THE UNIVERSITY OF CALGARY

Implementation of the Functional Architecture TIM

BY

Michael Johann Hermann

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA
December, 1991

0-315-75214-9

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "*Implementation of the Functional Architecture TIM*", submitted by Michael Johann Hermann in partial fulfillment of the requirements for the degree of Master of Science.

G Birtwistle

Dr. G. M. Birtwistle, Supervisor
Department of Computer Science

John Kendall

Dr. J. Kendall
Department of Computer Science

[illegible]

Dr. P. C. K. Kwok
Department of Computer Science

P. Prusinkiewicz

Dr. P. Prusinkiewicz
Department of Computer Science

L E Turner

Dr. L. E. Turner, External Examiner
Department of Electrical Engineering

Date December 13, 1991

Abstract

Functional languages have enjoyed increasing popularity over the last few years, due to their advantages from the viewpoint of user, verifier, and implementer. There has likewise been an increase in the demand for special purpose architectures to efficiently execute them.

The abstract functional architecture TIM (Three Instruction Machine) is a developmental culmination in both sides of the dichotomy formed by notational representation for functional languages and procedural evaluation for functional architecture. TIM was proposed by Jon Fairbairn and Stuart Wray at Cambridge University, and is a compact and efficient frame-based graph reduction processor which executes SuperCombinators.

This thesis is an attempt to give TIM a concrete architectural form, with particular emphasis on the general design issues and methods of attack to be addressed in designing a functional architecture for practical application. Improvements in speed, efficiency, and implementability of the abstract machine are made, via changes to the organisation of memory, the structure of physical objects, and the contents of the instruction set.

Finally, I make some arguments as to the accessibility of the architecture, its merits as a research tool and a representative of a new sub-class of functional machine.

Acknowledgements

I would like to express my deepest appreciation to my supervisor Graham Birtwistle, whose unfailing help and encouragement made this degree possible. Many students have enjoyed the benefit of your support and genuine concern, Graham, and I'm grateful for these *and* your eternal patience and enthusiasm.

A special thanks to my love Patricia Garner, who has supported me during the most difficult times and helped me see the light at the end of the tunnel. Your turn next, Hon.

Thanks to my parents, Gail and Kip, who respectively kept me there and got me there in the first place. A truly bright person is Cindy, who has always cared and will always deny it. The most capable guy I know is Arnold, who has taught me far more about life and people than I can gauge (he also cooks a mean brunch!). And, Paula is a true optimist who showed me the need for a balanced outlook.

Jon Fairbairn and Stuart Wray created the base on which this thesis is built; for this and their early help, I thank them. A lot of folks in the lab have made it interesting and enjoyable: Todd Simpson, Glenn Stone, Inder Dhingra, and Brian Graham. In particular, thanks to Cameron Patterson (an insightful, determined and thoroughly nice guy), for the opportunity to work on the ultimate sieve machine. Sieving über alles! Konrad Slind has been an acerbic wit, and the source of insightful commentary on my ideas and good common sense (sometimes we even talked about work). Simon Williams is an expert without par in VLSI, hardware design, TeX and more importantly motorcycles, who initiated me to the finer points of zooming. Thanks to Dan Marken for the early start in hardware, and Mrs. Olive Corriveau for a warm hearth and home.

Last but not least are Masoud Sahebkar, Winslowe Lacesso (hey, Windows!), Ian Olthof, David Pauli and Anja Haman who have been (variously) compatriots, co-conspirators, guides, rogues, confidantes, and always friends.

Financial support for this work was provided by the *Alberta Microelectronic Center*, the *National Sciences and Engineering Research Council* of Canada and the *University of Calgary.*

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter I introduce *functional programming languages*, outlining the main advantages and drawbacks these have over other programming language styles. From this I explain the need for machines specialised to execute functional languages, and describe the salient features of such *functional architectures.* The focus of this thesis is the *Three Instruction Machine* (TIM), of which I give a brief overview, and state the goals of my thesis work on TIM. I conclude with a list of the contributions of this work, and an outline of the thesis text.

1.1 Functional Languages

Functional programming languages derive their name from the basic operation intrinsic to their structure, that of applying a function to a single argument to produce a result:

$$f\,a \Longrightarrow r$$

(r may itself be another function). Functional programs consist entirely of such function definitions, which execute exclusively through the application of one to another.

The fundamental difference between functional languages and their ancestors in computer science arises from their mathematical nature: they contain no variables. While *imperative* languages such as Pascal and C use variables to maintain state through assignment, *applicative* languages maintain state only as *local arguments* to the currently executing function. Furthermore, the content of this state remains invariant, and is valid, only during the lifetime of the function call.

A more subtle distinction of functional languages is their strong basis in the mathematical theory of computation. As opposed to the *post facto* derived mathematical basis of imperative languages, applicative languages have evolved from an abstract notation or *semantics* called λ-calculus. This has bestowed them with some surprising and very beneficial characteristics:

Convenience The applicative programming environment is very simple to use. The absence of variables removes the effort normally required to define the type and scoping of storage. Functions are "first-class" entities, which may be treated as arguments, produced as results, and applied to other functions. This is a powerful tool that makes programming more intuitive, and makes programs more succinct and expressive. Conventional programming structures such as lists and abstract data types still exist. The result is a class of languages with clearer semantics, fewer details to look at, more compact and expressive operations, and shorter development times.

Typing and polymorphism These form a hierarchy of compile-time techniques that provide an additional convenience feature to the programmer. *Type* refers to the class of information represented, ranging from base types (*ie.* integer) to conglomerate types specifying functions or abstract data structures. The techniques range from simple *type checking* based on user-specified type information, to *type inference* in which the compiler automatically resolves all type information itself. *Polymorphic* types incorporate "wild cards" which match other types, allowing the re-use of single function definitions on multiple types. By rigorously enforcing rules of function application and usage, the compiler quickly flags "type clash" errors, while simultaneously guaranteeing that no (often costly) run-time type errors can occur.

Nondeterminism and optimised execution This encompasses a number of areas such as *lazy evaluation*, *strictness*, and *sharing analysis.* "Laziness" or nondeterminism allows computations to be postponed until absolutely necessary, preventing unnecessary work and providing bonuses such as infinite structures and partial evaluations. To avoid redundant work, each computation result may be shared amongst all references to that work. Strictness analysis determines which work *must* be done, so that the extra overhead of laziness and sharing can be avoided where unnecessary, and work may be more optimally scheduled.

Correctness Software verification is assuming growing importance, but faces difficult complexity problems. Imperative languages suffer from "side-effects" (unforeseen state changes amongst sections of code) and semantic weaknesses that complicate program proofs. In contrast, the lack of variables and applicative semantics of functional languages make them attractive to the verification community.

Simple parallelism The underlying λ-calculus notation provides some characteristics, illustrated by the lack of "side-effects", which allow for arbitrary execution mechanisms (including n-ary parallelism), and arbitrary partitioning of any functional program. This makes functional languages very attractive for multi-processing environments.

Functional languages have left their infancy as a *de facto* standard set of features and capabilities have emerged. Languages such as Miranda [Tur85], Hope [BMS80] and SML [HMM86, Har86] are far better than their predecessors. Functional languages are still not as competitive as they could be with their imperative counterparts, partly from a lack of exposure in the user community. Improvements in the clarity and versatility of semantic structures, provision of larger standard function libraries, and educating users on the functional programming style will encourage expanded use of functional languages. However, the largest problem is that of providing comparable compiler efficiency and execution performance.

The advantages of functional languages don't come for free. On a first approximation, the use of local state, and the applicative style itself make functional programming more expensive in terms of memory usage and the degree of "copying" (of local state) that is required. In the search for performance, software has traditionally migrated to hardware; with the settling of functional languages, and their unique execution requirements, the design of special-purpose *functional architectures* is a logical undertaking.

1.2 Functional Architectures

The major business of any architecture is the movement and storage of data. Hardware is simply a crystallisation of algorithm, and a physical machine must reflect the execution behaviour and storage use patterns of the source language which it evaluates. The patterning of physical structure after logical structure is thus central to the success of any architecture.

Backus [Bac78] makes this observation with regard to "conventional" architecture, grouping Von Neumann and Harvard style machines together as variants on the familiar Central Processing Unit (CPU)–Bus–Memory theme. On this alone the argument would fail, as this structure is common to nearly all computer architectures. However, Backus continues by arguing that the imperative programming style on which these machines is based is deeply ingrained in their design. The semantics of imperative programming naturally engender the use of such simple things as single-word wide instruction streams, user-accessible general purpose registers, and so forth. Further, this natural reflection of structure forever colours the character and capabilities of these architectures towards the service of the imperative style.

It follows that functional languages should have their own native style of architecture, which would execute them with greater utility. In many cases, a target non-native architecture can provide unnecessary *obstacles* to the efficient execution of a functional language. These obstacles frequently manifest themselves in instructions that require specialised sequences of non-native operations or unique logical structures; a good illustration is tag manipulation. Thus even a simple implementation of a functional architecture will be better than a non-native machine, if the basic needs of functional languages are being served, and the transistor budget is not being wasted on unnecessary functions.

What are these basic needs? The logical characteristics of functional languages indicate the physical form that a generic native functional architecture should take.

There are a number of observations from which the general nature of a functional architecture is derived:

Applicative style The functional style encourages short-bodied and short-lived functions. The resulting sustained high rate of function calls predicts a need for rapid context changes, with attendant copying operations and movement of logical entities. This predicates large amounts of storage with fast access times, and extensive CPU support for executing function calls.

Allocatable store The transient nature of most storage usage, and the additional storage requirements for laziness and sharing, means there must be at least one (and perhaps several) groups of allocatable logical storage elements. There will certainly be firmware/hardware support for allocation and garbage collection, including special control and storage use information. In some instances, the use of multiple physical memories may be beneficial.

Simple instruction sets Due to the proximity of functional languages to their common ancestor λ-calculus there is a small "semantic gap" [Mye78] between language and machine. Programs typically compile into a handful of basic operations, where the operations themselves usually require only a 0- or 1-operand specification (typically a constant or a reference to memory). Thus, the instruction set of the CPU will be small and limited in scope, there will be no user-reachable general purpose registers, and most details of the CPU will be hidden from the user.

Complex internal structure In contrast with the last point, while instruction decoding may be simplified, the instructions themselves often represent highly complex operations. When using techniques such as lazy evaluation and sharing, many internal tests and the movement and manipulation of much information is necessary, especially for those instructions controlling context changes. Such instructions will naturally require many clock cycles, the use of internal book-keeping registers, and a correspondingly more complex control unit.

Limited locality In comparison to their imperative cousins, we can expect applicative languages to exhibit much different memory reference patterns. Spatial and temporal locality will be affected by the rapid function calls, arguing against the utility of hardware accelerators (such as caching) as they are commonly used in conventional architecture. Improving memory performance will present new challenges to the hardware designer.

Hooks for parallelism With the isolation of functions from each other, parallel execution is made more tenable by the lack of code interdependence. While the scheduling of instructions on a fine-grained multiple execution unit processor would still retain about the same difficulty, breaking the program into function-sized chunks or larger for scheduling on a coarse-grained multiprocessor is much easier.

To summarise, the semantics of functional languages places special emphasis on the quantity and structure of storage, the uses to which it is put, and the efficient movement of data within and between the CPU and memory. Judging by the large numbers of abstract functional architectures that have been designed [Hen80, CGMN80, Tur84, Car84, Joh87, CCM87, Tra85], these lessons have not been lost on the functional programming community. A few of these abstract machines have also been built [Sto85, Sch86, Ram86, GWB+89].

A full blown implementation of a functional architecture would be able to make use of the same performance enhancements that have been used in other architectures for years. A speed-optimised machine design would use microcoding for the complex instructions and subsidiary concurrent operations, pipelining to accelerate the data and control paths, virtual memory and hardware allocation/collection of memory for the more storage-consumptive execution, control unit redundancies to hold multiple contexts and enable faster function calls and returns, and instruction and data memory caching. However, while these are all valuable tools to use, the critical effort comes in designing the machine concept properly for the functional programming paradigm.

1.3 The Three Instruction Machine

The Three Instruction Machine (TIM) was proposed by Jon Fairbairn (Cambridge University) and Stuart Wray (Olivetti, UK). TIM is an abstract functional machine tailored to execute a specialised form of λ-calculus called Combinatory Logic (specifically SuperCombinators, see Chapter 2). TIM uses *graph reduction* as an evaluation mechanism, a technique in which a program is distributed in a logical tree structure, and execution proceeds through a "pruning" process that gradually evaluates portions of the program and replaces them with their results. This is a common mechanism amongst functional architectures, but TIM is distinctive in both the way that it represents programs as graphs, and the method it uses to evaluate them. The major drawbacks with previous graph reduction schemes are the amount of time spent traversing the tree structure, and the amount of memory consumed in its representation. TIM uses an innovative approach to abstract most of the program code from the graph, leaving essentially only the call-structure of the program. This greatly abbreviates the tree structure. In addition, TIM supports nondeterministic evaluation and sharing, addressing the unique control problems of each within the new graph reduction scheme.

The three TIM instructions are called PUSH, ENTER, and TAKE. These roughly represent the three stages in a function call:

1. PUSH some number of arguments to the function into temporary storage.
2. ENTER (call) the function, the code portion of the new context.
3. TAKE the supplied function arguments, as the environment portion of the context.

So for the generic functional program, the instructions operate as in Figure 1.1. There are also a host of built-in ALU operations.

The abstract machine is outwardly simple, but it holds a few surprises for the unwary. "Three Instruction Machine" is a somewhat deceptive title, as there are actually a few variants or "flavours" of both PUSH and ENTER, which specify the type of logical entity being referenced (*ie.* a constant, a combinator "function" reference,

```
                PUSH a
                ENTER f
f a ⟹ r         TAKE 1
                  ⋮
                (return) r
```

Figure 1.1: The simplest TIM program

or a proper argument). Since TIM implements sharing, there are supplementary arguments to each instruction that provide control information, and specify which computations are shared.

The basic instructions have very different complexity; while some instructions are very simple and execute in one or two machine cycles, others are very complex and can take tens of cycles. Thus some instructions are natural bottlenecks, and will draw heavily on hardware resources to prevent detrimental effects to machine performance.

However, the basic abstract machine is reasonably straightforward. The most complex problems arise from implementing the many architectural optimisations that are discussed in Chapters 4 and 5. TIM holds a number of challenges for the designer charged with implementing the abstract definition as a practical working architecture.

1.4 Contributions of the thesis

The goals of this thesis are:

1. *to evaluate potential improvements to the abstract architectural definition of TIM*, and

2. *provide a concrete design for selected best optimisations.*

The central contributions of the thesis are:

1. evaluation of a number of optimisations to TIM proposed in the literature, and where applicable, comparison of reported results to replicate or refute.

2. evaluation of a number of optimisations proposed by the author.
3. specification of some improvements to the abstract TIM design.
4. design of the instruction format, logical objects and a partitioned memory system to support the storage and high throughput needs of the TIM processor.
5. development of a DEMOS-based [Bir79] logic simulation package, with performance measurement instrumentation and design debugging support.

1.5 Structure of this thesis

Chapter 2 is an introduction to functional languages, their theoretical characteristics, and some practical examples of their use. I quickly examine the underlying notation of λ-calculus, touching on the work of Church and Rosser, the specialisation of λ-calculus to combinators, and place special emphasis on SuperCombinators, the source notation of TIM. A brief discussion of "lazy" evaluation of expressions, sharing analysis and strictness analysis is included.

