three main differences between the register sets of different processors: first, the size of registers (usually, they are all the same size); secondly, the number of registers, and finally, the uses to which each register may be put: some processors assign special rôles to certain registers. The last of these varies too widely to be dealt with in a general way; each translator must deal with it ad hoc. The first is dealt with by allowing two register sizes, as discussed in section 4.2.4. The second is by far the most difficult to deal with: the only practical way seems to be to allow an unlimited number of virtual registers (section 4.2.1), which in turn introduces further problems, which are the subject of the rest of this section. The handling of registers is also affected by the varied ranges of immediate constants on different machines (section 4.2.5), and by argument and result passing under system calling conventions (section 4.2.2).

4.2.1 Unlimited virtual registers

Since supporting unlimited virtual registers creates some of the trickiest problems in Mite's design, it is necessary to justify their use in the first place.

There are at least four possible candidates for the "right" number of registers:

Few Have only as many as the most register-starved architecture (say, 6 for IA-32). GNU *lightning* [16] does this, because it speeds up translation by allowing a fixed mapping from virtual to physical registers, and still gives adequate native code quality. However, *lightning* is aimed at dynamic code generators which typically apply few optimizations, and hence tend to use few registers. GNU C (`-O2`) needs to spill for most programs even with 10 registers, as on the ARM,[5] and figure 6.2

[3]The same used to be true of processor architectures, but in recent years the majority have converged on the load-store register model.

[4]In fact it has two general-purpose registers that behave like a two-element stack.

[5]Admittedly, GNU C does not have a graph colouring register allocator, so its output is not the acme of register allocation.

shows that performance of Mite-generated code increases as the number of registers is increased, at least up to 10.

Enough Have as many registers as are needed. 16 or 32 might be plausible numbers. But since some programs will still cause spilling, both compilers and translators must cope, independently, with register spilling. Worse, compilers must spill virtual registers, while translators spill physical registers; these two types of spilling could easily interact badly. VCODE effectively takes this approach by allowing the physical registers to be claimed and released. This however means that, in general, different virtual code will be generated on machines with different numbers of physical registers; to obtain portable code, the minimum number of registers must be assumed once more.

Many Have a large fixed number of registers, say 256 or 1,024. Compilers could then reasonably fail if they run out of virtual registers, or generate much poorer code. Nevertheless, this is just the previous option with different tradeoffs, and special-case code is still required in both compiler and translator, even if only to detect the limit being exceeded; also, building in a fixed limit does not seem satisfactory, and could pose problems for machine-generated code, such as the output of compilers that use C as a target language.

Unlimited Have an unlimited number of registers. This simplifies code generation, though at the same time it subtly alters the notion of what a register is, as a compiler can use as many registers as it likes, although as with Mite, register assignment and co-location may still be an issue (see section 4.3.1.1). The translator is more complicated than with a minimal number of registers, but no worse than for the other options, as from a translator's perspective the number of physical registers is fixed, as is the number of virtual registers in any given program. To achieve a good translation the virtual registers must be ranked, but this is also true for the previous two alternatives.

The last option is the most aesthetically attractive, gives the best potential performance, and is no worse to implement than other options that give good performance. Stack machines (virtual and physical) effectively adopt this solution, but with the overheads that come with using a stack rather than registers.

4.2.2 Stack

Mite's stack is its most complex structure. From the code generator's point of view it combines virtual registers, argument and result passing, and stack frames. From the translator's point of view it supplies the information necessary to arrange physical register allocation and spilling, and to perform function call, entry and return. These functions are interrelated in most systems via the system calling convention. Hence, as at several places in Mite's design, the twin requirements of portable virtual code and efficient native code force a certain degree of complexity. Mite must match the machines it targets, so these features are best combined into a single structure.

The stack, therefore, combines the following elements:

Unlimited number of registers A stack allows an unlimited number of registers without needing to specify the total number used in advance, or a fixed maximum.

Register live ranges Stack elements can be popped as well as pushed, so registers' live ranges can be delimited, though registers must be created and destroyed in stack order. This is a reasonable restriction, as most virtual registers correspond either to temporaries which are live for the evaluation of a single expression, or register variables, whose live range corresponds to a nested block (either lexical or dynamic), which itself obeys a stack discipline. In any case, virtual registers can be reused (see section 4.3.1.1).

System stack The system stack, whose requirements generally correspond to those of conventional languages such as C and Pascal, tends to hold a series of frames for active procedures, which contain a saved program counter and perhaps other registers, local variables, and incoming and outgoing procedure arguments. It is also used for block-local storage, such as temporary data areas, and spilled register values. Modelling the system stack in the virtual register stack simplifies the spilling of virtual registers (see section 4.3.1.1), and makes virtual register sets correspond to stack frames (section 4.2.3 explains why this is desirable).

Function calling Certain virtual registers correspond to incoming and outgoing procedure arguments and return values; chunks allow structures to be easily passed by value. Making arguments and return values correspond to the top-most virtual registers on the stack makes it easy for the translator to move arguments and return values into the right place for function calls; usually it can be arranged so that they are already in the correct virtual registers when the call or return instruction is reached.

Stack allocation Chunks allow limited stack allocation (since the size of block allocated must be statically determined, they cannot be used to implement C's `alloca`), suitable for run-time scratch space, local structures and arrays, as well as passing structure arguments by value, as mentioned in section 3.1.3.

Most other systems fall into one of three categories: either they use a processor-like model that splits stack and registers, like Cintcode, VCODE, PASM and TAL, or they use a computation stack as a simple unifying structure, as the JVM does. Dis goes one step further by eschewing temporary storage in favour of using only memory locations. Finally, systems such as ANDF and Juice avoid referring to temporary storage by describing computations with trees.

4.2.3 Numbering

Mite allows an unlimited number of registers, as discussed in the previous section. Most processors have global register numbers, but this is useful only because a given number always refers to the same physical register. With an unlimited number of virtual registers, it is necessary for good register allocation to be able to map virtual registers to different physical registers, so the advantage of global numbering is lost. Globally numbered registers are also difficult to spill without wasting time and memory if they are spilt to some global area, and are tricky to spill locally. Frame-local numbering allows efficient, simple spilling to the current stack frame: the procedure is exactly the unit within which stack offsets to spill locations are statically determined. This is why there is no point making the numbering finer grained (for example, block-local).

The fact that registers must be explicitly passed to and returned from subroutines and functions may seem like a disadvantage of using local numbering. In fact, to allow subroutine and function calls to be both portable and efficient, it must be possible to use different calling conventions on different machines, and in particular, to support each system's function calling convention. Hence, the number and type of arguments passed to each function needs to be specified. Since the function's type may not be known at the call site (for example, if it is declared later, or in another module), and since in any case most C calling conventions permit functions to be called with different numbers of arguments, the number and type of arguments must be specified by the call. Actually, the type is already known (because the type of each stack item is known), so only the number of arguments needs to be given in the `CALL` instruction.

Another consequence of local register numbering is that registers declared in one subroutine or function may not be accessed by its callees. Allowing such access would complicate the design (for example by adding a notation to name variables in an outer frame) with little return; it is in any case possible to access registers in outer stack frames via displays or static chains, as discussed in section 5.4.1.

Most other systems either lack explicit registers, like the JVM, directly expose the machine's register set, as VCODE does, or have a fixed small set, like Cintcode and GNU *lightning* [16]. ICODE and PASM use a similar approach to Mite. The JVM and PRACTICAL impose their own calling conventions, which are therefore simpler to use, but require glue to interwork with native libraries. VCODE, ICODE and GNU *lightning* abstract from the native calling convention (though *lightning* in particular places restrictions on the sort of calls that may be made, forbidding more than six arguments, and not allowing structures to be passed by value). These are all rather more complicated to use than Mite, but allow slightly more efficient code to be generated, by revealing more details of the way that arguments and return values are marshalled.

4.2.4 Sizes

Mite restricts registers to a single size from a choice of two: 32 or 64 bits. These match the word size of most modern microprocessors, while simplifying address calculations and the difficulties of performing word-size-independent arithmetic (see section 4.1.3). With a little care registers can be treated as 32-bit words most of the time;[6] in the worst case only two versions of code are needed, one for each word size. The only major problem

[6]The results of most operations can be converted to a 32-bit result by simple truncation; only division and right-shifting need special attention, as the contents of particular digits of the operands can affect the contents of less significant digits of the result.

is the lack of machine-independent 64-bit arithmetic, but this seems to be rarely used. If it is desired, it can be implemented using two sets of code, one for 32-bit systems, and one for 64-bit systems; the correct code can be chosen at run-time.

Most other VMs use completely fixed types, like the JVM, Dis and Cintcode; some, like VCODE, also allow access to the natural machine word. Having a range of types is more flexible than Mite's solution, but requires the translator to generate special code for types not directly supported by the machine (for example, 64-bit arithmetic on 32-bit machines), and type conversion code.

4.2.5 Constants

Since the range of immediate constants varies between processors and even between instructions and addressing modes on the same processor, compilers cannot know when to use an immediate constant and when to load the constant into a register. Simplicity demands that Mite adopt one or the other mechanism uniformly. Tying a constant to a virtual register allows it to be assigned permanently to a physical register if it does not fit in a particular immediate field, rather than forcing it to be loaded repeatedly. The `DEF` instruction (introduced in section 3.2.2) declares constants with static scope so that they can be more easily optimized, as the translator knows exactly where a constant register's value is fixed.

Support for constants is limited in other systems: while most have some form of constant, these tend either to be ordinary immediate constants, or global constant values; no other system discussed here gives constants a live range as Mite does.

4.3 Instruction set

The instruction set's main features are:

Minimal abstraction The instruction set allows Mite programs to be portable, while making the translation into native code trivial for the majority of instructions on most machines.

Few instructions The instruction set follows the common observation that complex instructions are rarely used, and provides just enough instructions to permit good code to be generated, rather than attempting to exploit CISC instruction sets directly (though a sophisticated translator may do so by techniques such as peephole optimization; see sections 5.5.2.1 and 6.2.3.2).

Three-operand instructions Most instructions take three operands, as discussed in section 4.1.1.

Restricted addressing modes Immediate constants and register indirect are confined to special instructions, and these are the only addressing modes (other than register immediate; see section 4.1.1). The restriction on the use of immediate constants is discussed in section 4.2.5.

Unlike many VM instruction sets, Mite's is untyped. Typing is generally provided to aid safety, as in the JVM, and to allow the correct code to be generated for different types of quantity, as in VCODE. Most instruction sets contain many more specialized instructions, such as the JVM's Java method dispatch instructions. Even VCODE has `hton` and `ntoh` for changing host byte order to network order and vice versa. Mite omits them in the interests of language neutrality and simplicity. Unlike ANDF and Juice, whose virtual instruction sets are focused on programs, Mite's is focused on the machine. In this respect, it is very like `C--`, despite the superficial difference that `C--` looks like a high level language while Mite is a virtual assembly language.

The rest of this section first discusses some more specific features: register management, flags, function calling, division and the escape instruction (`ESC`). It ends with an examination of some features that at first sight might seem superfluous.

4.3.1 Registers

Section 4.2.1 discussed the need for an unlimited number of registers; here we examine the instructions needed to support their use. There are two main aspects to register management: creation and destruction, and ranking. Finally, the `REBIND` instruction is explained.

4.3.1.1 Creation and destruction

The creation and destruction of registers performs three functions: it defines the live ranges of virtual registers, declares subroutine and function arguments, and gives the types of subroutines and functions.

This lumps a lot of functionality together, and omits some obvious distinctions. Sections 5.2.3.2 and 6.3 discuss the disadvantages of not distinguishing temporaries from register variables, and not identifying function arguments. The use of `NEW` and `KILL` to give the types of a function or subroutine's arguments may seem odd at first, but it simplifies the translator, by not requiring extra code to read and construct a new virtual stack configuration for each function and subroutine entry point.

The `NEW` instruction is also used to create chunks, which were discussed in section 4.2.2. Using `NEW` and `KILL` for chunks, and using the same numbering scheme for registers and chunks, simplifies stack allocation in the translator: spill slots for virtual registers can be allocated contiguously in the stack frame, along with chunks. If such a scheme is used, then the space used by a stack item cannot be reused until all the items above it have been destroyed. This leads to the requirement that items are `KILL`ed in stack order (as mentioned in section 4.2.2), which again simplifies the translator. Note that the abstract semantics of KILL allow any stack item to be destroyed (see section A.5.7). Similarly, chunks are forced to have a statically determined size so that the address of each spill slot within the stack frame is known statically, and code to access stack slots can be generated by the translator in a single pass.

4.3.1.2 Ranking

Register ranking is perhaps Mite's most important innovation. It enables virtual register allocation to be performed by the compiler, and in particular, allows any register allocation algorithm to be encoded so that it can be performed in by the translator in time roughly linear in the length of the program.

The key problem that ranking attempts to overcome is that, whereas a native compiler knows how many registers it has to allocate, and can thus perform register allocation and assignment accurately, a code generator targeting Mite does not know. This makes virtual register *allocation* trivial: all quantities that can occupy a register may do so. Virtual register *assignment* is trickier: although the supply of virtual registers is unlimited, performance is improved by minimizing the number used, and by arranging registers on the stack so that their live ranges nest as well as possible (see section 6.3.3).

Physical register allocation and assignment, however, become rather more difficult. There are several problems. Fundamentally, the difficulty is that register allocation is usually performed entirely by one program, either the compiler, in direct native code compilation, or by the translator, where the compiler generates virtual code (for example, Java JIT translators must perform register allocation and assignment). However, good register allocation and assignment are expensive: traditional algorithms such as heuristic-aided graph colouring have at least quadratic cost [131], and more recent algorithms trade off performance against running time [52, 100, 131]. A Mite translator performing full register allocation and assignment would therefore have to make the same tradeoff. Hence it is necessary to find some way for the compiler to do most of the work, allowing the translator to use a quick and simple register allocation algorithm to obtain a good result.

This leads to another difficulty: traditional algorithms assume a constant number of physical registers, and perform spilling in tandem with allocation. Mite must allow for all possible numbers of physical registers at compile time. Furthermore, register allocation is not necessarily stable: an allocation for 8 registers might well bear no resemblance to that for 9.

A register allocation algorithm can be turned into a ranking algorithm as follows: perform register allocation assuming that there is only one register available. Then, fixing this allocation, run the algorithm again, this time with two registers. Continue until all the virtual registers have been allocated. The order in which virtual registers are allocated to physical registers at each point in the program then gives their ranks.

Fixing the allocation after each pass gives stability, but means that the allocation is not necessarily as good as the algorithm can achieve. The allocation is also limited by the fact that stack items must be killed in stack order, and hence live ranges must be nested (though this limitation could be removed; see section 7.1.1.1). However, the success of basically linear algorithms such as [100, 131], which are perforce stable, suggests that requiring stability need not mean a huge drop in code quality. By this method, register allocation algorithms can be encoded directly using ranks, without needing explicit support in the translator, so Mite can take advantage of improvements in this field not only without changing its design, but without changing its implementation. This ability seems to be unique to Mite.

A naïve compiler can simply ignore ranking. In this case, virtual registers are allocated to physical registers in the order in which they are declared. This is what the LCC Mite back end does (see section 6.2.2), and even then, the code quality is not disastrously poor.

Other systems rely on either an intelligent translator, like the JVM and Dis, or ad hoc mechanisms, like VCODE, whose dynamic code generation interface forces its clients to manage register allocation and spilling. ICODE is closest to Mite: it allows its input to be annotated with information about usage frequency of code, and then performs its own live range analysis and register allocation. It is rather more complex than Mite, while allowing less communication with the compiler.

The quantitative effects of ranking on execution speed are discussed in section 6.1.2; section 6.1.3 discusses their effect on code density.

4.3.1.3 `REBIND`

Several different schemes for indicating the relative importance of different parts of the program were considered. The aim was to enable the translator to generate the best native code for the parts of the program executed most frequently. Most of these schemes involved a code priority being attached to each basic block. This seemed awkward to implement without global analysis of the Mite object code, which is typically slow and hence runs counter to Mite's goal of fast translation (but see section 7.2.3).

The problem can be simplified: without inter-function optimization, the only reason for one section of code to be executed more frequently than another is that it is in a more often executed loop. The translator is not concerned with optimizations such as finding invariants or unrolling; these are the compiler's job. The translator's main concern is physical register allocation and assignment. In straight line code it does not matter in what order register allocation is performed, nor where spill code is placed. In the presence of loops, however, it makes sense to perform register allocation for inner loops before outer loops, to give the translator more freedom on code that will be executed more often. Also, spills and restores should be moved out of loops wherever possible.

Performing register allocation for inner loops before outer loops is not easy in the current design; a possible mechanism is discussed in section 7.2.3. The `REBIND` instruction (introduced in section 3.2.10) is a simple way to move spills out of loops. Instead of trying to move spill code once it has been generated, a `REBIND` hints that the mapping of virtual to physical registers should be brought up to date at that point. Hence, virtual registers that happen to have a lower rank than the number of physical registers are spilled, while virtual registers with a high rank that are not currently assigned to physical registers are reloaded. This register traffic takes place only once, outside the loop, and the loop is entered with the best possible register binding (assuming that the ranks are optimal). Then, spills and restores should only be generated in the loop if more virtual registers are used inside the loop than can fit in physical registers, when spilling is inevitable anyway.

The quantitative effects of `REBIND` are discussed in sections 6.1.2 and 6.1.3.

4.3.2 Flags

Flags are normally computed as the result of an arithmetic or logical operation. They are mostly used to decide the outcome of conditional branches, although their value may be used directly, as when the results of several comparisons are combined, or the carry out of an addition is used to perform multi-word arithmetic.

The implementation of flags varies widely. The Intel processor has a dedicated flags register while the Alpha writes the result of comparisons to a general purpose register specified by the instruction. The Intel sets the flags after each instruction while many other processors do not; the ARM allows any instruction to be executed conditionally on the contents of the flags, while most processors provide only conditional branches and compare-and-set instructions.

Mite's flags model is compatible with all these implementations. There is a virtual flags register, which implements the four commonest flags: zero, negative, carry and overflow. Each instruction's effect on the flags is compatible with most common processors; where their behaviour differs, Mite's is undefined: for example, most processors agree on how all four flags are set by addition, so `ADD` has a defined effect on all four flags, whereas there is little agreement about multiplication, so `MUL` has a completely undefined effect on the flags. Use of the flags register is heavily restricted: it may only be read by a conditional branch occurring immediately after the instruction that set the flags. Hence, on machines that lack a flags register, there is no need to simulate the virtual flags register at run time. Instead, a temporary register can be used to store the result of the instruction before a conditional branch, and then released immediately after the branch. In addition, many common comparison and branch pairs can be implemented as a single compare-and-branch instruction (as available on the MIPS, for example).

Most systems, like the JVM and VCODE, use compare-and-branch instructions rather than an explicit flags register. Section 7.1.3 discusses modifying Mite to use this approach.

4.3.3 Function calling

In order to be able to interwork with system calling conventions, which are typically geared to C, Mite needs special call and return instructions, and special function labels, as described in section 3.2.7. The main features that these are needed to support are:

Callee-saved information Since many calling conventions save certain machine registers along with the return address, entering a function causes a chunk of indeterminate size to be placed on the stack directly above the arguments. Since the contents of the chunk is largely system-dependent, it is not specified (except that it contains the return address), and writing to it is prohibited. This can cause problems for programs that wish to inspect the contents of stack frames (see section 7.1.2).

Variadic functions Calling conventions often treat variadic arguments differently from normal arguments: for example, they may always be passed on the stack, even if argument registers are available. Because of this, and in order to allow a native function to access variadic arguments passed to it by a Mite function, the layout of variadic arguments must be system-specific. This is unfortunate, as it means that in general there is no portable way for Mite functions to read variadic arguments. However, since function arguments tend to be passed as a series of words, all this means in practice is that the stack direction of the host machine must be computed at run time, to discover the order in which the variadic arguments are laid out within the chunk.

Structure-returning functions Like variadic arguments, structure return values are treated specially by most calling conventions, but not uniformly, so they need special annotation in Mite code. If the address at which the result is stored is determined by the caller, then the return item specifier given to the `CALLFC` instruction can be passed to the function; otherwise, the necessary manipulations can be performed at the call site. Some rare conventions are not supported by Mite's scheme: for example, the ARM Procedure Call Standard [9] allows small structures to be returned in a register rather than on the stack under some circumstances. Fortunately, this option is not widely implemented by compilers precisely because it makes calling between native code from different sources error-prone.

4.3.4 Division

The `DIV` instruction is intended to work well with both hardware and software division. It takes advantage of the fact that software division routines usually calculate both quotient and remainder, while allowing optimization when using hardware division which usually calculates one or the other. Two types of signed division are provided because both are used. Mite is rare in providing this degree of flexibility and precision: the JVM provides only rounding-to-zero division, and neither Dis nor VCODE documents the type of division provided.

4.3.5 Escape

`ESC` is a general-purpose trap-door instruction. Its main purpose is to allow system calls to be made inline: since the function number is encoded in the instruction, it can be directly translated to similar machine instructions such as the ARM's `swi` [56] and the Motorola 680x0's `trap` [140]. However, there is nothing to stop implementations using it in a more portable manner, for example to access functions in a run-time system; it is sometimes more convenient to do this by number than by name, using ordinary branches.

There is no set mechanism for passing parameters to `ESC`. In the ARM implementation of Mite, the given system call is performed with whatever is the current contents of the physical registers; by knowing how the translator allocates subroutine parameters and results, it is straightforward to use ESC by wrapping each invocation in a subroutine. An alternative, presented in section 4.6, is to pass parameters to ESC in the same way as calling a function. In the case of the ARM, this avoids loading and storing ten registers around system calls that only use one, for example.

Several other systems provide comprehensive I/O libraries. The JVM as part of Java and Dis as part of Inferno are exemplary in this respect.

4.3.6 Seeming superfluities

4.3.6.1 CATCH and THROW

CATCH and THROW may at first seem rather high level constructs to implement in Mite, but they are in fact the only mechanism for non-local return from a subroutine or function. In addition, as demonstrated in section 5.4.2, they can be used to implement most common styles of exceptions in high-level languages.

The reason for such high-level primitives is that non-local return involves unwinding the stack. Although this is generally implemented by simply setting the stack pointer to a previously saved value, then branching to the return address, in Mite it must be handled rather more delicately.

Most of the difficulties arise from Mite's register stack. There are three main problems. First, the physical to virtual register binding must be restored correctly when a handler is reached by a THROW. Secondly, unlike a CALL instruction, which calls a routine with a known return type, a handler may be reached from anywhere, with different types of return value. Thirdly, the stack pointer must be reset correctly by a THROW, a tricky operation when the virtual register stack is taken into account.

When a RET is executed, the physical register set current at the return site must be restored, and any return values written to the correct registers and memory locations. The same process must occur when a THROW is made to a handler label. However, while a RET instruction returns to code just after a CALL, which can restore caller-saved registers and store results, a THROW instruction goes straight to a handler label. Hence, the label itself must cause native code to be inserted to deal with the result passed to the THROW, and the virtual to physical register binding. This is easily achieved by having a standard register binding enforced at handler labels, with just the return value mapped into a register, and all other virtual registers spilt. It is most natural for THROW to pass its result in a register rather than on the stack, since it alters the stack pointer as part of its operation. This method has the further advantage that handler labels may also be reached by a normal branch, or by falling through from the previous instruction (provided that the translator inserts code just before the handler label to spill all the virtual registers, except the one that is normally overwritten by the result, which should be moved into the appropriate register). This allows a natural implementation of C's setjmp and longjmp (see section 5.4.2.1).

There is one more piece in the puzzle of ensuring the correct virtual to physical register binding when a handler label is reached: the virtual registers that are assumed to

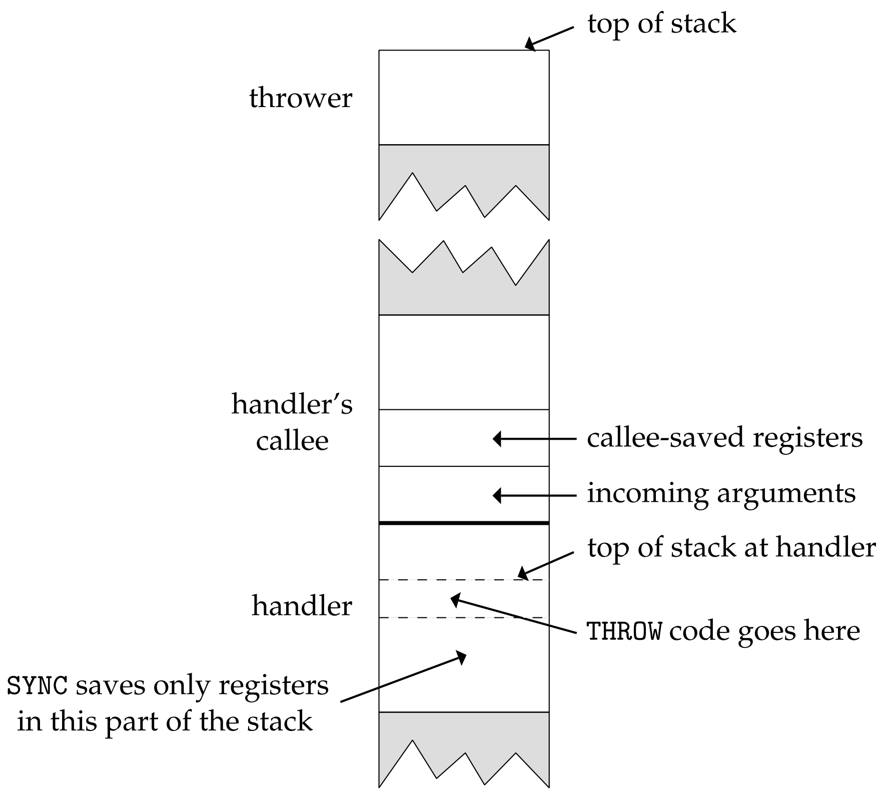


Figure 4.1: Throwing from one stack frame to another

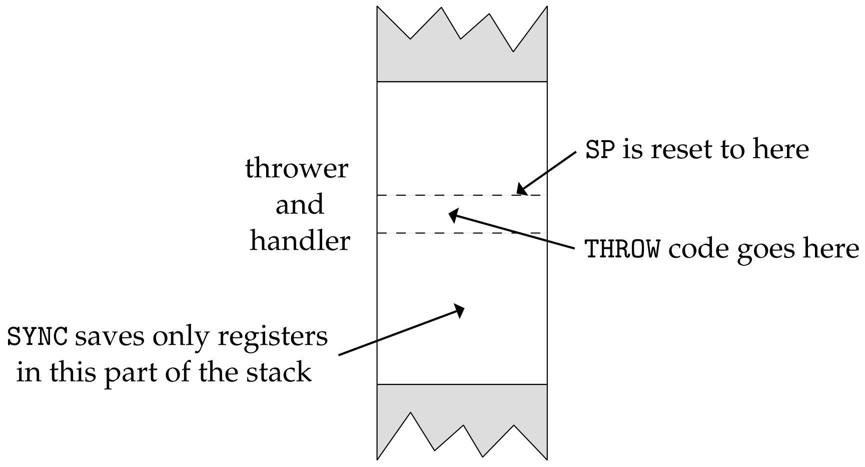


Figure 4.2: Throwing within a single stack frame

be spilled when the handler label is reached must indeed be spilled, and to the correct location.

To see how this is done, let the routine that executes the THROW be called the 'thrower', and that in which the handler label is found the 'handler'. The two possible situations are illustrated in figures 4.1 and 4.2: either the thrower is the same routine as the handler, or it is deeper in the call chain. Clearly, the registers live at the handler label must have been spilt by the time the THROW is performed. If the thrower is the same as the handler, this is easy: the THROW instruction itself can perform the necessary spilling. In order to minimize the amount of spilling, this is done by means of a SYNC annotation, which effectively enforces a caller-saves calling convention; indeed, it is superfluous if the system calling convention is purely caller-saves.[7] Only registers live at both the THROW instruction and at the handler label need be spilt. If the thrower is not the same as the handler, it is not possible to wait until the THROW instruction to perform the spilling: the thrower has no way of knowing which registers need to be spilt and where their current values reside. Instead, the SYNC annotation is placed on the CALL that leaves the handler.

Whether it is used on THROW or CALL, SYNC is optional: for THROW it need only be used when the handler could be the same as the thrower; for CALL, when the callee (or any more deeply nested callee) could THROW to the caller.

The second problem, dealing with the return type of a THROW, is much simpler. At a normal RET, the return type is known; handlers, however, may be "returned" to from anywhere. Mite's solution is to fix the "return type" of THROW to be a single register value. If more than a single value needs to be passed, the value can be a pointer, and the actual result placed in the heap, or deeper in the stack.

The third problem is to ensure that the stack pointer is correctly reset by a THROW. A CALL instruction implicitly saves the current value of SP, to be restored by the corresponding RET. Obviously this is not possible for THROW, where the return site is not known, nor who will return to it. Hence saving and restoring SP for non-local return must be done manually. Unfortunately, it is not possible just to read the value of SP, and then write it back later. First, making SP directly readable and writable would involve devising rules for its use in portable code, which would be hard to get right, as its use would have to be heavily restricted. Secondly, how would one calculate the value of SP needed at the handler label from elsewhere in the function? The value of SP will typically vary during a function as virtual registers and chunks are created and destroyed. Hence, the CATCH instruction is used to obtain the correct value of SP, by allowing the translator, with its knowledge of the generated code, to calculate it.