Chapter 3 outlines the abstract TIM machine as proposed by the original developers. I initially describe two notable functional architectures which have been built, which represent opposite ends of the spectrum of functional architectures, and illustrate two important components in the TIM architecture. I explain the philosophy of the TIM machine, describe the structures and basic instruction set, additional instructions for ground types, and those used to implement lazy evaluation.

Chapter 4 begins with a brief overview of the set of possible optimisations, followed by the design issues in TIM, goals I have specified for the design and a design philosophy developed to attain them. The bulk of this chapter discusses structural optimisations surrounding the TAKE instruction, covering aspects of context changes, creation and maintenance of sharing information, and the updating of shared results. Approximately one half of this material is new and original work, in the form of extension and analysis.

Chapter 5 focuses on lower-level optimisations, and implementation of the TIM machine including the instruction set and use of storage. I propose a new model of the frame heap and a new instruction, used to speed context changes and the use of shared evaluations in TIM. Approximately 90% of this material is new and original.

Chapter 6 summarises the contributions of this thesis, outlines some possible future work, and gives some final comments.

Chapter 2

Functional Programming

Functional programming has developed in the areas of software notations, evaluation strategies, and hardware structures, with growth in one area encouraging growth in the others. This is no accident; in functional programming the notation encodes its own evaluation algorithms, and hardware is simply a hardwired form of algorithm. Thus functional language and special architectural support are intimately related.

In the past, mainly notations and methods of evaluation were explored, and the interest in hardware was limited to the pervasive *abstract machine*. Now that some stable notations and evaluation strategies have settled out of the previous work, real hardware is being more aggressively explored for the next efficiency gains.

The purpose of this chapter is to illustrate key concepts of functional languages, by examining the notational genealogy of the Three Instruction Machine. This will supply the background information necessary for the reader to understand the first two of the three facets of TIM: notation (supercombinators), methods (lazy evaluation, sharing) and structure (stack-based graph reduction), preparatory to concentrating on the third in Chapter 3.1.

I will outline the development of notations up to supercombinators, illustrating the gradual improvements in efficiency and discussing some of the tools used to achieve these gains. A great deal of work has gone into understanding functional languages, how best to apply them to problems, creating type inference mechanisms, and so forth. Much of this work has been aided by the fact that all functional languages have a common mathematical basis, from which each may gain the same power, and be amenable to the same analysis techniques. This basis is the λ-calculus, and it is the intrinsic properties of λ-calculi that are in large part responsible for the success of functional programming languages.

The knowledgeable reader may safely skip this chapter. Those who wish to know more are referred to [Hug89], which outlines the arguments for functional programming, and [Kle81], which gives an historical account of its development. [Bar81b] is a comprehensive introduction to the λ-calculus, and [CF58, CHS72] are the standard references for combinatory logic. [HS86] treats theoretical aspects of the above, while [Pau87] and [Sto77] focus on program verification and typing of functional languages, respectively. [Gor88] is a teaching text for semantic analysis which covers λ-calculus, combinatory logic and supercombinators. [BW88], and more so [Pey87] and [FH88] are comprehensive, broad overviews of functional programming, including notations, evaluation strategies and architectures.

2.1 The Lambda-calculus

Functional programming got its start from a simple notation called the Lambda Calculus or λ-calculus. The λ-calculus is a formal abstraction originated by Church [Chu41] to provide a theoretical basis for mathematics, a task for which it proved inadequate [Ros84]. It lay dormant for many years, before computer scientists found that it was useful for reasoning about computation and the semantics of algorithms. λ-calculus could be used to separate program syntax (how an algorithm is written) from program semantics (how the algorithm operates, and what it *does*), isolating the meaningful from the extraneous.

An early exploration by Landin used the λ-calculus to provide an operational semantics for ALGOL60 [Lan65a, Lan65b], with which ALGOL60 programs could be understood and verified correct through "abstract evaluation". This developed from previous work where Landin introduced the use of λ-calculus as a semantic analysis notation, and created an abstract machine called the SECD which could execute the new notation directly [Lan64]. The notion of a language which held both a "program" and the method of its execution implicit in the program definition was to become central to the field of functional programming, and λ-calculus was to become the common denominator in much of the research to follow.

2.1.1 The Pure λ-calculus

The "pure" λ-calculus has only three constructs, as they appear in Figure 2.1. These are argument names (any tag symbol V introduced in an abstract function), function applications (read as function E_1 applied to argument E_2, where E is any λ-expression), and abstract function definitions, in which V is the tag name for zero or more occurrences of an argument in the function body E. These three semantic structures are the simplest expression of the essentials of functional computation, and form the common denominator for all functional languages.

$$\begin{array}{rcll} E & ::= & V & \text{(argument name)} \\ & | & (E_1 E_2) & \text{(function-argument application)} \\ & | & \lambda V.E & \text{(abstract function)} \end{array}$$

Figure 2.1: Pure λ-calculus BNF

To evaluate an λ-calculus expression, it is converted to another semantically equivalent form; the "conversion" rules (see Figure 2.2) governing λ-calculus are likewise few and simple. The first and third, α-conversion and η-conversion, are used to rename a argument, and to add or delete arguments as needed, respectively. The most important rule is β-conversion, which applies functions to arguments. The notation "$E_1[E_2/x]$" is short for "the expression E_1 with argument expression E_2 substituted for all free occurrences of x". In other words, reading from left to right this retrieves the outside argument E_2, "binds" it to the internal argument name x, and replaces free occurrences of the symbol x within E_1 by the expression E_2.

$$\begin{array}{llll} \lambda x.E & \overset{\alpha}{\Longleftrightarrow} & \lambda y.E[y/x] & \text{(renaming)} \\ & & & \text{where } x \text{ is not free in } E \text{)} \\ (\lambda x.E_1)E_2 & \overset{\beta}{\Longleftrightarrow} & E_1[E_2/x] & \text{(function application)} \\ \lambda x.Ex & \overset{\eta}{\Longleftrightarrow} & E & \text{(argument abstraction} \\ & & & \text{where } x \text{ is not free in } E \text{)} \end{array}$$

Figure 2.2: λ-calculus conversions

The execution of λ-calculus programs is called "reduction", and uses the left-to-right forms of β-conversion and η-conversion with the normal α-conversion. Execution proceeds as a search for reducible expressions, or "redexes", which are then rewritten using the rules. As an example, consider a simple predicate:

Operation	*Comment*
if false then A else B	
$\Longrightarrow$ $(\lambda txy.txy)$ false A B	(if $= \lambda txy.txy$)
$\overset{\beta}{\Longrightarrow}$ (false A B)	(3 β-reductions)
$\Longrightarrow$ $(\lambda xy.y)$ A B	(false $= \lambda xy.y$)
$\overset{\beta}{\Longrightarrow}$ $(\lambda y.y)$ B	(β-reduction)
$\overset{\beta}{\Longrightarrow}$ B	(β-reduction)

Figure 2.3: λ-calculus if-then-else

The process of "reduction" implies that the size of the expression is reduced, but this is not always the case. Program execution is complete when we run out of redexes or further applications of the rules to existing redexes no longer change the expression; this is known as "normal form".

Despite its simplicity, every computable function can be expressed in λ-calculus ("Church's Thesis", [Chu36]). That is, it possesses the same power as any other programming language in use. As with "if-then-else", integers, lists, pairs, datatype constructs and others can all be expressed in terms of the λ-calculus, albeit inefficiently.

2.1.2 Practical Aspects

In practice, a fourth construct (Figure 2.4) is added to the pure λ-calculus. Constants are used for ground types and their atomic operations, such as the natural numbers and the operations $+, -, x, /$.[1]

Implementing recursion in the λ-calculus is a sticky problem. Unlike higher-level languages, λ-calculus is *nameless* apart from the tags used for arguments. A recursive

[1]Pure λ-calculus is restricted to an inefficient unary integer representation; this is a common practical optimisation.

$$\begin{array}{lll} E & ::= & \ldots \\ & | & C \qquad\qquad \text{(constant)} \end{array}$$

Figure 2.4: Impure λ-calculus partial BNF

function refers to itself, and so cannot be directly expressed in the λ-calculus. We must use a "trick" to provide recursion, by converting recursive functions into a non-self-referential form, in which the function obtains its own definition as an argument. This argument is then used for the "recursive" function call. As an example, consider a recursive multiply function (Figure 2.5a):

$$\begin{array}{lll}
 & \mathsf{mpy} = \lambda\, p\, q\ .\ \textit{if}\ (\,p = 0\,)\ \textit{then}\ 0\ \textit{else}\ (\,q + (\,\mathsf{mpy}\ (\,p - 1\,)\ q\,)\,) & \\
\text{let} & \mathsf{M} = \lambda\, f\, p\, q.\ \textit{if}\ (\,p = 0\,)\ \textit{then}\ 0\ \textit{else}\ (\,q + (\,f\ (\,p - 1\,)\ q\,)\,) & (\beta\text{-conversion}) \\
\text{then} & \mathsf{mpy} = \mathsf{M}\ \mathsf{mpy} & \text{(by definition)} \\
\text{and} & \mathsf{mpy} = \mathsf{Y}\ \mathsf{M} & \text{(using a fixed-point function)}
\end{array}$$

Figure 2.5a: Developing a non-recursive definition

To provide the copies of the function definition as needed, we use the artifice of a "fixed-point operator". Denoted "Y", each time this operator is applied to an argument, it returns a copy of the argument as well as the original application. Thus the initial application pair remains unchanged:

$$\underline{\mathsf{Y}f} \overset{\beta}{\Longrightarrow} f(\underline{\mathsf{Y}f}) \overset{\beta}{\Longrightarrow} f(f(\underline{\mathsf{Y}f})) \overset{\beta}{\Longrightarrow} \cdots$$

The (Yf) pair is a *fixed-point* of M, and is in effect a catalyst for the recursive computation, moderating it but remaining unchanged. To see how a recursive function is translated, we start with the definition of a recursive function called mpy (Figure 2.5a). First β-conversion is used to abstract away the function name, and replace it with an argument; the external parameter "mpy" represents the mpy function definition needed for a recursive call to succeed. The last step is to discard the function name, and add the Y operator to implicitly replace it with the mpy definition. Simplified, the process is as in Figure 2.5b.

Under execution, the fixed-point pair (YM) is applied to the regular function arguments, and immediately rewrites itself as the function M applied to the fixed-

$$\begin{array}{rll}
\mathsf{mpy} & = \mathsf{mpy\text{-}body} & \mathsf{mpy\text{-}body} = \lambda\ m\ n\ .\ \mathit{if} \cdots \mathsf{mpy} \cdots \\
& \overset{\beta}{\Longleftrightarrow} \mathsf{M\ mpy} & \mathsf{M} = \lambda\ f\ .\ \mathsf{mpy\text{-}body}\ [\, f \,/\, \mathsf{mpy}\,] \\
& = \mathsf{Y\ M} & \\
& = \mathsf{M\ (Y\ M)} & \text{proof: invoke } \mathsf{Y} \\
& = \mathsf{M\ mpy} & \text{proof: } \mathsf{mpy} = \mathsf{Y\ M}
\end{array}$$

Figure 2.5b: Fixed-point equivalence

point pair and the other arguments. Execution continues with the body of M. See Figure 2.5c.

$$\begin{array}{lll}
\multicolumn{2}{l}{\mathsf{mpy}\ \mathit{3\ 2}} & \\
= & \mathsf{Y\ M}\ \mathit{3\ 2} & \\
= & \mathsf{M}(\mathsf{Y\ M})\ \mathit{3\ 2} & \text{(invoke Y)} \\
= & [\,\lambda\, f\, p\, q\ .\ \mathit{if}\ (\ p = 0\ )\ \mathit{then}\ 0 & \\
 & \quad \mathit{else}\ (\ q + (\ f\ (\ p - 1\ )\ q\ )\ )\,]\ \ (\mathsf{Y\ M})\ \mathit{3\ 2} & \\
= & \mathit{if}\ (\ = 3\ 0\ )\ \mathit{then}\ 0\ \mathit{else}\ (\ 2 + (\ (\mathsf{Y\ M})\ (\ 3 - 1\ )\ 2\ )\ ) & \text{(apply arguments)} \\
= & 2 + (\ (\mathsf{Y\ M})\ 2\ 2\ ) & \text{(resolve predicate)} \\
= & 2 + (\ \mathsf{M}(\mathsf{Y\ M})\ 2\ 2\ ) & \text{(invoke Y)} \\
= & 2 + (\ [\,\lambda\, f\, p\, q\ .\ \mathit{if} \cdots f\ (\ p - 1\ )\ q\,]\ (\mathsf{Y\ M})\ 2\ 2) & \text{(the full definition)} \\
\vdots & \qquad\qquad\qquad \vdots & \\
= & 2 + (\ 2 + (2 + (\ (\mathsf{Y\ M})\ 0\ 2\ )\ )\ ) & \\
= & 6 &
\end{array}$$

Figure 2.5c: Evaluation under Y fixed-point operator

Mutually recursive functions are handled by encapsulating the definitions as a single argument, to be extracted for function calls as needed.

One λ-expression definition[2] for Y is $(\lambda h.(\lambda x.h(xx))(\lambda x.h(xx)))$ and others have been suggested by [CF58, CHS72, Bar81b]. [Hud89] states that fixed-point operators are inadequate for typed λ-calculi and non-standard evaluation mechanisms, and suggests other methods of implementing recursion.

2.1.3 Theoretical Implications

The mathematical basis of λ-calculus is responsible for many of the assets of functional languages, mainly through the ease of developing and extending formalisms. The greatest asset of the λ-calculus is derived from a set of basic results called the

[2]from [Pey87], pg. 26

"Church–Rosser" properties [CR36][3], which address the evaluation of λ-expressions and their convertibility one to another:

Theorem 1 (Church-Rosser) *Given two λ-expressions X and Y, if $X \Longleftrightarrow Y$, then there exists a λ-expression Z such that $X \Longrightarrow Z$ and $Y \Longrightarrow Z$.*

Corollary 1 (Church-Rosser) *No expression can be converted to two distinct normal forms.*

Theorem 2 (Church-Rosser) *If A* **red** *B, and B is in normal form, then there exists a normal order reduction from A to B.*

Theorem 1 states that any two interconvertible ("$\Longleftrightarrow$" denotes any α-, β- or η-conversions) expressions will have a common result expression through reduction ("$\Longrightarrow$" using β,η-reduction). More generally, a single expression may be evaluated in many different ways, but all the interim results are interconvertible, and so will (eventually) reduce down to a single result. An inductive argument on theorem 1 states this clearly in corollary 1, where *normal form* means "fully evaluated".

Theorem 2, or the "Normalisation Theorem", states if an expression can be reduced down to its normal form, then there is a well-defined method of reduction that always attains the normal form. *Normal-order* reduction always applies β-reduction to the leftmost-outermost redex first, until no more such redexes exist.

Taken together, these three statements provide two guarantees:

1. evaluate the redexes of an expression in any order. If the evaluation produces a result (doesn't loop infinitely), the result is correct.

2. if you evaluate a (terminating) expression in normal order, you will terminate with a result, and that result will be correct.

The same holds for any operations we split in pieces and do in parallel (the most obvious place being amongst arguments at a function application).

The practical upshot of the Church-Rosser theorems as they apply to λ-calculus, and by extension to all functional programming languages, is that the evaluation

[3]more accessible proofs are available in [CF58, HLS72, HS86]

mechanism we use is immaterial to our results. We can break a program up in any way we want, execute the pieces in any order, and if we get a result, it is guaranteed to be correct.

In other words, functional programs can be broken up and mapped onto a parallel processor in any arbitrary manner, so we can attain the benefits of concurrent evaluation without the typical costs of partitioning and communication. The λ-calculus asset of cheap parallelism is precisely what motivates much of the continued interest in functional programming.