For similar reasons, THROW is needed. It may seem to be merely an abbreviation for:

```
                    return value in 1
MOV SP, 2           set SP
BAL 3               branch to handler
```

[7]SYNC is similar to C--'s cuts to annotation. This is used at a call site to specify variables that are live at other points in the procedure, which may be reached by a non-local return from the call about to be made. SYNC only allows one such point to be specified.

but it covers up some nasty surprises: what if having executed the first instruction it turns out that register 3 in the next instruction is currently spilt? It has to be reloaded from its spill location, probably on the stack, but that is no longer accessible, as `SP` has already been reset to the value it takes at the handler.

Having obtained a model of non-local return that is rather high-level compared with the rest of Mite's instruction set, it must be demonstrated that it is expressive enough. As the names "`CATCH`" and "`THROW`" suggest, the aim is to support exceptions; this is demonstrated in section 5.4.2. Other sorts of non-local return, such as tail call (see section 7.1.5), continuations and coroutines, are not covered. Simple continuations can be implemented with indirect branches; coroutines, which require multiple stacks, are beyond the scope of Mite's design, and would currently require the multiple stacks to be emulated for a portable implementation.

`CATCH` and `THROW` are unlikely to be used by back ends for existing high-level language compilers, where exceptions are generally implemented by the run-time system, which may even be part of the operating system, as in the case of `longjmp` and `signal` on many operating systems. However, if Mite is to be used in OS kernels, it can usefully provide an exception mechanism to be used by all language implementations.

Given the lengths to which Mite's design has to go to provide portable non-local return, it is hardly surprising that other VMs tend not to, apart from those whose model of control flow is completely concrete, like Cintcode, and hence have no need of explicit support. There are exceptions: `C--` models non-local return explicitly, and, unlike Mite, allows the stack to be unwound one frame at a time. The JVM's Java exceptions mechanism can also encode many sorts of non-local return. Dis and VCODE, on the other hand, do not provide any sort of non-local return.

4.3.6.2 `SWAP`

`SWAP` is rarely useful, but it is cheap to implement, because Mite must provide routines to swap two registers for register shuffling. Other ways of expressing the operation will tend to generate less efficient native code, so it seems more of a waste to omit it than to include it. Having a more general register-permutation operation, though by the same argument it would use extant routines in the translator, would have almost no practical benefit.

4.4 Object format

The object format's main aims, as stated in section 1.2.2, are to be simple, quick to read and write, and endianness-independent. The encoding described in appendix C achieves these aims by being a simple byte code. Multi-byte quantities are encoded as single bytes in a uniform way rather like the UTF-8 encoding of Unicode [54]; uniform methods of packing structures such as lists are also used, and instructions have a one-byte opcode, which is quick to read and decode; bit-field patterns within opcodes are reused where possible. Hence, object files can be read with a few primitive routines, thus making it easier to write a correct decoder. The lengths of lists are stored before their contents; similarly, the length of the object file and the number of labels it contains is given in the header. This means that most data structures can be allocated to their final size in advance, and means that the end of lists (or the object file, when loading it) do not have to be detected by a marker, which makes reading an object file quicker, and writing a robust translator simpler.

The object format is not particularly compact, although it compares reasonably with native code (see section 6.1). However, it is arguably better to use compression techniques such as SSD [71] rather than make the object code itself extremely compact. The advantages are first, that a better compression method can be introduced without changing the format; conversely, a new format can be introduced without needing to rework the compression scheme; finally, ease of decoding and traversing the object code is not hampered by built-in compression. SSD-compressed code can be decoded at a fine grain: it is not necessary to decompress an entire program before starting to translate it. Finally, SSD exploits the structure of both the instruction set and individual programs to achieve compression ratios comparable with good general-purpose data compression algorithms; it would be time-consuming to design a special-purpose encoding that gave as good compression.

Other VMs use a wide range of object formats. At the highest level, `C--` has no object format, but stores its programs in conventional text files, using a C-like notation. Juice's "slim binary" format takes advantage of the compressibility of abstract syntax trees. The JVM's class file format is complicated, with detailed support for the Java type system; Cintcode is a simple byte-code in which literal data and instructions are intermingled. Dis's object format is a half-way house: structured, but simply, separating code, data, types and symbols. It makes little sense to compare the different object formats directly, as they simply reflect the widely differing ends for which their VMs were designed.

4.5 Semantics

The aim of Mite's semantics is not primarily to allow proofs about programs, but simply to make Mite's definition as brief and unambiguous as possible. Mite needs a mathematical definition because it is designed for multiple implementations on widely differing machines.

The semantics given in appendix A is a small-step operational semantics; it defines Mite's behaviour in terms of changes in state on an instruction by instruction basis. In linguistic terms, it can be thought of as a dynamic semantics. Unfortunately there is a tension between the needs of the designer and user, for whom a dynamic semantics is easier to specify, understand, and reason about, and the needs of the translator, which must reason about the program statically. The meaning ascribed to Mite's assembly language in appendix B is therefore static, and in some places it conflicts with the dynamic semantics, such as over the meaning of `NEW` and `KILL`. Other elements, such as register ranking, are deliberately omitted from the semantics, as they do not affect the meaning of programs, but are rather hints to the translator.

Overall, the semantics provides a clear and simple specification of Mite's behaviour, but needs further work to turn it into a basis on which proofs can be made. The obstacles are discussed further in section 6.3.3, and their resolution in section 7.1.7.

Other than Mite, only TAL was designed with formalization in mind. However, many attempts have been made to give a formal semantics to the JVM [5], which illustrates the value that is increasingly attached to formal descriptions of key software components.

4.6 Shortcomings

Mite's design has some shortcomings, which have arisen for different reasons. Some are due to compromises between the different goals, and lack of time; these are discussed in section 6.3.3. Some lie outside the scope of Mite's goals (section 1.2.2); nevertheless, it is worth looking at how hard they would be to rectify. Dynamic code generation is discussed in section 7.2.5, sandbox execution in 7.2.6, and verifiable code in 7.1.7. Garbage collection is considered in section 5.4.3. Other improvements and extensions to Mite are presented in chapter 7.

Finally, some features have simply been omitted or implemented in a less than ideal way. As well as the more important deficiencies, there are a myriad tiny lacunae in the design, and simply not enough time to deal with them all. A few examples are:

`LD` with sign extension Allowing `LD` to sign-extend the quantity it loads would be used by most compilers, and is directly supported by most processors.

Strict constant registers Allowing constants to be loaded with `MOV` conflicts with the intent of section 4.2.5, and was only left in because LCC could not use constant registers (see section 6.2.2). It should only be possible to make a register variable with `UNDEF`.

Indicate stack space usage in functions If each function had the total amount of stack space it used encoded at its start, the translator could easily implement stack pollution (see section 5.2.3.1), and avoid the problem of frequent stack pointer updates discussed in section 6.2.3.2. Since the assembler can calculate this information itself, no change would be required to the semantics or assembler syntax.

Parameters to `ESC` The `ESC` instruction could be more efficiently implemented for OS call mechanisms that take a variable number of parameters if it took parameters in the same way as `CALL`, as only those virtual registers holding parameters would need to be passed. However, the first parameter should still be immediate, as discussed in section 4.3.5.

Code sharing As mentioned in section 3.5, code sharing between subroutines and functions is restricted in Mite. Code sharing was omitted because it is tricky to implement and only of use to compilers that perform inter-procedural optimization. Most systems avoid this problem, either through having simpler semantics for the stack than Mite, like Dis, or by being lower level, like Cintcode. Systems similar to Mite, such as PASM and VCODE, tend to forbid code sharing; in any case, it is hard to reconcile with their code generation interfaces, which translate one function at a time. Sections 6.1.6 and 6.2.3 suggest that Mite should also translate one function at a time, and per-function translation is one of the changes to the implementation proposed in section 7.1.6. Code sharing is discussed further in section 7.1.4.

4.7 Summary

This chapter has shown why, given the goals set out in section 1.2.2, Mite was designed the way it is. Some of the features discussed, such as the load-store architecture, were chosen from the range of existing solutions to the problem of VM design. These can also be thought of as features which classify Mite, or Mite's view of the "right way" to do things. Others, such as the register stack and three-component numbers, add flexibility to Mite's design, and make it harder to classify precisely, moving tradeoffs out of the hands of the VM designer and into those of the compiler writer. While the second class of features may seem more important, and certainly contains Mite's novel contributions to VM design, the first contributes just as much to its usefulness. The success of a new design often rests more on the skill with which it chooses the best elements of the state of the art than the degree to which it innovates. Indeed, as mentioned in section 2.1, perhaps the greatest strength of the JVM is that it contains no innovations, but is a novel and skilful combination of existing techniques. Mite must innovate, as it attempts to combine a range of features and abilities previously seen as mutually incompatible. However, it nevertheless recognizes the dangers of pointless innovation, and, within the limits set by its design goals, is as conservative as possible.

5 Implementation

Mite's implementation consists of two programs: the assembler, which works like a conventional assembler, producing object files from assembly source, and the translator, which loads, translates and executes object files. The translator currently produces code only for the ARM processor. To test Mite's suitability as a compiler target, a Mite back end was added to LCC [38], a well-documented retargetable ANSI C compiler. The assembler and translator are literate programs. The system is available under the GNU General Public License from http://rrt.sc3d.org/.

This chapter is organized as follows. First, section 5.1 gives a brief description of each component; then in section 5.2 a sample translation is followed from C source through Mite assembler to ARM code, to illustrate the translation process. Section 5.3 then discusses highly optimizing compilation for Mite, by taking the output of LCC's Mite back end and hand-optimizing it using the same tricks as GNU C.

This, however, only illustrates the translation of one language for one processor, and Mite is supposed to be language and architecture-neutral. To show how Mite can be used for other languages, section 5.4 discusses three mechanisms not found in C: static chains, exceptions and garbage collection. In each case, several possible implementations are discussed, a few involving additions to Mite's design.

Section 5.5 demonstrates Mite's architecture neutrality in a similar way. Since most current workstation architectures are RISC machines like the ARM, translation for them is mostly straightforward. However, most other RISC processors do not have a dedicated flags register, so Mite's flags register must be treated differently. This is discussed before turning to the more daunting prospect of translating Mite code on Intel IA-32 machines. Two of the main difficulties presented by this architecture are discussed: the paucity of registers, along with the special purpose nature of several of them, and the fact that its instructions are not three-operand, but two, one or even zero-operand.

5.1 Mite's components

An overview of each component is given below; more details of key algorithms are given in the next section.

5.1.1 Assembler

The assembler is written in ANSI C. Its only system dependency is on the word length; this could be removed by using the new ISO C headers [55]. The module which writes object files can be used separately, for example by a compiler which generates binary code directly.

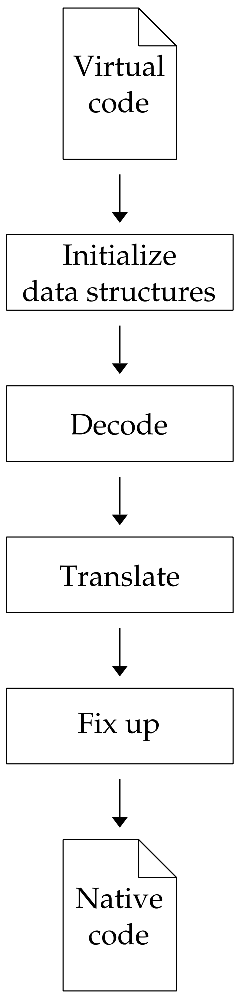


Figure 5.1: Block structure of the Mite translator

5.1.2 Translator

The translator is also written in ANSI C, and most of its system dependencies are isolated in a single module. Its block structure is shown in figure 5.1; it is about 3,500 lines long. It operates as follows:

1. The object file is read in, and its header's validity is checked.
2. Based on the header, some data structures are initialized, such as the array that holds label information.
3. The instructions are decoded and translated (these phases are combined) into a series of native code fragments, roughly corresponding to basic blocks.
4. The code is fixed up by inserting the correct values for branch targets and address constants; at the same time, it is concatenated into a single block of native code, which includes literal data.
5. The native code is executed.

5.1.3 Compiler back end

The LCC back end was written using LCC's back end generator system, `lburg`. This generates the back end from a tree pattern grammar for turning LCC's intermediate code into assembler, and some hand-written routines that implement part of LCC's code generation interface for instructions that cannot be handled by `lburg`, such as function call and block copies. A typical back end is about 1,000 lines, of which 150 or so are always the same; Mite's is less than 600, because many details that must be dealt with by native back ends, such as calling conventions, are left to the translator.

Writing a back end for an existing compiler tested Mite's goal of integrating well with current compiler technology. Two other reasonable courses of action would have been to write a custom back end, or to have used LCC as the basis for a new compiler. The success of the approach used is analysed in sections 5.2.3 and 6.2.2.

5.1.4 Run-time system

The translator has a minimal run-time system, containing just implementations of Mite instructions whose implementation is too long to inline each time the instruction occurs. In the ARM implementation division and memory block copy routines are provided (the ARM has no divide instruction). On most architectures, probably just the block copy would be required; on CISC machines such as the IA-32, no run-time system would be needed.

5.1.5 Standard library access

One problem with compiling C portably is that there is no portable way of calling library routines. Some operating systems provide dynamic linking; others do not, and where it is provided, the mechanisms differ. Mite needs dynamic linking in some form because it translates its programs at load time, and hence also needs to link them to system libraries at load time.

The current Mite translator uses a simple trick to provide portable access to the standard ANSI library routines (except those which may be defined as macros): it initializes an array of pointers with the entry point of each routine, and then looks up all external function calls (those where the label is preceded by `x`) in the symbol table.

With static linking, this results in the entire standard library being linked into the translator, but with dynamic linking, this is avoided. There is no speed penalty in either case, as the translator compiles normal function calls to the addresses stored in the lookup table.

Another problem with compiling C portably (as with any language) is marshalling. In general this problem must be solved by forcing libraries to present a portable interface, whether directly, or via an interface description language. Mite ignores the problem; indeed, for most functions on most machines, it can be avoided by assuming standard representations (for example, `char` is generally a byte, `int` four bytes, `long` and pointer types one word), but this is not good enough for fully portable code.

5.2 A sample translation

To see Mite in action, we now follow a sample translation from C to Mite code, and then to ARM code. So that the translation process can be followed in some detail, only a single short function is examined. It is the `main()` function of the `wf1` benchmark (see section 6.1), which counts the number of times each word occurs in a text file.

5.2.1 C program

The C program is as follows:

```
main() {
  struct node *root;
  char word[20];

  root = 0;
  next = 0;
  while (getword(word))
    lookup(word, &root)->count++;
  tprint(root);
  return 0;
}
```

Here, `root` is the root of a binary tree, whose nodes are statically allocated, and `next` is the number of the next unused node. The tree is initialized by the two assignments, and then the main loop is entered: `getword()` returns the next word in the input stream, and `lookup()` looks it up in the tree, installing it if it is not already there. Finally, `tprint()` displays the words in frequency order.

5.2.2 Translation to Mite virtual code

The output of LCC for this program is as follows:

```
fvp.main
NEW                             return value
NEW_20                          word
NEW_4                           root
NEW                             three temporaries
NEW
NEW
MOV    5, 4                     root = 0
MOV    6, #0
ST_4   6, [5]
MOV    5, .next                 next = 0
MOV    6, #0
ST_4   6, [5]
BAL    .13                      go to test of while loop
.12
MOV    5, 4                     get address of root
NEW                             declare argument register
MOV    8, 5                     load argument &root
MOV    5, 3                     get address of word
NEW                             declare argument register
MOV    9, 5                     load argument word
CALLF  .lookup, 2, [1]          call lookup
MOV    5, 8                     get returned value
KILL                            discard returned value
LD_4   6, [5]                   indirect through pointer
MOV    7, #1                    load 1 into temporary
ADD    6, 6, 7                  increment counter
ST_4   6, [5]
.13
MOV    5, 3                     get address of word
NEW                             declare argument register
MOV    8, 5                     load argument word
CALLF  .getword, 1, [1]         call getword()
MOV    5, 8                     get returned value
KILL                            kill returned value register
MOV    6, #0                    load 0 into temporary
SUB    , 5, 6                   test of while loop
BNE    .12                      loop while true
MOV    5, 4                     get &root
```

```
LD_4   5, [5]
NEW                                  declare argument register
MOV    8, 5                          load argument root
CALLF  .tprint, 1, [1]               call tprint()
MOV    5, 8                          get returned value
KILL                                 discard returned value register
MOV    2, #0                         load return value of main()
.l1
RETF   1, [2]                        return from main()
KILL                                 kill remaining stack items
KILL
KILL
KILL
KILL
KILL
KILL
```

In examining this translation, the focus will be mainly on the issues raised by writing a Mite back end for LCC. Details of LCC's workings that are not pertinent to code generation for Mite are not elaborated.

As shown by the comments, the C program has been translated straightforwardly into Mite code. There are several interesting features of this translation:

Function prologue The function itself starts with a label preceded by `fvp`. `f` indicates a function label, `v` that the function is variadic (since it was not given a proper ANSI declaration), and `p` that it is publicly visible (since it is implicitly declared `extern`). Stack items are then declared: first, the return value (a register, since `main()` returns an `int`), then the automatic variable. Two of these are chunks: `word` because it is an array, and `root` because its address is taken, and Mite registers, like ordinary machine registers, do not have an address. Note that the bottom-most stack item is the return chunk, which is implicitly declared by the function label (see section 3.2.7). It contains the return address and any callee-saved registers or other information required by the system calling convention.

Temporaries Temporaries are declared at various places in the function, starting with three immediately after the prologue. Because LCC does not handle variable numbers of registers gracefully,[1] registers once declared remain live until the end of the function, with the exception of outgoing function arguments and incoming return values. A better approach would be to declare registers used to hold automatic variables at the beginning of the block in which they are declared, and to destroy them at the end. In this example, there is only one block, so this strategy would not help; here, the other register allocation problems described below are more relevant.

[1] Among other problems, the hooks `blockbeg()` and `blockend()`, which are run at the beginning and end of each code block respectively, are called before the number of registers used by the block is known.

Constants Where constants are used, as in the test of the `while` loop, the constant value is always loaded into a virtual register. In a simple-minded translator, this will generally result in a physical register being allocated to hold the constant. Mite provides constant registers to avoid this problem, but LCC does not use them because it has no way of distinguishing constant from variable registers.

Register targeting Since LCC's intermediate code does not give information about the number of arguments to a function, the Mite back end cannot use LCC's register targeting mechanism for function calls, because the relevant register numbers are not known until the code has already been generated. This results in inefficient code, as shown in the example below on the left; a better compiler might generate the code on the right.

```
MOV     5, 3              get address of word
NEW                       declare argument register    NEW
MOV     9, 5              load argument word           MOV 9, 3
```

Return values are similarly handled inefficiently; again, the examples below show the code actually generated on the left, and that which a more intelligent compiler could generate on the right.

```
CALLF   .getword, 1, [1]  call getword()               CALLF  .getword, 1, [1]
MOV     5, 8              get returned value
KILL                      kill returned value register
MOV     6, #0             load 0 into temporary        MOV    6, #0
SUB     , 5, 6            test of while loop           SUB    , 8, 6
                          kill returned value register KILL

CALLF   .tprint, 1, [1]   call tprint()                CALLF  .tprint, 1, [1]
MOV     5, 8              get returned value
KILL                      discard returned value       KILL
```

(In the second case, it might seem even better just to generate

```
CALLF  .tprint, 1, []          call tprint()
```

but this would violate `tprint()`'s type, and might not work on some systems.)

Common sub-expression elimination LCC is rather poor at this optimization; the immediate constant 0 is loaded once redundantly while initializing `next` near the start of the function.

Functions and function calls While all the above points concern limitations of LCC's Mite back end, most of which are at least partly due to Mite's design, there is one respect in which the Mite back end is superior to those for other architectures: it has far less work to do to generate code for function entry, exit and call. The resulting virtual code is also simpler in these areas; the translator must take care of all the fiddly details. This is a good example of reuse arising from Mite: the code to handle functions is implemented just once, in the translator, and can be used by any number of compiler back ends. This is especially beneficial as functions and function calls are often one of the trickiest and most time-consuming parts of a compiler back end to implement. Indeed, `function()`, which generates function prologues and epilogues, averages 112 lines long in the native LCC back ends, but is only 49 in Mite's. The only native back end whose `function()` is less than 100 lines is that for the Intel IA32, which is only 35 lines.

5.2.3 Translation to ARM assembly

The Mite translator produces the following ARM code from the Mite code above:

```
.main
mov     ip, sp                          function prologue
stmfd   sp![2], {r0-r3}
stmfd   sp!, {r4-r9, fp, ip, lr, pc}
sub     fp, ip, #20
cmp     sp, sl
bllt    x.stack_check
subs    sp, sp, #40                     reserve stack space
adds    r9, sp, #12                     root = 0
mov     r8, #0
str     r8, [r9, #0]
adr     r9, &0000031c                   load address of next
add     r9, r9, #&0400
nop
mov     r8, #0                          next = 0
str     r8, [r9, #0]
b       .l3                             go to test of while loop
.l2
adds    r9, sp, #12                     get address of root
movs    r7, r9                          load argument &root
subs    sp, sp, #4                      reserve spill slot for argument register
adds    r9, sp, #20                     get address of word
movs    r6, r9                          load argument word
sub     sp, sp, #4                      reserve spill slot for argument register
mov     r1, r7                          load arguments into correct registers
mov     r0, r6
add     sp, sp, #8                      remove argument register spill slots
bl      .lookup                         call lookup()
movs    r9, r0                          get returned value
ldr     r8, [r9, #0]                    indirect through pointer
mov     r7, #1                          load 1 into temporary
adds    r8, r8, r7                      increment counter
str     r8, [r9, #0]
.l3
adds    r9, sp, #16                     get address of word
```

[2]The `!` causes the stack pointer (`sp`) to be updated by the store instruction, so that the instruction is effectively a multi-register push.

```
movs    r6, r9                         load argument word
sub     sp, sp, #4                     reserve spill slot for argument register
mov     r0, r6                         load argument into correct register
add     sp, sp, #4                     remove argument register spill slot
bl      .getword                       call getword()
movs    r9, r0                         get returned value
mov     r8, #0                         load 0 into temporary
cmp     r9, r8                         test of while loop
bne     .l3                            loop while true
adds    r9, sp, #12                    get root
ldr     r9, [r9, #0]
movs    r6, r9                         load argument root
sub     sp, sp, #4                     reserve spill slot for argument register
mov     r0, r6                         load argument into correct register
add     sp, sp, #4                     remove argument register spill slot
bl      .tprint                        call tprint()
movs    r9, r0                         get returned value
mov     r6, #0                         load return value of main()
.l1
mov     r0, r6                         put return value into correct register
ldmdb   fp, {r4-r9, fp, sp, pc}^[3]    return from main()
```

For clarity, labels have been inserted and branch targets turned into symbolic addresses. The following special registers are given names: pc the program counter, lr the link register (saved return address), sp is the stack pointer, fp the frame pointer, sl the stack chunk limit and ip, a scratch register. The other registers are named r0 to r9 (the ARM has sixteen registers).

By way of comparison, the ARM code generated by LCC's ARM back end is:

```
.main
mov     ip, sp                         function prologue
stmfd   sp!, {v5-v6, fp, ip, lr, pc}
sub     fp, ip, #4
cmp     sp, sl
bllt    x.stack_overflow
sub     sp, sp, #24                    reserve stack space
mov     v6, #0                         root = 0
ldr     ip, [pc, #0]                   load constant 8 bytes ahead[4]
mov     pc, pc                         branch past constant
dcd     0                              constant 0
str     v6, [sp, ip]
ldr     v6, [pc, #0]                   load address of next
mov     pc, pc
dcd     .next                          constant address of next
mov     v5, #0
```

[3]The ^ causes the condition flags, which are stored in the top 6 bits of pc, to be overwritten by the load, thus preserving the flags across the function.

[4]pc holds the address of the current instruction plus 8 bytes.

```
str     v5, [v6, #0]                next = 0
b       .l3                         go to test of while loop
.l2
add     a1, sp, #4                  load argument word
mov     a2, sp                      load argument &root
bl      .lookup                     call lookup()
ldr     v5, [a1, #0]                indirect through pointer
add     v5, v5, #1                  increment counter
str     v5, [a1, #0]
.l3
add     a1, sp, #4                  load argument word
bl      .getword                    call getword()
cmp     a1, #0                      test of while loop
bne     .l2                         loop while true
mov     v6, sp                      load argument root
ldr     a1, [v6, #0]
bl      .tprint                     call tprint()
mov     a1, #0                      load return value of main()
.l1
ldmea   fp, {v5-v6, fp, sp, pc}^    return from main()
```

The following sections compare the two translations.

5.2.3.1 Idiosyncratic code

Most of the ARM instructions clearly correspond to Mite instructions, usually one-to-one. Others have a less clear correspondence, or none at all; they are as follows.

The first five instructions are the function prologue, including stack checking code (the fourth and fifth instructions) that allocates more stack space if necessary.

The adjustments of sp generally correspond to the creation and destruction of registers. Mite's translator reserves a spill slot for each register, in the position on the stack corresponding to the register's number. This avoids needing a spill slot allocator, at the cost of higher stack usage. Many compilers allocate all the stack space required by a function at the beginning, and deallocate it at the end; in addition, they often let outgoing function arguments accumulate until the present function returns, a practice known as "stack pollution". Mite uses the stack more conservatively, which incurs a slight time penalty, because of the more frequent adjustments of the stack pointer, and more than claws back the extra stack usage caused by using fixed spill slots.

Two other features of stack management are worth noting. First, Mite updates sp lazily, waiting until its value is needed. This avoids updating it at all in leaf functions that do not spill or use stack-allocated chunks. That is why the instructions that allocate the spill slots for argument registers often appear after the register has been given a value, whereas the NEW instruction declaring the register in the Mite code must appear before the register is first used. Secondly, the spill slots for argument registers must be removed before the function is called, so that arguments passed on the stack appear in the right place: this is the reason for the adjustment of sp (add) just before the call (bl).

This code could be omitted for functions which take all their arguments in registers, but is not. Also, some function calls, such as that to `getword()`, cause code that updates `sp` to be generated needlessly: `sp` is updated before the arguments are moved into the correct registers, in case any argument needs to be loaded from or stored to the stack.

5.2.3.2 Physical register usage

Unfortunately, rather more physical registers are used than is strictly necessary, and quantities such as argument and return values are often copied more than they need be. This is partly due to the lack of virtual register targeting in the LCC back end, which was discussed in section 5.2.2. The rest, such as the copying of arguments into the correct physical registers before each function call, is due to Mite's lack of physical register targeting: the virtual code has no way of signalling to the translator that a particular quantity is a function argument. There is also a lot of physical register spilling, reloading and shuffling at branches; this is explained further in section 5.2.3.5. A change to the design to allow physical register targeting is discussed in section 7.1.1.3.

Any implementation of Mite also has to deal with register allocation issues specific to the host machine. These tend to centre on the calling convention, which dictates both how registers are used within a function, and how they must be arranged at the instant of procedure call and return. Most ARM-based operating systems use the ARM Procedure Call Standard [9]. This allocates the first four registers as procedure arguments; any that are not used for arguments may be used freely by the callee, and they are therefore caller-saved. The next six registers are callee-saved, and therefore generally used as register variables. The remaining six registers are reserved by the calling convention, as described in section 5.2.3.

Since Mite has no register typing, the translator uses the first ten registers without distinguishing register variables from temporaries, but to avoid excessive saving and restoring around calls the registers are allocated in order from highest to lowest, so that the register variables are used before the argument registers.[5] The translator tries to reload spilled values into the registers they occupied before the call, so that at the end of loops the virtual registers tend to be held in the same registers as at the start, and little shuffling is required before the branch back to the start (see section 5.2.3.5). Apart from this, there are few optimizations. The translator avoids the expense of counting how many registers each function uses, and thus saves all the callee-saved registers in every function (see sections 5.3.3.2 and 6.2.3.2).

Function calls leave little room for manoeuvre: the arguments must be moved into the correct registers, any unused argument registers that are currently in use must be spilled, then the call is made. On return the first argument register holds the return value. This interacts well with the translator's trying to reload values into the register they last occupied, as the first argument register is the last to be allocated, and is hence rarely used.

[5]The test results in figures 6.4 and 6.2 which are discussed in sections 6.1.7 show that most of the time it is better to use the caller-saved registers, as the generated code tends to be both quicker and smaller.

5.2.3.3 Immediate constants

As discussed in section 5.2.2, LCC's Mite back end does not use constant registers, which means that the translator does not use immediate constants. For example, the example in section 5.2.2 contained the Mite code shown below on the left, for which the translator emitted the code on the right:

```
MOV    7, #1            define constant 1        mov     r7, #1
ADD    6, 6, 7          increment counter        adds    r8, r8, r7
```

If LCC's Mite back end used constant registers, it could generate the following Mite code, whose ARM translation saves a physical register and an instruction:

```
DEF    7, #1            define constant 1
ADD    6, 6, 7          increment counter        adds    r8, r8, #1
```

5.2.3.4 Address constants

Address constants are tricky to deal with on the ARM, which lacks absolute addressing and does not have an load effective address instruction. Compilers tend to store address constants in literal pools, which can be placed between functions and indexed with PC-relative addressing (relative address offsets may be up to 4Kb). For simplicity and speed of translation, the translator turns addresses into immediate constants. On the ARM, these have eight significant bits, so it takes up to four instructions to load an arbitrary 32-bit constant. Since the addresses of labels are not known until code generation is complete, the maximum number of instruction slots must always be reserved, and padded with no-ops (see section 6.2.3.2). The ARM translator restricts immediate addresses to 24-bit offsets from the program counter, a reasonable limit. This means that it must allow three instructions for each address load. This explains the `nop` instruction in the code above.

The elimination of no-ops is discussed in section 6.2.3.

5.2.3.5 Register allocation and spilling

One important aspect of the translator has been only tangentially dealt with: the mechanisms for register allocation and spilling. There are three points of particular interest: spilling, register bindings around branches, and dealing with registers at calls. Since the first two are not well illustrated by the example used above (in particular, there is no spilling there), a different code fragment is used, from the `fft` benchmark program. Register bindings at calls were covered in section 5.2.3.2.

The C program is shown below. The Mite code under consideration does not correspond exactly to a contiguous fragment of C, so the extract below is rather schematic. LCC compiles the test of `while` and `for` loops after the body of the loop; the Mite code goes from the test at the end of the first `for` loop below up to part way through the second loop, as shown.

```
{
  ...
  for (i = 0; i < n-1; i++) {
    ...
  }
}

for (s = 1; s <= ln; s++) {
  int m = 1<<s;
  int m2 = m>>1;
  ...
}
```

The Mite code produced by LCC for this program is shown below on the left, and the ARM code it translates to on the right:

```
SUB     , 11, 14     comparison of for loop          cmp     r6, r3
                                                     ldrlt   r2, [sp, #128]
BLT     .119                                         blt     &00000274
                     spill r1                        str     r1, [fp, #8]
MOV     7, #1        initialize s                    mov     r1, #1
REBIND               reorganize registers to         str     r0, [sp, #120]
                     match the virtual-to-physical   str     r7, [sp, #72]
                     register binding at the end     str     r8, [sp, #64]
                     of the loop                     str     r9, [sp, #56]
                                                     str     r6, [sp, #52]
                                                     str     r4, [sp, #48]
                                                     str     r5, [sp, #44]
                                                     mov     r0, r1
                                                     mov     r1, r3
                                                     mov     r3, r2
BAL     .131         branch to the loop test         b       &000005d8
.128
                     spill r0                        str     r0, [fp, #4]
MOV     20, #1       m = 1<<s                        mov     r0, #1
                     spill r7                        str     r7, [sp, #72]
SL      16, 20, 7                                    movs    r7, r0, lsl r1
MOV     20, #1       m2 = m>>1                       mov     r0, #1
SRA     14, 16, 20                                   movs    r3, r7, asr r0
```

At first sight the ARM code appears to contain a large number of register moves and stores which have no counterpart in the Mite code. It turns out that they arise from the points mentioned above. The two stores (`str`) near the end of the native code are spills of `r0` and `r7`. The translator generates spill code at the point in the code where the register is needed again, rather than trying to find a better position for the code. This is only sub-optimal if the spill is in the middle of a loop, and could have been moved outside the loop; that case is partly handled by the `REBIND` directive (see section 4.3.1.3). The series of seven stores and three moves near the start of the ARM code are caused by a `REBIND` directive rearranging the registers before the second `for` loop. The second instruction, `ldrlt`, is a conditional load. It is caused by the following conditional branch, which goes back to the beginning of the first `for` loop. During the loop the register `r2` is spilled, but it is held in a register at the beginning of the loop, so it must be reloaded before the branch.

5.2.4 Summary

There is a straightforward correspondence between the native ARM code produced by LCC's ARM back end and that emitted by Mite's ARM translator. Most of the differences arise from the treatment of registers, and in particular, in the register shuffling code generated by Mite for spilling, and at branches and function calls. Virtual registers are also the source of most difficulties with the LCC Mite back end, such as the inability to perform physical register targeting or to use constant registers. These problems are discussed further in section 6.2.2.

5.3 Optimizing compilation

While the translation above demonstrates that Mite produces a reasonable approximation to LCC's native back end, it is not yet clear that it would do as good a job on the output of a heavily optimizing compiler, compared with that compiler's native code generator. In this section, the gradual optimization of Mite's translation of one of the benchmark programs used in chapter 6 is used to show the sort of Mite code an optimizing compiler might produce.

The optimizations were applied by hand in two stages: first, the optimizations that LCC makes when generating ARM code, to show that very similar code can be obtained from Mite. Secondly, some of GNU C's more aggressive optimizations were applied to give dramatically better code; the result is compared with the code produced by GNU C's native ARM back end. The quantitative results of the optimizations are discussed in section 6.1.

5.3.1 Test program

The test program used to demonstrate Mite's potential for optimizing compilation is `fft`, a fast-Fourier transform program, adapted from a BCPL program by Martin Richards. It was designed to be a good test of register allocation: most of the code resides in two functions, which contain nested loops and many local variables. Section D.1 gives a listing of the program.

The performance of the various forms of the program are shown in figure 6.1, and discussed in section 6.1.2. `fft-1` is the original program, `fft-3` has the LCC-style optimizations added, and `fft-7` has GNU C-style optimizations added. `fft-6` is the same as `fft-7`, but without virtual register rankings.

5.3.2 LCC-style optimization

The first set of optimizations was mostly restricted to those which LCC makes in its ARM back end (as noted above, and discussed further in section 6.2.2, LCC's ARM back end is able to make some optimizations that LCC's Mite back end cannot), namely:

Redundant move elimination Move instructions that move a register to itself were removed; LCC's Mite back end is forced to generate some such instructions after the optimization phase that removes them.

Constant register use Constants were put in constant registers, just as the LCC's ARM back end puts constants in immediate operand fields.

Precompute manifests Manifest constants were computed by the compiler. LCC relies on the assembler to do this, which Mite's assembler cannot.

Register targeting Function argument and result registers were targeted directly as in LCC's ARM back end, rather than using intermediate temporary registers, as LCC's Mite back end does.

Shorter live ranges for temporaries The live ranges of temporaries were reduced to the length of the statement in which each was created, rather than lasting from the point of creation to the end of the function.

Added `REBIND`s A `REBIND` instruction was added just before each loop. This causes the virtual to physical register binding to be brought in line with the ranks, thus potentially avoiding register spilling inside the loop (see section 4.3.1.3).

The resulting improvements in the virtual and native code are shown by the following fragment, in which the left-hand column is the original code, and the right-hand column the hand-optimized version:

```
                           create constant register   NEW
MOV    4, #1               constant 1                 DEF    3, #1
                           create result register     NEW
MOV    5, .ln              load ln                    MOV    4, .ln
LD_4   5, [5]                                         LD_4   4, [4]
SL     5, 4, 5             (1<<ln)                    SL     4, 3, 4
SUB    4, 5, 4             (1<<ln) - 1                SUB    4, 4, 3
MOV    4, 4                (redundant move)
MOV    5, #2               constant 2                 DEF    3, #2
SL     4, 4, 5             ((1<<ln) - 1) << 2         SL     4, 4, 3
NEW
MOV    6, 4                load argument of malloc
CALLF  x.malloc, 1, [1]    call malloc                CALLF  x.malloc, 1, [1]
MOV    4, 6                get return value
KILL
```

The corresponding native code translations are:

```
mov    r9, #1           constant 1
adr    r8, &00000000    load ln                            adr    r9, &00000000
nop                                                        nop
nop                                                        nop
ldr    r8, [r8, #0]                                        ldr    r9, [r9, #0]
                        constant 1                         mov    r8, #1
movs   r8, r9, lsl r8   (1<<ln)                            movs   r9, r8, lsl r9
subs   r9, r8, r9       (1<<ln) - 1                        subs   r9, r9, r8
movs   r9, r9           (redundant move)
mov    r8, #2           constant 2
movs   r9, r9, lsl r8   ((1<<ln) - 1) << 2                 movs   r9, r9, lsl #2
                        make spill slots for NEWed registers sub  sp, sp, #8
movs   r7, r9           load argument of malloc
sub    sp, sp, #4       reserve spill slot for argument
mov    r0, r7           copy argument into its register    mov    r0, r9
add    sp, sp, #4       get rid of spill slot              add    sp, sp, #4
bl     x.malloc         call malloc                        bl     x.malloc
movs   r9, r0           copy return value
```

Notice that the redundant `MOV` has disappeared, so that the corresponding ARM `mov` has also gone; the constant 2 now appears as an immediate constant, while 1 is still loaded into a register, as the left-hand operand of a shift cannot be immediate on the ARM. The unnecessary intermediate register used to hold `malloc`'s parameter and result is no longer present, resulting in shorter, faster code. Finally, the temporary constant 1 has a shorter live range than before. In the event, it does not affect the efficiency of the generated code, but it could well have done if there had been more demand for registers at that point in the program.

5.3.3 GNU C-style optimization

The second set of optimizations was much more pervasive and thorough. The ARM assembly code produced by GNU C at its highest optimization level (`-O3`) was examined, and its structure applied to the Mite virtual code. This ranged from low-level optimizations such as more efficient register use to higher-level code transformations such as moving invariants out of loops and inlining short functions. Additionally, the registers were ranked according to a simple algorithm that GNU C could easily use: inside each block, register variables used in that block were given the highest rank. After discussing some successful optimizations and how they were made possible by Mite's design in section 5.3.3.1, some of the GNU C optimizations that could not be expressed in Mite are examined in section 5.3.3.2.

5.3.3.1 Hits

Many common compiler optimizations can be expressed directly in Mite code, and the optimizations carry through as expected to native code. This section shows some examples of how optimizing the virtual code resulted in optimized native code.

The function add was inlined. The body of the function before was

```
ADD     5, 2, 1                   add the arguments
MOV     7, #65537                 compare result with 65,537
SUB     , 5, 7
BGE     .l9                       if greater, go to l9
MOV     6, 5                      copy result into return value
BAL     .l10                      branch to exit
.l9
MOV     7, #65537                 subtract 65,537 from result
SUB     6, 5, 7
.l10
MOV     4, 6                      copy return value into the correct register
```

A typical inlined call is shown below on the left, with its translation on the right.

```
NEW
DEF     5,  #1
SL      4, 4, 5       (add optimized into a shift)        movs    r9, r9, lsl#1
NEW                   create result register
NEW                   create constant register
DEF     6, #65537     constant 65,537                     mov     r8, #1
                                                          orr     r8, r8, #65536
SUB     5, 4, 6       result −65,537                      subs    r7, r9, r8
KILL                  kill the constant register
BLT     .l51          if sum less than 65,537, finish     sublt  sp, sp, #4
                                                          blt     .l51
MOV     4, 5          otherwise load result −65,537       movs    r9, r7
                      reserve space on stack              sub     sp, sp, #4
.l51
```

This compares with GNU C's

```
mov      r2, r5, lsl#1            (add optimized into a shift)
sub      ip, r2, #1               result −65,537
sub      ip, ip, #65536
cmp      r2, #65536               if sum less than 65,537, load sum
movle    r5, r2
movgt    r5, ip                   otherwise load new result
```

Another stretch of code was originally

```
MOV     14, #2                    constant 2
SL      14, 10, 14                make count into address offset
ADD     14, 14, 3                 add offset to base address
LD_4    13, [14]                  load contents of address plus offset
MOV     14, #2                    constant 2
SL      15, 10, 14                make count into address offset
ADD     15, 15, 3                 add base address to offset
SL      14, 11, 14                make count into address offset
ADD     14, 14, 3                 add base address to offset
LD_4    14, [14]                  transfer first quantity
ST_4    14, [15]
```

```
MOV     14, #2              constant 2
SL      14, 11, 14          make count into address offset
ADD     14, 14, 3           add base address to offset
ST_4    13, [14]            transfer second quantity
```

This was transformed into the following Mite code (again, shown with its ARM translation):

```
DEF     16, #2         constant 2
SL      12, 8, 16      make count into address offset          movs    r3, r7, lsl #2
                       reserve spill slots for new registers   sub     sp, sp, #20
                       spill r2                                str     r2, [fp, #12]
SL      13, 9, 16      make count into address offset          movs    r2, r6, lsl #2
KILL                   constant no longer needed
                       reserve spill slot for constant register add    sp, sp, #4
                       spill r1                                str     r1, [fp, #8]
LD_4    14, [3, 12]    load first quantity                     ldr     r1, [r0, r3]
                       spill r8                                str     r8, [sp, #36]
LD_4    15, [3, 13]    load second quantity                    ldr     r8, [r0, r2]
ST_4    14, [3, 13]    store first quantity                    str     r1, [r0, r2]
ST_4    15, [3, 12]    store second quantity                   str     r8, [r0, r3]
```

Note that there are far fewer actual instructions in the optimized version, which contains a lot of stack directives. This compares with GNU C's

```
ldr     r2, [sp, #0]            load addresses
ldr     r0, [sp, #0]
ldr     ip, [r2, r4, lsl #2]    transfer quantities
ldr     r2, [r2, lr, lsl #2]
str     ip, [r0, lr, lsl #2]
str     r2, [r0, r4, lsl #2]
```

There is little point in giving other examples; by now it should be clear that Mite's success stems from its similarity to real processors: the optimizations that GNU C applies to ARM code can be applied in exactly the same way to Mite code. This does not by itself vindicate Mite's approach; it is this ease of optimization combined with the performance figures (see section 6.1.2), which demonstrates that these optimizations are sufficient to give good performance. It is possible that low-level machine-dependent optimizations such as those discussed in the next section are also sufficient to give good performance. Such optimizations can be implemented in the translator, and hence used with all compilers. Using code transforming optimizations has the opposite advantages, that they are implemented in the compiler, so do not slow translation down, and are applicable to all machines.

5.3.3.2 Misses

Some of the optimizations made by GNU C cannot easily be imitated by the translator, at least not quickly, owing to limitations of Mite's design. Such optimizations can be divided into two classes: machine-specific optimizations and those of more general applicability.

The first class represents Mite's tradeoff between simplicity and universality: accommodating such machine-specific features would generally come at the expense of making Mite more complex or less machine-neutral. In the case of the ARM, the most obvious examples are conditional execution and multiple register load and store. Conditional execution is used in GNU C's version of the inlined add function in section 5.3.3.1, where both possible results are calculated, and only one of the last two instructions is executed to move the result into the result register. Multiple register load and store are mostly used for function entry and exit, where Mite uses them too, but GNU C also uses them for spill and restore code, as well as a way of combining memory transfers. This requires peephole analysis which Mite currently does not do, because it is slow (but see section 5.5.2.1). Optimal use of multiple loads and stores requires interaction with register assignment, since registers are transferred to and from memory in numeric order (the lowest numbered register corresponds to the lowest address, and so on), which would slow the translator down.

The most important examples of the second class are physical register targeting and function entry and exit sequence optimization. The former has already been discussed in section 5.2.2, and changes to Mite's design to accommodate it are discussed in section 7.1.1.3. There is already limited support for the latter in the form of leaf routines, but other measures, such as saving only those callee-saved registers that are actually used, would require Mite do to more work at translate time (see section 5.2.3.2). Sophisticated methods such as moving the entry sequence inside a top-level conditional are probably out of Mite's reach.

5.4 Other languages

Rather than sketching the translation of whole languages, it seems more sensible to treat the implementation of a few key mechanisms in detail. The mechanisms chosen are static chains, often used to implement Modula-like languages that allow nested procedure definitions; exceptions as found in C, Java and ML; and garbage collection, which is required by almost all modern languages.

5.4.1 Static chains

Languages such as the Modula family and ML allow procedure definitions to be nested, and lexical scoping means that variables declared in one procedure are visible in nested procedures. This means that such variables must be accessible even when they are not in the current stack frame. Two common methods used to implement this are static chains, where a pointer in each stack frame points to the suspended frame of the lexically enclosing procedure, if any, and displays, where each stack frame contains a pointer to each of the lexically enclosing procedures. Since displays are simply a way

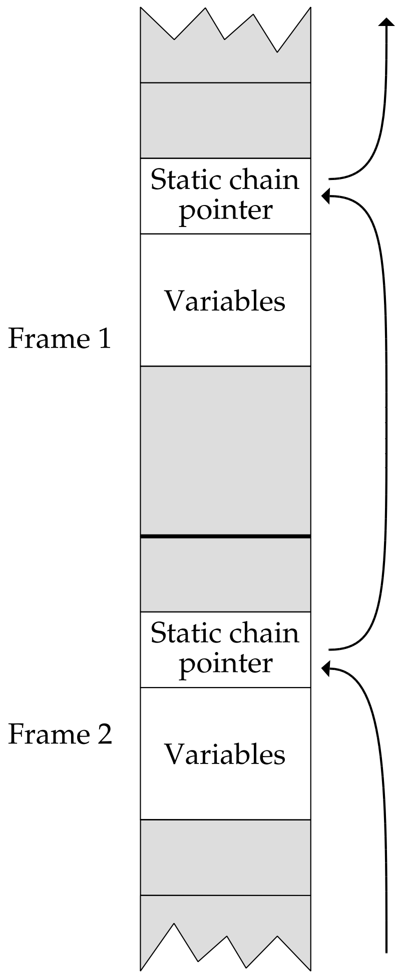


Figure 5.2: Static chaining

of flattening the static chain, their implementation is just a variation on that of static chains, and need not be elaborated separately.

The mechanism of static chains is illustrated in figure 5.2. Each stack frame contains some variables which must be accessible via the static chain from inner procedures. These are stored in a block which starts with a static chain pointer, which points to the corresponding block in the next innermost procedure. Each time a procedure is called, a pointer to the current variable block is passed as an implicit argument, and becomes the value of the new static chain pointer. Now, a variable in an outer stack frame can be accessed by following the static chain back to the procedure to which it is local, and then indexing off the static chain pointer in that procedure.

Two implementations of static chains are discussed. The first (section 5.4.1.1) is a little awkward, but requires no changes to Mite's design to implement portably. The second (section 5.4.1.2), which involves walking the stack directly, is simpler and lighter weight, but cannot be implemented portably in the current model.

5.4.1.1 Putting variables in a chunk

A simple solution, which can be implemented portably in the current model, is to place the variables to be accessed via the static chain in a chunk. A chunk large enough to hold all the variables is declared, and they are stored at fixed offsets. The first word of the chunk is used to hold the static chain pointer, and the chunk's address is passed to inner procedures. The layout of the chunk is independent of whether the system stack happens to be ascending or descending. However, some overhead is incurred: the variables are not directly accessible, but must be loaded into virtual registers and stored back when they change, which is a potential waste of space and time; for example, variables which are passed as parameters to the procedure must be stored into the chunk as part of the entry sequence. This means that the virtual register mechanism of Mite must effectively be duplicated manually for variables that are to be accessible via the static chain. A partial solution to this problem is proposed in section 7.1.1.4.

5.4.1.2 Walking the stack

It would be better if virtual registers could be used directly for variables to be accessed via the static chain, but this requires a way to find the value of a virtual register in an outer procedure. Mite's design denies access to virtual registers in outer procedures for efficiency reasons; is it possible to allow such access without it imposing overheads where it is not needed?

One way to enable this is to allow stack frames to be navigated directly. This breaks down into four requirements. First, the stack layout must be fixed. Secondly, it must be possible to find the start of a stack frame. Thirdly, the stack direction must be known. Fourthly, it must be possible to ensure that when a function call is made, any variables in the current procedure whose values are accessible via the static chain are up to date. Otherwise, under a callee-saves convention, a variable whose value is held in a register just before a call may have its current value saved in the callee's stack frame; meanwhile, the value accessible via the static chain is out of date.

The stack walking mechanism introduced in section 7.1.2 solves all these problems. The stack layout is fixed by requiring all virtual registers to have a stack slot, allocated in order of register number. The register `FP`, which points to the start of the current stack frame, is made visible in the assembly language. Manifest constants are extended to allow them to be multiplied by the stack direction. Finally, `SYNC` is extended to allow the registers to be `SYNC`ed to be specified, so that variable values can be saved before a call without requiring all registers to be flushed.

Section 7.1.2 discusses the modest implementation effort that would be required and, along with section 5.4.3.3, some other benefits that would accrue from its introduction.

5.4.2 Exceptions

Most languages have non-local exits, or exceptions, but the details of how they work vary widely. In C, only a simple value can be passed by an exception. In Java and ML exceptions can pass values of arbitrary type. Each requires a different implementation, but all can be built in terms of Mite's CATCH and THROW instructions.

5.4.2.1 C style exceptions

Although C has already been implemented on Mite, the implementation of exceptions has not been discussed, and it is worth comparing it with the others. C uses the `setjmp` and `longjmp` macros to implement non-local return. Since these macros vary from system to system, they cannot be used in portable code. The current ARM translator ignores the issue, as `setjmp` and `longjmp` are simply function calls on the system on which it runs, and are hence accessed just like any other standard C library routine (see section 5.1.5).

This approach is compatible with natively compiled code, but is not portable: it only works on systems where `setjmp` and `longjmp` are true functions. A portable mechanism can be built quite simply with CATCH and THROW, although it is hard to see how an implementation could be both portable and interwork with natively compiled code efficiently.

A call to `setjmp` is translated as follows:

```
                          register 1 will contain the result of setjmp
                          register 2 contains the address of the jmp_buf
NEW                       scratch register
NEW                       constant
DEF       4, #0+1         one-word offset
CATCH     3, .handler
ST_a      3, [2]          store the address in the jmp_buf
MOV       2, .handler     get the address of the handler
ST_a      3, [2, 4]       store the handler address in the jmp_buf
KILL                      kill registers that are no longer needed
KILL
KILL                      the top stack item is now register 1
MOV       1, #0           set result of setjmp to 0
h.handler                 the point reached by longjmp
```

When control reaches `.handler`, register 1 contains either 0, if the code was entered at the top, or the `longjmp` value, if the handler was reached by a THROW instruction (which overwrites the top-most stack item with the throw value). The call to `longjmp` is implemented as:

```
                          register 1 contains the address of the jmp_buf
                          register 2 contains the return value
NEW                       register to hold the stack state
NEW                       constant
DEF       4, #0+1         one-word offset
LD_a      3, [1]          get stack state
LD_a      1, [1, 4]       get handler address
THROW     1, 3, 2         perform the THROW
```

This causes the handler to be reached with the given return value.

There is a further subtlety: to ensure that the stack state is consistent when a `longjmp` is executed, all `CALL`s and `THROW`s in a function that calls `setjmp` must be followed by `SYNC .handler`.

5.4.2.2 Java style exceptions

Exceptions in Java work as follows: a code block guarded by `try` can raise an exception, which is an object whose type is a sub-class of `Exception`. A `try` block is followed by a number of `catch` blocks, each of which has an associated exception type. The first whose type is a super-class of that of the exception object is executed. After the `catch` blocks there may be a `finally` block, which is always executed, whether the `try` block terminates normally, or with a `return` or `break`, or by an exception. This applies even if a further exception is raised in one of the `catch` blocks. Exceptions may be raised anywhere by `throw`, which is given the exception object. This is often created at the same time:

```
throw new MyExceptionClass("we made a booboo");
```

is a common idiom.

Since user-supplied exception classes can add extra instance variables and methods, exceptions are naturally value-passing.

As exceptions have a special syntax in Java, the implementation is more straightforward than that for C. The `try` block starts with a `CATCH`, and all method calls and `throw`s inside it are `SYNC`ed. The first `catch` block is preceded by a handler label, whose address is used as the current innermost handler. When an exception is thrown to this handler, it determines which `catch` block to run, according to the type of the exception, and then branches to it. Each `catch` block ends with a branch to the end of the last such block, where the `finally` block occurs, if any. If no suitable exception value is found, the exception must be re-`throw`n to the next innermost handler.

Any `return`s, `break`s or `continue`s within the `try` block must also cause the `finally` block to be run before the appropriate action is performed. Thus it might be best to translate the `finally` block as a subroutine, or alternatively to pass it a continuation address. Since an exception may be raised inside a `catch` block, an extra handler must be installed for the duration of the `catch` blocks, which causes the `finally` block to be executed before the exception is re-raised.

The addresses of handlers can be passed to `THROW` sites in a number of ways. The currently active handler could be passed as an implicit parameter to every method call, or the handler chain could be kept as a linked list on the stack. It would also be possible to have a separate handler stack. Most methods used by compiler writers are applicable to Mite.

Note that although Mite's `THROW` instruction only allows a single register to be passed, rather than the compound values allowed in Java, no run-time penalty is incurred, since the value, being an arbitrary Java object, must in any case be allocated on the heap, so the exception value is naturally just a pointer to the exception object.

5.4.2.3 ML style exceptions

In ML, an exception is simply an exceptional value. Exceptions are datatype constructors, and may thus pass arbitrary data. An exception *e* is raised with `raise` *e*. An exception causes immediate termination of expression evaluation, and the value of the expression is the exception value. Exceptions thus propagate outwards like any other result, except that they prevent any further evaluation.

An exception handler is a guard on an expression of the form

$$E \texttt{ handle } P_1 \texttt{ => } E_1 \texttt{ |} \ldots \texttt{| } P_n \texttt{ => } E_n$$

where E is the guarded expression, the P_i are patterns whose top-level constructor is an exception, and the E_i are expressions. There is no equivalent of `finally` in ML.

When an exception value is propagated into an expression that has a handler, the exception value is matched against each clause in the handler; if a match is found, the corresponding handler expression is evaluated, and its value becomes the value of the expression. Otherwise, the exception value becomes the value of the whole expression, just as if there were no handler.

The implementation is similar to the Java case. Since exceptions are propagated until they reach a handler, intervening unguarded expressions can be ignored, and exceptions can be `THROW`n straight to the next innermost handler, just as in Java. When a handler is reached, the exception is dispatched by ML pattern matching rather than according to the Java class hierarchy, but this does not affect the implementation per se.

Unlike the Java case, since ML exceptions need not be constructed on the heap, there is a potential speed penalty in having to place them there, rather than simply treating them as return values. On the other hand, if an exception has to be propagated through several handlers before being handled, it may well be quicker to allocate space for it on the heap than have to copy it between stack frames once for each handler.

5.4.3 Garbage collection

Most modern languages have automatic memory management, and this generally means having a garbage collector. There are two main types of garbage collection: tracing collection, which periodically scans data structures to discover garbage, and reference counting, which acts at each pointer update, and frees storage immediately it is no longer in use by the program. Garbage collectors may also be categorized as conservative, meaning that they do not have a precise idea of the layout of memory, and must occasionally leave garbage uncollected, or accurate, meaning that all garbage is collected. [58] is a thorough guide to the field.

5.4.3.1 Reference counting

Reference counting requires full support from the compiler (and sometimes the programmer), as it affects every pointer update; this has the advantage that it can be implemented entirely portably, and is no harder to implement on Mite than on any other system.

5.4.3.2 Conservative

Conservative garbage collectors such as the Boehm–Dehmers–Weiser collector [15] are generally used for languages that do not explicitly support garbage collection, such as C and FORTRAN. They make more or less weak assumptions about the contents of memory and machine registers and in case of doubt must leave garbage uncollected. Hence, using such a collector with Mite is straightforward: provided that calls to the garbage collector are `SYNC`ed, so that the contents of all virtual registers are available on the stack, the usual heuristics can be used to find garbage by scanning the stack and heap.

Less conservative collection, which requires more accurate information about the layout of the stack, would be made possible by the techniques used for accurate tracing that are discussed in the next section.

5.4.3.3 Accurate tracing

Accurate tracing collection is sometimes not merely desirable, but necessary to avoid serious space leaks, for example in functional languages implemented by graph rewriting, such as Haskell [94] and KRC [133]. In order to work, the collector needs to know what every value on the stack and in registers is. In Mite terms, this reduces to knowing what every item on the virtual stack is, and where it is. The compiler obviously knows what each virtual stack item contains, but the garbage collector cannot, under Mite's current design, know where they are stored.

The changes to Mite's design proposed in section 7.1.2 fix the stack layout, with one exception, and allow garbage collectors to traverse the stack. The one part of the stack whose layout is not fixed is the return chunk, whose format is system-dependent. This may contain callee-saved registers, which may hold the values of virtual registers belonging to the caller. Although they have stack slots in the caller's stack frame, the values held there may not have been up to date when the call was made, so the correct value is now only available in the return chunk, whose format is unknown.

This problem can be overcome by `SYNC`ing all calls. Then, all virtual registers' current values are accessible via the stack walking mechanism, and the return chunks can be ignored. However, as discussed in section 7.1.2, `SYNC`ing all calls is inefficient. Hence, selective `SYNC` is introduced, which allows only those virtual registers containing values that might be of interest to the garbage collector to be `SYNC`ed.

This still leaves some inefficiency, and does not permit the use of such common tricks as finding the stack layout of the caller by looking at the return address. These and other subtleties of supporting garbage collection have been examined by the designers of `C--` [93]. To be as efficient as `C--`, Mite would need to allow the return chunk to be deciphered; this adds considerable extra complexity, though it also goes a long way towards supporting even more facilities, such as multi-threading. Whether it is necessary to make these further extensions to achieve a reasonable level of performance, or whether the changes suggested so far would suffice, remains to be seen.

5.5 Other target processors

Implementing Mite on processors other than the ARM introduces a new range of problems. Here, some of them are considered to show that Mite is indeed implementable on a wide range of processors. First, a problem typical of other RISC machines is investigated: that many do not possess a dedicated condition codes register. Secondly, some of the difficulties thrown up by the most problematic common architecture, the Intel IA-32, are discussed.