2.1.4 Strengths and Limitations

There are a few practical problems involved when considering the λ-calculus as a notation that would be executed in a real or abstract machine. The size of λ-calculus programs increases compared to that of the original source program. Even when ground types and associated operations are included, the nameless nature of λ-calculus requires that function bodies be replicated wherever they are called. Even so, λ-calculus code expansion is not that much worse, symbol for symbol, than with imperative-style machine code on Von Neumann machines.

λ-calculus programs consist of many very small-bodied functions, and this translates to an abundance of function calls, all of which are short. Aside from ground type operations, evaluating λ-calculus programs consists almost entirely of function calls. Qualitatively, this means that most effort is spent rearranging complex expressions simply to filter arguments "down" to where they are needed. In addition, the use of fixed-points for recursion is expensive, as each recursive call requires that the function be copied in its entirety.

The largest cost is the use of β-reduction to evaluate each function call, as it is inherently expensive. Substituting arguments requires a time-consuming search for variable names throughout the body of the λ-expression, which can be several thousand symbols at the start of a program. β-reduction is also susceptible to the "name-capture" problem [Pey87], which arises when one or more symbols in a substituted expression become erroneously bound to remaining parameters. As illustrated on the left in Figure 2.6 below.

With capture		Without capture	
$(\lambda x.(\lambda y. + x\ y))\ y\ 3$	$\surd$	$(\lambda x.(\lambda y. + x\ y))\ y\ 3$	$\surd$
$\xRightarrow{\beta}\ (\lambda y. + y\ y)\ 3$	!	$\xRightarrow{\alpha}\ (\lambda x.(\lambda w. + x\ w))\ w\ 3$	$\surd$
$\xRightarrow{\beta}\ +3\ 3$	X	$\xRightarrow{\beta}\ (\lambda w. + y\ w)\ 3$	$\surd$
$\Longrightarrow\ 6$	X	$\xRightarrow{\beta}\ +y\ 3$	$\surd$

Figure 2.6: Name-Capture

The argument y is substituted for the locally free argument name x in the first β-reduction and has become bound to the local parameter y. It should remain a "free variable" at this level (although it may be bound by a surrounding expression). To prevent this name clash, α-conversion is used to rename y prior to substitution, as shown on the right in Figure 2.6. Thus, β-reduction is made more expensive by having to detect name clashes and apply α-conversion as needed. Performing this task over an entire program is very expensive in time and memory usage.

Of course, many of these problems may be reduced by an intelligent implementation of the evaluation methods. A machine architecture proposed by DeBruijn [DeB72] uses annotated variable names and an environment-lookup method to track the positions of argument symbols in the body of an expression, to avoid searching and quickly detect name clashes. Aiello and Prini [AP81] maintained expensive run-time variable scoping lists for each expression, and extensively applied α-conversion to avoid name-capture. Both of these machines resorted to reincarnating information that the λ-calculus had originally abstracted out, and using more intelligent evaluation mechanisms than simple term-rewriting. However, the real problem is with the notation; the use of β-reduction, and the raw numbers of function calls must be avoided.

While the λ-calculus is not a very good implementation language, it is a good notation for representing and reasoning about programs. For this reason, many functional programming implementations currently rely on λ-calculus as an intermediate functional language (IFL). Higher-level functional languages are translated down to λ-calculus, where type inference and similar tasks are performed, and from there translated to a specific implementation language for evaluation. In the rest of this chapter, we look at some more suitable implementation languages.

2.2 Combinatory Logic

Combinatory Logic (CL) predates the λ-calculus in the search for an abstract theory of mathematics, and was developed independently by Schonfinkel and Curry [Cur29, Sch24, Ros84]. Summarily discarded, CL lay dormant for many years until Petznick [Pet70] revived it by suggesting it as the basis for a computing engine. The modern use of combinators in computer science stems mostly from Turner [Tur79a, Tur79b] who expanded upon the original set of combinators and developed an abstract graph-reduction machine to execute them.

The Combinatory Logic notation contains only constants, combinator names and application (Figure 2.7).

$$\begin{array}{rcll} E & ::= & C & \text{(constant, combinator name)} \\ & | & (E_1 E_2) & \text{(application)} \end{array}$$

Figure 2.7: Combinatory Logic BNF

The definition of a combinator is a λ-expression which contains no occurrences of a free variable. For example, "$\lambda y.yx$" would not be a combinator since x occurs free in the abstraction, and makes the expression prone to name-capture. Each combinatory logic program is formed of combinators represented only by name, the actual λ-expression definitions being "hidden" from the user. In this sense, combinatory logic is variable-free λ-calculus, where combinators are fixed reduction formulae, used in combination to implement larger, more complex λ-expressions. Only a finite set of these combinators are used for any given application, so that each may be hard-wired into the evaluation mechanism. This and the lack of free variables means that the expense of β-reduction and substitution is avoided, making CL a potentially attractive replacement for λ-calculus.

The basic set of combinators are called S, K and I (Figure 2.8). These encode the

$\mathsf{S}fgx$	$= fx(gx)$	(distribute and apply)	S	$\equiv \lambda fgx.fx(gx)$
$\mathsf{K}xy$	$= x$	(elimination)	K	$\equiv \lambda xy.x$
$\mathsf{I}x$	$= x$	(identity)	I	$\equiv \lambda x.x$

Figure 2.8: Basic Combinators and λ-calculus equivalents

simple operations necessary to manipulate symbols in the manner of computation, and are sufficient to represent any computable function. Actually, I is a convenience, and only S, K are necessary [Sch24, Cur29], as demonstrated:

$$\mathsf{SKK}x \Longrightarrow \mathsf{K}x(\mathsf{K}x) \Longrightarrow x = \mathsf{I}x \qquad \therefore\ \mathsf{SKK} = \mathsf{I}$$

Lacking substitution, combinators must control the movement of arguments directly. The equivalent of a λ-expression in CL is a string of combinator applications,

$$\lambda a_1 a_2 \cdots .body = (\cdots((C_1 C_2) C_3) \cdots C_n)$$

which on execution will incrementally accept, copy and rearrange arguments.

For example, when evaluating an application $E_1 E_2$ in environment σ, both E_1 and E_2 must be evaluated in σ before application:

$$E\sigma \Longrightarrow (E_1 E_2)\sigma \Longrightarrow (E_1\sigma)(E_2\sigma)$$

Once an application $E = (E_1 E_2)$ has been stripped of all argument names, it must be reconstructed to "automagically" restore arguments to their correct places. This is the task of the S combinator:

$$\mathsf{S}E\sigma = \mathsf{S}E_1 E_2 \sigma \Longrightarrow (E_1\sigma)(E_2\sigma)$$

If the argument σ does not appear in the sub-expression (trivially, when the expression is a simple constant), K is used to eliminate the argument(s):

$$E\sigma \Longrightarrow C\sigma$$
$$KE\sigma = KC\sigma \Longrightarrow C$$

Lastly, I is used where the argument is passed on unchanged.

The procedure for translating λ-calculus to combinatory logic is known as *bracket abstraction.* The abstraction rules for the simple SKI logic (from [Gor88]) are shown in Figure 2.9. Rules 1–3 parse the λ-expression "$(\cdots)_c$", deleting λ-abstractions and identifying variables "$\langle x\rangle$" to be removed. Rules 4–7 parse the expression a second time to convert appearances of the variables to combinators which will restore the appearances on execution. The notation "$\langle x\rangle\, E$" denotes the abstraction of a variable name "x" from expression "E".

1)	$(x)_c = x$	atomic
2)	$(E_1E_2)_c = (E_1)_c(E_2)_c$	parse the parts of the application seperately
3)	$(\lambda x.E)_c = \langle x\rangle\,(E)_c$	parse the λ-abstraction
4)	$\langle x\rangle\, x = \mathsf{I}$	x matches itself
5)	$\langle x\rangle\, y = \mathsf{K}y$	x is not found here
6)	$\langle x\rangle\, \mathsf{M} = \mathsf{KM}$	M is a combinator
7)	$\langle x\rangle\,(E_1E_2) = \mathsf{S}(\langle x\rangle\, E_1)(\langle x\rangle\, E_2)$	abstract the application

Figure 2.9: SKI λ-abstraction rules

Thus, λ-abstraction converts an interpreted λ-expression, into a string of CL rules that explicitly manipulate parameters. Applied to a λ-program, the entire expression will eventually become "flattened", consisting of one long string of combinators which expects to receive all the top-level program inputs in the same order as previously. Figure 2.10 illustrates the abstraction process and resulting code for a simple example.

$$\begin{aligned}(\lambda fa.faa)_c &= \langle f\rangle\,(\lambda a.faa)_c \\ &= \langle f\rangle\,\langle a\rangle\,(faa)_c \\ &\overset{*}{=} \langle f\rangle\,(\mathsf{S}((\mathsf{K}f)\mathsf{I})\mathsf{I}) \\ &\overset{*}{=} \mathsf{S}(\mathsf{S}(\mathsf{KS})(\mathsf{S}(\mathsf{S}(\mathsf{KK})\mathsf{I})\mathsf{KI}))\mathsf{KI}\end{aligned}$$

Figure 2.10: Example of SKI abstraction

2.2.1 Strengths and Limitations

The major advantage of Combinatory Logic is that it dispenses with variables in order to avoid both β-reduction and substitution, the most expensive components of λ-calculus evaluation. Complex λ-expressions can now be represented using a small fixed set of simple reduction rules. The rules can thus be "hard-wired" into the evaluation algorithm, whether this is in an abstract machine, or as microcode in a real architecture.

Unfortunately, the basic SKI combinator set is too fine-grained. The restricted capability of these simple operators means much execution time is spent simply rearranging expressions to slowly filter arguments "down" to where they are needed. Also, there are high associated memory costs (see Section 3.1.2), and the same overhead as λ-calculus for the basic evaluation mechanism and implementing fixed-point recursion.

It is possible to do better by modifying the set of combinators, and using the new combinators to optimise inefficiencies in the abstraction process. Many such alternative logics have been proposed [Abd74, Abd76, Tur79a, Tur79b, Ken82]. However, while it is tempting to design new combinators for every special case that arises, the usefulness of each is difficult to predict, and each addition complicates the already costly translation algorithm; Joy [Joy84] has shown that optimising combinator code is NP-complete, and that in any case, fixed-set combinatory logics have unpromising size complexities (see Table 2.1, where n is the λ-expression size, and m is the number of variables being abstracted). These limitations have led to the development of combinators tailored to the program code, or *supercombinators* (see Section 2.4).

worst case	$O(n3^m)$	(unadorned SKI logic)
typical case	$\Theta(nm^2)$	(Turner[Tur79a] logic)
best case	$\Theta(n \log m)$	(theoretical best achievable)

Table 2.1: Size Complexity of Combinatory Logic

2.3 Lazy Evaluation, Sharing, Strictness Analysis

Lazy Evaluation (or *laziness*), *sharing* and *strictness analysis* are a collection of concepts intrinsic to the convenience and efficiency of functional programming. I have neglected these thus far, but they are necessary to understanding the material that follows.

2.3.1 Lazy Evaluation and Sharing

In conventional programming languages, arguments to a function can be evaluated either before application of the function (*call-by-value*) or after (*call-by-name*). Each method can suffer from inefficiency, call-by-value when an argument is evaluated and subsequently left unused, and call-by-name when each use of the argument requires a re-evaluation.

Lazy evaluation does exactly what its name suggests, postponing the evaluation of a named argument until its value is required. Furthermore, once an argument is evaluated, the result is retained for future use, which is known as *sharing*. Typically, laziness/sharing is implemented by representing argument bindings with a pointer to the argument, rather than the argument itself. Figure 2.11 shows this pictorially.

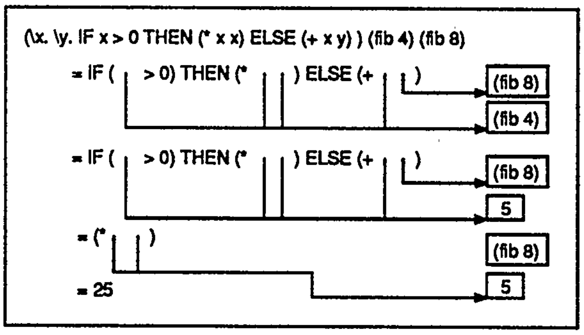

(\x. \y. IF x > 0 THEN (* x x) ELSE (+ x y)) (fib 4) (fib 8)
= IF (> 0) THEN (*) ELSE (+)
(fib 8)
(fib 4)
= IF (> 0) THEN (*) ELSE (+)
(fib 8)
5
= (*)
(fib 8)
= 25
5

Figure 2.11: Example of Lazy Evaluation

This *"call-by-need"* method avoids needless work both for unused arguments (as with y) and redoing work for multiple occurrences of an argument (as with the three uses of x). There are degrees to the laziness concept: *ie. full laziness* is an important goal of functional language implementations. Full laziness avoids the construction of multiple instances of the same expression. As an example, consider the expression below:

$$(\lambda y.\lambda x. + (*2y)x)E$$
$$(\lambda x. + (*2E)x)$$

This is a *partial application*, so named because only one argument is supplied. The result is the function that adds $*2E$ to x, where laziness will postpone the evaluation of E. If this partial application is shared amongst two or more contexts that will supply the x argument, the first to execute will trigger E's evaluation, but each would be forced to evaluate the local subexpression $*2E$. Full laziness will avoid this needless work, by constructing only one instance of the subexpression and treating it lazily.

The overhead of implementing laziness and sharing is non-trivial. There are the costs of suspending and "storing" argument evaluations, passing pointers, and updating arguments with results. Distinguishing between an unevaluated and evaluated argument requires some variety of book-keeping, which becomes more complicated when (as seen above) the argument is not a simple expression but a partial application. Similarly, sharing of expressions adds to the complexity. The needless use of sharing (and laziness) for expressions or arguments which are only used once can be avoided by tabulating the dependencies of expressions on its neighbours. Unshared code can then be moved into the body of the expression that requires it. An analogous, but much more general technique is used to restrict the application of laziness and sharing, and is described in the next section.

Section 3.1 will discuss some of the methods for implementing laziness and sharing.

2.3.2 Strictness Analysis

The cost of laziness is enough that when it can be avoided, it is worthwhile. Expressions sometimes *require* an argument to produce a result, and it is cheaper to *eagerly* evaluate the argument, while maintaining sharing. Such expressions can be detected using a body of theory called *strictness analysis*. Informally,

$$\boxed{\text{A function } f \text{ is \textit{strict} in an argument } x \text{ if and only if the value of } x \text{ is necessary to produce the value of } f.}$$

For example, operations such as integer addition are always strict in all of their arguments (*ie.*. "$\lambda x.\lambda y. + xy$") since a result for each is needed for an expression result. The expression "$\lambda xns.$ if x then n else s" is strict only in x, so nothing is gained by lazy evaluation of x. Strictness analysis is an interpretive technique that predicts only argument evaluations which are *mandatory*, so that these arguments may be automatically executed as soon as is practical.

2.4 Supercombinators[4]

Supercombinators were introduced by Hughes [Hug84] when he combined a new technique called *lambda-lifting* (λ-lifting) with full laziness to produce *custom* combinators. The motivation is prohibit free variables so as to allow simultaneous substitutions, while avoiding the limitations of small combinators. Rather than mimic a λ-calculus expression with a composition of fixed-set combinators, the λ-calculus is reorganised so each abstraction has the characteristics of combinators, and translated directly. These *super*combinators have more variables and larger bodies, so that more substitutions can be done simultaneously.