5.5.1 Flags without a dedicated register

Some RISC architectures, such as the Alpha and MIPS, do not have a dedicated flags register, and conditional branches take one or two register operands: either a single register which is compared with zero, or two registers which are compared with each other. On such architectures, the translator must take the instruction before a conditional branch into account when generating the branch. If it is a comparison instruction, that is, it has no destination register, it may be possible to emit a single compare-and-branch instruction; for example, the Mite code on the left could generate the MIPS code on the right:

```
SUB     , 2, 3                                  beq     $8, $9, .smaller
BEQ     .smaller
```

Otherwise, the result of the operation must be examined by the conditional branch, as the MIPS has no "compare and branch on less than" instruction. In this case, the translation might look like this (the MIPS register `$0` always has the value zero):

```
SUB     , 2, 3      set $7 if $8 < $9               slt     $7, $8, $9
BLT     .smaller    branch if comparison was true   bne     $7, $0, .smaller
```

In some cases, it may be necessary to generate rather more code than for the ARM, for example, when implementing a `BVS` (branch on overflow set) instruction on the MIPS:

```
ADD     4, 2, 3     get carry bit in $10            add     $7, $8, $9
                                                    lsr     $11, $7, #31
                                                    and     $11, $11, $10
BVS     .overflow                                   bne     $10, $0, .overflow
```

Such instructions are rare, however, and must sometimes already be synthesized by native code compilers, as in this case. Similarly, the Alpha has no built-in carry detection, so carry after an add must be detected by performing an unsigned comparison of the result with each operand to see if it is less than either.

Finally, note that machines which lack a flags register are helped by Mite's prohibition on chained conditional branches (see section 4.3.2).

5.5.2 Targeting the IA-32

The Intel IA-32 architecture is probably the hardest common architecture on which to implement Mite. It throws up many problems; just two of the most important will be considered here: register allocation, and implementing three-operand instructions using two-operand instructions.

When generating code for the IA-32 it is worth remembering that the difficulty of code generation is compensated for to a certain extent by the intelligence of the hardware; most IA-32 processors perform many basic block optimizations in hardware, so that what appears prima facie to be terribly naïve code will often execute reasonably well.

5.5.2.1 Register allocation

Register allocation is difficult on the IA-32 architecture because it has only six general-purpose registers; in addition, each of these registers has a special purpose for certain instructions. Many IA-32 instructions can have memory operands and destinations, so it may sometimes be better to access a virtual register directly on the stack (as a constant offset from the stack pointer) rather than load it into a physical register. Mite's translator has no time to analyse the code deeply in order to determine when to allocate virtual registers to physical registers, and when to access them on the stack, so heuristics must be used. For example, a virtual register could be accessed directly on the stack if its rank is outside the range 1–6. It might also be sensible to spill the physical registers in a fixed order rather than according to rank, spilling first those registers which are most frequently needed for their special-purpose operations.

One example of an operation which uses fixed registers is division, which uses `eax` and `edx` to hold a double-length dividend (the dividend must be double-length); the same registers hold the quotient and remainder. Another is shift by a variable amount, which uses `cl` (the low byte of `ecx`) to hold the shift amount.

Division on the IA-32 will in practice be no worse than on RISC machines that lack hardware division, such as the ARM, where division is implemented as a subroutine, and hence uses fixed registers for the operands and results.

The converse operation, namely assigning virtual registers to the physical registers they must occupy, is harder: for example, if a virtual register is first used as the destination of an `add`, then the divisor of a `div`, the translator must scan ahead to find the second virtual instruction before it can know that assigning the virtual register to `edx` is a good idea. This effectively involves peephole optimization of virtual code.

It may be better simply to have a peephole post-pass on the native code, as mentioned in section 1.2.2, and discussed in sections 5.3.3.2 and 6.2.3.2. This would have two particular benefits on the IA-32, over and above the benefits of peephole optimization on a RISC machine. First, it could locally improve register allocation, and secondly, it could take advantage of the rich instruction set and addressing modes, for example combining scaling a load offset with the load instruction:

```
DEF     3, ashift           scaling constant
SL      2, 2, 3             shift the offset
LD_a    1, [1, 2]           load with word offset      mov     eax, [eax+4*ecx]
```

5.5.2.2 Three-operand instructions

Most of Mite's instructions have three operands, but with a few exceptions, such as multiplication by a constant, the IA-32 instruction set is two-operand, and sometimes fewer, when one or both operands are in fixed registers. Three-operand instructions must be synthesized by first moving one of the operand registers to the destination register, then performing the operation using the destination register and other operand register:

```
                           copy left-hand operand   mov    eax, ecx
ADD    1, 2, 3             perform addition         add    eax, edx
```

When the destination is the same as the left-hand operand, it may not be necessary to generate an extra `mov`; this case can easily be spotted by the translator. Compilers can easily be made to generate instructions with the destination the same as the left-hand operand wherever possible. The opportunity arises quite frequently, as operands are often discarded after being read, and operand registers can then be reused for results.

5.6 Summary

This chapter has shown in detail the workings of the LCC back end for Mite, demonstrating that it produces reasonable code relative to the native ARM back end. It was then shown that several optimizations applied by a more aggressive compiler, GNU C, could also be applied to Mite code. Then, the implementation of key features of some other languages was discussed. These can all be accommodated within the present design, but could be supported more simply and efficiently by a mechanism for stack frame traversal that would be both simple and cheap to implement. Finally, some apparent difficulties in translating Mite for other architectures than the ARM were investigated; in particular, it was shown how Mite could be translated effectively for the Intel IA-32.

It seems that, from a design point of view, Mite is sufficiently flexible to allow both fast translation and good native code across a wide range of languages and machines. The next chapter tests Mite more practically, by examining its quantitative performance in a series of benchmarks.

6 Assessment

This chapter assesses Mite's performance in two ways. First, in section 6.1, a series of tests run on the ARM implementation of Mite is described, and the results are analysed. Next, section 6.2 evaluates the implementation in the light of the results; then, section 6.3 evaluates the design in the light of both, considering how well it has performed and what compromises have been made. The findings are summarized in section 6.4.

6.1 Tests

The test machine was an Acorn RISC PC with a 200MHz StrongARM processor and 16MHz system bus running Acorn RISC OS 3.7.

Three sets of tests were run: some hand-written assembler programs, to test the correctness of the assembler and translator, some of LCC's test suite, to test that the C back end and translator were working, and some benchmark programs. The tests discussed here are drawn from the last two sets. As well as being run on Mite, they were compiled natively for the ARM with LCC[1] and GNU C [35]. Since LCC was also used to produce the Mite code, its native back end was the main point of comparison. The results of GNU C, which was used with full standard optimizations (`-O2`), were intended mainly as an indication of the maximum performance that could be expected from the test platform; LCC performs few optimizations, so its code, whether native or for Mite, cannot be expected to compete with GNU C's.[2] As a broad measure of comparison, and to indicate the absolute speed of the test platform, the Dhrystone 2.1 benchmark [138] was also run, giving the results in table 6.1. To provide a comparison with more familiar hardware, the benchmark was also run on a IBM PC compatible with a 150MHz Pentium processor running Red Hat Linux 6.2.

The tests are as follows. `switch`, `wf1` and `14q` are from the LCC test suite (respectively a switch statement test, a word frequency counter and a 14-queens solver). `stan` is the Stanford Integer Benchmarks [47], which are implemented as a single program. `fft-1` to `fft-7` are versions of the `fft` benchmark, an integer fast Fourier transform (see section D.1), whose Mite translations have been successively hand-optimized, as discussed

[1]It is unfortunate that I wrote LCC's ARM back end, doubly so because the combination of peculiarities of the ARM architecture and deficiencies in LCC's code generation interface and the assembler used meant that code quality was not as good as might have been hoped. Nonetheless, correspondence with the author of another ARM back end [115] and with the compiler's authors indicated that my back end was reasonable under the circumstances.

[2]Section 5.3 considered how well a Mite back end for an optimizing compiler such as GNU C would perform.

System	Speed/VAX MIPS	Relative to Mite
GNU C	100	1·81
GNU C -O	187	3·40
GNU C -O2	191	3·47
GNU C -O3	195	3·54
LCC ARM	69	1·25
LCC Mite	55	1·00
150 MHz IBM PC (GNU C -O2)	200	3·64

Table 6.1: Dhrystone 2.1 results

in section 5.3.1. `pyram` and `pyr-bad` comprise an artificial test of register allocation. The results of `fft-3`, `fft-6`, `fft-7` and `pyr-bad` are excluded from quoted averages.

The next section explains the principles by which the measurements were made and the figures that summarize the results were drawn. Next, the various sets of measurements are explained, and the results are analysed. In each case a reasonable expectation is compared with what actually happened. Where performance was worse than expected, the reasons are investigated and remedies discussed.

6.1.1 Measurements

All timings were obtained with the computer running just the test process (other than interrupt-driven tasks), and each test was performed several times consecutively to minimize differences between runs due to caching and buffering. The timing resolution was 0.01s; the translation times in figure 6.6 were obtained by performing the translation fifty times in a row and dividing by fifty. The measurement data collected from the tests are shown in appendix E.

Relative rather than absolute figures are used for the most part, to concentrate attention on the relative performance of Mite, LCC and GNU C. Absolute measurements often seem to exercise a seductive fascination far beyond their importance: witness the current craze for marketing processors purely on the strength of their clock speed. Mite must be taken in context, and its success judged according to how well it performs relative to its established competitors. Any intuitive advantage that concrete data may have is quickly eroded by rapid advances in the size and speed of hardware, so that grasping absolute performance measurements even a year after they were taken requires a mental effort to remember what typical systems were like at the time. Note that where averages are quoted for relative measurements, they are geometric means [34].

6.1.2 Execution speed

The similarity of Mite's machine model to a real processor suggests that a compiler should be able to generate Mite code similar to the native code it generates for other

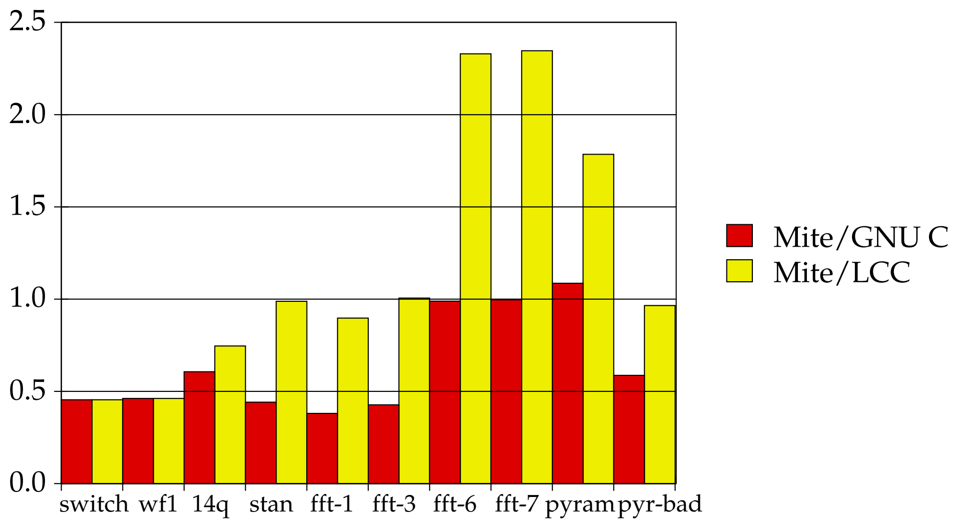


Figure 6.1: Relative execution speed

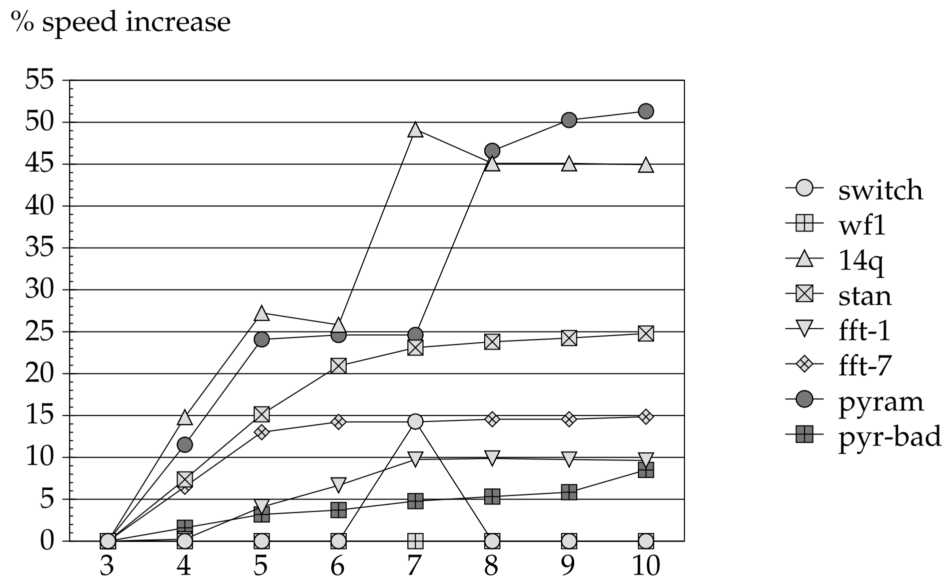


Figure 6.2: Variation in running time with number of physical registers

platforms. This is certainly borne out by comparing the ARM and Mite code generated by LCC for the examples in chapter 5. As Mite's translator naïvely maps Mite instructions to native instructions, the resulting native code should also be similar to that generated directly by a compiler's native back end. Hence, the execution speed of Mite's native code should be about the same as that of LCC's. On the other hand, Mite's looser coupling to the machine, combined with its insistence on rapid translation, could mean that it ends up a little slower. What is an acceptable slow-down? Mite is aiming for performance comparable to that of native compiled code, so it must not be too great; 10–20% seems a reasonable figure. If the slowdown were much bigger than that, use of Mite versus ordinary compilation would turn into a serious tradeoff; on the other hand, for mid-range CPU speeds,[3] a 20% faster processor costs only a few percent more.

Figure 6.1 shows the relative execution speed of each program. The total time taken to load and run each program was measured; in the case of Mite, this included loading the translator and translating the program into native code. Figure 6.2 shows the running time of Mite's code for each test when the number of physical registers available to the translator was artificially restricted, from a minimum of 3, which, on the ARM, is the fewest registers it can work with, up to a maximum of 10. Note that only the time taken to run the test was measured; translation and loading time was excluded.

From figure 6.1 it transpires that, on average, Mite's code runs 21% slower than LCC's. However, in the `switch` and `wf1` benchmarks, translation takes a large proportion of the total execution time (see figure 6.6). If these tests are neglected, along with the artificial boost to Mite given by `pyram`, the mean slow-down is 13%, well within the acceptable range. The Dhrystone benchmark runs 20% slower on Mite than the native LCC version; this is also just within the acceptable range.

Whether the slow-down in `switch` and `wf1` is important is largely a matter of perspective. If they are considered to be interactively-run user commands, then the overall running time is so short that doubling it hardly matters. If on the other hand they are taken to be frequently-run system processes, then the native code version should probably be cached in any case; as can be seen from figure 6.6, this would nearly double their speed, bringing them back in line with the desired performance.

It is also interesting to note the effect of the optimization directives. `fft-1` and `fft-7` were timed with and without `REBIND`. There was no measurable difference, possibly because there were no loops in which substantial spill code was being generated in any case. Alternatively, the register shuffling required at the end of loops in order to match the virtual to physical register binding in effect at the start of the loop may have outweighed the effect of spills. `RANK` on the other hand turned out to have a slight effect: `fft-7`, which is just `fft-6` with ranking added, ran 0.7% faster than `fft-6`.

To see the full potential of ranking, the `pyram` test was written; its source code is given in section D.2. It consists of a series of assignments which gradually involve more and more variables, hence the name, a contraction of "pyramid". The idea is that there should be too many variables to hold in physical registers; furthermore, the variables nearer the start of the alphabet are used much more frequently, and so should be given

[3]Say, 400–500MHz at the time of writing.

higher priority during register assignment. The alternating use of addition and subtraction, coupled with the initial assignment of random values to the variables, aims to prevent a cunning compiler from generating trivial code. `pyram` is an optimally ranked version of the test, and `pyr-bad` a pessimally ranked version. The results are dramatic: not only does `pyram` run much faster, but its performance increases much more than `pyr-bad`'s as the number of physical registers is increased (see figure 6.2). The same effect applied to the amount of code produced (see figure 6.4).

Overall, Mite's execution speed is acceptable. The program that gives most cause for concern is `14q`: it is long-running, and the Mite version runs 25% slower than LCC's native version. This is mostly due to a large number of register spills around function calls caused by LCC's lack of virtual register targeting (see section 6.2.2), as `14q` has a high proportion of function calls. `fft-3`, which has this optimization added by hand, and runs as fast as LCC's ARM version of `fft`, suggests that adding virtual register targeting to LCC's Mite back end would nullify this effect. The prospects for the performance of an optimizing back end targeting Mite were discussed in section 5.3.

This, though, is not the end of the story. While Mite's performance relative to LCC is reasonable, it must be remembered that LCC is not a heavily optimizing compiler. It is far easier for a portable system to be competitive with a relatively straightforward native compiler than with a highly optimizing code generator such as GNU C's. Fortunately, figure 6.1 shows that the optimized version of the program discussed in section 5.3, `fft-7`, runs almost exactly as fast as `fft` compiled by GNU C `-O2`. This is only one benchmark, and it was hand-optimized (though in a mechanical way, as explained in sections 5.3.2 and 5.3.3). Nevertheless, it is an indication that Mite's virtual code can indeed benefit from conventional optimizing compilation to much the same degree as native code.

Finally, it should be remembered that the benchmarks are all compute-intensive; most real-world applications contain a far higher degree of I/O, which will tend to reduce the difference between Mite and its competitors.

6.1.3 Code size

By the argument used in the previous section, the size of Mite's native code should be about the same as that of LCC's, quite probably slightly greater. Bloat of 10–20% over natively compiled code seems reasonable: memory is cheap, and at any rate, this sort of figure is no worse than the cost incurred by moving from a typical imperative language, such as C, to an object-oriented or functional language, such as Java or ML.

Figure 6.3 shows the relative size of the native code produced for each test. The counts exclude out-of-line data, and those for Mite are shown with and without no-op instructions (see section 6.1.3). The counts for Mite were obtained by instrumenting the translator, and those for LCC and GNU C by hand-counting the number of instructions in the assembler output. Run-time routines, start-up code and the like were not counted, to make the comparison between the code generators more accurate. Figure 6.4 shows the size of the native code obtained by Mite for each test when the number of physical registers was varied as in section 6.1.2.

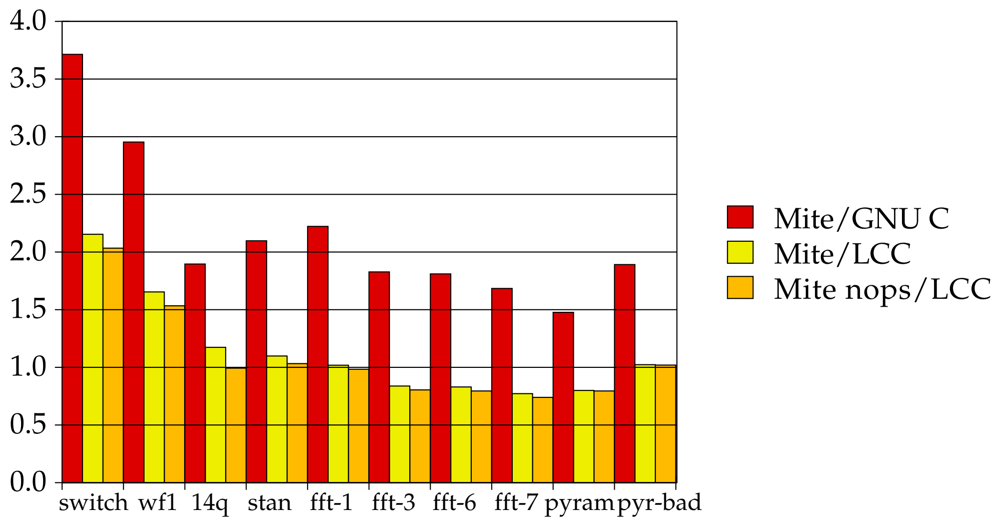


Figure 6.3: Relative code size

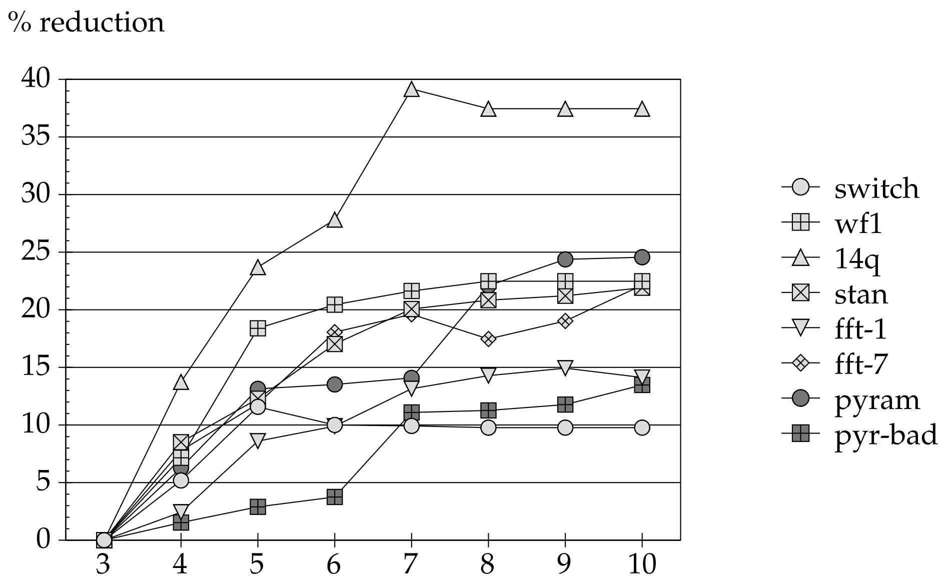


Figure 6.4: Variation in code size with number of physical registers

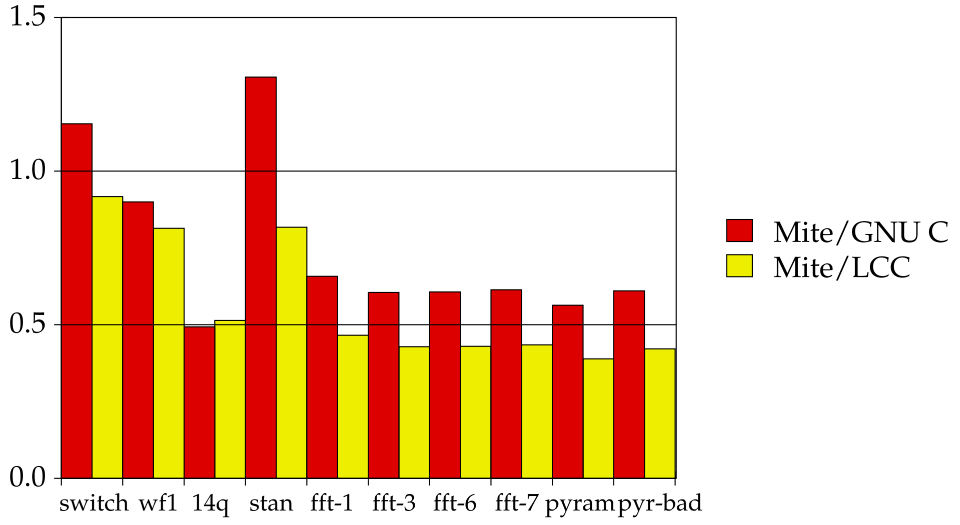


Figure 6.5: Relative executable file size

Figure 6.3 shows that, on average, Mite produces 25% more native code than LCC. In the worst case, `switch`, Mite produces over twice as much code as LCC's native back end; if this test is neglected, the average is 12%.

While the average increase is acceptable, it is worth seeing how it could be improved, especially in the worst case. There are four main reasons for the bloat. First, LCC's Mite back end does not use constant registers, so immediate constants are never used in the native code (see section 6.2.2); this means that constants must always be loaded into a register, which takes an extra instruction, and may incur spills. Secondly, the translator takes up to three instructions to load the address of a label, but since it does not know the address until after code generation, it has to leave space for the instructions and patch them afterwards; the final code must be padded with no-ops (see section 6.2.3). Thirdly, Mite's lack of physical register targeting (see section 6.3) causes excess register shuffling around branches and calls. Finally, as above, the lack of virtual register targeting in LCC's back end (see section 6.2.2) causes unnecessary virtual register traffic.

Optimization directives have little effect on code size: `REBIND` made `fft-7` 1.0% bigger, while the code for `fft-1` remained identical in size when `REBIND` directives were added. The `RANK`ed `fft-7` produced 3.6% less native code than `fft-6`, which has no rankings.[4]

[4]This does not agree with the figures in appendix E, because there `fft-6` has the `add` function compiled in even though all calls to it are inlined; it is removed from `fft-7`, and for a fair comparison should also be removed from `fft-6`.

6.1.4 Executable size

As for the native code, it seems reasonable to expect the virtual binaries to be about the same size as the native binaries. Figure 6.5 shows the relative size of the executable files for each test. In Mite's case this is simply the object file output by the compiler, which is loaded and translated by the translator. It includes optimization directives. For the other two compilers, it is the size of the program executable, which includes start-up code and some run-time routines, though not library code, as the executables are dynamically linked.

In fact, Mite's virtual binaries are much smaller: on average, they are only 65% of the size of LCC's native executables. Since most of the tests are less than 2Kb long, this might be thought to be caused by the standard initialization code present in each LCC binary (a minimal "hello world" program is about 500 bytes long), but even if 500 bytes is subtracted from each of LCC's binaries, Mite's are still 19% shorter on average.

It is also interesting to see how much optimization directives add to the size of a program: `fft-7` is 5.5% longer than the same program without the optimization directives.

Mite's virtual code density is therefore perfectly acceptable. Were greater compactness required, there are several options available. First, the encoding has some room for improvement: successive `NEW`s and `KILL`s, of which there are often several in a row, could be combined, and it could be made possible for register numbers to occupy less than a byte. Secondly, a general-purpose compression algorithm could be used to compress the files on disk; decompression algorithms exist that have a tiny fixed space overhead, and decompress data faster than it can be read from disk [86]. Finally, a compression scheme such as SSD [71] could be used, as discussed in section 4.4. However, none of these measures is urgently needed.

6.1.5 Translation speed

Mite is designed to translate virtual code rapidly into native code, though not as rapidly as a system like VCODE that does not need to decode a virtual binary before generating code. It should generate code fast enough that it does not add significantly to program execution time; this effectively means that it should not significantly reduce execution speed, which was discussed in section 6.1.2. However, for interactive use the translator must also start running the program quickly enough not to annoy the user, so the program's start-up time must not noticeably be increased. Without going deeply into psychology, the following rule of thumb seems reasonable: allow 0.1s for programs up to 10Kb, 1s for programs up to 100Kb, and so on. Few programs are larger than 10Mb, which this logarithmic rule allows 3s.

Figure 6.6 gives a breakdown of the time Mite took to perform each test into translation time and run time; loading time is ignored. Figure 6.9 shows the time taken to translate each test when the number of physical registers was varied as in section 6.1.2.

It turns out from figure 6.6 that Mite generates about 270Kb/s of native code on average, and that code generation speed is roughly linear in the size of the program. For all but the largest applications this is an insignificant addition to startup time. Figure 6.9

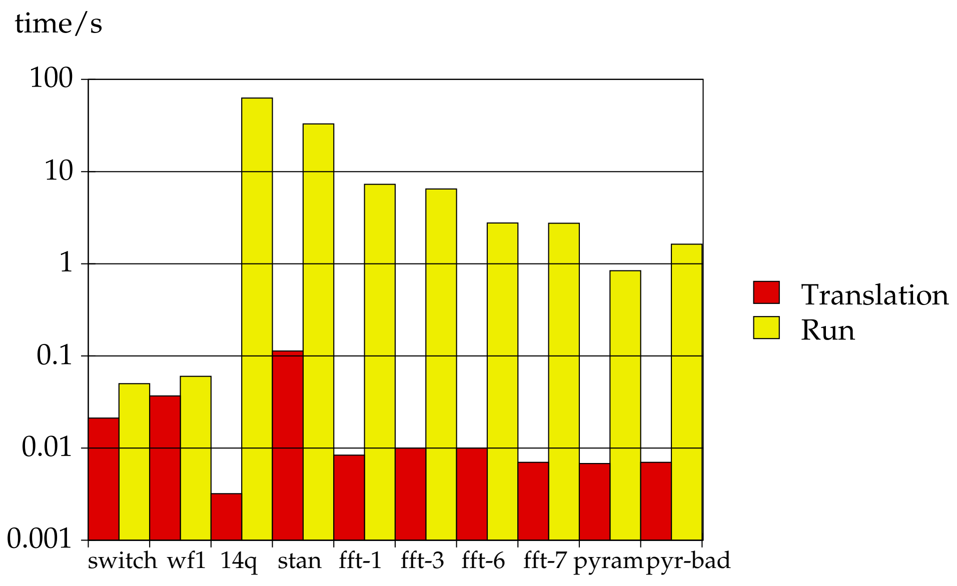


Figure 6.6: Translation versus running time

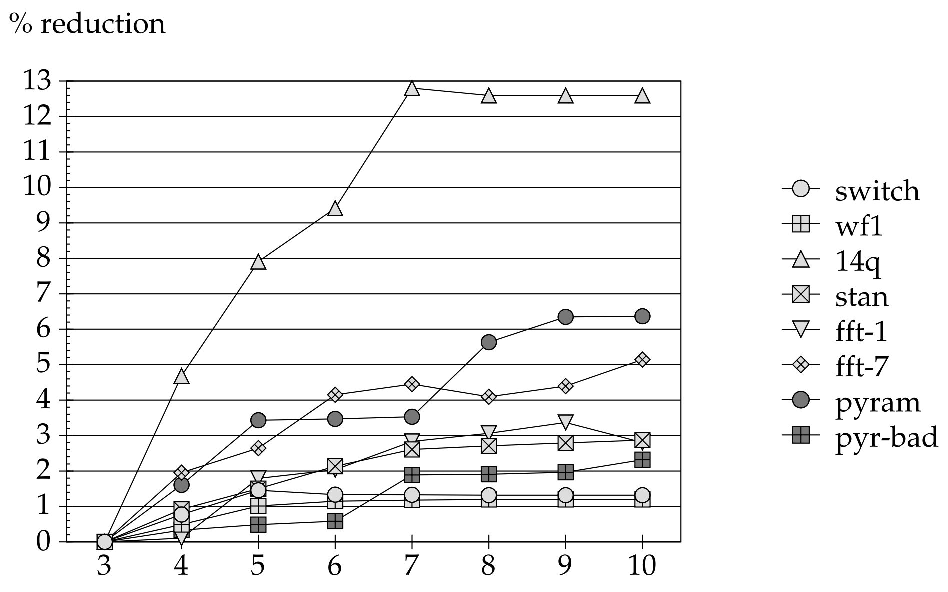


Figure 6.7: Variation in memory consumption with number of physical registers

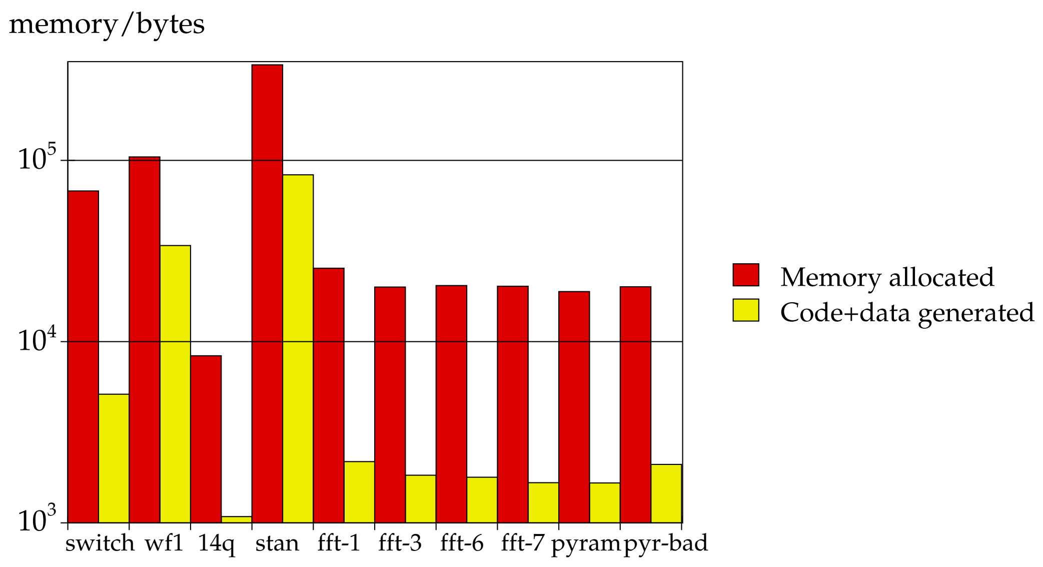


Figure 6.8: Memory consumption of the translator

shows further that translation time does not vary greatly with the number of physical registers. Indeed, profiling the translator (as discussed in section 6.2.3.1) suggests that the number of physical registers is irrelevant: what counts is the amount of code generated. As can be seen from figure 6.4, using more registers tends to reduce code size; thus, the translator tends to run faster when it has more registers are available.

By way of comparison, the test machine loads from hard disk into memory at a rate of about 1Mb/s, or about 3.7 times the rate of translation. Given that Mite's virtual binaries are generally about 65% the size of the generated native code, this makes starting a program with Mite about 6.1 times slower than simply loading it from disk (neglecting the time taken to load the translator). This figure would be rather lower when loading from other media, or downloading code over a network, where latencies are typically much higher; it could also be reduced by compressing the virtual code on disk, as suggested in the previous section, using a decompression algorithm that is faster than reading from disk.

6.1.6 Memory consumption

Low memory consumption is not an explicit goal of Mite; nevertheless, it should stay within reasonable bounds. So that Mite can remain memory-resident in a system when it is used frequently, static memory usage should be low, say less than 0.5Mb. Peak memory consumption during translation might reasonably reach 1Mb.

Figure 6.8 shows the amount of memory dynamically allocated by the Mite translator while translating each benchmark, compared with the memory required by the final code image (including static data, unlike the counts of generated native code in

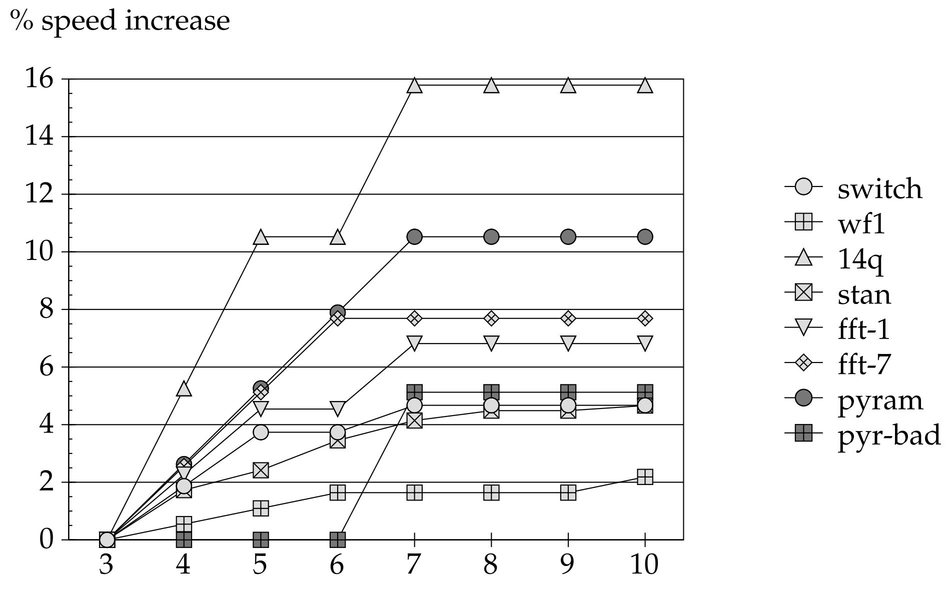


Figure 6.9: Variation in translation time with number of physical registers

figure 6.3). Measured memory consumption excludes memory allocated internally by the `malloc` system. Figure 6.7 shows the translator's memory consumption for each test when the number of physical registers was varied as in section 6.1.2.

In fact, static memory consumption is modest: the translator uses about 80Kb for the program, static data, C heap and stack. Peak dynamic memory usage on the other hand, as shown in figure 6.8, varies roughly linearly with code size, and averages about 12 times the size of the program produced. This may seem excessive, but it is largely due to easily removed inefficiencies in the design of the translator. First, although Mite's translator performs no inter-function optimizations, each module is translated as a whole. Secondly, the code is generated as a series of fragments which are later concatenated to form the binary image; these fragments are of fixed size, and many are partly unused. Changes discussed in sections 6.2.3 and 7.1.6 would greatly reduce peak memory consumption by generating almost all code in-place and translating one function at a time. This would limit total peak memory consumption to well under 100Kb except for programs with pathologically large functions (perhaps produced by compilers that output C).

6.1.7 Usage of physical registers

Figures 6.2 and 6.4 show how execution time and code size vary when the number of physical registers available to the translator is artificially limited. The translator makes good use of extra registers on the whole, as both execution time and code size are almost always reduced by increasing the number of registers available. As already noted in the case of the `pyram` and `pyr-bad` test in section 6.1.2, the effect can be dramatic.

There is an exception to this: some benchmarks perform best with about 7 registers, or experience a performance dip in the range 6–8 registers before climbing again. This is most noticeable in the `14q` benchmark, for which both the best performance and smallest code is obtained with 7 registers.

This strange result is an artefact caused by the lack of virtual register typing. The ARM calling convention allows ten registers for general use (hence the maximum number), of which six are callee-saved and four caller-saved. The translator uses callee-saved registers in preference, as they are saved efficiently on entry to a function with a single multiple register store instruction, whereas caller-saved registers must be individually saved and restored around each call. When the translator is not allowed to use more than six registers, it uses only callee-saved registers, and hence never has to save caller-saved registers around calls. When more registers are added, this becomes a possibility, and if the caller-saved registers happen to be heavily used, a large amount of spill code may result.

If the virtual registers were typed, as suggested in section 7.1.1.2, this would be less of a problem, as the compiler would be able to specify whether virtual registers are short-lived or long-lived, and hence whether they should be mapped to callee-saved physical registers or caller-saved physical registers. In addition, as discussed in sections 5.2.2 and 5.2.3.2, physical and virtual register targeting would alleviate the problem by reducing the number of physical registers used, and the number of inter-register moves. A way to add targeting is outlined in section 7.1.1.3.

6.2 Evaluation of the implementation

This section evaluates the implementation of Mite in the light of the test results. The three parts of Mite are discussed successively: the assembler in section 6.2.1, the LCC back end in section 6.2.2, and the translator in section 6.2.3.

6.2.1 Assembler

There is little to say about the assembler, as the demands on it are slight. It is required to assemble correct code, and extensive checks both inside it and in the translator ensured that this was so. Its performance is of little importance, as it is used only during compilation. In any case, it assembles each of the tests in less than a second.

The only problem experienced with the assembler was its lack of an expression evaluator: when writing LCC back ends it is often convenient to insert constant expressions in the generated assembler code. Strictly speaking, this is a failing of the assembler syntax, or of the compiler, according to one's point of view. In either case, it would be straightforward to remedy.

6.2.2 LCC back end

Although LCC's Mite back end is simpler than the native back ends, there are several problems with it, as discovered in section 5.2, and a few changes had to be made to the machine-independent part of the compiler to make it work at all.

The major difficulties were as follows:

Register live range Ideally, each virtual register should be created when the value it holds becomes live, and destroyed when it ceases to be live (or, since registers must be destroyed in stack order, as soon afterwards as possible). Unfortunately, LCC's back end generator does not provide hooks to do this easily. Hence, the back end waits until the end of each function to kill registers. This lack of control leads to greater run-time stack space usage, and many unnecessary loads and stores of physical registers whose contents are dead.

Register targeting Although LCC supports register targeting, it could not be used for Mite, because the register numbers to be targeted for function parameters and return values are not known when the appropriate back end routine is called. The lack of targeting causes many extra virtual registers to be declared, and many pointless register move instructions, as discussed in section 5.2.2. A solution is discussed in section 7.1.1.3.

Ranking LCC provides no register ranking information, as it has an extremely simple spiller. Support for ranking really requires a more heavily optimizing compiler, such as GNU C, as discussed in section 4.3.1.2.

Number of registers LCC allows as many machine registers as bits in an unsigned integer, typically 32. This has not so far been a problem for the Mite back end, as LCC does not allocate registers aggressively, and none of the tests needed more than 32 registers. However, since one register is allocated for each automatic variable, this limit could easily be exceeded.

Immediate constants LCC has no notion of constant registers. Without them, the Mite translator cannot use immediate constants, and small constants take an extra register and instruction to load.

Fixed-size types LCC requires the sizes of all types to be fixed, so cannot easily cope with variable-sized registers. The code generated by the current back end assumes a 32-bit machine word. A 64-bit back end could be obtained by merely altering the type sizes, but to produce fully portable code would require variable-sized types to be added to LCC. In fairness to LCC, this is likely to be a problem with any compiler, as discussed in section 4.1.3. Also, obtaining portable binaries from C programs is problematic; this is discussed further below.

`extern` **objects** LCC does not note (or in some cases, know) whether a particular object is defined in the current source file or not. Mite demands that references to labels in the current object module are local. The LCC back end therefore assumes that a given object is defined outside the current module if it is declared `extern`. Problems occur mostly with old-style functions which are used before they are defined; this was solved in the tests by declaring each function at the beginning of the C source file. To support all standard C programs would require LCC to obtain this information.

Overall the back end was easy to write, despite the trickery required to make it work: Mite's close resemblance to a conventional processor made it a straightforward target. In retrospect it was perhaps unwise to use LCC's back end generator; a custom version of LCC would have been better, generating code that fully exploited Mite. However, using the back end generator made the task easier. In the end, no version of LCC can hope to exploit Mite fully. The advantages of using GNU C to target Mite are discussed in section 7.2.4.

C itself is problematic, irrespective of the compiler: Mite has the same problems as ANDF (see section 2.4) with differences such as the size of types on different machines, as discussed in section 5.1.5. For Mite they are harder to solve, as the virtual code is so low-level. C is simply not a good choice of language for binary-portable code; however, it can still benefit from Mite's other advantages on a single machine architecture.

6.2.3 Translator

The two most important criteria by which the translator should be judged are speed of translation and quality of the generated code. Speed of translation, along with the subsidiary concern of memory consumption, is discussed in the next section, and the quality of the generated code in the following section.

6.2.3.1 Speed and memory consumption

In section 6.1.5 the translator was judged to be fast enough for all but the largest applications. The only other reason to require faster translation would be for dynamic code generation, as discussed in section 7.1.6. VCODE, which is designed for dynamic code generation, executes on average about 10 machine instructions per generated instruction [29], which on the test platform equates to about 10Mb of code per second, assuming sustained performance of 100 MIPS. This is roughly 35 times more code per second than Mite. A rough hand-count of the number of instructions Mite needs to generate a single machine instruction gives about 200 instructions in the optimal case, a similar ratio of 20 : 1. As mentioned in section 6.1.5, an important reason for this is that Mite has to decode the virtual binary before generating code; nevertheless, it could take a leaf from VCODE's book, and use macros for code generation rather than functions; this would speed up code generation by inlining many of the code generation functions in the translator.

Profiling the translator shows that there are three main bottlenecks:

Decoding virtual instructions The main loop of the virtual binary decoder is an inefficient multiple conditional statement. In addition, some instructions are decoded in two or three stages, being dispatched to the final code generation routine by intermediary functions. This structure made the translator easier to write, and to modify as the design evolved, but is inefficient; a simple 256-way switch based on the opcode of the current instruction (each opcode occupies a byte) would be much faster.

Emitting code As it is generated, the code is emitted into small 32-byte `malloc`ed chunks, which are held in a linked list. A new chunk is started after each branch and before each label, and at any other place where register shuffling code may need to be inserted after code generation. The resultant `malloc`ing of many small blocks wastes both memory and time. By emitting code into a single block, this overhead could be removed. The block could be large enough for the majority of functions, and would only need to be enlarged for exceptionally large functions. This also requires code generation to be performed on a per-function basis. This was already seen to be a good idea in section 6.1.6, and is part of the list of suggested changes to the translator in section 7.1.6.

Producing the final image The final code image is produced by copying the code fragments into a single memory block, at the same time generating fix-up code as described in section 5.1.2. Generating the code into a single block in the first place, as suggested above, would leave the same amount of code to be copied into the final image, but vastly reduce the number of blocks, and hence the administrative overhead.

These shortcomings are peripheral, however, and do not detract from the translator's good performance.

6.2.3.2 Quality of the generated code

Section 6.1 judged the quality of the native code generated by Mite's translator to be adequate. It is interesting to see how much the code quality is dependent on the translator; after all, Mite aims to make code quality dependent on the compiler, not the translator (see section 1.2.2).

It turns out that there is very little more the translator could do without considerably greater effort, which would make it slower, contrary to another of Mite's goals. Compared with GNU C, Mite's native code output has the following shortcomings:

No-ops The ARM translator loads address constants using one to three immediate move instructions. The number of instructions needed is not known until the labels have been generated. Rather than repeatedly adjust the number of instructions used and then the label addresses, with the attendant possibility of non-termination, the translator simply reserves three instruction slots during code generation, and any unused slots are padded with no-ops. This has little effect on execution speed, but increased the code size by 4% on average. It is hard to see how to improve code size without affecting speed of translation, other than by adding an optional post-pass to compress the native code, perhaps as part of a peephole optimiser, as discussed in section 5.5.2.1.

Incomplete use of instruction set As discussed in section 5.3.3.2, GNU C makes better use of the ARM's instruction set than Mite. In particular, it uses the ARM's ability to execute any instruction conditionally on the contents of the flags, which often helps to avoid short compare-and-branch sequences; it uses multiple-register loads and stores for spilling and reloading; it makes good use of instructions that Mite does not possess, such as the ability to combine shifts and arithmetic operations; and it uses addressing modes that Mite lacks, such load and store with write-back to the index, which allows an increment to be added to the base register as part of a load or store instruction. A peephole optimiser, as discussed in section 5.5.2.1, suffices to make all these optimizations (as it does in GNU C); it is hard to see how they could be made without one.

Conservative callee saving In every function except leaf functions, Mite saves all the callee-saved registers on the stack. GNU C saves only those that were used. Mite's translator could do the same, but currently does not, owing to a design flaw centred on the fact that it does not model the ARM's frame pointer in the machine-independent part of the translator. If the proposals of section 7.1.2 were adopted, Mite would have a virtual frame pointer which the translator would have to model, and this problem would disappear.

Excessive stack pointer updates The Mite translator updates the stack pointer every time a register is created or destroyed, rather than claiming all the space required by a function at the start, as LCC and GNU C do. This leads to Mite making more parsimonious use of stack space, but at the expense of frequent updates to the stack pointer, even though they are combined into a single instruction where possible. In retrospect, this is probably a mistake, although to prevent the translator having to scan the virtual code for a function to calculate its maximum stack usage before translating it, it would be worth storing this information at the start of each function, as suggested in section 4.6.

None of these shortcomings is disastrous for Mite. The last two can be easily rectified, and it is hard to see how the first two could be improved without peephole optimization, which would significantly decrease Mite's translation speed. In any case, the higher level optimizations that can already be expressed in Mite code, such as constant folding, loop invariant removal and inlining, are generally more significant for performance than these micro-optimizations, as demonstrated by the increase in speed obtained in section 5.3 during the optimization of the `fft` benchmark, which brought Mite's performance close to GNU C's without peepholing.

On the positive side, Mite is able to do about as well as GNU C in its use of immediate fields, both in arithmetic instructions, and loads and stores. These optimizations are performed solely on a per-instruction basis, and hence are easy to implement in Mite's simple-minded translator. Also, Mite optimizes register shuffling code to use the minimum number of loads and stores.

Hence it seems that the quality of Mite's native code is not critically dependent on its translator (provided that it is implemented sensibly!). There is nothing particularly clever about the current translator, yet it generates good native code when given good virtual code. Indeed, because of Mite's insistence on fast translation, there cannot be anything too clever about the translator. Hence the quality of its output is less a vindication of the translator's implementation than of Mite's design, which is what the next section goes on to consider.

6.3 Evaluation of the design

Mite's design is fundamentally sound: most of the problems with its implementation and performance have been traced to the translator or the LCC back end, as discussed in the previous two sections. However, it is still valuable to weigh the design carefully in the light of experience, and this section does so from three angles. First, the ease of implementation is discussed: the best design in the world is useless if it is too hard to implement. Secondly, the importance of Mite's more innovative features is considered: the considerable effort that went into their conception and implementation will have been worth little if they did not repay that investment by improving Mite's performance. Finally, the design compromises that were due to conflicts between Mite's goals are examined, along with ways of solving the weaknesses that they introduce.

6.3.1 Implementability

There are two sorts of implementability to consider: first, how easy the design was to implement; and secondly, how easy it was to map on to the target architecture.

Mite was straightforward to implement. The only algorithm of any complexity was that required to deal with register shuffling, and it was complex principally in order to avoid needing temporary storage (other than a single physical register) while moving the values of virtual registers around. Other parts of the translator that dealt with registers and their ranks, such as the physical register spiller, also proved tricky, but were in fact less complicated than the equivalent code in many compiler back ends. Finally, implementing function call and return was tedious and error prone, but again, this is a common problem when writing compiler back ends.

The ARM posed few difficulties as a target for Mite. The greatest annoyance was the lack of two-byte load and store instructions. Other oddities included the special code needed to implement shifts of 32 places when the shift amount was specified by an immediate constant, and the fact that the multiplication instruction's destination register must be different from its first operand register.

Over two-thirds of the code in the translator (2,500 lines out of 3,500) is machine-independent, and could be reused when targeting another architecture (although see section 7.1.6). The routines that deal with register rebinding, while not completely portable, could be reused with slight adaptations; only the functions that perform instruction selection would have to be rewritten from scratch.

In summary, Mite's design, though novel in some respects, presented few challenges to the implementor. Difficulty of implementation is not an obstacle to Mite's adoption.

6.3.2 Importance of innovative features

To see how important Mite's more innovative features are, it suffices to imagine what would happen without each of them in turn. Most of the relevant points have already been discussed; this section draws them together briefly, with references to the earlier, fuller discussions.

Stack Mite's stack is most important to the efficacy of its design. Section 4.2.2 explained how it unifies the unlimited-size register file, registers' live ranges, the system stack, function calling and stack allocation. Without this unification Mite's design would have been much more complex, and since the complexity of the translator is directly related to the complexity of the design, this would have made the translator much harder to write. Most importantly, separating the register file from the system stack and doing without the stack discipline of register creation and destruction would have made register spilling and live range management much harder for the translator (see section 4.3.1.1), and led to either less efficient code, or a more complex, and hence slower, translator. Finally, the changes to Mite's design proposed in section 7.1.2, which provide stack frame traversal, which is required for efficient accurate garbage collection, and would also be required for concurrency and debugging, require the virtual register file to be integrated with the stack.

Constant registers Without constant registers, code generators would have to move all immediate values into registers in order to use them. This would cause much more register usage, as discussed in section 5.2.2. Section 5.2.3.3 demonstrates the savings possible when constant registers are used; while the same benefits could be achieved by allowing instructions such as `ADD` and `SUB` to take a constant operand, other problems would then arise, as discussed in section 4.2.5.

Register ranking The importance of ranking was discussed in section 4.3.1.2, with an explanation of how it can be used by an optimizing compiler. The difference in performance between `pyram` and `pyr-bad` (see section 6.1.2), shows its value clearly; the same effect is also shown there in the difference between `fft-6` and `fft-7`. Without ranks, Mite's native code would simply not be able to run as fast.

Three-component numbers The importance of three-component numbers for the generation of portable code was discussed in section 4.1.3. Without them machine-dependent offsets, such as those dependent on the size of pointers, would have to be calculated at run time. This would add considerable overhead to accessing pointer arrays, such as the branch tables often used to implement C's `switch` statement. An extra shift would be required for every access to a pointer array, and the size of the shift would have to be held in a register, since it would only be known at run time. More complex data structures could require an extra two shifts and two adds (to calculate a full three-component number dynamically). The only alternative would be to generate non-portable code (or never to use addresses!).

6.3.3 Design compromises

Given the degree to which Mite's goals conflict, according to conventional wisdom, surprisingly little compromise between them was necessary. Nevertheless, compromise was inevitable, and there was enormous tension at times, with different goals pulling in different directions. Below, the main compromises are listed, and the conflicting goals that caused them are discussed.

Lack of support for modern languages The twin goals of a low-level VM model and portable virtual code militated against support for some mechanisms required by modern languages. Lazy functional languages tend to allocate large amounts of memory, and so need accurate garbage collection. As discussed in section 5.4.3, providing this is possible, but to do so efficiently would require some changes to Mite's design. Other features that would require changes to support efficiently include full continuations (with environment passing) and light-weight concurrency. However, the experience of `C--` [92, 93] shows that it is extremely difficult to reconcile efficiency and portability for these mechanisms.

Limited optimization directives The optimization directives (see section 3.2.10) are rather simple and limited in scope. This is largely because the low-level instruction set is directly optimizable; at the same time, however, the loss of information such as types means that some optimizations used in higher level systems like ANDF and Juice cannot be expressed in Mite. Also, to keep the translator fast, there are no optimization directives that would require substantial translate-time effort to implement. Some further directives that would not increase translation time greatly are proposed in sections 7.1.1.3, 7.1.1.2 and 7.2.3.

Stack order creation and destruction of registers As mentioned in section 4.3.1.1, in order to keep the translator fast, registers are created and destroyed in stack order; therefore, registers' live ranges must be nested, even when the live ranges of the values they hold are not. It might be worth allowing out of order `KILL`s in order to get better native code.

No explicit 64-bit quantity support In order to have efficient portable support for explicit 64-bit quantities, it would be be necessary to have 64-bit virtual registers. This would be a heavy burden on the register allocator of translators for 32-bit systems, so was omitted. As observed in section 4.2.4, 64-bit quantities are rarely required in practice.

Statically sized chunks As mentioned in section 4.2.2, chunks must have a statically determined size to keep translation fast. Since variable-sized stack allocation is generally just used as a low-overhead alternative to allocation in the heap, this is not a serious problem.

High level features The combination of requiring portable virtual code, good quality native code, and a fast translator led to conflict with the ideal of a low-level VM model in several instances. Non-local return had to be handled by `CATCH` and `THROW`; special versions of labels and call and return instructions were required for implementing and calling C-style functions; it was necessary to make virtual registers variable width (see section 3.1.1), which led directly to three-component numbers (see section 4.1.3); and an unlimited number of registers was required (see section 4.2.1). Most of these features tend to make more work for the translator, but on the other hand, they simplify code generation for Mite, with the exception of variable-width registers and three-component numbers, which are awkward for compilers to handle.

Lack of support for faulting Mite has no support for signalling exception conditions such as page faults or division by zero. It is hard to see how to handle them portably,[5] and easier for portable compiled code to check for errors, as this can be achieved with a normal compare and conditional branch, and requires no special semantics. Mite's `THROW` instruction can also be used to deal with exceptional conditions.

Overall, then, it seems that the hardest combination of goals to meet was that of retaining fast translation and good native code quality; this is unsurprising, as they are, in general, diametrically opposed.

One cause of compromise not yet mentioned is that of lack of time, which forced some omissions from the design:

Floating point To support most programming languages fully, and for many real-world programs, Mite needs floating point support. Its addition is discussed in section 7.2.1.

Register typing and targeting As discussed in section 6.1.7, virtual register typing and physical register targeting would both improve the quality of code generated in the presence of function calls. Additions to support them are proposed respectively in sections 7.1.1.2 and 7.1.1.3.

Consistent semantics The formal presentation of Mite's semantics has the advantage of brevity and precision when compared with conventional expositions such as

[5]Of the other systems considered here, only `C--` has systematic support for faulting, and it proved extremely difficult to specify well [103].

those in [67,72,104]. However, the abstract semantics as presented in appendix A are not fully consistent with their extension by the assembly language as presented in chapter 3 and appendix B. The problem, as mentioned in section 4.5, is that the abstract semantics are entirely dynamic, while those of the assembler are partly static. This leads to differences in the meaning of some instructions between the two. For example, `NEW` has a static effect in the assembly language, so that a register is created at the point in the program where the `NEW` is. In the semantics, `NEW`, like all instructions, is dynamic, so that *each time* a `NEW` instruction is reached, a register is created. The difficulty arose because the semantics, which is a small-step operational semantics, is easier to specify in dynamic terms, whereas the translator, in order to generate efficient code, and generate it in a single pass, requires the program's meaning to be easily determined statically. A way of fixing up the semantics is outlined in section 7.1.7.

6.4 Summary

The tests have shown that Mite performs well in practice, with slight reservations about translation speed and the dynamic memory consumption of the translator, both of which have been shown to be easy to improve. Furthermore, it was not a great struggle implementation-wise to achieve these results, and several pointers to ways of improving both the translator and code generator have been given, especially with regard to the use of a more optimizing compiler.

It is extremely difficult to give a comprehensive objective assessment of Mite, because of its broad goals which aim, effectively, to make it all things to all people. An evaluation of it that took into account all the contexts in which it might be used would fill many volumes. Nevertheless, it has been demonstrated that having fast translation and fast execution together is possible, and it seems that Mite's design has indeed managed to reconcile its divergent goals in a practical and workable system.

The next chapter gathers up the loose ends of the design and implementation into a plan for their consolidation and improvement.

7 Future work

This chapter suggests ways in which both the design and implementation of Mite could be improved and extended. The recommendations fall into two categories: improvements (section 7.1) which aim to address problems observed in previous chapters, and extensions (section 7.2) which either improve Mite's performance, or widen its application domain by broadening its functionality.

7.1 Improvements

Most of the improvements discussed here were among many features considered and rejected while designing Mite. Now that Mite has been implemented and tested, it is easier to see which of them are most needful, add least complexity to the design, and are easiest to implement.

7.1.1 Registers

Several problems have been identified with the model of virtual registers, in particular the lack of virtual register typing and physical register targeting. This section suggests some solutions.