Supercombinators form a proper subset of the infinite set of all possible combinators, and have the same semantics (see Figure 2.7). The definition (Figure 2.12) of a supercombinator differs slightly from that for combinators, to specify in rule (2) that the entire λ-calculus program contain no free variables.

For example, the λ-calculus expression $\lambda fa.faa$ is already a supercombinator, which can be translated directly and named "R".

$$\lambda fa.faa \quad\longrightarrow\quad \mathsf{R}fa = faa$$

Conversely, $\lambda y.\lambda x. + x(*yy)$ has y as a free variable of the inner abstraction, which must be bound to make it a supercombinator.

[4]This section draws from the structure and examples of Chapter 13 in [Pey87].

A supercombinator S is an expression of the form

$$S = \lambda x_1.\lambda x_2.\cdots\lambda x_n.E$$

where E is not a λ-calculus abstraction, and

1. S contains no free variables.
2. Any lambda abstraction in E is a supercombinator.
3. $n \geq 0$, *ie.* S can be a simple constant.

Figure 2.12: Supercombinator definition

Lambda lifting is the translation process that converts λ-calculus to supercombinators. It binds all free variables in an expression, in effect "lifting" them to the same level as the other variables in the abstraction. By applying η-conversion to create a new y abstraction, the free variable is removed from the expression (where $\overset{\lambda L}{\Longrightarrow}$ denotes the λ-lifting translation):

$$\lambda y.\lambda x. + x(*yy) \overset{\lambda L}{\Longrightarrow} \lambda y.(\lambda yx. + x(*yy))y$$

The translation continues as follows:

$\mathsf{R}yx = +x(*yy)$	$(\lambda y.(\lambda yx. + x(*yy))y)$
$\lambda y = \mathsf{R}y$	
$\mathsf{R}yx = +x(*yy)$	
$\mathsf{T}y = \mathsf{R}y$	$(\lambda y.\mathsf{R}y)$
$\mathsf{Main} = \mathsf{T}$	

Which can be simplified by noticing that $\mathsf{T}y = \mathsf{R}y$ implies that $\mathsf{T} = \mathsf{R}$, and so we can remove T to have $\mathsf{Main} = \mathsf{R}$.

In practice, we would also like to maintain full laziness, which means that common subexpressions must be sharable. We can have full laziness and better supercombinators besides by noticing that not just variables but *sub-expressions* can be "free", usually the portions of the λ-expression immediately surrounding free variables. These can likewise be lifted out, just as with variables. In the case of *fully lazy lambda-lifting*, we are only concerned with *maximal free expressions* (MFE's), which are the largest free subexpressions that can be identified. The expressions below all have their MFE's underlined:

$$(\lambda y.\lambda x.+x\underline{(*y\ y)})\quad(\lambda y.\lambda x.\underline{+(*y\ y)}x)$$
$$(\lambda x.\underline{(\lambda x.x)}x)\qquad(\lambda x.\lambda y.+y\underline{(*(*xx)3)})$$

MFE's are precisely the common subexpressions that can be shared between one or more evaluations. Lifting the MFE of the last example, the result is:

$\mathsf{R}wx = +xw$	$(\lambda y.(\lambda wx.+xw)\underline{(*yy)})$
$\lambda y = \mathsf{R}(*yy)$	
$\mathsf{R}wx = +xw$	
$\mathsf{T}y = \mathsf{R}(*yy)$	
$\mathsf{Main} = \mathsf{T}$	

The decision as to which variables or MFE's are lifted first can greatly affect both the size and numbers of translated supercombinators. To decide lifting order, *lexical level numbers* (LLN) are assigned to each unit of the λ-expression, as follows:

1. the level number of constants and the "empty" λ-abstraction are 0.
2. the LLN of a variable is the LLN of the λ-abstraction that binds it.
3. the LLN of a λ-abstraction is 1 more than the number of textually-enclosing λ-abstractions.

This concept is intrinsic to the λ-lifting procedure working effectively. In the example of Figure 2.13, $\mathrm{BODY}_{xyz} = +(*yx)(+z(*yy))$ (note that y is free). The LLN's of the variables and subexpressions appear respectively above and below the λ-expression:

$$\lambda x.\lambda z.\underbrace{\underbrace{+\underbrace{(*\overset{0}{y}\ \overset{1}{x})}_{0}}_{1}\underbrace{(\underbrace{+\overset{2}{z}}_{2}\underbrace{(*\overset{0}{y}\ \overset{0}{y})}_{0})}_{2}}_{2}$$

The idea behind ordering parameters is that we wish to make MFE's and the resulting supercombinators both *larger* and *fewer*. For λ-lifting, ordering parameters according to increasing lexical level numbers means abstracting the "most free" variables first. Since freer variables have more flexibility for later λ-lifting, they are

$\lambda x.\lambda z.BODY_{xyz}$	$LLN(y) = 0, LLN(x) = 1, LLN(z) = 2$
	Incorrect λ-lifting order
$\mathsf{R}xyz = \mathrm{BODY}_{xyz}$ $\lambda x.\mathsf{R}xy$	$\lambda z.\mathrm{BODY}_{xyz} \overset{\lambda L}{\Longrightarrow} (\lambda xyz.\mathrm{BODY}_{xyz})xy$ but $LLN(x) > LLN(y)$!
$\mathsf{R}xyz = \mathrm{BODY}_{xyz}$ $\mathsf{T}yx = \mathsf{R}xy$ $\mathsf{T}y$	$\lambda x.\mathsf{R}xy \overset{\lambda L}{\Longrightarrow} (\lambda yx.\mathsf{R}xy)y$ 2 definitions
	Correct λ-lifting order
$\mathsf{R}yxz = \mathrm{BODY}_{xyz}$ $\lambda x.\mathsf{R}yx$	$\lambda z.\mathrm{BODY}_{xyz} \overset{\lambda L}{\Longrightarrow} (\lambda yxz.\mathrm{BODY}_{xyz})yx$ $\surd LLN(x) < LLN(y)$
$\mathsf{R}yxz = \mathrm{BODY}_{xyz}$ $\mathsf{T}yx = \mathsf{R}yx$ $\mathsf{T}y$	$\lambda x.\mathsf{R}yx \overset{\lambda L}{\Longrightarrow} (\lambda yx.\mathsf{R}yx)y$
$\mathsf{R}yxz = \mathrm{BODY}_{xyz}$ $\mathsf{R}y$	since $\mathsf{T}yx = \mathsf{R}yx \vdash \mathsf{T} = \mathsf{R}$ 1 definition

Figure 2.13: Effect of parameter ordering on produced supercombinators

the natural choice for lifting first to get them "out of the way" so we can concentrate on the more constrained variables which remain. These less free variables are those bound at "inner" levels, and can be thought of as having "fewer degrees of freedom" for manipulation. These will be lifted later and appear later in the list of supercombinator parameters. Informally, λ-lifting follows the natural flow of arguments and function calls in the source λ-calculus, resulting in fewer and more efficacious supercombinators. Figure 2.13 shows two possible ways to lift the example expression, where the second makes use of LLN's to reduce the number of supercombinators produced.

2.4.1 Strengths and Limitations

Importantly, the use of recursion with supercombinators is now very much simpler, since supercombinators are themselves named, and may reference each other directly. There are a few variants of the translation process that produce slightly different code, but with self-referential supercombinators, there is no need to resort to crude artifices such as fixed-point operators.

Lambda-lifting and the use of MFE's conspire to compact the supercombinator code we produce while maintaining full laziness. The rearranged applications are more economical in the movement of parameters and the flow of work. Redexes thus incorporate more work, and are less frequently reduced, improving both the time and space costs of execution. The translated code contains more redexes than the λ-calculus source, but far fewer than that when using straight combinatory logic. Joy [Joy84] has shown that when allowed an "infinite" set of combinators, the number of combinators definitions introduced is $O(m)$, where m is the number of variables being lifted. This compares favourably with the typical *size complexity* of $\Theta(m^2)$ for fixed-set logics.

In some cases, λ-lifting in order to retain full laziness is counter-productive. This can result in larger supercombinator redexes which can slow execution and/or excessive numbers of supercombinators being produced (the latter being the combinatory logic problem we originally wished to avoid). It is sometimes cheaper to allow *some* extra work by not λ-lifting fragments of shared code when nothing is gained. Dependency and strictness analysis are also used to restrain laziness, and there are a number of smaller special-case refinements to the translation process. The features which moderate laziness are expensive and place more work on the compiler, in addition to the tasks of type inference, pattern matching, *etc.*. Supercombinator translation can produce multiple solutions, with a lot of latitude in efficiency from one translation to another; I suspect that deriving an optimal supercombinator translation is NP-complete (judging from the similar result for fixed-set logics derived in [Joy84]).

Laziness, the overhead of sharing, and strictness-derived eager evaluations manifest themselves as *annotations* to supercombinators, to properly direct their evaluation at run-time. The many "special cases" that can arise at runtime imply a more complex architecture, with more scratchpad registers, machine instructions, microcode, and memory structures to implement the required techniques.

More importantly, the underlying architecture will have to be changed. Whereas a small set of combinators could be "hard-wired" as machine microcode, supercombinator programs are all large, unique collections of redexes with varying size and complexity. The execution architecture must be able to store and *interpret* the supercombinators themselves in some way. A few general approaches are:

- by expressing the supercombinators as microstore control words, and designing the machine to have a loadable control store which is rewritten for each new program (imprudent).
- by expressing them as individual graphs whose operations are implicit in their structure (inefficient).
- by expressing them in terms of some developed machine language which are instructions to control the CPU and storage of the machine.

These topics are discussed in greater detail in the next chapter.

2.5 Functional Language Summary

Backus [Bac78] distinguishes between λ-calculus-based and higher-level or *pure* functional languages, and asserts that the semantic advantages of λ-calculus can be obtained without expressing programs at any time in the λ-calculus. While pure functional languages may be ultimately superior, they are outside the scope of this thesis. However, as notations go λ-calculus is certainly the most *elegant* in its simplicity, and combinatory logic is conceptually fascinating for its behaviour. Supercombinators have elements of both, but are chiefly a practical compromise. λ-calculus has been used as an intermediate functional language (IFL) with success for over 20 years [Lan64, Mos75, Pau87, Pey87], a task into which it has today settled. The advantage of an IFL is the common bridge it forms between pure functional languages and λ-calculus-based notations, and the work that has been done on these notations. Thus, all functional languages benefit from the set of software, hardware, analysis techniques and optimisations that have been developed for λ-calculus-based notations.

We have examined the notations of λ-calculus, combinatory logic and supercombinators, and shown their development to have been motivated by a search for more compact and efficient evaluation behaviours. The move from λ-calculus to combinators reduced the cost of function calls by substituting simple application for β-reduction, at the expense of greatly increased numbers of applications. The move to supercombinators reduced the sheer numbers of function calls, still the most expensive component of the functional language equation, it being the *cusp* around which most work is done, in a compiler or a machine program. The next chapter discusses the development of hardware and evaluation mechanisms to provide efficient implementations of supercombinators, and describes the Three Instruction Machine as one culmination of these efforts.

Chapter 3

The Three Instruction Machine

Over the last decade, functional programming has become an established field, with standardised languages such as SML, Miranda, HOPE and recently Haskell. As with previous paradigm developments in computer science, there is an impetus for functional language operations and constructs to migrate into hardware. Motivated by a need for greater efficiency and higher execution speeds, a number of special purpose functional architectures have been designed. In their turn, these have become active research instruments that encourage new designs, and the development of better and more capable functional languages. In this chapter, I describe the abstract Three Instruction Machine, one of the latest developments in the functional architecture field.

3.1 Functional Architecture

To properly explain how TIM works, I must illustrate a few key concepts, namely *term rewriting*, *closures* and *environments*, *stack-based evaluation*, *graph reduction* and finally the *"frame-based"* evaluation of TIM. To simplify this task, I give an overview of two representative architectures from the literature, the SECD machine and the Combinator Machine. This will contrast the environment and stack-based architecture of the SECD, with the graph reduction architecture of the Combinator Machine, and pre-instruct the reader on the underlying concepts of TIM, which draws on elements of both.

The underlying concept of all functional language evaluation is *term rewriting*, the process by which an expression in whatever form is recognised to match some equivalent (and hopefully "simpler") form. The expression is rewritten to the new form, and rewriting continues until no more "simpler" form exists. The concept that all functional language expressions and their simpler or "reduced" forms are equivalent is implicit to functional programming. Evaluation methods, whether utilising environments, graph reduction or whatever, are simply ways to implement this underlying term rewriting. The common requirements of all of these methods are: to represent the expression so it may be manipulated; delay the substitution of arguments and easily bind them when needed; and to control and remember the order of expression evaluation.

3.1.1 Environments and the SECD Machine

Originally introduced by Landin [Lan64], the stack-based SECD machine was one of the first attempts at a functional architecture. I consider here the version proposed by Henderson [Hen80], whose instruction set is tailored to execute a limited dialect of LISP called LispKit [HJJ83a].

The source language has a static scoping and contains no global definitions or variables, with definitions supplied prior to use in an enclosing LET (or recursive LETREC) block. With variables and definitions, the machine maintains an *environment* to facilitate substitutions. Definitions of variables are stored in the environment, and variables become run-time references to the current environment, whose run-time structure is controlled explicitly by the compiled instructions. The SECD instructions and machine code programs closely match that of the source code; this small semantic gap means that abstract interpretation of the source and actual execution appear nearly identical. The machine has integer and list operations, and instructions for function entry/exit and support for recursion.

SECD is an acronym from the designations of the four principal registers in the architecture (Figure 3.1). Every structure in the machine is formed of dyadic objects in lists, after the *s*-expressions of LISP. The environment E maintains a nested set of contexts for each function called, which are formed of the argument lists for each function. E is a list of lists (Figure 3.2), indexed by a pair of numbers "(m.n)" denoting the n^{th} member of the m^{th}-deep context (the 0^{th} context is the argument list for the current context).

Stack S: holds intermediate results when evaluating expressions, argument lists and closures prior to a function call, and function return results.

Environment E: stores the context for the current expression. All definitions in the function are bound to some location in the environment.

Control C: holds the current SECD machine code expression to be evaluated, stored as a list. C is a pointer into the code list, and acts like a program counter.

Dump D: used to store the contents of the other registers on context changes, pending their return.

Figure 3.1: SECD registers

$$\begin{array}{lllll} & \vdots & \vdots & \vdots &) \\ & (2.0) & (2.1) & (2.2) & \cdots \\ & (1.0) & (1.1) & (1.2) & \cdots \\ \mathsf{E} \longrightarrow (& (0.0) & (0.1) & (0.2) & \cdots \end{array}$$

Figure 3.2: The SECD environment

The Henderson SECD machine contains 21 instructions, fifteen of these being basic integer and list operations, predicates and a branching "decision" function. All work is in reverse polish form on the contents of the stack, and assumes the previous preparation of necessary arguments. Three variants of the LD (load) instruction are used to place objects on the stack. These may be constants (LDC), bindings for variables from the environment (LD), or *function closures* (LDF). A *closure* is simply the combination of function body (code) and an environment to supply any pending bindings in the body.