7.1.1.1 Out of order `KILL`

Section 6.3.3 pointed out that forcing stack items to be destroyed in stack order can make the native code less efficient. The restriction was made to simplify, and hence speed up the translator (see section 4.3.1.1). Allowing out of order `KILL`s, by adding a parameter specifying the stack item to be killed, might not harm the performance of the translator if it did not attempt to reuse the stack space thus freed until all intervening items had been killed. The benefit would be that the translator would not generate spill and restore code for registers whose contents is no longer needed, but which cannot currently be declared dead until all the stack items above them have been killed as well.

7.1.1.2 Typing

As discussed in section 6.3, registers should be given a type, temporary or variable, allowing callee and caller-saved registers to be used more prudently by the translator. The type can be given in the register's declaration by splitting `NEW` into `NEWT` and `NEWV`.

7.1.1.3 Targeting

Register shuffling could be reduced by targeting virtual registers whose physical binding is known, such as function parameters and return values. This could be done by adding an optional target to the `NEW` instruction, so that registers could be declared as `NEW a3/5`, to indicate the third argument out of five. Alternatively, Mite's `CALL` and `RET` instructions could take a list of the instructions used to calculate each argument and return value. Within each list, register 0 could be used to mean the value being calculated. The lists would not be allowed to contain further `CALL`s or `RET`s. This would allow the translator to evaluate arguments left-to-right or right-to-left as appropriate, as well as performing register targeting.

Then, code such as the following, which is taken from the example in section 5.2.2:

```
MOV    5, 4                  get address of root
NEW                          declare argument register
MOV    8, 5                  load argument &root
MOV    5, 3                  get address of word
NEW                          declare argument register
MOV    9, 5                  load argument word
CALLF  .lookup, 2, [1]       call lookup
```

might become

```
CALLF  .lookup, 2, [1]       call lookup
{ MOV   5, 4                 get address of root
MOV    0, 5 }                load argument &root
{ MOV   5, 3                 get address of word
MOV    0, 5 }                load argument word
```

Here, the braces group the code that produces each argument, and the argument registers are declared implicitly. The following ARM code could be generated:

```
adds    r9, sp, #12          get address of root
movs    r1, r9               load argument &root
subs    sp, sp, #4           reserve spill slot for argument register
adds    r9, sp, #20          get address of word
movs    r0, r9               load argument word
add     sp, sp, #4           remove argument register spill slot
bl      .lookup              call lookup
```

With a better compiler back end that used virtual register targeting too, the following code would be generated:

```
CALLF  .lookup, 2, [1]       call lookup
{ MOV    0, 4 }              get address of root
{ MOV    0, 3 }              get address of word
```

resulting in

```
adds    r1, sp, #12          load argument &root
subs    sp, sp, #4           reserve spill slot for argument register
adds    r0, sp, #20          load argument word
add     sp, sp, #4           remove argument register spill slot
bl      .lookup              call lookup
```

Now, since the function arguments would be guaranteed to be in the right registers or stack locations by the time of the function call, it would be easy to improve the intelligence of the stack pointer modification code, leaving just:

```
adds    r1, sp, #12             load argument &root
adds    r0, sp, #16             load argument word
bl      .lookup                 call lookup
```

7.1.1.4 Addressing chunks directly

Chunks are often used to hold records containing register-sized fields which are manipulated individually. Allowing these fields to be accessed directly would improve native code quality, and could be seen as a less radical alternative to the changes proposed in the next section. Registers could alias a word in a chunk, which would be used for the register's initial value and its spill location. It would have to be possible to force the register's value to be written back, either by adding forced read and write instructions, or by declaring a register "always read" or "always write", as in PASM [21]. More simply, chunks could be allowed as the base address in `LD` and `ST` instructions; for chunks located at small offsets from the stack pointer, this could frequently save a register and an instruction.

7.1.2 Walking the stack

Sections 5.4.1.2 and 5.4.3.3 discussed the need for a stack-walking mechanism. It turns out that such a mechanism could be added to Mite with modest implementation effort, minimal overheads, and benefits beyond those already discussed.

There are two main elements to stack walking: knowing the layout of the stack, and traversing the stack frame. The first is achieved by fixing the stack layout: each virtual register is required to have a stack slot, and the slots are allocated in numerical order from the start of the stack frame. This may seem wasteful, as even registers that are never spilt must have a stack slot (indeed, even constant registers whose value may be held entirely in instructions), but this is exactly what the current implementation does anyway, for simplicity and speed of translation. There is no intrinsic time overhead, only a space overhead, and it is still permissible not to allocate stack slots in leaf procedures. The only part of the stack whose layout is not now fixed is the return chunk; section 5.4.3.3 discusses the consequences for garbage collection, and how this lack of knowledge can be overcome.

Stack traversal is provided by making the register `FP`, which points to the top of the stack on entry to the current function, visible in the assembly language. In addition, manifest constants are extended so that they can be multiplied by the stack direction. With these two mechanisms, stack items can be addressed using manifest offsets from `FP`.

There is one final problem: since virtual registers are cached in physical registers, the translator must be able to ensure that the values in their stack slots are up to date when the stack is traversed. The `SYNC` modifier to `CALL` ensures that this is the case at a procedure call; it remains to rule that traversing the current stack frame may result in incorrect values being read for registers (though it may be useful for chunks). Also, attaching a SYNC to every CALL is expensive, as it effectively forces a caller-saves convention, which must be implemented on top of the system calling convention; most systems use a mix of caller and callee saving. To minimize the overhead, a second argument is added to SYNC, which gives a list of registers that should be updated. The syntax of SYNC becomes:

$$\langle \textit{reg-list} \rangle = \langle \textit{reg} \rangle\texttt{,}^{*}$$
$$\langle \textit{sync} \rangle = \texttt{SYNC}\ [\langle \textit{handler} \rangle][\texttt{,}\langle \textit{reg-list} \rangle]$$

With this new SYNC, just those registers whose value may be needed in an inner procedure can be updated, such as local variables that must be available to a lexically nested procedure, or pointers that may be inspected by the garbage collector.

Finally, note that by giving knowledge about the stack layout and allowing it to be traversed, these mechanisms support rudimentary debugging, although more information about the return chunk would be needed to give a stack backtrace, for example.

7.1.3 Flags

Processors without a dedicated flags register would be better served by combined compare-and-branch and compare-and-set instructions, as discussed in section 4.3.2. Since these instructions decompose straightforwardly into a compare followed by a branch or set on machines that do have a flags register, it seems sensible to adopt this model (in a similar way to the use of three-operand instructions, as discussed in section 4.1.1). Thus, there would no longer be a flags register in Mite.

For each current conditional branch instruction, two instructions would be created: one of the form $\boxed{\texttt{B}c\ d\texttt{,}x\texttt{,}y}$, and one of the form $\boxed{\texttt{T}c\ r\texttt{,}x\texttt{,}y}$. Here, c is a condition code other than AL, and x and y are the operands; d is a destination address, and r is a result register. In each instruction, $x - y$ is calculated. Then, for a branch instruction (B), the branch is taken if the result is true according to the given condition code, c; for a test instruction (T), 1 is written to register r if the result of the test is true, and 0 otherwise.

This scheme only gives one way of generating flags directly: subtraction. Other operations that can currently generate flags, such as addition and masking, will require an extra comparison to do so. However, that is all that many processors offer, and even Mite's restricted flags model is too demanding for some: for example, on the SPARC [137], logical shifts do not set the flags. Another disadvantage of compare-and-branch is that it can cause extra register shuffling, because if the registers used in the comparison are dead at the destination, they cannot be killed before the branch, and so are likely to be spilled before the branch is taken. The extension to KILL proposed in section 7.1.7 would overcome this problem, as explained there.

It might seem that always requiring two operands to the comparison would result in spurious comparison instructions being inserted when no comparison is actually needed. Consider the following code fragment:

```
DEF     4, #0
AND     1, 2, 3                 calculate a bit mask
BEQ     .zero, 1, 4             branch if all bits clear
```

When generating code for a machine such as the ARM, which has a flags register, it seems that the following code might be generated:

```
and     r1, r2, r3              calculate bit mask
cmp     r1, #0                  check if all bits clear
beq     .zero                   branch if so
```

whereas in the current system, the virtual code

```
AND     , r4, r5                calculate bit mask
BEQ     .zero                   branch if all bits clear
```

could be generated, assuming that the bit mask's value is not required, and this would obviously translate to

```
tst     r4, r5                  calculate bit mask
beq     .zero                   branch if all bits clear
```

(`tst` ands its operands and sets the flags accordingly.) However, it is easy for the translator to elide spurious comparisons with zero (this is the only case common enough to be worth optimizing), and note that most processors do not have an equivalent of the ARM's `tst` instruction, which makes the current model more problematic for them: they have to use an extra register. The use of the extra register `r1` could similarly be elided on the ARM.

7.1.4 Code sharing

Code sharing between subroutines and functions, as discussed in sections 3.5 and 4.6, should be allowed if compilers use it. A way would be needed to indicate functions that share a `RET`, or which functions a given `RET` can return from. It might be necessary to restrict code sharing to functions with the same parameter types, as some calling conventions' procedure epilogues need to know the size of the stack frame.

7.1.5 Tail call

A tail call instruction, like that provided by `C--`, would allow many function calls to be optimized. The `TCALL` instruction could be like the `CALL` instruction, except with no return values.

7.1.6 Translator

The translator should be rewritten, to achieve the following aims (some of which were discussed in section 6.2.3):

Layering The translator would be better organized as a series of layers: at the bottom, an assembler, essentially a series of macros to generate machine instructions given physical registers; above this, register allocation for a finite number of registers and label handling; next, spilling, with the object-file decoder on top. It should be possible to use the layers independently; for example, the instruction generation macros could be used as a *lightning*-like system on their own.

Isolation of machine dependencies It should be possible to port the translator by simply replacing the assembler layer; other changes can always be made later to improve the translator or the code it generates.

Faster translation If Mite is to be used for load-time translation of large binaries and rapidly reconfigurable systems, its translation speed should be improved. Layering gives control over translation speed, as it allows the translator to be used at different levels, but standard translation should also be sped up as discussed in section 6.2.3.1.

Reduced memory consumption Translating into native code a function at a time would make peak memory consumption proportional to the size of the largest function translated rather than that of the largest program. Memory allocation overheads could be reduced by using arenas [45] for dynamic memory allocation. Further savings could be made by writing most of the native code into a single memory block, using code fragments only for register rebinding, as discussed in section 6.2.3.1.

Dynamic code generation support This is discussed in section 7.2.5.

Sandbox execution This is discussed in section 7.2.6.

When the translator has been rewritten, the first priority should be to port it to other architectures. An IA-32 port should be made first, partly because it is the commonest workstation architecture, and partly because as the only mainstream CISC processor family it is a stern test of Mite's RISC-like design.[1]

7.1.7 Consistent semantics

An ideal resolution of the problems discussed in section 6.3.3 would make the assembly language merely a sugaring of the abstract syntax, plus the optimization directives; at the moment, the assembly language adds to and changes the abstract semantics.

[1]As noted in section 2.3, VCODE has no IA-32 implementation.

The major problem is to find a unified semantics for the stack instructions. One way to do this is to make the semantics of the assembly language dynamic, like the abstract semantics. This can be achieved by two changes. First, the assembly language is changed so that labels are declared before the start of the program. For subroutines and functions, this includes their parameter and return types, rather like a function prototype. Thus, the static `NEW`s associated with function labels are no longer needed, and the awkward semantics of `CALL`, which effectively performs a number of static `KILL`s at the moment, become purely dynamic. Secondly, branches are allowed to be decorated with `KILL`s, that are performed only if the branch is taken. This allows any change in the stack state between a branch and its destination to be given explicitly in a dynamic way, rather than being static and implicit.

Labels would also have to be added to the abstract semantics to unify it with the concrete; this is beyond the scope of the present discussion, but is not difficult. Finally, following TAL, it might be useful to define Mite by a series of axioms which could be used to derive a typed assembly language, of the sort mentioned in section 2.6. This would give a formal system more suited to making proofs about programs than the current definition.

7.2 Extensions

Mite is at the moment still quite limited in some ways. Partly, this is due to features which its design completely omits, such as floating point arithmetic. On the other hand, certain additions to the implementation would make Mite more attractive to users without requiring changes to the core design: for example, the ability to perform dynamic code generation, or to run virtual code in a sandbox. This section outlines some extensions of both kinds.

7.2.1 Floating point

If Mite is to be generally useful, it must support floating-point operations, which were omitted for simplicity, and because of lack of time. Like almost all the systems discussed in chapter 2, Mite should use the IEEE floating point model [53]. Floating-point instructions and registers are straightforward to add: the instructions can use the same three-operand format as the current arithmetic instructions, and floating-point registers can be declared and used much like integer registers. If 64-bit IEEE representation was required, then no change would be needed to three-component numbers (see section 4.1.3) to accommodate floats in data structures. Extra flags would be needed to indicate error conditions such as underflow; these would fit into the current model, and the changes discussed in sections 7.1.3 apply equally well to these new flags.

The most difficult area of any floating point implementation is error handling. Java and VCODE do not fault, while Dis can, under the control of its floating-point libraries. For Mite it is more useful to report errors than to fault, as errors can be more easily dealt with by portable compiled code (see section 6.3.3).

7.2.2 Instruction scheduling

Instruction scheduling is obviously lacking. VCODE provides functions to perform instruction scheduling, but they slow code generation down, and make it more awkward. It seems better either to perform limited scheduling automatically, or to perform it as a post-pass, which could be combined with global optimization on some systems, or omitted when code generation speed is crucial.

7.2.3 Global optimization

Mite's current design makes no provision for global optimizations, because most would force the translator to make at least one extra pass over the code. The best global optimization to make would probably be global register allocation, which could remove much register shuffling at control flow joins. It should be possible to add this as an optional pass to the translator; ICODE [99] adds a similar option to VCODE.

Some simpler global optimizations would not require extra passes. Interpreters would benefit from being able to bind key virtual registers permanently to machine registers. Another register type to go with those introduced in section 7.1.1.2 could hint that the translator should binding the virtual register permanently to a machine register. Register usage in loops and shared code could be improved by giving an order in which to translate sections of the virtual code. The most important parts would be translated first, and thus have greatest freedom of register use.

Since it requires considerable effort to implement, and will slow down the translator considerably, full-scale global optimization should be postponed until the other suggested improvements have been made and measured. In any case, it may be better to use an independent native code global optimiser [12], or perform run-time specialization [66], using Mite as the dynamic code generator.

7.2.4 Compiler back end

As discussed in section 6.2.2, LCC does not produce well-optimized virtual code. GNU C [35] performs a much wider range of code transformations, and has a peephole optimiser. LCC works on a limited range of machines, and only compiles C; GNU C also supports C++, Objective C, Java, Pascal, FORTRAN and Ada, and is widely ported and heavily used.

GNU C was not used because with its greater power and flexibility comes extra complexity, and though its documentation is comprehensive, it is less exegetic than LCC's. However, for high performance Mite needs a good compiler, and GNU C seems the best choice for development of a new back end.

7.2.5 Dynamic code generation

Allowing Mite to perform dynamic code generation directly, rather than needing to save out an object file and translate that, would make it a competitor to VCODE and PASM. In any case, if Mite is to be used as the sole code generator in a system, it must support dynamic code generation, which is increasingly widely used, for example to translate portable code downloaded across a network. Mite's design is suited to dynamic code generation, and the layering of the translator proposed in section 7.1.6 would allow the code generation functions to be called directly. Multiple clients should be able to use Mite simultaneously in a thread-safe manner.

7.2.6 Sandbox execution

In order to run untrusted code safely, it is useful to be able to run it in a controlled environment, or sandbox, in which it cannot make illegal memory references, or in any other way affect the operation of the host in an unauthorized manner. Such code is common in several applications, including user-written network packet filters that must run in the operating system kernel, and applets downloaded from the internet.

There are two main approaches to sandboxing, which are usually used together. One is to use an interpreter to run the virtual code, and check all operations such as memory accesses. This results in poor performance; however, it is the simplest way to ensure the security of the host machine, and so is widely used, for example by Cintcode. The other common approach is to verify the virtual code according to some security policy. This is how proof-carrying code [84] works: the program comes with a proof that it has certain properties. This proof can be checked by the recipient, and if it is either invalid for the given program, or does not prove strong enough properties, the program is not run. Otherwise, the program, which is ordinary native code (with some restrictions) can be run normally, at full speed, and the host can be confident that it will be safe. The JVM uses a combination of the two approaches, using type-based byte-code verification to catch many potential errors before running the program, which can then be run with far fewer dynamic checks than it would otherwise have needed.

Mite's low-level design means that it can easily be interpreted, but this would lose all its performance benefits. On the other hand, its precise specification makes it possible to design security policies based on properties of the virtual code. In either case, though infrastructure would be needed that is beyond the scope of Mite's original design, its openness would allow the necessary additions to be made without having to change the core specification.

8 Conclusion

To conclude, an appraisal of Mite with reference to its goals is followed by a final perspective on its place in the space of virtual machines.

8.1 Appraisal

This section appraises the degree to which Mite meets each of its goals (see section 1.2.2).

A low-level processor-based VM model Mite's VM model is indeed low-level, and very similar to that of typical processors: especially on RISC machines, most of Mite's computational instructions translate into a single machine instruction. As discussed in section 6.3.3, some high-level features had to be introduced to combine portable virtual code with high-performance native code, but these do not compromise the reason for having a low-level model, which was to make translation of virtual into native code simple.

Architecture neutrality Mite achieves a high degree of architecture neutrality largely by dint of implementing a subset of the facilities provided by most processors. Despite its RISC-like appearance, section 5.5 demonstrated that translating Mite for CISC architectures is straightforward.

Language neutrality Mite is less language neutral than it might be, because mechanisms such as accurate tracing garbage collection and closures, which are relied on by many languages, especially the more modern ones, cannot currently be implemented both efficiently and portably. Section 7.1.2 described how stack walking, which would aid efficient implementation of these and similar mechanisms, could be added to Mite. This outcome largely reflects Mite's similarity to real processors, which lack direct support for these mechanisms, and therefore force them to be programmed in a machine-dependent way if they are to be efficient.

Portable virtual code It is straightforward to generate completely portable code for Mite. As discussed in section 5.1.5, this does not completely solve the portability problem; for that, any libraries which are used by the portable code must be callable in a machine-independent manner.

Fast JIT translation Section 6.1.5 showed that Mite gives reasonably fast single-pass JIT translation, and section 6.2.3.1 discussed ways of improving the speed further.

High-quality native code Mite's performance was shown in section 6.1.2 to be good in the current implementation; however, an optimizing compiler back end and translators for more machines are needed to make this demonstration unequivocal.

Virtual code annotation Mite's virtual code annotations were shown to be both necessary and sufficient to ensure that good native code is produced (see section 6.1.2). Further annotations to improve the code generated around function calls were suggested in section 7.1.1.

Interworking with native code Since Mite can use the system calling convention on its host machine, and can address data anywhere in its host address space (if permitted to), it can easily and fully interwork with native code. All the tests in section 6.1 were linked against the unmodified system libraries, using the method described in section 5.1.5.

Portable object format Mite's simple byte-oriented object format is fully portable.

Precise definition Mite's definition is precise and unambiguous. Some problems with it were identified in section 4.5, but these affect only the semantics' use for proof, and a solution was given in section 7.1.7.

Mite has substantially met all its goals. To be practically useful, however, it needs to be more implemented for more processors, and targeted by better compilers for more languages.

8.2 Perspective

The emergence of a new wave of VMs in the last few years shows that a portable, fast execution platform is desirable. Their proliferation and lack of general acceptance points to the need for a more general-purpose system.[1] Recent interest in typed assembly languages [82] and the many attempts to formalize the JVM [5] suggest that a formal definition is important. After a period of homogenization, the processor market appears to be on the crest of a new wave of architectural diversity; at the same time, the long-predicted shift from 32 to 64-bit architectures is underway. Both these factors increase the need for portable code, and in particular, a way of making legacy code easily portable to new architectures. Networks and distributed applications are now of central importance: these too demand portability, along with distribution and security. However, any system offering all these features is doomed to fail; it cannot be flexible enough to be universal, nor can it evolve as better answers to these problems are found.

[1]Despite sustaining enormous interest for several years, Java still shows no sign of fulfilling its promise of delivering "write once, run anywhere" applications, destroying the barriers between different OSs and machines for user and developer alike. I think that the most important reasons are the over-specialization and poor resource consumption of the Java VM.

Mite provides a flexible way to address all these needs, without attempting to be a complete solution. Its similarity to independent recent work such as VCODE [29] suggests that its design is on the right track; at the same time it is simpler than comparable systems. Its performance is adequate, and readily bettered. The improvements recommended in the previous chapter would make Mite a compelling choice for all its target applications. Instead of being a solution, a rightly maligned concept, Mite is a tool: simple and small enough to be adopted and adapted as a central part of every software system. Only the simple should be ubiquitous.

A Semantics

A.1 Introduction

Mite's semantics are defined in terms of an abstract machine, which consists of a state and a set of rules for transforming it according to a program.

A.2 Definitions

Quantity a string of bits

$q[i \ldots j]$ the quantity consisting of bits i to j inclusive of quantity q

Width the number of bits in a quantity

A either 32 or 64

Word an A-bit quantity

w**-aligned** a multiple of w

Size an expression of the form $b + w + r$, where b, w, and r are non-negative integers, whose value is $b + Aw + 32 \lfloor A/64 \rfloor r$

ρ an undefined quantity of infinite width

$[S]$ a bit representing the truth of statement S; if S is true then $[S]$ is one, otherwise it is zero

$q \leftarrow E$ the assignment of expression E to quantity q; before assignment, E is truncated or zero-extended to make it the same width as q

Stack a last-in-first-out stack whose items are said to be added to the top, and are numbered from one, counting from the bottom

$s[i]$ the ith item of stack s

$s[i \ldots j]$ the stack consisting of items i to j inclusive of stack s

$s \oplus E$ the stack s with an extra item added whose value is E

A.3 State

Mite's state consists of the following elements:

Flags f_Z, f_N, f_C, f_V one bit each

Execution pointer EP a word

Temporary register T a word

Memory M a quantity

Permutation functions p_8, p_{16}, p_{32}, p_A

Stack S a stack

Frame stack F a stack

An index into M is called an **address**.

The function p_w takes a quantity of width w and returns it with its bytes permuted.

F is a stack of pairs of naturals. FP is the index of the top-most item in F. $F_S(i)$ and $F_N(i)$ denote respectively the first and second component of the ith item of F. A **stack position** is a natural p in the range $1 \ldots F_N(FP)$.

S holds items of two sorts: a **register** is a word created by NEW(), and a **chunk** is a quantity of arbitrary size created by NEW(c) (see section A.5.7). The stack items are held in M at word-aligned addresses.

S_p, where p is a stack position, denotes $S[F_S(FP) + p - 1]$; $\&S_p$ denotes the address of S_p. SP is an abbreviation for $F_S(FP) + F_N(FP) - 1$.

A.4 Program

An **instruction** consists of an operation and a tuple of operands. Each operand has a **type**, given by its name; a subscript is added to distinguish operands of the same type. The allowable instructions are given in section A.5. The program P is an array of instructions; $P[i]$ denotes the ith element of P.

The types are:

Stack position p a stack position

Natural n a non-negative integer

Register r a stack position p such that S_p is a register, or T

Chunk c a stack position p such that S_p is a chunk

Width w a member of the set $\{8, 16, 32, A\}$

Size s a size

A.5 Instructions

The state is transformed by repeatedly performing $EP \leftarrow EP + 1$ then the semantics of $P[EP - 1]$. The semantics of each instruction are given below in terms of assignments to state elements, and other instructions; the operations are performed sequentially. An underlined expression is a predicate that must evaluate to true when the instruction is executed; otherwise the instruction has no effect.

Arithmetic is integral, performed on A-digit binary numbers using two's complement interpretation. Quantities are evaluated with bit zero as the least significant digit.

The semantics of every instruction have the assignment $T \leftarrow \rho$ prepended, and also, for all instructions except branches (see section A.5.4), the following:

$$\begin{aligned} f_Z &\leftarrow \rho \\ f_N &\leftarrow \rho \\ f_C &\leftarrow \rho \\ f_V &\leftarrow \rho \end{aligned}$$

For branches, the four instructions above are added to the end of the instruction's semantics.

A.5.1 Assignment

$$\begin{aligned} \mathsf{MOV}(r_1, r_2) \ : \ & S_{r_1} \leftarrow S_{r_2} \\ & f_Z \leftarrow [S_{r_1} = 0] \\ & f_N \leftarrow [S_{r_1} < 0] \\ \mathsf{MOV}(r, c) \ : \ & S_r \leftarrow \&S_c \\ & f_Z \leftarrow [S_{r_1} = 0] \\ \mathsf{SWAP}(r_1, r_2) \ : \ & T \leftarrow S_{r_1} \\ & S_{r_1} \leftarrow S_{r_2} \\ & S_{r_2} \leftarrow T \end{aligned}$$

A.5.2 Data processing

All the data processing instructions except MUL, DIV[S[Z]] and REM[S[Z]] have

$$\begin{aligned} f_Z &\leftarrow [S_{r_1} = 0] \\ f_N &\leftarrow [S_{r_1} < 0] \end{aligned}$$

appended to the end of their semantics.