The most interesting instructions are the 5 concerned with function calls. A function call, or *application*, requires that the function arguments and a closure be placed on the stack S. The arguments are supplied as a list constructed (including any necessary eager evaluations) prior to the call. For non-recursive functions, the LDF (load function) instruction then places on the stack the function body (following it in the code stream C), and a reference to the current environment E. These three elements are all that is needed to execute the function. The call proceeds with the AP (apply) instruction, which will execute a context change by saving the current contents of S, E, and C onto the dump D, and distributing the arguments and closure into the state registers. The function body resides in C, the environment resides in E and the argument list is added as the first (0th) level of the new environment. On completion, a single result will remain on the stack and the last instruction in the code stream will be RTN (return), which will reinstate the last saved context in D, with the result on top of the old stack contents. Thus the SECD is much like the familiar Von Neumann machine, with the exceptions that all structures are held as s-expressions, function code is "carried around" with each function call site, and environments are passed through pointers, in total. Figure 3.3a describes the function call instructions as state-transitions in the SECD, and Figure 3.3b an example of a non-recursive function call.

s	e	(LDF fun).c	d	→	(fun.e).s	e	c	d
((fun.env) args.s)	e	AP.c	d	→	nil	(args.env)	fun	(sec). d
(res)	e'	RTN.c	(sec).d	→	(res.s)	e	c	d
s	e	DUM.c	d	→	s	(nil.e)	c	d
((fun.(nil.e)) args.s)	(nil.e)	RAP.c	d	→	nil	(args.env)	fun	(sec). d

Figure 3.3a: Function call instructions in SECD

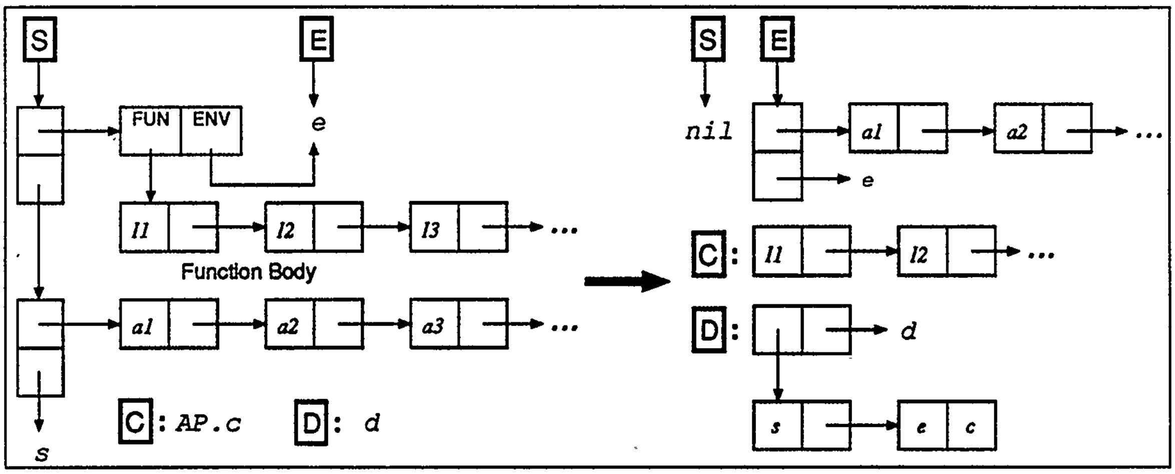


Figure 3.3b: Function preparation and application in SECD

When only enclosing definitions and the argument bindings of elder functions are required, the current environment E is sufficient for the function closure. To properly execute a recursive function, it must have access to its own definition. The SECD constructs a circular list in the environment prior to the function call to make it self-referential, using the DUM (dummy) and RAP (recursive apply) instructions. First, DUM is used to push a *dummy* environment on the stack, an object containing nil. The recursive function definition(s) are placed on the stack with LDF, as if they were an argument list, but each closure references the (dummy.E) environment rather than just E. A "priming" function is then pushed, and RAP is executed. RAP behaves precisely as AP does, with the exception that instead of pushing a *new* object on the environment to hold the code body "arguments", it rewrites the top object (the dummy) with these. In other words, DUM pre-allocates the new environment for the coming function call, but leaves it empty. All subsequent preparations for the function use the dummy environment. RAP completes the process by unpacking the arguments and priming closure, places the arguments on top of the closure environment to form the new environment, and overwrites the dummy physical object with this new environment. This completes the circular structure, as each member of the new environment references the top of the new environment (see Figure 3.4). The priming function we now execute will simply load (one of) the newly recursive definitions onto the stack for the coming *real* application and exit. Execution then continues with gathering of arguments for the recursive function and execution of a regular AP, and the RAP is never executed more than once for each set of recursive functions.

The SECD is somewhat like a high-level language (HLL) machine, in that there is very little difference between its instructions and those of the source language. This is more a function of being based on a purely functional language rather than being λ-calculus-based. However, the language is very simple, limiting the usefulness of SECD as a general-purpose architecture.

SECD uses an environment to implement variables and bindings, and thus avoid the expense of substitution in β-reduction. In addition, SECD dispenses with cumbersome fix-point operators for implementing recursion, by directly creating self-referential environments. The fact that machine code is interpreted, rather than the *structure* of the execution expression (as with λ-calculus evaluation) makes the SECD more efficient in machine cycles and memory consumption.

One major drawback is the use of s-expressions for all storage. There is no

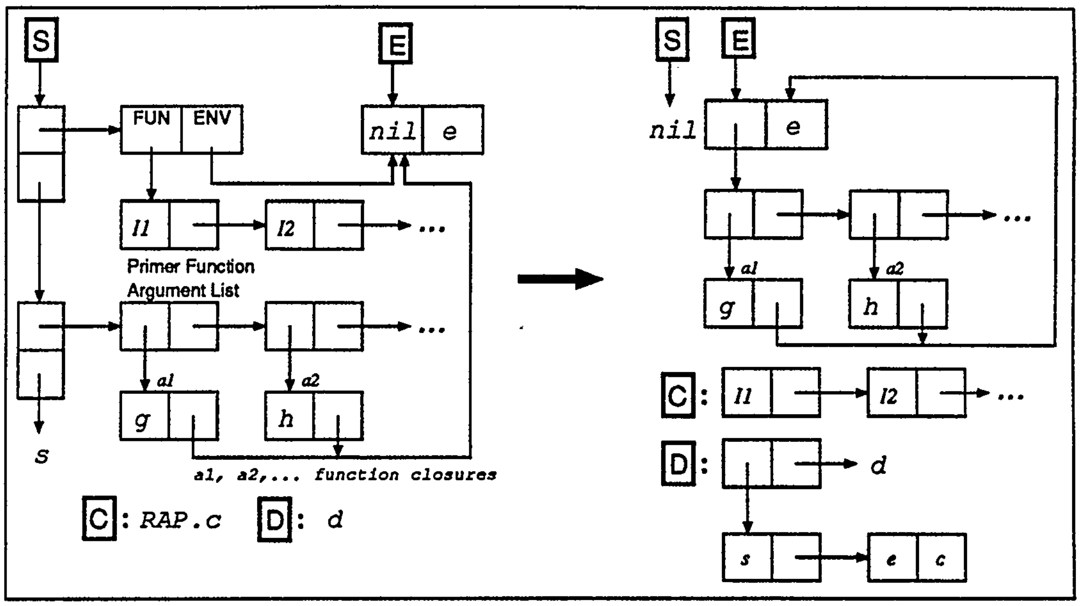


Figure 3.4: Recursive function preparation SECD

design reason to store machine code streams as lists in C, aside from the appearance of consistency with the rest of the machine. Once loaded, the code is never garbage-collected or re-allocated. Linear memory and absolute code addresses would be logically equivalent and cheaper to use than lists. The use of allocatable objects to hold each member of the environment is unnecessary; once defined, each level in the environment remains constant in size (although content may vary in lazy versions of the SECD). Variable-sized objects would be better, and require only minimal changes to the machine. However, optimising the environment would add little benefit, as nearly all references are to the "(0.0)" slot. [BGJ+89][1]. Similarly, the stack S could occupy a linear memory, a more economical approach considering its high usage. This overuse of allocatable objects increases memory consumption, complicates garbage collection, and uses an extra memory access (to read the pointer) for every cell referenced that could otherwise be avoided.

The second major drawback is that the Henderson SECD I describe is eager, but can easily support lazy evaluation. This requires an updatable environment and

[1]For slot $(m.n)$, $\overline{m} = 0.21$ and $\overline{n} = 0.55$, and 80% of all references are to the current LET or LETREC block

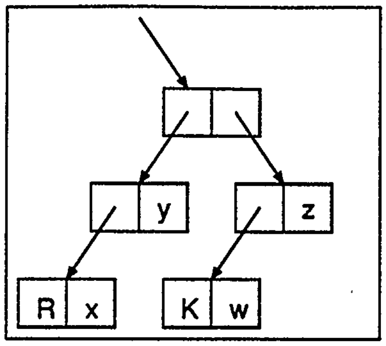
a few new instructions to provide function "wrappers" that delay and then force their execution [Hen80]. Control of suspensions, updating and providing fully lazy evaluation complicate the implementation further.

The notion of the SECD is a good one, but it was meant as an abstract research vehicle, and not a real architecture, a fact reflected in the limited language and impractical design philosophy. The SECD has been built on a full-custom monolithic IC [HBGS89, SBGH89]. Related machines are the Categorical Abstract Machine (CAM) [CCM87] and the Functional Abstract Machine (FAM) [Car83, Car84].

3.1.2 Graph reduction and the Combinator Machine

A conceptually simple way to achieve laziness and sharing is to use *graph reduction* as an evaluation scheme. First suggested by Wadsworth [Wad71](ch. 4), this method distributes the entire code expression into a tree of allocatable nodes, with each node being either a branch, or a terminal holding one or more code symbols. In practice, the expression will be "curried" as follows, $Fxyz \Rightarrow (((Fx)y)z)$ to make the context of each node into an explicit application of a "function" to an "argument". Figure 3.5 shows an example expression and corresponding graph. Graph reduction proceeds with a (usually) depth-first preorder search of the graph for the leftmost-bottommost reducible expression. The path of the search (or parse of the expression) at any time forms the "spine" of the graph, and is retained so that the arguments of the redex are accessible, and our path through the graph may be retraced. The redex itself is defined by the terminal symbol ("head") of the search; this will match one of a set of rewrite rules encoded in the machine. Each will require a certain number of arguments, which immediately precede the head in the graph, and may be accessed from the spine. The redex is then the subgraph which holds the head and the arguments it requires to be reduced. When recognised, the redex is rewritten according to the rule, and reflected in the graph by overwriting the root node of the redex with the reduced graph. Figure 3.6 shows the example graph (a) with spine, and its reduction (b).

Sharing of expressions is done simply by passing a reference to the subgraph in question, and updating happens automatically when the subgraph is reduced and



R x y (K w z)
= ((R x) y) ((K w) z)

[Rabc = ab(ac), Kab = a]

Figure 3.5: The Graph Representation

the root of the redex overwritten. Since only a pointer to the cell is shared, only a single execution will be performed. Furthermore, the graph does not differentiate between expressions and results, relying on the redex search to detect reducible expressions. Laziness is thus implicit in graph reduction, with the evaluation of expressions automatically postponed, right up until the first redex which shares it requires the result to be reduced. No extra annotations to control the suspension and continuation of evaluations are necessary, and the graph thus implements laziness and sharing in a nearly transparent way. Figure 3.7 shows the reduction of a shared expression in the example.

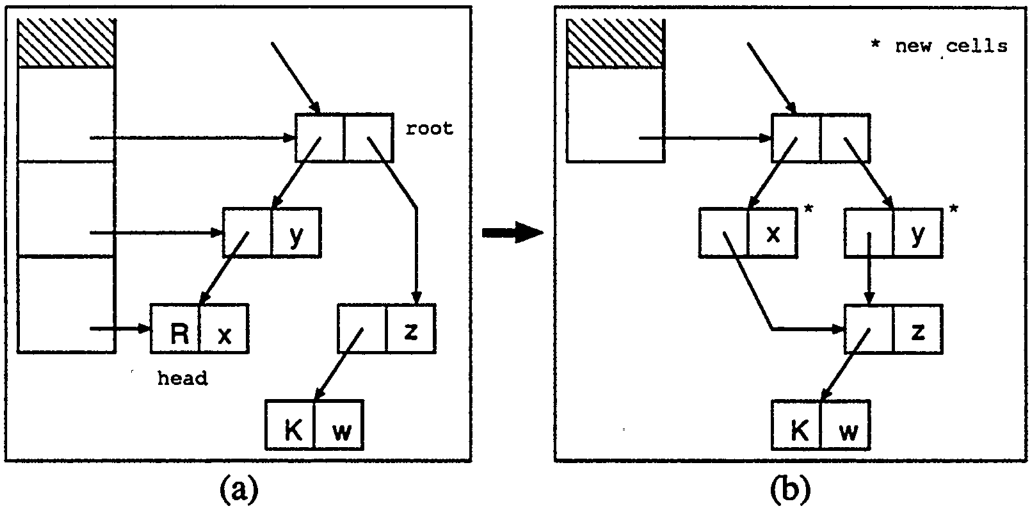


Figure 3.6: Graph Reduction

As an example of a graph-reduction architecture, I consider the machine of Turner

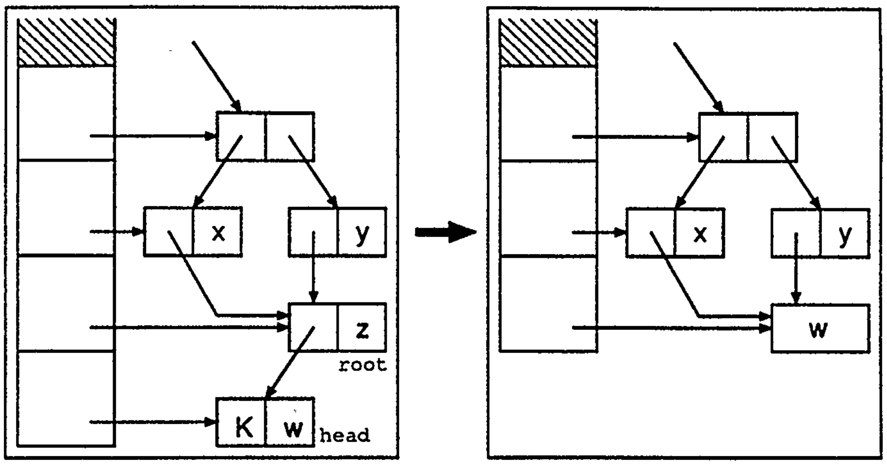


Figure 3.7: Graph Reduction with Sharing

[Tur79a, Tur79b, Tur84], which he suggested while repopularising the use of combinatory logic. This *Combinator Machine* uses a fixed set of combinators as rewrite rules based on the basic set of SKI (see Section 2.2) to reduce combinatory logic programs. The architecture itself is very simple, consisting of only:

- a large allocatable store with garbage collection,
- a stack to hold the "spine" of the evaluation,
- a set of rewrite rules, microcoded as "instructions".

Nearly all the complexity of the machine is in the microcoded combinators, which consist only of microoperations that manipulate the spine, examine the graph through the spine, allocate new nodes, and rewrite the contents of existing nodes. Depending on the combinator, *several* memory cycles are necessary to search back through the spine for the arguments, retrieve the references desired, allocate any new nodes, and overwrite the root of the redex.

The combinator machine implementation of graph reduction has a few inadequacies. First, only a finite library of combinators can be held in microcode at any time, and their complexity is limited, thus restricting the capability to use more or larger combinators. To implement recursion, the combinator machine uses a fixed-point operator such as Y (see Section 2.1.2) to construct a circular self-referential graph.

A subtle problem arises here when it is realised that reduction is inherently "destructive" to the combinator program. The recursive function must not be overwritten if it is to be used more than once, and the simplest (and most expensive) solution is to force Y to copy its function for successive calls. For each node in the recursive definition, this incurs an extra traversal, allocation and garbage collection, and since most programs are contained entirely within recursive definitions, this results in an immediate doubling of execution cost [Her87].

Maintaining a stack to hold the graph spine consumes significant memory bandwidth, considering how quickly the stack contents change at the site of reductions. Fortunately, the stack can be easily avoided by using a technique known as *pointer reversal* [Sch86], in which the nodes of the graph hold the spine temporarily by referencing the last node visisted.

The major drawbacks are not the responsibility of the combinator machine, but due to graph reduction, and combinatory logic itself. While graph reduction is very versatile, and allows any variety of expression to be executed with automatic full support for lazy evaluation and sharing, the cost is high. Allowing *any* node to be shared, suspended and updated is unnecessary and very expensive. When applied to fine-grained combinatory logic, graph reduction spends most of its time simply reorganising the graph rather than doing "useful" work, and most of this effort is expended on supporting laziness that won't be needed.

A number of combinator machines based on graph reduction have been designed and built. The earliest was the Cambridge SKI Machine (SKIM) [Sto83, Sto85, CGMN80], built from discrete components. More recently Ramsdell at MITRE Corp. fabricated the CURRY Chip [Ram86], a full-custom monolithic device. The NORMA machine [Sch86] includes many of the graph reduction optimisations mentioned, including a cache memory for the spine.

3.1.3 Summary

The SECD machine represents everything as s-expressions, and consumes similar resources compared to the combinator machine using graph reduction. The basic SECD does not have laziness, sharing, or simple structure, so in other words, it pays all of the costs of graph reduction without enjoying any of its benefits.

On the other hand, the graph reduction that forms the core of the Combinator Machine is elegant, but an overkill. Not all expressions (or portions thereof) need be shared, updated with results, or lazily evaluated. Since these are all expensive tasks to perform and regulate, it makes more sense to recognise those expressions that will possibly be shared, and only spend the resources on these. While Henderson's SECD has no support for laziness, the combinator machine has too much.

3.2 Three Instruction Machine

The Three Instruction Machine (henceforth TIM) is a functional architecture designed to execute supercombinators. It was initially presented as an abstract machine by Jon Fairbairn (Cambridge University) and Stuart Wray (Olivetti Research UK) [FW87]. TIM uses a unique application of graph reduction technique to perform normal-order fully lazy expression evaluation. The machine supports eager evaluation where it is expedient, and minimizes the overhead of passing unevaluated expressions with its simple, elegant design.

The TIM machine contains elements of both graph reduction such as in the combinator machine, and stack-based environment machines such as the SECD. Its nearest competitor is the G-Machine, which may loosely be described as an SECD machine with support for updating. Rather than being a hybrid of SECD and combinator machines, TIM is more of a graph reduction architecture which utilises environments.

The remainder of this chapter will discuss the abstract machine as proposed, describing the reasoning that goes into TIM's unique design, the three core instructions and their variants, the translation mechanism from supercombinators to TIM machine instructions, and the additional instructions to implement ground types, lazy evaluation and sharing.

3.2.1 The Three Instruction Rationale

The priority goals in the design of the TIM abstract machine were to achieve graph reduction without the graph, and make reduction as efficient as possible. To this end supercombinatory logic (SCL) was employed as code.

There are two expensive inter-related tasks that we will want to avoid if we are to make graph reduction more efficient:

1. Copying graph expressions for execution, to protect the original expression.

2. Constructing graphs on each reduction only to have them discarded or immediately parsed and rewritten.

The point behind using a graph is *reduction*, which happens when we overwrite the root node of a graph expression with its result (thus yielding the familiar advantages of sharing, *etc.*). Thus, if we can dispense with the graph nodes that will never be rewritten, we can save the costs of allocation, collection, and traversal of the expression which they form. TIM greatly reduces the costs of graph construction and reduction by avoiding them. Instead, only function closures are held in environments rather than embedded in a graph. As with graph nodes, environment entries are updatable.

Supercombinators require only a local environment, with arguments and pointers to access other environments (through continuations) passed around as arguments. The operations implicit in the structure of supercombinators can be performed with only a few machine instructions, replacing complicated supercombinator rewrite rules with individually simple operations that perform the same task. Thus the expense of constructing and interpreting graphs holding mostly code is removed by abstracting the code out of the graph.

The designers of TIM built the machine around two central concepts. The first is that everything in the machine be built around a single logical structure that would be used to represent everything. This is a closure *object*, comprising a pair of references, one to a code expression and one to an environment for the expression. Ultimately, objects will be a pair of words holding two pointers to storage in a real machine. Second, that the instructions of the machine focus on function application, the nexus around which most work is done in functional programs.

The placement of the instructions breaks this work up into three distinct phases:

PUSH	prepare arguments on the stack
ENTER	enter the new context
TAKE	retrieve arguments

The three instructions (really instruction *types*, as there are many "flavours" of each) are used to construct combinator contexts as needed, and execute them. As we shall see below, the use of a machine macrocode to moderate the graph reduction process, rather than the raw combinators, yields some particularly effective optimisations towards a more efficient implementation of sharing.

Since any node may be the root of a combinator evaluation, it is usually impossible to determine *a priori* if an arbitrary node will not be overwritten (and is thus a candidate for possible optimisations). However, some progress can be made. Certain nodes will hold evaluated expressions or constants, and will not need updating, so we should try to dispense with these. Similarly, many graph reductions are very simple rearrangements of arguments; for instance, each simple combinator represents a needless cost in graph construction and especially traversal, when one considers that we just had each of these arguments "in our hands" (on the graph reduction spine) a moment ago. Taking this one step further, in lazy graph reduction arguments are passed as pointers to graph expressions, which are evaluated or discarded as the applied combinator dictates. If possible, we would like to entirely avoid constructing the argument graphs until they are needed.

The only place we *need* to overwrite a node, is precisely where the result of the expression is shared amongst other expressions. The central idea is this: if we can detect sharing ahead of time, we should (ideally) be able to allocate, reduce and update only these shared nodes. Each such individual will "hold" its expression (in some as yet undefined non-graphical way) and be overwritten after evaluation with a result sub-expression or constant. Of course, an expression graph contains more than just nodes holding subexpressions, it also controls the computation. Once these interconnecting graph links are abstracted away, we will need some method of maintaining these prior associations.

An SCL program contains an arbitrarily large number of unique, arbitrarily complicated supercombinators. Furthermore, these combinators are distinguished from their simpler counterparts by having names by which they may reference themselves and their brethren. The standard local context formed by combinator parameters is now augmented by a single, large global context of all the supercombinators. Where combinators may only rearrange or replicate their own arguments, a SC definition may *introduce* new symbols in their definitions as well. In effect, supercombinators are very much like functions, and this observation has great bearing on design of the TIM.

3.2.2 Architectural blocks

Code memory

Existing graph reduction machines rely on microcode to implement a small fixed library of simple combinators, and the control algorithm. This both creates and is controlled by the graph it traverses when it recognises and initiates reductions, rewrites nodes with results, and continues to search for other reductions. Given the complexity of a supercombinator program, and the fact that most of the graph structure is now gone, microcode seems inadequate to the task of implementing TIM. Using graph expressions to interpret and reduce other graphs is impractical, so we are left with the need for some variety of "macrocode" to describe how to rewrite each graph. This instruction set will stand between the supercombinator source code and the TIM microcode.

Figure 3.8 shows the old and new situations, where "*T" denotes the transformation between schemes. Supercombinators are compiled into this TIM code, to become graph-evaluation formulae. These are stored as sequential vectors in a linear **program code memory** or CMEM (*not* in an allocatable store as *s*-expressions), indexed by a **program counter** or $\boxed{\text{PC}}$.

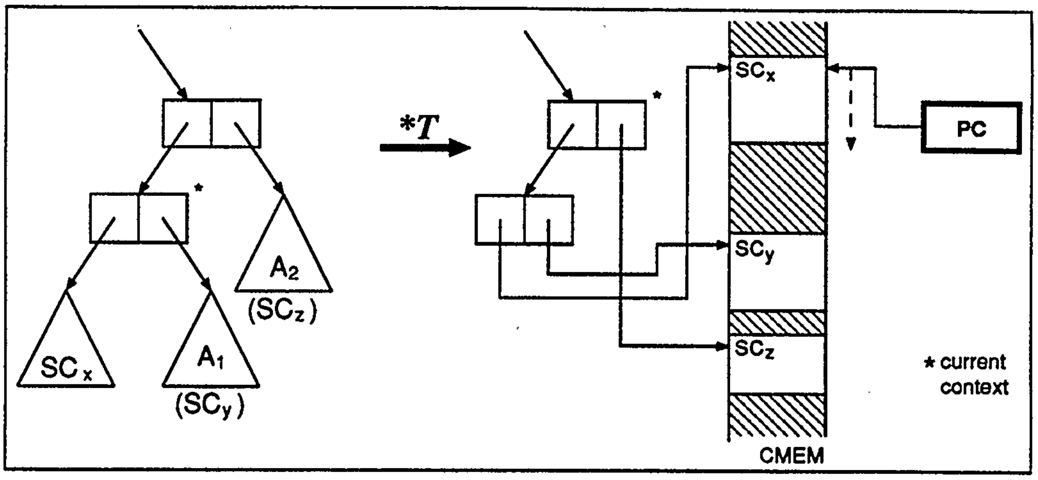


Figure 3.8: Argument graphs built with code sequences

Frame Heap

These frames hold objects, and are referred to by objects. Objects | code | frame | can be updated with new code and frame for any shared code which has been evaluated.

The Frame Heap provides part of what we need to remove argument graphs, but we must still address keeping track of arguments and the rest of the computation. Consider a generic recursive combinator such as the following:

$$\mathsf{REC\ a} = (\mathsf{TEST\ a})(\mathsf{TERM\ a})(\mathsf{REC\ a})$$

which takes a single argument a, and uses a boolean expression (TEST a) to select between either a terminal expression (TERM a) or another recursion (REC a). Figure 3.9 contains a single iteration of this function as a standard graph reduction, where the rectangles represent nodes, and the triangles abbreviated sub-expression graphs.

Aside from the multiple allocations and traversals, the essential work of the reduction can be seen with the three node rewrites that occur (marked "◇" in Figure 3.9). Returning to the notion of supercombinator as function, we can think of these as context change boundaries in the function call history. In this example, we execute a bit from REC, make a call to the function TEST with argument a, which constructs a context for itself (we assume) and a while later returns to the original context with

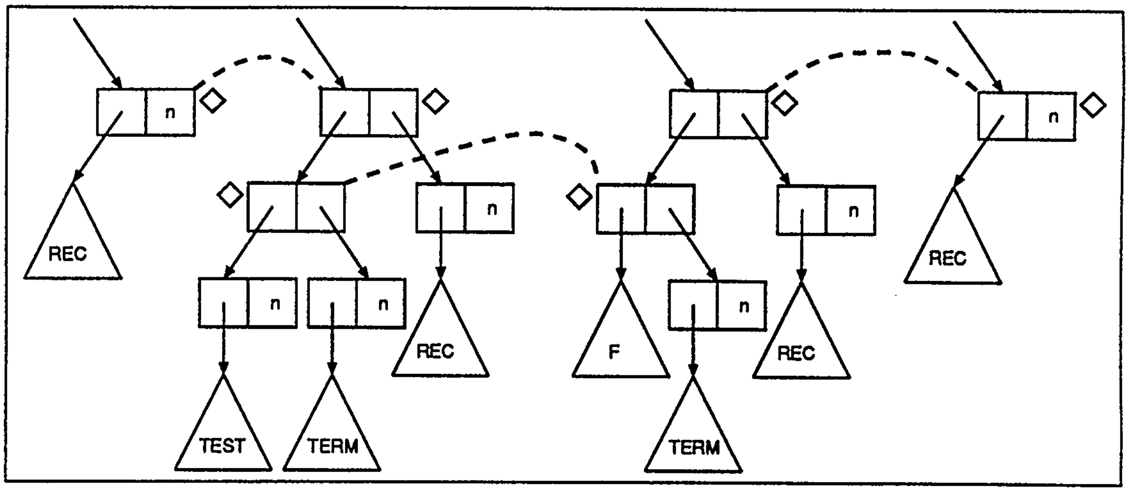


Figure 3.9: Standard graph reduction

the result F (false). Here, the return result uses (TERM a) and (REC a) as arguments, and selects the latter fragment as the one to obtain control of the processor.

Recalling the description of graph reduction (Section 3.1.2), a *spine* is used to keep track of the computation by stacking visited nodes. The spine consists of two conceptual pieces: the local graph, and all the rest. Most of the time we are concerned with the former, where the currently executing combinator is held, along with pointers to its arguments and the expression root (rewriting) node. This "local" part of the graph/spine forms our context, while the remainder holds our execution tree, and tells us where to work next when the current sub-expression is evaluated.

During execution, we need some place to hold *at least* the local context, without the benefit of a graph. This is necessary for two reasons, first that we must be able to perform our "node" updates somewhere, and second that supercombinator applications are no longer "indivisible". That is, contexts must be maintained between excursions to other contexts, whilst evaluating sub-expressions passed as arguments. In orthodox architectures, function arguments are held in an "activation record". Now that we are employing the concept of combinator-as-function, this is a reasonable idea to use in the TIM. On initiation, each supercombinator will require some number of arguments n; once these are located (see below), the TIM permanently stores these arguments (in order) in an activation record called a **frame**. The graph structure is now converted to look something like that which appears in Figure 3.10, where each frame represents a combinator application that has been (at least partially) evaluated. The dashed boundaries denote closures, with code pointers and frames combined.

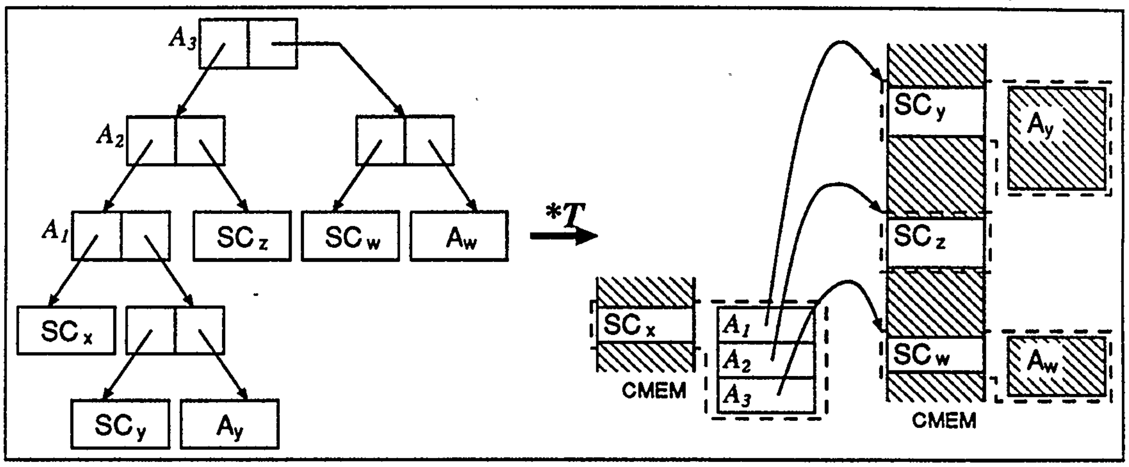


Figure 3.10: Context frames are graph node conglomerates

The Three Instruction Machine contains a large physical memory, called the **frame heap** or HMEM, in which to store all of the frames for all of the active contexts in a program execution. The memory is a garbage-collected store from which variable size context frames are allocated. The frame associated with the currently executed context is always referenced by the machine register **Current Frame** or $\boxed{\text{CF}}$. The combination of a supercombinator code address, and a frame holding its arguments, is known as an **object**, and is sufficient to completely describe a machine context. Thus the machine register pair $\boxed{\boxed{\text{PC}} \mid \boxed{\text{CF}}}$ tells us everything we need to know. The new arrangement appears in Figure 3.11.