A.5.2.1 Arithmetic

$\mathsf{NEG}(r_1, r_2)$: $S_{r_1} \leftarrow -S_{r_2}$
$f_C \leftarrow [S_{r_1} = 0]$
$f_V \leftarrow [S_{r_1} = -2^{A-1}]$

$\mathsf{ADD}(r_1, r_2, r_3)$: $S_{r_1} \leftarrow S_{r_2} + S_{r_3}$
$f_C \leftarrow$ carry out of most significant bit
$f_V \leftarrow$ [signed overflow occurred]

$\mathsf{SUB}(r_1, r_2, r_3)$: $S_{r_1} \leftarrow S_{r_2} - S_{r_3}$
$f_C \leftarrow$ carry out of most significant bit
$f_V \leftarrow$ [signed overflow occurred]

$\mathsf{MUL}(r_1, r_2, r_3)$: $S_{r_1} \leftarrow S_{r_2} \times S_{r_3}$

$\mathsf{DIV}(r_1, r_2, r_3)$: $$\frac{S_{r_3} \neq 0}{S_{r_1} \leftarrow S_{r_2} \div S_{r_3}\text{, treating } S_{r_2} \text{ and } S_{r_3} \text{ as unsigned, and rounding the quotient to } 0}$$

$\mathsf{DIVS}(r_1, r_2, r_3)$: $$\frac{S_{r_3} \neq 0}{S_{r_1} \leftarrow S_{r_2} \div S_{r_3}\text{, treating } S_{r_2} \text{ and } S_{r_3} \text{ as signed, and rounding the quotient to } -\infty}$$

$\mathsf{DIVSZ}(r_1, r_2, r_3)$: $$\frac{S_{r_3} \neq 0}{S_{r_1} \leftarrow S_{r_2} \div S_{r_3}\text{, treating } S_{r_2} \text{ and } S_{r_3} \text{ as signed, and rounding the quotient to } 0}$$

$\mathsf{REM}(r_1, r_2, r_3)$: $\mathsf{DIV}(T, r_2, r_3)$
$r_1 \leftarrow r_2 - T \times r_3$

$\mathsf{REMS}(r_1, r_2, r_3)$: $\mathsf{DIVS}(T, r_2, r_3)$
$r_1 \leftarrow r_2 - T \times r_3$

$\mathsf{REMSZ}(r_1, r_2, r_3)$: $\mathsf{DIVSZ}(T, r_2, r_3)$
$r_1 \leftarrow r_2 - T \times r_3$

A.5.2.2 Logic

$\mathsf{NOT}(r_1, r_2)$: $S_{r_1} \leftarrow$ one's complement of S_{r_2}

$\mathsf{AND}(r_1, r_2, r_3)$: $S_{r_1} \leftarrow$ bitwise and of S_{r_2} and S_{r_3}

$\mathsf{OR}(r_1, r_2, r_3)$: $S_{r_1} \leftarrow$ bitwise or of S_{r_2} and S_{r_3}

$\mathsf{XOR}(r_1, r_2, r_3)$: $S_{r_1} \leftarrow$ bitwise exclusive-or of S_{r_2} and S_{r_3}

$$\mathsf{SL}(r_1, r_2, r_3) \ : \ \dfrac{0 \leq S_{r_3} \leq A}{\begin{array}{l} S_{r_1} \leftarrow S_{r_2} \text{ shifted left } S_{r_3} \text{ places} \\ f_C \leftarrow \text{carry out of most significant bit, if } S_{r_3} > 0 \end{array}}$$

$$\mathsf{SRL}(r_1, r_2, r_3) \ : \ \dfrac{0 \leq S_{r_3} \leq A}{\begin{array}{l} S_{r_1} \leftarrow S_{r_2} \text{ shifted right logically } S_{r_3} \text{ places} \\ f_C \leftarrow \text{carry out of least significant bit, if } S_{r_3} > 0 \end{array}}$$

$$\mathsf{SRA}(r_1, r_2, r_3) \ : \ \dfrac{0 \leq S_{r_3} \leq A}{\begin{array}{l} S_{r_1} \leftarrow S_{r_2} \text{ shifted right arithmetically } S_{r_3} \text{ places} \\ f_C \leftarrow \text{carry out of least significant bit, if } S_{r_3} > 0 \end{array}}$$

A.5.3 Memory

$$\mathsf{LD}(w, r_1, r_2) \ : \ \begin{array}{l} S_{r_1} \leftarrow 0 \\ S_{r_1}[0 \ldots w-1] \leftarrow p_w(M[S_{r_2} \ldots S_{r_2} + w - 1]) \end{array}$$

$$\mathsf{ST}(w, r_1, r_2) \ : \ M[S_{r_2} \ldots S_{r_2} + w - 1] \leftarrow p_w^{-1}(S_{r_1}[0 \ldots w-1])$$

$$\mathsf{COPY}(s, r_1, r_2) \ : \ \dfrac{S_{r_1} + s \leq S_{r_2} \text{ or } S_{r_2} + s \leq S_{r_1}}{M[S_{r_1} \ldots S_{r_1} + s - 1] \leftarrow M[S_{r_2} \ldots S_{r_2} + s - 1]}$$

$$\mathsf{COPY}(s, r, c) \ : \ \dfrac{S_r + s \leq \&S_c \text{ or } \&S_c + s \leq S_r}{M[S_r \ldots S_r + s - 1] \leftarrow M[\&S_c \ldots \&S_c + s - 1]}$$

$$\mathsf{COPY}(s, c, r) \ : \ \dfrac{\&S_c + s \leq S_r \text{ or } S_r + s \leq \&S_c}{M[\&S_c \ldots \&S_c + s - 1] \leftarrow M[S_r \ldots S_r + s - 1]}$$

$$\mathsf{COPY}(s, c_1, c_2) \ : \ \dfrac{\&S_{c_1} + s \leq \&S_{c_2} \text{ or } \&S_{c_2} + s \leq \&S_{c_1}}{M[\&S_{c_1} \ldots \&S_{c_1} + s - 1] \leftarrow M[\&S_{c_2} \ldots \&S_{c_2} + s - 1]}$$

A.5.4 Branch

$$\mathsf{BAL}(r) \ : \ EP \leftarrow S_r$$

$$\mathsf{BEQ}(r) \ : \ \dfrac{f_Z = 1}{EP \leftarrow S_r}$$

$$\mathsf{BNE}(r) \ : \ \dfrac{f_Z = 0}{EP \leftarrow S_r}$$

$$\mathsf{BMI}(r) \ : \ \dfrac{f_N = 1}{EP \leftarrow S_r}$$

$$\mathsf{BPL}(r) : \frac{f_N = 0}{EP \leftarrow S_r}$$

$$\mathsf{BCS}(r) : \frac{f_C = 1}{EP \leftarrow S_r}$$

$$\mathsf{BCC}(r) : \frac{f_C = 0}{EP \leftarrow S_r}$$

$$\mathsf{BVS}(r) : \frac{f_V = 1}{EP \leftarrow S_r}$$

$$\mathsf{BVC}(r) : \frac{f_V = 0}{EP \leftarrow S_r}$$

$$\mathsf{BHI}(r) : \frac{f_C = 1 \text{ and } f_Z = 0}{EP \leftarrow S_r}$$

$$\mathsf{BLS}(r) : \frac{f_C = 0 \text{ or } f_Z = 1}{EP \leftarrow S_r}$$

$$\mathsf{BLT}(r) : \frac{f_N \neq f_V}{EP \leftarrow S_r}$$

$$\mathsf{BGE}(r) : \frac{f_N = f_V}{EP \leftarrow S_r}$$

$$\mathsf{BLE}(r) : \frac{f_Z = 1 \text{ or } f_N \neq f_V}{EP \leftarrow S_r}$$

$$\mathsf{BGT}(r) : \frac{f_Z = 0 \text{ and } f_N = f_V}{EP \leftarrow S_r}$$

A.5.5 Call and return

$$\begin{aligned}\mathsf{CALL}(r, p) : {} & \mathsf{NEW}(s) \\ & S[SP][o \dots o + A - 1] \leftarrow EP \\ & EP \leftarrow S_r \\ & F[FP] \leftarrow (F_S(FP), F_N(FP) - (p+1)) \\ & F \leftarrow F \oplus (SP + 1, p + 1)\end{aligned}$$

$$\begin{aligned}\mathsf{RET}(c) : {} & EP \leftarrow S_c[o \dots o + A - 1] \\ & \mathsf{KILL}(c) \\ & F \leftarrow F[1 \dots FP - 2] \oplus (F_S(FP - 1), F_N(FP - 1) + F_N(FP))\end{aligned}$$

o and s may vary between instructions, but should be the same for corresponding CALLs and RETs; s is at least A, and $0 \leq o \leq s - A$.

A.5.6 Catch and throw

$$\begin{aligned}
\mathsf{CATCH}(r) &: S_r \leftarrow FP \\
\mathsf{THROW}(r_1, r_2, r_3) &: EP \leftarrow S_{r_1} \\
&\quad F \leftarrow F[1 \ldots S_{r_2}] \\
&\quad S \leftarrow S[1 \ldots SP - 1] \oplus S_{r_3}
\end{aligned}$$

A.5.7 Stack

$$\begin{aligned}
\mathsf{NEW}() &: S \leftarrow S \oplus \rho[0 \ldots A - 1] \\
&\quad F[FP] \leftarrow (F_S(FP), F_N(FP) + 1) \\
\mathsf{NEW}(s) &: S \leftarrow S \oplus \rho[0 \ldots s - 1] \\
&\quad F[FP] \leftarrow (F_S(FP), F_N(FP) + 1) \\
\mathsf{KILL}(p) &: S[F_S(FP) + p - 1 \ldots SP - 1] \leftarrow S[F_S(FP) + p \ldots SP] \\
&\quad F[FP] \leftarrow (F_S(FP), F_N(FP) - 1) \\
&\quad S \leftarrow S[1 \ldots SP]
\end{aligned}$$

B Assembly language

B.1 Introduction

Mite's assembly language is based on the abstract syntax. Where the two correspond exactly the semantics are the same; the semantics of departures from and extensions to the abstract syntax are given below.

B.2 Metagrammar

The grammar is described in a BNF-like notation. **Terminal tokens** are shown `thus`, and **non-terminal tokens** <*thus*>. Space or lack of it between tokens, including line breaks, is significant. **Terms** are formed from tokens and the following operators, given in decreasing order of precedence:

Zero or more repetitions of a term are denoted by appending an asterisk, thus: A^*.

One or more repetitions of a term are denoted by appending a plus sign, thus: A^+.

Lists are denoted by a single terminal character before a repetition symbol: for example, <*ship*>`,`$^+$ denotes a comma-separated list of one or more ships.

Concatenation is denoted by textual concatenation, thus: AB.

Alternation is denoted by a vertical bar, thus: $A \mid B$.

Optional terms are enclosed in brackets: `A cat's[-tail [causes]] wounds!`

Parentheses may be used to override precedence: for example, (<*A*> | <*B*>)<*C*> means "<*A*> or <*B*>, followed by <*C*>".

A **production** consists of the non-terminal being defined, followed by an equals sign, followed by the defining term: <*insect*> = <*head*><*thorax*><*abdomen*>.

B.3 Identifier

```
      <d-digit> = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
      <h-digit> = <d-digit> | A | B | C | D | E | F
       <letter> = a |...| z | A |...| Z
<alphanumeric> = <letter> | <d-digit> | _ | .
   <identifier> = (<letter> | _)<alphanumeric>*
```

An identifier is a string of letters, numbers, underscores and full stops, starting with a letter or underscore.

B.4 Number

```
<natural> = <h-digit>+[:(b | o | d | h)]
<integer> = [-]<natural>
   <size> = <natural>[@<natural>[@<natural>]]
 <offset> = <integer>[@<integer>[@<integer>]]
  <width> = 1 | 2 | 4 | a
```

A natural number is a string of hex digits (see section B.3) optionally followed by a colon and a base (`b` for binary, `o` for octal, `d` for decimal and `h` for hexadecimal); numbers may only contain digits allowed by the base. If there is no base the number is decimal. An integer is a natural with optional initial minus sign. Three-component numbers have the components separated by `@`.

Widths are given in bytes; `a` represents $A/8$.

B.5 Item

```
<item> = <natural>
 <reg> = <item>
```

A *<reg>* is a register; T is not directly accessible.

B.6 Label

<l-label> = `.`*<identifier>*

<x-label> = `x`*<l-label>*

<label> = *<l-label>* | *<x-label>*

<label-exp> = *<label>*[+*<size>*]

A local label (*<l-label>*) is the address of a location (see section B.9). An external label (*<x-label>*) refers to a label in another module.

When a label is used as an address the semantics of the instruction are preceded by $T \leftarrow l$, and the label is replaced by T in the instruction's signature.

B.7 Manifest

<constant> = `ashift`

<manifest> = `#`*<offset>* | *<constant>* | *<label-exp>*

The value of the constant `ashift` is $\log_2 A/8$.

B.8 Instruction

<instruction> = *<assignment>* | *<dataproc>* | *<memory>* | *<branch>* | *<callret>* | *<throwcat>* | *<stack>* | *<escape>* | *<datum>*

B.8.1 Assignment

<assignment> = `MOV` *<reg>*`,`(*<reg>*|*<manifest>*) |
`DEF` *<reg>*`,`*<manifest>* |
`UNDEF` *<reg>* |
`SWAP` *<reg>*`,`*<reg>*

The instruction $\boxed{\texttt{MOV}\ r\texttt{,}m}$, where m is a manifest, has the semantics

$$S_r \leftarrow m$$

The instruction $\boxed{\texttt{DEF}\ r\texttt{,}m}$ has the same semantics as $\boxed{\texttt{MOV}\ r\texttt{,}m}$, but the register is made constant. The instruction $\boxed{\texttt{UNDEF}\ r}$ makes r non-constant, as does `MOV` when applied to a constant register. Constant registers may only be modified by `DEF`, `UNDEF`, or `MOV`.

B.8.2 Data processing

```
<dataproc> = <2-op> <reg>,<reg> |
             <3-op> [<reg>],<reg>,<reg> |
             <4-op> [<reg>],[<reg>],<reg>,<reg>

   <3-op> = <arithmetic> | <logical> | <shift>

   <2-op> = NEG | NOT
```

When a destination is omitted, it is T. The only 3-operand instructions whose destination may be omitted are SUB, AND and XOR. At most one destination may be omitted in a 4-operand instruction.

B.8.2.1 Arithmetic

```
<arithmetic> = ADD | SUB | MUL

      <4-op> = DIV[S[Z]]
```

The instruction $\boxed{\texttt{DIV[S[Z]]}\ q, r, x, y}$ has the semantics

$$\text{DIV[S[Z]]}(q, x, y)$$
$$\text{REM[S[Z]]}(r, x, y)$$

q and r must be distinct.

B.8.2.2 Logical

```
<logical> = AND | OR | XOR

  <shift> = SL | SRL | SRA
```

B.8.3 Memory

```
<memory> = LD_<width> <reg>,[<reg>[,<reg>]] |
           ST_<width> <reg>,[<reg>[,<reg>]] |
           COPY_<size> <item>,<item>
```

$\boxed{\texttt{LD or ST\_}w\ r_1, [r_2, r_3]}$ has the semantics

$$T \leftarrow S_{r_2} + S_{r_3}$$
$$\text{LD or ST}(w, r_1, T)$$

B.8.4 Branch

$<$*condition*$>$ = `AL` | `EQ` | `NE` | `MI` | `PL` | `CS` | `CC` | `VS` | `VC` | `HI` | `LS` | `LT` | `GE` | `LE` | `GT`

$<$*address*$>$ = $<$*reg*$>$ | $<$*label*$>$

$<$*branch*$>$ = `B`$<$*condition*$>$ $<$*address*$>$

A branch to the value of a register must be to an indirectable label (see section B.9.2). The types of stack items active at both branch and destination must match (including the constancy of registers and values of constants).

B.8.5 Call and return

$<$*type-list*$>$ = `[(`$<$*natural*$>$`,`$<$*size*$>$`),`*`[`$<$*natural*$>$`]]`

$<$*reg-list*$>$ = `[`$<$*reg*$>$`,`*`]`

$<$*callret*$>$ = `CALL[F[V]]` $<$*address*$>$`,`$<$*natural*$>$`,`$<$*type-list*$>$ `[`$<$*sync*$>$`]` |
`CALLFC[V]` $<$*address*$>$`,`$<$*natural*$>$`,`$<$*item*$>$ `[`$<$*sync*$>$`]` |
`RET[F]` $<$*item*$>$`,`$<$*reg-list*$>$

In the instruction $\boxed{\texttt{CALL}\ a, p, [t_1, \ldots, t_n]}$, a must be the value of a subroutine label (see section B.9.4). If the address is a register, it must hold the address of an indirectable subroutine (see section B.9.2). p is the number of parameters. The type list gives the format of the return values, from bottom-most to top-most on the stack. The list items give alternately a number of registers followed by the size of a chunk. All chunk sizes must be non-zero. Any registers returned are ranked in descending order from the top of the stack downwards, and the return values are ranked above registers already on the stack. $<$*sync*$>$ is described in section B.8.6.

`CALLF` has the same effect as `CALL` except that the system calling convention is used, and the number of return values must be zero or one. The first argument goes on top of the stack. `CALLFV` and `CALLFCV` are used to call a variadic function (see section B.9.4); in this case the second operand is the total number of parameters being passed. `CALLFC` and `CALLFCV` are used when the function returns a chunk, and the third operand gives either a register holding the address to which the return value should be copied, or the chunk in which it should be stored.

In the instruction $\boxed{\texttt{RET}\ c, [r_1, \ldots, r_n]}$, c must be the chunk placed on the stack on entry to the subroutine or function. The register list gives the return values, which must be in ascending stack order, and match the types in the corresponding `CALL` instruction. A `RET` is assumed to return from the textually most recently declared subroutine or function. `RET` must be used to return from subroutines, and `RETF` from functions.

B.8.6 Catch and throw

```
<throwcat> = CATCH <reg>,<l-label> |
             THROW <reg>,<reg>,<reg> [<sync>]

    <sync> = SYNC <l-label>
```

In the instruction $\boxed{\texttt{CATCH}\ s\texttt{,}l}$, *l* must be a handler's label (see section B.9.3) in the current subroutine. *s* is set to the corresponding stack pointer. In $\boxed{\texttt{THROW}\ l\texttt{,}s\texttt{,}c}$, *l* must be the value of a handler's label, and *s* the address returned by `CATCH` for that label. The `CATCH` must have been executed in the current subroutine or function, or one of its callers.

A `SYNC` is performed before the semantics of the instruction to which it is attached. In $\boxed{\texttt{SYNC}\ l}$, *l* must be a handler's label in the current subroutine. When a handler is reached via a `THROW` instruction, registers other than the top-most stack item have the same value as just before the last `SYNC` performed for that handler's label in the instantiation of the subroutine or function which is thrown to, provided they have not been altered since.

B.8.7 Stack

```
<stack> = NEW[_<size>] |
          KILL |
          RANK <reg>,<natural> |
          REBIND
```

The instruction $\boxed{\texttt{NEW}}$ creates a register with undefined value. The instruction $\boxed{\texttt{NEW\_}s}$ creates an *s*-byte chunk. $\boxed{\texttt{KILL}}$ kills the top-most item on the stack. Items further down may not be killed.

Registers are ranked, the ranking giving the order in which they would ideally be assigned to physical registers. The rankings are distinct and contiguous, the highest being 1. The instruction $\boxed{\texttt{RANK}\ r\texttt{,}n}$ changes the rank of register *r* to *n*. *n* must be between 1 and the number of registers. When a register is killed or ranked, the rankings of the other registers are adjusted accordingly. Newly created registers have rank 1. $\boxed{\texttt{REBIND}}$ causes the bindings of virtual to physical registers to be updated to reflect the current ranking.

Stack instructions are interpreted statically: the `NEW` and `KILL` for a register must textually enclose all other uses. Apart from `NEW` and `KILL` the only instruction that affects the state of the stack seen by the textually next directive (see section B.10) is `CALL`, which kills the parameters and creates the return values.

B.8.8 Escape

```
<escape> = ESC #<natural>
```

$\boxed{\texttt{ESC}}$ performs arbitrary actions.

B.8.9 Datum

$<$*datum*$>$ = `LIT_`$<$*width*$>$ $<$*manifest*$>$`,`$^+$ |
`SPACE[Z]_`$<$*width*$>$ $<$*size*$>$

The instruction $\boxed{\texttt{LIT\_}w\ v_1\texttt{,}\dots\texttt{,}v_n}$ places the values v_1 to v_n in contiguous locations, starting at the next w-aligned address after the preceding datum, if any. Label values may only be used when w is `a`.

The instruction $\boxed{\texttt{SPACE[Z]\_}w\ n}$ reserves n w-words, starting at the next w-aligned address after the preceding datum, if any. If the `Z` modifier is used the space is zero-initialised.

B.9 Location

$<$*location*$>$ = $<$*code*$>$ | $<$*handler*$>$ | $<$*subroutine*$>$ | $<$*function*$>$ | $<$*data*$>$

A location assigns the address of the next piece of code or data to the given label.

A datum (see section B.8.9) may only appear after a $<$*data*$>$, and other instructions may only appear after a non-$<$*data*$>$ location. The flow of control must never fall through to a location other than a code or handler labelling.

B.9.1 Labelling

$<$*labelling*$>$ = `[p]`$<$*l-label*$>$

If `p` is used the label is public, and may be visible outside the module.

B.9.2 Code

$<$*code*$>$ = `[i]`$<$*labelling*$>$

If `i` is used the label may be used as the target of an indirect (register) branch.

B.9.3 Handler

$<$*handler*$>$ = `h`$<$*labelling*$>$

A handler is the same as a labelling, except that it may also be given as the label for a `CATCH` instruction (see section B.8.6). The top-most stack item at a handler must be a non-constant register.

B.9.4 Subroutine and function

<subroutine> = `s[l]`*<labelling>*

<function> = `f[l][c][v]`*<labelling>*

The code in a subroutine or function extends from its labelling to the next subroutine or function labelling. The state of the stack directly before the subroutine or function specifies the number and type of its parameters (with the exception of a function's variadic parameters). Subroutines must be reached by `CALL`, and functions by `CALLF`.

If `l` is used in a subroutine or function labelling, the subroutine or function is a leaf routine, and may not perform any `CALL` instructions.

If `c` is used in a function labelling, the function returns a chunk. If `v` is used, the function is variadic, and the `V` modifier must be added to `CALLF`. On entry to a variadic function the variadic arguments are stored in chunk 1, which should be declared with size `0`. Their layout is system-dependent. The non-variadic arguments should be declared as normal.

The return chunk, which is placed on top of the stack on entry to a function or subroutine, must not be written to.

B.9.5 Data

<data> = `d[r]`*<labelling>*

If `r` is used the data up to the next data location are read-only; otherwise they are writable. The data in a program define the initial contents of the memory. A data labelling has the same alignment as the first datum following it (see section B.8.9).

B.10 Directive

<directive> = *<instruction>* | *<location>*

B.11 Module

<module> = *<directive>*$^+$

B.12 Comments

A comment, starting with a semicolon, may be placed at the end of any line or on a line by itself.

C Object format

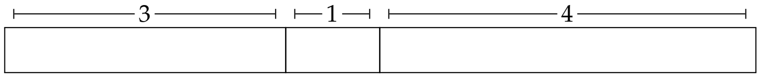
C.1 Introduction

Mite's assembler writes object modules in the format given below, which is a direct encoding of the concrete syntax.

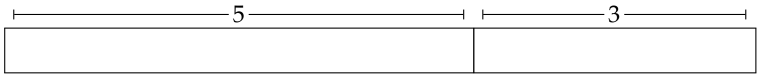
C.2 Presentation

Hexadecimal numbers are followed by an "h"; for example, 100 = 64h. Binary numbers are similarly suffixed "b".

The encoding is presented diagrammatically. Boxes representing bit fields are concatenated to make bytes or larger words:

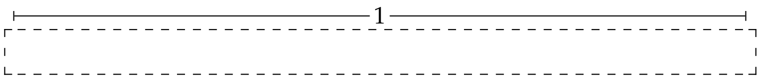


These in turn are listed vertically. The contents of a bit field is given either as literal binary digits (`01101001`), or as a name. Fields are usually labelled with their width:

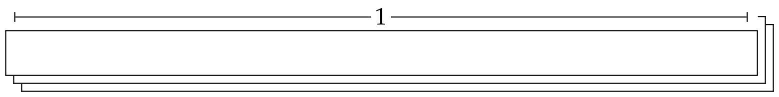


The most significant bit is at the left-hand end of the word, and the least significant at the right-hand end. Multi-byte words are stored with their bytes in little-endian order.

Boxes labelled in ordinary type (box) represent units which themselves have internal structure. Boxes labelled in italics (*scatola*) are numbers (see section C.3). Optional units are shown as dashed boxes:



Lists are denoted by a stack:



C.3 Number

Unsigned numbers are encoded as follows:

1. A list of 7-bit words is formed by repeatedly removing the least significant seven bits of the number until none of the remaining bits is set.
2. The 7-bit words are turned into bytes by the addition of a bit at the most significant end, which is zero for all the quantities except the first.
3. The bytes are stored in the reverse order to that in which they were generated.

Signed numbers are encoded in the same way except that the list is formed by repeatedly removing the least significant seven bits of the number until all the remaining bits are the same as the most significant bit of the previous 7-bit word. Three-component numbers are encoded as three consecutive numbers.

C.3.1 Width

Widths of quantities are encoded as

Width	**Code**
`1`	00b
`2`	01b
`4`	10b
`a`	11b

C.4 Identifier

All strings are ASCII-encoded, preceded by a number giving their length.

C.5 Item

Stack items are encoded as a number (see section C.3).

C.6 Address

Local addresses give the number of a label (see section C.10), and are stored as a number. External addresses are stored as an identifier.

Address types are encoded as

Type	Code
Register	00b
Local label	01b
Global label	10b

C.7 Manifest

The type of a manifest quantity is given by an op type field, which is encoded as

Operand type	op type
number	000b
3-component number	001b
constant	010b
local label	100b
local label plus offset	101b
external label	110b
external label plus offset	111b

Constants are encoded as a single byte; for `ashift` the byte is 00h. A label expression is represented as the number or name of the label followed by the offset, which is a three-component number (see section C.3).

C.8 Lists

The list elements are stored consecutively. The length is encoded as a number directly before the list elements.

C.9 Instruction

Instructions are encoded as the opcode followed by the operands, encoded in order from left to right. The operands are encoded as in the preceding sections; lists of items and types enclosed in brackets are stored as lists.

C.9.1 `MOV` and `DEF`

A `MOV` instruction whose second operand is a register is encoded as 0000 0000b. `DEF` and `MOV` with a manifest second operand are encoded as

1	3	1	3
0	`011`	inst	op type

where the inst bit is clear for `MOV` and set for `DEF`, and the op type field gives the type of the value, encoded as in section C.7.

C.9.2 Data processing

C.9.2.1 Three-operand

1	3	3	1
0	inst	`011`	dest

The inst field indicates the instruction:

Instruction	**inst**
`ADD`	000b
`SUB`	001b
`AND`	010b
`OR`	100b
`XOR`	101b

The dest bit is set if the destination is present.

C.9.2.2 Four-operand

1	2	3	1	1
0	inst	`011`	quot	rem

The inst field indicates the instruction:

Instruction	**inst**
`DIV`	00b
`DIVS`	01b
`DIVSZ`	10b

The quot bit is set if the first destination is present, and the rem bit if the second destination is present.

C.9.3 Memory

1	1	3	1	2
0	inst	`011`	off	width

The inst bit is clear for `LD` and set for `ST`. The off bit is set if a third (offset) register is given. The width field gives the width of the quantities being transferred, encoded as in section C.3.1.

C.9.4 Branch

2	2	4
`11`	adr	condition

The adr field encodes the address type as in section C.6. The condition is encoded as

Condition	condition
`AL`	0001b
`EQ`	0010b
`NE`	0011b
`MI`	0100b
`PL`	0101b
`CS`	0110b
`CC`	0111b
`VS`	1000b
`VC`	1001b
`HI`	1010b
`LS`	1011b
`LT`	1100b
`GE`	1101b
`LE`	1110b
`GT`	1111b

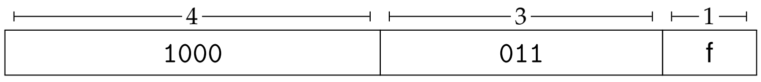
C.9.5 Call and return

`CALL` is encoded as

3	1	1	1	2
`011`	f	c	v	adr

where the f bit is set for `CALLF`, the c bit is set if the `C` modifier is used, and the v bit if the `V` modifier is used. The adr field encodes the address type as in section C.6. The argument types are alternately 1 and 3-component numbers.

`RET` is encoded as

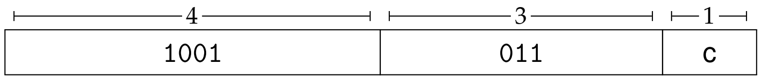


where the `f` bit is set for `RETF`.

C.9.6 `SYNC`

`SYNC` is encoded as a separate instruction immediately following the instruction to which it is attached. Its opcode is 0001 0101b. It is not counted as a separate directive in the count in the module header (see section C.12).

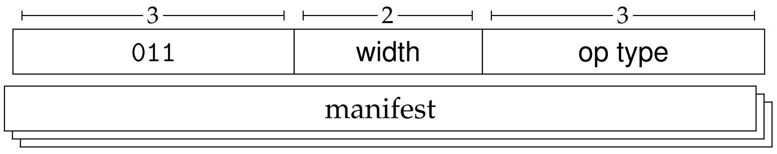
C.9.7 `NEW`



If the c bit is set, a chunk is being declared.

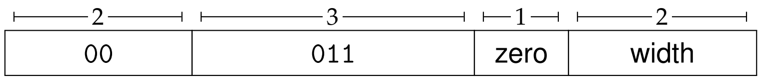
C.9.8 Datum

C.9.8.1 Literal



The width field gives the width of the literals. The op type field gives the literal type, encoded as in section C.7. The list of manifests follows.

C.9.8.2 Space



The zero bit is set if the space is zero-initialised; the width field gives the width of the words being reserved.

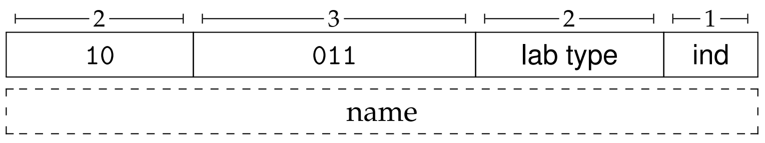
C.9.9 Other instructions

The remaining instructions are encoded thus:

Instruction	inst
`UNDEF`	0000 0001b
`SWAP`	0000 0010b
`NEG`	0000 0100b
`NOT`	0000 0101b
`MUL`	0000 1000b
`SL`	0000 1001b
`SRL`	0000 1010b
`SRA`	0001 0000b
`COPY`	0001 0001b
`CATCH`	0001 0010b
`THROW`	0001 0100b
`KILL`	0010 0000b
`RANK`	0010 0001b
`REBIND`	0010 0010b
`ESC`	0010 0100b

C.10 Location

Labellings, handlers and subroutines are encoded thus:



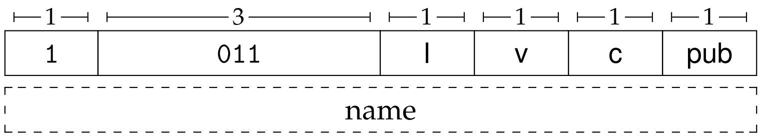
The lab type field gives the type of label, encoded as

Label type	lab type
ordinary	00b
handler	01b
subroutine	10b
leaf subroutine	11b

If the ind bit is set the label is indirectable. A public label has an identifier after the opcode byte, starting with an underscore. (This means that the encoding is ambiguous, as the underscore could also represent part of another instruction.)

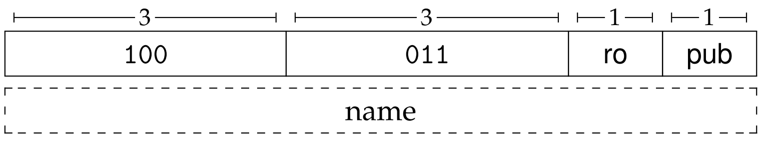
The labels are numbered consecutively from one.

Functions are encoded



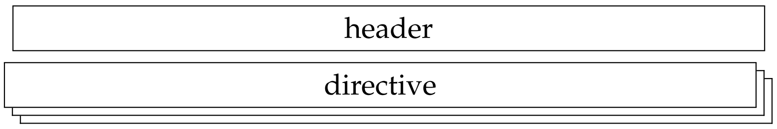
where the l bit is set if the function is a leaf, the v bit is set if it is variadic, the c bit is set if it returns a chunk, and the pub bit if the label is public. A public function has an identifier after the opcode byte.

C.11 Data

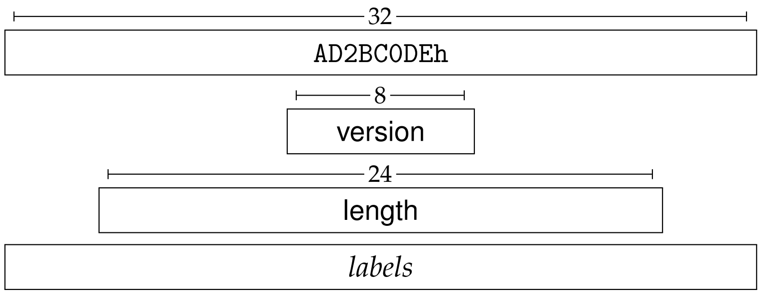


If the ro bit is set the following data is read-only; otherwise it is read-write. If the pub bit is set it is public, otherwise it is private. A public data label has an identifier after the opcode byte.