It is important to mention that all arguments to a combinator will themselves be combinators, continuations of combinators, or constants and that each will have its own set of arguments. With reference to Figure 3.11, SC_z is a "new" combinator, and so has no arguments associated with it as yet. On the other hand, SC_y and SC_w are partial evaluations of combinators which have already been started and so own some arguments. Should SC_x evaluate an argument, the code at that re-entry point will "continue" execution in the argument context, providing the needed result.

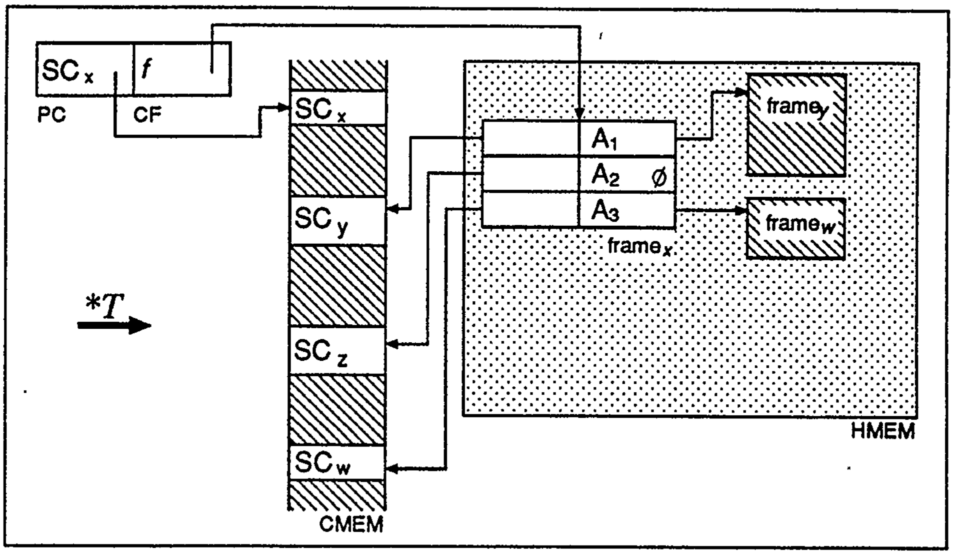


Figure 3.11: The frame scheme

Thus objects holding code and frame references are the units of communication in the TIM, for every argument, every frame inhabitant, and the current context.

System stack

Having abstracted the code away from the graph, and dispensed with those portions that hold arguments and represent the local context, we now turn to dealing with the spine. The spine of the graph (and the stack normally used to hold it) performs the important task of maintaining the calling tree, so that execution can continue in the correct context after an argument evaluation. Further, without the use of microcoded operations, we cannot call a supercombinator in a single machine clock cycle. Its arbitrary arrangement of arguments must be compiled together one cycle at a time, which requires some sort of a temporary staging area.

So while it would be nice to delete the spine, the TIM maintains a vestige in a **system stack.** This stack deals exclusively with objects (as before), is held in a linear memory called SMEM, and its top is referenced by the machine register $\boxed{\text{ARGP}}$(see Figure 4.1). Aside from temporary storage for combinator arguments (the only remnant of the local-context portion of the spine), the stack is also used for ALU operations, to hold return results of evaluations, and continuations. These last are placed on the stack prior to a combinator call or argument evaluation by the parent context, as a "return address" for the child context when it completes with a result. A continuation is accessed either explicitly as an argument subsequently entered, or by virtue of running out of code in the $\boxed{\text{PC}}$, and retrieving the first available continuation on the stack.

3.2.3 The Basic Instruction Set

The three instructions of the TIM architecture each have several options, used to specify the instruction argument source, and any optional behaviours. In the following descriptions, instructions make make reference to the following logical entities:

"COMB," used to designate a combinator code sequence. These are always "new" combinators, having no arguments and thus no context other than the usual initial flat-domain of combinator definitions. Instructions will always use a CMEM pointer to the beginning of the supercombinator, and the implicitly used frame is always null or ϕ.

"ARG," an argument of the currently executing combinator. Instructions with this designator find their referents in the current frame $\boxed{\text{CF}}$.

"LABEL," a continuation of the current combinator. Combined with the current frame, labels form re-entry points used by a child context as one of its arguments (lazy argument passing) or a return address.

"CONST," a machine constant, used to represent ground types such as integer, boolean, lists, *etc.* (see Section 3.2.5).

Here, I describe the machine instructions using the notation of [FW87], with the exception of objects. This is a semantic description of the instructions as machine state transitions. Examples will revert to a concrete representation using machine registers and pictorial descriptions. The machine state appears as a four-tuple:

$$\{\text{Program Code, Current Frame, Stack, Frame Heap}\}$$

with code and stack shown as a list "[...; ...; ...]", object internals as "$\boxed{c_x \mid f_x}$", and frames as "$\boxed{a_1, a_2, ..., a_n}$", where the abbreviations C, A, and F are used for code, argument stack and frame heap. Following the path of a function application, the first task is to provide arguments to the combinator via the stack.

The PUSH *instruction* is used to place arguments on the system stack, drawing from several sources. The arguments can be continuations, code labels, combinator references, or passed arguments (from the frame of another context further up the call tree).

PUSH ARG **n** Place the nth object of the current frame on the stack. In other words, one of the current arguments is pushed on the stack.

$$\left\{[\mathsf{PUSH\ ARG}\ i; C], f, A, F\left[f : \boxed{\ldots, a_i, \ldots}\right]\right\} \Rightarrow \{C, f, [a_i; A], F\}$$

PUSH COMB **c** Push a combinator closure, $\begin{array}{|c|c|}\hline \mathit{codeptr} & \phi \\\hline\end{array}$, where codeptr is the address of the code for combinator "c", and ϕ is the empty environment (*ie.* "c" will retrieve its own arguments to form an environment).

$$\{[\mathsf{PUSH\ COMB}\ l; C], f, A, F\} \Rightarrow \left\{C, f, \left[\begin{array}{|c|c|}\hline c & \phi \\\hline\end{array}; A\right], F\right\}$$

PUSH LABEL **l** Push a continuation of $\begin{array}{|c|c|}\hline \mathit{codeptr} & CF \\\hline\end{array}$ (the current execution context), where codeptr is the address of label "*l*" in the current combinator.

$$\{[\mathsf{PUSH\ LABEL}\ l; C], f, A, F\} \Rightarrow \left\{C, f, \left[\begin{array}{|c|c|}\hline l & f \\\hline\end{array}; A\right], F\right\}$$

The ENTER *instruction* is responsible for executing a context change, where the context may be either a new combinator, a suspension of a shared argument, or a continuation in a calling parent context. Whichever of these the context is, it is represented as a code+frame object, retrieved from one of two places, and becomes the new $\boxed{PC}$ and $\boxed{CF}$.

ENTER ARG **i** Enter the ith argument object in the current frame.

$$\left\{[\mathsf{ENTER\ ARG}\ i; C], f, A, F\left[f : \boxed{\ldots, \begin{array}{|c|c|}\hline c_i & f_i \\\hline\end{array}, \ldots}\right]\right\} \Rightarrow \{c_i, f_i, A, F\}$$

ENTER COMB **c** Enter the combinator "c" ($\begin{array}{|c|c|}\hline \mathit{codeptr} & \phi \\\hline\end{array}$), where codeptr is the address, and the environment is empty.

$$\{[\mathsf{ENTER\ COMB}\ c; C], f, A, F\} \Rightarrow \{c, \phi, A, F\}$$

The TAKE *instruction* fulfills the second half of the context change when necessary. For a fresh call to a combinator, the environment is initially empty. TAKE will pop the desired number of arguments from the stack, allocate a new frame of that size, place the arguments into the frame, and force $\boxed{\text{CF}}$ to reference the new frame.

$$\{[\mathsf{TAKE}\ n; C], f_0, [a_1, \ldots, a_n, A], F\} \Rightarrow \left\{C, f_1, A, F'\left[f_1 : \boxed{a_1, \ldots, a_n}\right]\right\}$$

These six basic forms of the three instructions form the normal order evaluation mechanism for TIM. Figure 3.12 demonstrates the execution of the Turner combinator expression $SKKx$, where the TIM code for S and K appears in Figure 3.13. On the left are the two registers defining machine state, the program counter and current frame. The argument stack appears on the right, as well as new frames allocated from the frame heap. The evaluation proceeds as "$SKKx \Rightarrow Kx(Kx) \Rightarrow x$".

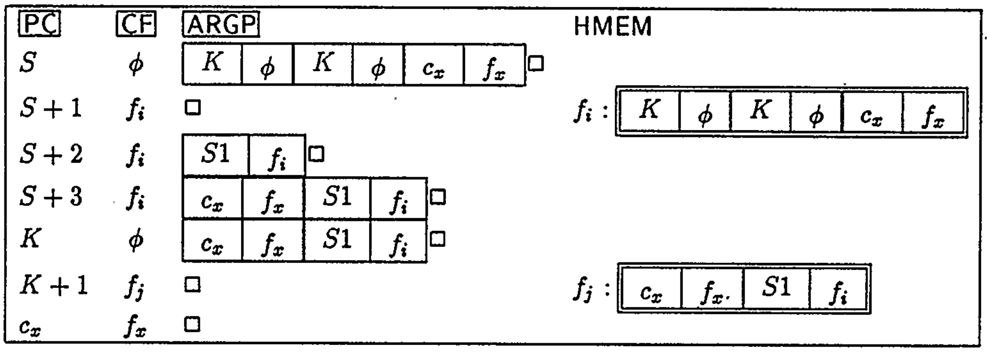


Figure 3.12: An example using Turner Combinators

3.2.4 TIM code compilation

The translation mechanism to convert from supercombinator source code to the TIM instruction code is quite straightforward, at least for the basic instruction set. For example, the TIM code representations for the two familiar combinators S and K appear in Figure 3.13.

The denotational semantics of the translation appear in Table 3.1, adapted from [FW87]; sharing and strictness analysis add significant complexity and have been neglected. The source code is a set of supercombinator definitions, of the form

Kxy = x	K:	TAKE 2	take two arguments
		ENTER ARG 1	execute second
Sxyz = xz(yz)	S:	TAKE 3	take three arguments
		PUSH LABEL S1	a continuation for (yz)
		PUSH ARG 3	push z
		ENTER ARG 1	apply x to [z;S1;A]
	S1:	PUSH ARG 3	push z
		ENTER ARG 2	apply y to [z;A]

Figure 3.13: Supercombinator and TIM code

"$c_i = expr_i$" followed by the main expression of the program. Each supercombinator is shown as a λ-abstraction to represent the argument list. **G** parses the definitions, **C** generates code for each supercombinator, **P** generates PUSH instructions, and **E** the ENTER instructions.

$\mathbf{G}[\![c_i = expr_i; next]\!]\sigma$	$\Rightarrow$	$\mathbf{G}[\![next]\!](\sigma[expr_i/C_i])$
$\mathbf{G}[\![expr]\!]\sigma$	$\Rightarrow$	$\mathbf{C}[\![expr]\!]\sigma$
$\mathbf{C}[\![\lambda a_1 \ldots a_n.expr]\!]\sigma$	$\Rightarrow$	[TAKE n; $\mathbf{C}[\![expr]\!]\sigma$]
$\mathbf{C}[\![e_1 e_2]\!]\sigma$	$\Rightarrow$	[$\mathbf{P}[\![e_2]\!]\sigma$; $\mathbf{C}[\![e_1]\!]\sigma$]
$\mathbf{C}[\![atom]\!]\sigma$	$\Rightarrow$	$\mathbf{E}[\![atom]\!]\sigma$
$\mathbf{P}[\![a_i]\!]\sigma$	$\Rightarrow$	[PUSH ARG i]
$\mathbf{P}[\![c_i]\!]\sigma$	$\Rightarrow$	[PUSH COMB ($\mathbf{C}[\![\sigma(c_i)]\!]\sigma$)]
$\mathbf{P}[\![k]\!]\sigma$	$\Rightarrow$	[PUSH CONST (k)]
$\mathbf{P}[\![expr]\!]\sigma$	$\Rightarrow$	[PUSH LABEL ($\mathbf{C}[\![expr]\!]\sigma$)]
$\mathbf{E}[\![a_i]\!]\sigma$	$\Rightarrow$	[ENTER ARG i]
$\mathbf{E}[\![c_i]\!]\sigma$	$\Rightarrow$	[ENTER COMB ($\mathbf{C}[\![\sigma(c_i)]\!]\sigma$)]

Table 3.1: Translation semantics for TIM code

As example of the compilation output appears in Figure 3.14. The function from n, which produces an infinite list of numbers "$n : n+1 : n+2 : \cdots$", is compiled first to a supercombinator and thence to TIM machine instructions. Note that we assume a "standard library" of supercombinator support functions has been defined and compiled for the use of programs. These are integer, boolean, list and I/O operations that are provided beforehand (see Section 3.2.5).

```
from n = n : from (n + 1)                            original source code
from n = (cons n (from (+ n 1)))                     supercombinator code

                         TIM machine code
      (         cons n                       (  from   (+ n 1) ) )
FROM : TAKE 1                         FROM1 :  PUSH COMB FROM
                PUSH LABEL FROM1                     PUSH ARG 1
                PUSH ARG 1                           PUSH CONST 1
                ENTER COMB CONS                      ENTER COMB +
```

Figure 3.14: Example TIM code translation

3.2.5 Ground Types and Operations

Of course, TIM must include ground types such as integers, booleans and lists to be useful. The original designers chose to maintain consistency with the object concept in the representation of ground types. Machine constants take the form of objects, with the constant held in the frame reference half of the pair. The constant object is distinguished from regular closures by the contents of the code reference, a special instruction called SELF: $\boxed{\mathsf{SELF} \mid \mathrm{k}}$. The larger purpose of the SELF instruction is to make a constant appear to have the same operational behaviour as a regular closure. When TIM attempts to evaluate a constant, it must recognise it and swap to an alternate context. Constants can be directly entered as passed arguments, or arrived at after a string of strict operations. In either case, the current evaluation is complete, and it is time to return to the previous context; this will have been saved as a continuation on the stack, immediately preceding the entrance to this context. Thus when executed, SELF has the sole task of swapping itself with the first object it finds on the stack. The two instructions dealing with ground types are:

PUSH COMB k is used to push a constant onto the stack, as a SELF object.

$$\{[\mathsf{PUSH\ CONST}\ k; C], f, A, F\} \Rightarrow \left\{C, f, \left[\boxed{\mathsf{SELF} \mid \mathrm{k}}, A\right], F\right\}$$

SELF construct a SELF object with the contents of the $\boxed{\mathsf{CF}}$ as precision, swap with the first object on the stack, and enter the stack object.

$$\left\{\mathsf{SELF}, k, \left[\boxed{c_x \mid f_x}, A\right], F\right\} \Rightarrow \left\{c_x, f_x, \left[\boxed{\mathsf{SELF} \mid k}, A\right], F\right\}$$

ALU instructions

Some of the basic operations from the TIM machine can be executed almost directly in hardware. These ALU instructions are the *strict* operators from the source language (those that require both their arguments to produce an answer), and consist of the familiar mathematical, list manipulation, and boolean operations we expect in any functional language. The operation P is used to define a pair, the left and right members of which are accessed with the L and R operators, respectively. Mathematical and boolean operations (like comparison) can be built in, along with branching operations for the selector IF, and so on. The operators included in the original abstract TIM are shown in Table 3.2. It should be noted that the various operand types (integer, character, list cell, pair) are recognised only at the source level, type inference and checking at compile-time obviate run-time machine type mechanisms and checking.