C.12 Module



A module consists of a header and a list of directives.



The header starts with a magic number. Next comes a byte containing the version number of the encoding. The current version number is 0. Next comes the length of the module in bytes excluding the header, and finally the number of labels.

D Source code of tests

D.1 The fast-Fourier transform test (fft)

```
#include <stdio.h>
#include <stdlib.h>

#define modulus 0x10001   /* 2**16 + 1 */
#define omega 0x00003
const int ln = 16;
#define N (1<<ln)         /* N is a power of 2 */
#define upb (N-1)
int *data;

int add(int x, int y) {
       int a = x+y;
       return a < modulus ? a : a - modulus;
}
#define neg(x) (modulus-(x))
#define sub(x, y) add((x), neg(y))
int mul(int x, int y) {
    if (x == 0) return 0;
    if ((x & 1) == 0) return mul(x>>1, add(y,y));
    return add(y, mul(x>>1, add(y,y)));
}

void fft(int *v, int ln, int w)  /* ln=log2 n, w=nth root of unity */ {
    int n = 1<<ln;
    int *vn = v+n;
    int s;

  { int j = 0, i;
    for (i = 0; i < n-1; i++) {
      int k = n>>1;
      if (i<j) { int t = v[j]; v[j] = v[i]; v[i] = t; }
      while (k<=j) { j -= k; k >>= 1; } /* "increment" j */
      j += k;
    }
  }


  for (s = 1; s <= ln; s++) {
    int m = 1<<s;
    int m2 = m>>1;
    int wk = 1, wkfac = w, i, j;

    for (i = s+1; i <= ln; i++) wkfac = mul(wkfac, wkfac);
    for (j = 0; j < m2; j++) {
      int *p = v+j;

      while (p<vn) { /* the butterfly operation */
        int a = *p, b = mul(p[m2], wk);
        *p = add(a,b);  p[m2] = sub(a, b);
        p += m;
      }
      wk = mul(wk, wkfac);
    }
  }
}

void pr(int *v, int max) {
    int i;

    for (i = 0; i <= max; i++) {
        printf("%5d ", v[i]);
        if (i % 8 == 7) putchar('\n');
    }

    putchar('\n');
}

int main(void) {
   int i;

   printf("fft with N = %d and omega = %d modulus = %d\n\n",
                    N,         omega,     modulus);

   data = (int *)malloc(upb * sizeof(int));

   for (i = 0; i <= upb; i++) data[i] = i;
   pr(data, 7);

   fft(data, ln, omega);
   pr(data, 7);
   return 0;
}
```

D.2 Pyramid register allocation test (pyram)

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y, loop;

    a= rand();  b= rand();  c= rand();  d= rand();  e= rand();
    f= rand();  g= rand();  h= rand();  i= rand();  j= rand();
    k= rand();  l= rand();  m= rand();  n= rand();  o= rand();
    p= rand();  q= rand();  r= rand();  s= rand();  t= rand();
    u= rand();  v= rand();  w= rand();  x= rand();  y= rand();

    for (loop= 0; loop < 1000000; loop++) {
    a= a+1;
    b= a-b+1;
    c= a+b-c+1;
    d= a-b+c-d+1;
    e= a+b-c+d-e+1;
    f= a-b+c-d+e-f+1;
    g= a+b-c+d-e+f-g+1;
    h= a-b+c-d+e-f+g-h+1;
    i= a+b-c+d-e+f-g+h-i+1;
    j= a-b+c-d+e-f+g-h+i-j+1;
    k= a+b-c+d-e+f-g+h-i+j-k+1;
    l= a-b+c-d+e-f+g-h+i-j+k-l+1;
    m= a+b-c+d-e+f-g+h-i+j-k+l-m+1;
    n= a-b+c-d+e-f+g-h+i-j+k-l+m-n+1;
    o= a+b-c+d-e+f-g+h-i+j-k+l-m+n-o+1;
    p= a-b+c-d+e-f+g-h+i-j+k-l+m-n+o-p+1;
    q= a+b-c+d-e+f-g+h-i+j-k+l-m+n-o+p-q+1;
    r= a-b+c-d+e-f+g-h+i-j+k-l+m-n+o-p+q-r+1;
    s= a+b-c+d-e+f-g+h-i+j-k+l-m+n-o+p-q+r-s+1;
    t= a-b+c-d+e-f+g-h+i-j+k-l+m-n+o-p+q-r+s-t+1;
    u= a+b-c+d-e+f-g+h-i+j-k+l-m+n-o+p-q+r-s+t-u+1;
    v= a-b+c-d+e-f+g-h+i-j+k-l+m-n+o-p+q-r+s-t+u-v+1;
    w= a+b-c+d-e+f-g+h-i+j-k+l-m+n-o+p-q+r-s+t-u+v-w+1;
    x= a-b+c-d+e-f+g-h+i-j+k-l+m-n+o-p+q-r+s-t+u-v+w-x+1;
    y= a+b-c+d-e+f-g+h-i+j-k+l-m+n-o+p-q+r-s+t-u+v-w+x-y+1;
    }

    printf("%d %d %d %d %d %d %d %d %d %d %d %d %d %d %d "
      "%d %d %d %d %d %d %d %d %d %d\n",
      a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y);
    return 0;
}
```

E Results of the tests

This appendix lists the data on which figures 6.1–6.7 are based. The data are tabulated in the same order as the figures.

Test	GNU C	LCC	Mite	Mite − nops
`switch`	1,292	2,228	4,800	4,532
`wf1`	616	1,100	1,820	1,688
`14q`	384	620	728	616
`stan`	6,364	12,144	13,352	12,544
`fft-1`	952	2,076	2,116	2,044
`fft-3`	952	2,076	1,740	1,672
`fft-6`	952	2,076	1,724	1,652
`fft-7`	952	2,076	1,604	1,536
`pyram`	1,072	1,980	1,584	1,576
`pyr-bad`	1,072	1,980	2,028	2,020

Table E.1: Native code size/bytes

Test	GNU C	LCC	Mite
`switch`	2,260	2,844	2,609
`wf1`	1,472	1,628	1,325
`14q`	1,168	1,120	576
`stan`	8,136	13,000	10,626
`fft-1`	1,844	2,604	1,213
`fft-3`	1,844	2,604	1,116
`fft-6`	1,844	2,604	1,119
`fft-7`	1,844	2,604	1,132
`pyram`	1,980	2,868	1,116
`pyr-bad`	1,980	2,868	1,209

Table E.2: Executable file size/bytes

Test	Memory allocated	Code + data generated
`switch`	67,754	5,128
`wf1`	104,392	33,877
`14q`	8,356	1,084
`stan`	336,390	83,191
`fft-1`	25,383	2,178
`fft-3`	20,006	1,834
`fft-6`	20,395	1,786
`fft-7`	20,207	1,666
`pyram`	18,882	1,660
`pyr-bad`	20,062	2,104

Table E.3: Memory consumption/bytes

Test	GNU C	LCC	Mite
`switch`	0·05	0·05	0·11
`wf1`	0·06	0·06	0·13
`14q`	38·21	46·97	62·96
`stan`	14·60	32·70	33·08
`fft-1`	2·79	6·57	7·32
`fft-3`	2·79	6·57	6·53
`fft-6`	2·79	6·57	2·82
`fft-7`	2·79	6·57	2·80
`pyram`	1·01	1·66	0·93
`pyr-bad`	1·01	1·66	1·72

Table E.4: Execution time/s

Test	Translation	Run
`switch`	0·021	0·05
`wf1`	0·037	0·06
`14q`	0·003	62·92
`stan`	0·113	32·93
`fft-1`	0·008	7·28
`fft-3`	0·010	6·48
`fft-6`	0·010	2·77
`fft-7`	0·007	2·75
`pyram`	0·007	0·84
`pyr-bad`	0·007	1·63

Table E.5: Translation and running time/s

Test	3	4	5	6	7	8	9	10
`switch`	0·07	0·07	0·07	0·07	0·06	0·07	0·07	0·07
`wf1`	0·09	0·09	0·09	0·09	0·09	0·09	0·09	0·09
`14q`	114·95	97·92	83·64	85·26	58·45	63·11	63·11	63·30
`stan`	44·46	41·18	37·73	35·16	34·19	33·88	33·68	33·44
`fft-1`	8·10	8·08	7·77	7·56	7·31	7·30	7·31	7·32
`fft-7`	3·23	3·02	2·81	2·77	2·77	2·76	2·76	2·75
`pyram`	1·91	1·69	1·45	1·44	1·44	1·02	0·95	0·93
`pyr-bad`	1·88	1·85	1·82	1·81	1·79	1·78	1·77	1·72

Table E.6: Effect of number of physical registers on execution time/s

Test	3	4	5	6	7	8	9	10
`switch`	5,320	5,044	4,704	4,788	4,792	4,800	4,800	4,800
`wf1`	2,348	2,180	1,916	1,868	1,840	1,820	1,820	1,820
`14q`	1,164	1,004	888	840	708	728	728	728
`stan`	17,100	15,644	15,000	14,188	13,668	13,536	13,472	13,352
`fft-1`	2,464	2,404	2,252	2,220	2,140	2,112	2,096	2,116
`fft-7`	2,060	1,900	1,816	1,688	1,656	1,700	1,668	1,604
`pyram`	2,100	1,968	1,824	1,816	1,804	1,636	1,588	1,584
`pyr-bad`	2,344	2,308	2,276	2,256	2,084	2,080	2,068	2,028

Table E.7: Effect of number of physical registers on code size/bytes

Test	3	4	5	6	7	8	9	10
`switch`	68,658	68,126	67,658	67,742	67,746	67,754	67,754	67,754
`wf1`	105,656	105,136	104,584	104,440	104,412	104,392	104,392	104,392
`14q`	9,560	9,112	8,804	8,660	8,336	8,356	8,356	8,356
`stan`	346,340	343,156	341,136	338,948	337,308	336,952	336,664	336,390
`fft-1`	26,115	26,087	25,647	25,583	25,375	25,315	25,235	25,383
`fft-7`	21,303	20,887	20,739	20,419	20,355	20,431	20,367	20,207
`pyram`	20,166	19,842	19,474	19,466	19,454	19,030	18,886	18,882
`pyr-bad`	20,538	20,470	20,438	20,418	20,150	20,146	20,134	20,062

Table E.8: Effect of number of physical registers on memory allocation/bytes

Test	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
`switch`	0·021	0·021	0·021	0·021	0·020	0·020	0·020	0·020
`wf1`	0·037	0·036	0·036	0·036	0·036	0·036	0·036	0·036
`14q`	0·004	0·004	0·003	0·003	0·003	0·003	0·003	0·003
`stan`	0·116	0·114	0·113	0·112	0·111	0·111	0·111	0·110
`fft-1`	0·009	0·009	0·008	0·008	0·008	0·008	0·008	0·008
`fft-7`	0·008	0·008	0·007	0·007	0·007	0·007	0·007	0·007
`pyram`	0·008	0·007	0·007	0·007	0·007	0·007	0·007	0·007
`pyr-bad`	0·008	0·008	0·008	0·008	0·007	0·007	0·007	0·007

Table E.9: Effect of number of physical registers on translation time/s

Bibliography

All the references amassed during my research are reproduced below in the hope that they form a useful collection. [24] contains an excellent bibliography of earlier work.

[1] Mike Acetta, Robert Baron, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: a new kernel foundation for UNIX development. Technical report, Carnegie Mellon University, 1986.

[2] A. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and language-independent mobile programs. In *Proceedings of the ACM SIGPLAN Symposium on Programming Language Design and Implementation (PLDI '96)*, pages 127–136, Philadelphia, PA, May 1996. ACM.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[4] M. Alfonseca, D. Selby, and R. Wilks. The APL IL interpreter generator. *IBM Systems Journal*, 30(4):490–497, 1991.

[5] J. Alves-Foss, editor. *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[6] American National Standards Institute. *ANS X3.159-1989: Programming Languages—C*, December 1989.

[7] American National Standards Institute. *ANS X3.215-1994: Programming Languages—Forth*, 1994.

[8] American National Standards Institute. *ANS X3.4-1986(R1997): Information Systems: Coded Character Sets—7-Bit American National Standard Code for Information Interchange*, 1997.

[9] Arm Limited. *The ARM–THUMB Procedure Call Standard*, 1998. http://www.arm.com/.

[10] John Batali, Edmund Goodhue, Chris Hanson, Howie Shrobe, Richard M. Stallman, and Gerald Jay Sussman. The Scheme-81 architecture—system and chip. In Paul Penfield, Jr., editor, *Proceedings of the MIT Conference on Advanced Research in VLSI*, Dedham, Mass., 1982.

[11] W. Gurney Benham, editor. *Cassell's Book of Quotations, Proverbs and Household Words*. Cassell, revised edition, 1914.

[12] Manuel E. Benitez and Jack W. Davidson. The advantages of machine-dependent global optimization. In *International Conference on Programming Language and Architectures (PLSA '94)*, pages 105–123, 1994.

[13] Lennart Benschop. Sod32, 1995. Posted to comp.sources.misc, Volume 46, Issue 7.

[14] Valer Bocan. Delta forth, 1999. http://www.dataman.ro/dforth/.

[15] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software—Practice and Experience*, 18(9):807–820, 1988.

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

[17] Barry B. Brey. *Programming the 80286, 80386, 80486 and Pentium-based Personal Computer*. Prentice-Hall, 1996.

[18] Leo Brodie. *Thinking FORTH*. Prentice-Hall, 1984.

[19] Leo Brodie. *Starting FORTH*. Prentice-Hall, second edition, 1987.

[20] Fred P. Brooks. *The Mythical Man-Month*. Addison-Wesley, anniversary edition, 1995.

[21] Michael Brown. PASM—portable runtime assembler. http://www.washery.com/projects/pasm/.

[22] P. J. Brown. *Writing Interactive Compilers and Interpreters*. Wiley, 1979.

[23] Harold Carr and Robert R. Kessler. An emulator for Utah Common Lisp's abstract virtual register machine. In *Proceedings of the 1987 Rochester Forth Conference*, pages 113–116, Rochester, NY, 1987.

[24] Eddy H. Debaere and Jan M. Van Campenhout. *Interpretation and Instruction Path Coprocessing*. MIT Press, 1990.

[25] distributed.net. http://www.distributed.net/.

[26] S. Dorward et al. Inferno. In *IEEE Compcon '97 Proceedings*, 1997.

[27] W. Earle. Compress ROM programs with a math-function interpreter. *Electronic Design News*, March 31st 1982.

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

[29] Dawson R. Engler. VCODE: A retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the 23rd Annual ACM Conference on Programming Language Design and Implementation*, 1996. http://www.pdos.lcs.mit.edu/~engler/.

[30] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Programming Languages*, 1996.

[31] M. Anton Ertl. A portable Forth engine. In *EuroFORTH '93 conference proceedings*, Mariánské Láznè (Marienbad), 1993.

[32] M. Anton Ertl. Stack caching for interpreters. In *EuroForth '94 Conference proceedings*, pages 3–12, Winchester, UK, 1994.

[33] Marc Feeley and James S. Miller. A parallel virtual machine for efficient Scheme compilation. In *Proceedings of the 1990 ACM Conference*, pages 119–130, Nice, France, 1990.

[34] P. J. Fleming and J. J. Wallace. How not to lie with statistics—the correct way to summarise benchmark results. *Communications of the ACM*, 29(3):218–221, March 1986.

[35] Free Software Foundation. GCC compiler collection. http://www.gnu.org/software/gcc/.

[36] Free Software Foundation. GNU C library. http://www.gnu.org/software/libc/.

[37] Michael Franz and Thomas Kistler. Introducing Juice, 1996. http://caesar.ics.uci.edu/juice/intro.html.

[38] Christopher Fraser and David Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, 1995.

[39] Richard M. Fujimoto. The virtual time machine. In F. Leighton, editor, *Proceedings of the 1989 ACM Symposium*, pages 35–44, Santa Fe, Mexico, 1989.

[40] Jim Galbreath. A high-level language benchmark. *BYTE*, 6(9):180–198, 1989.

[41] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[42] L. George. MLRISC: Customizable and reusable code generators. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, 1997.

[43] James Gosling and Henry McGilton. The Java language environment: A white paper, May 1996. http://java.sun.com/.

[44] Numerical Algorithms Group. NAG FORTRAN library. http://www.nag.com/.

[45] David R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software—Practice and Experience*, 20(1):5–12, 1990.

[46] Steven Hardy. The POPLOG programming system. Technical Report CSRP 003, University of Sussex, 1982.

[47] J. Hennessy and P. Nye. *Stanford Integer Benchmarks*. Stanford University. ftp://ftp.complang.tuwien.ac.at/pub/forth/.

[48] C. B. Hill, editor. *Apophthegms from Hawkins's edition of Johnson's works*, volume ii of *Johnsonian Miscellanies*. 1897.

[49] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.

[50] Rolf Hoffmann. A classification of interpreter systems. *Microprocessing and Microprogramming*, 12:3–8, 1983.

[51] Ian Holyer. *Functional Programming with Miranda*. Pitman, 1991.

[52] Wei-Chung Hsu, Charles N. Fischer, and James R. Goodman. On the minimization of loads/stores in local register allocation. *IEEE Transactions on Software Engineering*, 15(10):1252–1260, October 1989.

[53] Institute of Electrical and Electronics Engineers. *IEEE 754-1985(R1994): IEEE Standard for Binary Floating-Point Arithmetic*, 1994.

[54] International Organization for Standardization. *ISO/IEC 10646-1:1993 Information technology—Universal Multiple-Octet Coded Character Set (UCS)—Part 1: Architecture and Basic Multilingual Plane*, 1993.

[55] International Organization for Standardization. *ISO/IEC 9899-1999: Programming Languages—C*, December 1999.

[56] David Jaggar. *ARM Architectural Reference Manual*. Prentice Hall Europe, 1996.

[57] Chris Jobson and John Richards. *BCPL for the BBC Microcomputer*, 1983.

[58] Richard Jones and Rafael Lins. *Garbage Collection*. John Wiley and Sons Ltd, 1996.

[59] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice-Hall, 1992.

[60] Richard Kelsey, William Clinger, Jonathan Rees, et al. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, August 1998. http://www.swiss.ai.mit.edu/projects/scheme/.

[61] Paul Klint. Interpretation techniques. *Software—Practice and Experience*, 11:963–973, 1981.

[62] Andreas Krall. Efficient JavaVM just-in-time compilation. In *PACT '98 proceedings*, 1998.

[63] Glenn Krasner, editor. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, 1983.

[64] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, January 1964.

[65] Mark Leone and Peter Lee. Deferred compilation: The automation of run-time code generation. Technical Report CMU-CS-93-225, School of Computer Science, Carnegie Mellon University, December 1993.

[66] Mark Leone and Peter Lee. Lightweight run-time code generation. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 97–106, June 1994.

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

[68] R. G. Loeliger. *Threaded Interpretive Languages: Their Design and Implementation*. BYTE Books, Peterborough, NH, 1981.

[69] 180 Software Ltd. ORIGIN white paper, 2000. http://www.180sw.com/.

[70] S. Lucco, O. Sharp, and R. Wahbe. Omniware: A universal substrate for mobile code. In *Proceedings of Fourth International World Wide Web Conference*, Massachusetts Institute of Technology, 1995. http://www.w3.org/pub/Conferences/WWW4/Papers/165/.

[71] Steven Lucco. Split-stream dictionary program compression. In *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, pages 27–34, 2000.

[72] Dis virtual machine specification. In *Inferno User's Guide*, chapter 7. Lucent Technologies, 1997.

[73] Wayne Luk, David Ferguson, and Ian Page. Structured hardware compilation of parallel programs. In W. Moore and W. Luk, editors, *More FPGAs*, pages 213–224. Abingdon EE&CS Books, 1994.

[74] Wayne Luk and Ian Page. Parameterising designs for FPGAs. In W. R. Moore and W. Luk, editors, *FPGAs*, chapter 5.4. Abingdon EE&CS Books, 1991.

[75] Mark Lutz. *Programming Python*. O'Reilly & Associates, October 1996.

[76] J. McCarthy. *LISP programmers' manual*. MIT Computation Center, Cambridge, MA, 1960.

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

[78] Robin Milner. The polyadic π-calculus: A tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 203–246. Springer Verlag, 1993.

[79] Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, Cambridge, MA, 1991.

[80] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[81] Charles Moore and Jeff Fox. Preliminary specification of the F21, 1998. http://www.ultratechnology.com/f21data.pdf.

[82] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, USA, May 1999.

[83] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[84] George Necula. Compiling with proofs. Technical Report CMU-CS-98-154, School of Computer Science, Carnegie Mellon University, September 1998.

[85] K. V. Nori, U. Ammann, H. H. Nabeli, and Ch. Jacobi. Pascal P implementation notes. In D. W. Barron, editor, *Pascal—The Language and its Implementation*, pages 125–170. Wiley, 1981.

[86] Markus F. X. J. Oberhumer. LZO data compression library, 1999. http://wildsau.idv.uni-linz.ac.at/mfx/lzo.html.

[87] The Open Group. *Architecture Neutral Distribution Format (XANDF) Specification*, January 1996. Preliminary Specification P527.

[88] Ian Page and Wayne Luk. Compiling occam into FPGAs. In W. R. Moore and W. Luk, editors, *FPGAs*, chapter 5.3. Abingdon EE&CS Books, 1991.

[89] Nic Peeling. ANDF—some information, November 18th 1993. comp.compilers.

[90] S. Pemberton and M. C. Daniels. *Pascal implementation: the P4 compiler*. John Wiley & Sons, 1982.

[91] Simon Peyton Jones, Thomas Nordia, and Dino Oliva. `C--`: A portable assembly language. In *Proceedings of the 1997 Workshop on Implementing Functional Languages*, 1997.

[92] Simon Peyton Jones and Norman Ramsey. The `C--` run-time interface for concurrency, 2000. http://www.cminusminus.org/abstracts/c--concurrency.html.

[93] Simon Peyton Jones, Norman Ramsey, and Fermin Reig. `C--`: a portable assembly language that supports garbage collection. In *Proceedings of PPDP '99*, 1999.

[94] Haskell 98: A non-strict, purely functional language, 1999. http://www.haskell.org/definition/.

[95] Benjamin C. Pierce. The pict programming language, 1995. http://www.cis.upenn.edu/~bcpierce/.

[96] Matt Pietrek. A crash course on the depths of Win32 structured exception handling, 1997. http://www.microsoft.com/msj/0197/exception/exception.htm.

[97] Rob Pike. Private communication, 1997. Lucent Technologies.

[98] Thomas Pittman. Two-level hybrid interpreter/native code execution for combined space-time program efficiency. In *Symposium on Interpreters and Interpretive Techniques (SIGPLAN '87)*, pages 150–152, 1987.

[99] Massimiliano Poletto, Dawson R. Engler, and M. Frans Kaashoek. tcc: A system for fast, flexible and high-level dynamic code generation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation '97*, 1997.

[100] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, September 1999.

[101] Robin Popplestone. Specifying types by grammars: A polymorphic static type checker operating at a stack machine level. Technical Report FP-94-01, University of Glasgow, 1994.

[102] Robin Popplestone. A typed operational semantics based on grammatical characterisation of an abstract machine. Technical Report FP-94-02, University of Glasgow, 1994.

[103] Norman Ramsey and Simon Peyton Jones. A single intermediate language that supports multiple implementations of exceptions. In *Proceedings of PLDI '00*, 2000.

[104] Martin Richards. Intcode: an interpretive machine code for BCPL. Technical report, University of Cambridge Computer Laboratory, December 1972. Revised August 1975.

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

[106] Martin Richards and Colin Whitby-Strevens. *BCPL—the language and its compiler*. Cambridge University Press, 1979.

[107] Daniel Ridge, Donald Becker, Phillip Merkey, and Thomas Sterling. Beowulf: Harnessing the power of parallelism in a pile-of-PCs. In *Proceedings of IEEE Aerospace*, 1997.

[108] Theodore H. Romer, Geoffrey M. Voelker, Alec Wolman, Wayne A. Wong, Jean-Loup Baer, Brian N. Bershad, and Henry M. Levy. The structure and performance of interpreters. In *Proceedings of ASPLOS VII*, pages 150–159, University of Washington, Seattle.

[109] A. W. Roscoe and C. A. R. Hoare. The laws of occam programming. Technical monograph PRG-53, University of Oxford Computer Laboratory, 1986.

[110] Mark Roulo. Misty Beach Forth: An implementation in Java. *Forth Dimensions*, XIX(4), November–December 1997.

[111] David W. Sandberg. Experience with an object-oriented virtual machine. *Software—Practice and Experience*, 18(5):415–425, 1988.

[112] Richard L. Sites. *Alpha architecture reference manual*. Digital Equipment Corporation, 1992.

[113] Robert Smith, Aaron Sloman, and John Gibson. Poplog's two-level virtual machine support for interactive languages. Technical Report CSRP 153, University of Sussex, 1990.

[114] Robert M. Smith. A high performance version of the POPLOG virtual machine. Technical Report CSRP 117, University of Sussex, 1988.

[115] Stuart Smith. Private communication, 1998. Essex University.

[116] T. B. Steel, Jr. UNCOL. *Ann. Rev. Auto. Prog.*, 2:325–344, 1960.

[117] Tao Systems. Elate, 1999. http://www.tao.co.uk/.

[118] MLj team. The MLj compiler, 1999. http://www.dcs.ed.ac.uk/~mlj/.

[119] Tendra home page, 1998. http://alph.dra.hmg.gb/TenDRA/.

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

[121] Reuben Thomas. The melting machine: from PC to pronit, 1995. http://sc3d.org/rrt/.

[122] Reuben Thomas. Mite: a fast and flexible virtual machine. In *EuroForth '98 conference proceedings*, September 1998. http://sc3d.org/rrt/.

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

[124] Reuben Thomas. The TpForth project. In *EuroForth '99 conference proceedings*, 1999. http://sc3d.org/rrt/.

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

[126] Reuben Thomas. The Mite VM: bridging the complexity gulf. In *EuroForth '01 conference proceedings*, November 2001. http://sc3d.org/rrt/.

[127] Reuben Thomas. The beetle forth virtual machine, 2011. http://sc3d.org/rrt/.

[128] Reuben Thomas. An implementation of the beetle virtual machine for posix, 2011. http://sc3d.org/rrt/.

[129] Reuben Thomas. A simple user-interface for the beetle forth virtual machine, 2011. http://sc3d.org/rrt/.

[130] Tools Interface Standards Committee. *Executable and Linkable Format (ELF)*. http://developer.intel.com/vtune/tis.htm.

[131] Omri Traub, Glenn Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. In *Proceedings of the ACM SIGPLAN '98 conference on Programming language design and implementation*, pages 142–151, 1998.

[132] David A. Turner. A new implementation technique for applicative languages. *Software—Practice and Experience*, 9:31–49, 1979.

[133] David A. Turner. *Recursion equations as a programming language*, pages 1–28. Cambridge University Press, January 1981.

[134] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, second edition, September 1996.

[135] Bruce E. Wampler. The V reference manual, 1999. ftp://objectcentral.com/vref.pdf.

[136] D. H. D. Warren. An abstract Prolog instruction set. Technical Note 300, SRI International, 1983.

[137] David L. Weaver and Tom Gamond, editors. *The SPARC Architecture Manual*. Prentice-Hall, 1994.

[138] Reinhold P. Weicker. Dhrystone benchmark: Rationale for version 2 and measurement rules. *SIGPLAN Notices*, 23(8):49–62, August 1988.

[139] Daniel Weinreb and David Moon. *LISP Machine Manual*. Massachusetts Institute of Technology, 1979.

[140] Steve Williams. *68030 Assembly Language Reference*. Addison-Wesley, 1989.

[141] Phil Winterbottom and Rob Pike. The design of the Inferno virtual machine, 1997. Lucent Technologies.

[142] N. Wirth and M. Reiser. *Programming in Oberon—Steps Beyond Pascal and Modula*. Addison-Wesley, 1992.

[143] John Zukowski. *Java AWT Reference*. Java Series. O'Reilly & Associates, 1997.

Colophon

What is written without effort is in general read without pleasure

Johnson [48]

The thesis was prepared using NEdit on a Daewoo Chorus laptop running GNU/ Linux, Zap on an Acorn RISC PC running RISC OS, and Symbian's Text Editor on a Psion Revo running EPOC. It was typeset with LaTeX, using BibTeX to prepare the bibliography. The KOMA-Script report document style was used. The main text was set in Palatino, with Helvetica used for headings and Computer Modern Typewriter as the typewriter face. Acorn's Draw was used to prepare the figures (with the exception of figure 2.1, which was drawn with pstricks in LaTeX), and the graphs were generated from the original data by PipeDream.

Several LaTeX packages were used to improve the design, notably booktabs and dcolumn to improve the tables, and lips to give better text ellipses. mathpple was used to provide the typefaces, in a version kindly modified by its author to omit the kerning pair for ones which prevented columns of digits from lining up. Custom packages were written to typeset the code examples, the semantic and syntactic definitions, and the data structure diagrams.

Proofs were printed on a Brother HL-760 and a Hewlett-Packard LaserJet 5Si laser printer; the final copy was printed by the Hewlett-Packard on 80gsm Officeteam A4 laser copier paper, and bound by J. S. Wilson of Cambridge.