$+ \; - \; \times \; \div \; \%$	integer	&& || $\ll \; \gg \; \sim$	logical
$< \; \leq \; = \; \neq \; \geq \; >$	boolean (int)	$< \; \leq \; = \; \neq \; \geq \; >$	boolean (char)
P L R	pairs	hd tl null	lists
opt-in opt-out	IO		
get-file append-to-file make-file delete-file			file handling

Table 3.2: TIM built-in operators[2]

Although the strict machine operations are built into the architecture, just like any other function they require a supercombinator "wrapper" to evaluate the arguments. Each argument is processed and placed on the stack in turn, before invoking an ALU operation to produce a result (that for "+" is denoted "#+", for example). A stencil for an n-argument strict operation appears in Figure 3.15.

3.2.6 Implementing Laziness

TIM already provides call-by-name evaluation, by using suspensions to postpone the evaluation of arguments and alternate sections of supercombinators. These suspensions can be passed freely wherever they are needed to share expressions. All that

[2]Extracted from the Ponder system of Fairbairn and Wray [FW86]

```
op:     TAKE n              } Frame the context
        PUSH LABEL op1      }
        ENTER ARG 1         } Process argument 1
op1:    PUSH LABEL op2      }
        ENTER ARG 2         } Process argument 2
              :
        PUSH LABEL opn      }
        ENTER ARG n         } Process argument n
opn:    #alu_op             } execute ALU operation
```

Figure 3.15: Ground Type Operator Code

remains is to store the value of shared expressions so that they are evaluated but a single time. There are three modifications to the instruction set of TIM necessary to implement laziness, to ENTER, TAKE and PUSH.

Recall from Section 2.4 that only arguments are ever shared in a fully lazy set of supercombinators, since any expression which may be shared is λ-lifted to a new combinator definition. The result of a shared argument will always be either a constant (the trivial case) or a partial application. This can be visualised in the examples below, using the S supercombinator as context where its third argument is shared. In each case the third argument T will be eventually reduced to a new form T'. The first example produces a constant, while the second results as a partial application (the supercombinator T' which "adds 6 to its argument").

$$\mathsf{SC}_x\mathsf{C}_y\underline{\mathsf{T} = (+23)} \overset{*}{\Rightarrow} \mathsf{SC}_x\mathsf{C}_y\underline{\mathsf{T}' = 5}$$
$$\mathsf{SC}_x\mathsf{C}_y\underline{\mathsf{T}x = (+(*23)x)} \overset{*}{\Rightarrow} \mathsf{SC}_x\mathsf{C}_y\underline{\mathsf{T}'x = (+6x)}$$

To ensure that updating is performed properly, a shared expression must be distinguished from regular expressions, the expression result must be recognised when it is available, and the original location of the expression overwritten with this result. Since a shared expression is always retrieved from an argument slot in some frame, the result should be written over that same frame slot. The particular frame and argument to update is known at the time that the argument is pushed onto the stack. In TIM this update information is held in a *mark*, shown as "$\boxed{f \mid i}$", which is placed onto the stack immediately before a shared evaluation begins. The supercombinator compiler determines which arguments are potentially shared (using a "when in doubt" method). A modification to the ENTER instruction enables it to place marks:

$$\left\{[\text{ENTER ARG } i; C], f, A, F\left[f : \boxed{\ldots, \boxed{c_i \mid f_i}, \ldots}\right]\right\} \Rightarrow \left\{c_i, f_i, \left[\boxed{\boxed{f \mid i}}, A\right], F\right\}$$

Of course, the entry occurs in a child closure passed the shared expression as an argument, and the update information is no longer available to the child context. TIM must link the entry to the argument push, and a modification to the PUSH instruction provides the update location. Instead of pushing the literal argument, an *indirection* containing the appropriate ENTER instruction is pushed:

$$\{[\text{PUSH ARG } i; C], f, A, F\} \Rightarrow \left\{C, f, \left[\boxed{\text{ENTER ARG } i \mid f}, A\right], F\right\}$$

This insures a mark referencing the original frame slot is pushed onto the stack at evaluation time. In Figure 3.16, if the combinator c requires 3 arguments, we have a shared partial application, in this case retrieved from the m^{th} argument of frame f. On execution, c will attempt to consume extra arguments, and be prevented by doing so when its TAKE instruction encounters the mark.

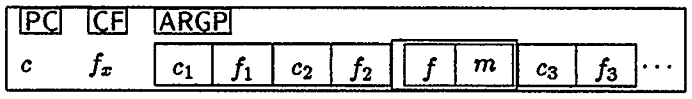


Figure 3.16: Update marker in the stack

The shared application is saved in a form that can be reconstructed on demand. This *suspension* has two elements: i) a special *update frame* allocated to hold the arguments until needed, and ii) a code sequence "created" to restore the arguments from the update frame to the stack, and enter the combinator. For the available arguments i less than requested, a modified TAKE instruction now provides TIM with sharing:

$$\begin{aligned}&\left\{[\mathsf{TAKE}\ n; C], f_x, \left[a_1, \ldots, a_i, \boxed{\boxed{f \mid m}}, A\right], F\left[f : \boxed{\ldots, a_m, \ldots}\right]\right\} \\ &\qquad \Rightarrow \left\{P, f_y, A, F\left[f : \boxed{\ldots, \boxed{P \mid f_y}, \ldots}, f_y : \boxed{a_1, \ldots, a_i}\right]\right\} \\ &\qquad \text{where } P = [\mathsf{PUSH\ ARG}\ i; \ldots; \mathsf{PUSH\ ARG}\ 1; \mathsf{TAKE}\ n; C]\end{aligned}$$

It should be noted that a suspension is the same as a continuation, except for its originating at runtime as a shared evaluation. After TAKE has saved the application, the current context will be the first to access the suspension. First, the arguments of the update frame will be restored to the stack in original order. TAKE will again attempt to get the full n arguments (and may discover another mark) and execution will continue into the body of the supercombinator with the next instruction C. Any other closure passed (an indirection to) this argument will now enter the suspension, and reconstruct the partial application for itself, specialising it to whichever arguments it chooses to place on the stack.

One of the immediately suggested optimisations in the original TIM machine is to maintain both lazy and eager versions of the instructions, so that the expense of lazy evaluation can be avoided where possible. In the full instruction set that appears in Table 3.3, the eager or unshared versions are denoted "UNS'.

3.2.7 Summary

On first examination, the TIM architecture compares favourably with both SECD and Combinator Machines. TIM does significantly more than the SECD with far fewer instructions, due mostly to the universal use of objects. The abstract machine applies the salient features of graph reduction while avoiding the needless expense of the naïve global approach in the combinator machine. Recursion is simplified by the use of supercombinators which have names, thereby avoiding fixed-point operators. In fact, much of the work of implementing full laziness has already been done for TIM by using supercombinators as source code.

However, the TIM instructions encode quite complex operations (notably TAKE), despite their conceptual simplicity. The instruction modifications for sharing appear to require self-modifying code, and objects which contain both references and packed

$$\left\{[\mathsf{PUSH\ ARG\ UNS}\ i;C], f, A, F\left[f:\boxed{\ldots,a_i,\ldots}\right]\right\} \Rightarrow \{C, f, [a_i;A], F\}$$

$$\{[\mathsf{PUSH\ ARG}\ i;C], f, A, F\} \Rightarrow \left\{C, f, \left[\boxed{\mathsf{ENTER\ ARG}\ i \mid f}, A\right], F\right\}$$

$$\{[\mathsf{PUSH\ COMB}\ l;C], f, A, F\} \Rightarrow \left\{C, f, \left[\boxed{c \mid \phi}; A\right], F\right\}$$

$$\{[\mathsf{PUSH\ LABEL}\ l;C], f, A, F\} \Rightarrow \left\{C, f, \left[\boxed{l \mid f}; A\right], F\right\}$$

$$\{[\mathsf{PUSH\ CONST}\ k;C], f, A, F\} \Rightarrow \left\{C, f, \left[\boxed{\mathsf{SELF} \mid \mathrm{k}}, A\right], F\right\}$$

$$\left\{\mathsf{SELF}, k, \left[\boxed{c_x \mid f_x}, A\right], F\right\} \Rightarrow \left\{c_x, f_x, \left[\boxed{\mathsf{SELF} \mid k}, A\right], F\right\}$$

$$\left\{[\mathsf{ENTER\ ARG\ UNS}\ i;C], f, A, F\left[f:\boxed{\ldots,\boxed{c_i \mid f_i},\ldots}\right]\right\} \Rightarrow \{c_i, f_i, A, F\}$$

$$\left\{[\mathsf{ENTER\ ARG}\ i;C], f, A, F\left[f:\boxed{\ldots,\boxed{c_i \mid f_i},\ldots}\right]\right\} \Rightarrow \left\{c_i, f_i, \left[\boxed{f \mid i}, A\right], F\right\}$$

$$\{[\mathsf{ENTER\ COMB}\ c;C], f, A, F\} \Rightarrow \{c, \phi, A, F\}$$

$$\{[\mathsf{TAKE}\ n;C], f_0, [a_1,\ldots,a_n,A], F\} \Rightarrow \left\{C, f_1, A, F'\left[f_1:\boxed{a_1,\ldots,a_n}\right]\right\}$$

$$\left\{[\mathsf{TAKE}\ n;C], f_x, \left[a_1,\ldots,a_i,\boxed{f \mid m}, A\right], F\left[f:\boxed{\ldots,a_m,\ldots}\right]\right\} \quad \text{for } 0 \le i < n$$

$$\Rightarrow \left\{P, f_y, A, F\left[f:\boxed{\ldots,\boxed{P \mid f_y},\ldots}, f_y:\boxed{a_1,\ldots,a_i}\right]\right\}$$

$$\text{where } P = [\mathsf{PUSH\ ARG}\ i;\ldots;\mathsf{PUSH\ ARG}\ 1;\mathsf{TAKE}\ n;C]$$

Table 3.3: TIM instruction set

instructions, a situation we would like to avoid. The three disparate storage spaces are necessary to the function of TIM, but present a wide latitude for efficient implementation options. Developing approaches to designing these will be the focus of the following chapters.

Argo [Arg88] has suggested many notational and hardware optimisations for TIM, a number of which will be addressed below. Chin [JGCH89, CJH89] has developed an abstract machine which is a hybrid of logic programming and functional architectures. This machine allows logic clauses and functions to be interchanged freely, and performs both resolution techniques and TIM-style expression evaluation.

The interested reader is referred to the following for further study of functional architecture, and machines related to TIM. The Johnsson G-Machine [Joh84, Joh87, Kie85] is a predecessor of the TIM machine, which I characterize as an environment-based SECD machine with support for graph reduction (whereas TIM is designed to do graph reduction with environment support). The G-machine has spurred a number of refinements, such as the spineless G-Machine [BPR88, Pey88], and the ν-G-machine [AJ89, Aug88, Bur88], a shared memory multiprocessor. An architecture currently under development is the GRIP machine [PCSH87], a shared memory parallel graph reducer. George [Geo89] suggests an abstract machine for multiprocessor graph reduction, derived from the G-Machine and focusing on parallel evaluation of strict arguments. An early survey paper not limited to functional architectures is [Veg84].

Chapter 4

Structural Optimisations

In this and the succeeding chapter, I discuss a number of optimisations to the TIM architecture, some of which are novel and some of which have already appeared in the literature, mainly in the work of Fairbairn and Wray [FW87, WF89] and Argo [Arg89].

I begin with a brief overview of the set of proposed optimisations, pointing out that set which is within the scope of this work. I then outline some of the architectural design issues in TIM, propose a set of objectives that define an efficient and practical hardware implementation, and present a more appropriate design philosophy directed towards achieving this goal.

The remainder of Chapter 4 is concerned roughly with structural issues surrounding the context change, specifically those operations performed during or near the TAKE instruction, including the creation of sharing information and updating of results. Approximately half of this material is new and original work.

Chapter 5 will be concerned with issues much closer to the practical implementation details of TIM. An assortment of optimisations concerned peripherally with TAKE and deferred from Chapter 4, and a host of small practical optimisations ranging over marking, result updating, and the SELF instruction are covered. A novel design for the TIM frame is developed, along with an instruction format, main memory hierarchy, and control unit with microcode. The majority of this material is original work.

Most of those proposals that have been put forward are directed at the *abstract* TIM. In contrast, I analyse each optimisation from an architectural design perspective.

4.1 Optimisations to TIM

For historical interests, the predecessor to TIM is the PONDER machine documented in [Fai86]. In the initial TIM paper [FW87], a number of suggestions for future improvements to the abstract machine were outlined. For the most part these were immediately obvious simple changes that were not pursued in any detail. In subsequent work, more complex improvements have been presented, and some of the initial work developed further. However, nearly all the work aimed at the abstract TIM neglects practical issues. Elegant improvements to the abstract design may well result in inelegant hardware design, and incur practical design problems. Furthermore, previous work has used a "shotgun" approach for TIM improvements; I try to unify the chosen diverse optimisations into a single underlying design philosophy.

The original TIM paper of Fairbairn and Wray [FW87] suggested a handful of optimisations for dealing with marks, update frames, and regular frames. Some good tricks to use in a hardware implementation are also mentioned. Most of these small changes are covered and credited below. A more detailed and readable description of the TIM machine, its links to lazy evaluation, and details of implementation appears in [WF89]. The ideas already presented for marks and update frames are extended, and the paper introduces the use of a stack for marks, a method of deferring mark operations to cut down on marking and the expense of handling them, and a way to use a single frame for all updatees recognised on a TAKE. These are all treated herein.

Both of the above papers also discuss the benefits of sharing and strictness analysis, which deal with the most efficient application of the normal order or lazy versions of the instructions. If an expression is not shared, normal order instructions are used to PUSH and ENTER it. For those evaluations which do not need to be updated, the code may be reorganised to place the task in question at the end of a list of internally strict operations. Sharing and strictness analysis restrict the costs of sharing to where it is necessary, and limit the memory and bandwidth consumption of frames, marks, and updates. These are software techniques which fall mainly within the domain of the compiler writer. While I avoid the techniques behind the prudent use of the instruction set, the examination of alternative versions of the instructions or details of implementation are within the scope of the thesis.

Argo [Arg89] provides the most thorough treatment of the TIM architecture to date, and includes experimental results lacking in [FW87] and [WF89]. Argo covers most of the optimisations suggested previously, focusing on optimisations to the instruction set. The first half of the paper concentrates on reducing memory activity in TIM in both the heap and stack, with treatments of partial applications and "fully applied" combinators. The second half presents a redesign of TIM that involved a different implementation of sharing, new markers and a method to place frames on the stack. The changes used to produce the "G-TIM" machine rely on analysis techniques that are likewise outside the scope of this thesis.

Each of the optimisations from the literature which I address, and those that I suggest, is presented using the following format:

Rationale a brief introduction of the problem, a rationale for the optimisation, and the explanation of its operation,

Implementation suggested modifications to the optimisation and implementation options, ending with a critique suggesting the best approach(es), and covering any possible significant effects on other parts of TIM and other optimisations,

Results the analysis and testing strategy (where necessary), and the results suggested by the experiments.

The symbols $\oplus$ and $\ominus$ are sometimes used to represent the pros and cons, respectively, of a particular approach.

Each result is presented as an estimation of the efficacy of the optimisation. For the original ideas, I compare the new approach to any competitors in the literature, or provide a prediction on measurable improvement. Optimisations suggested in the literature are first subjected to a practical design process, considering all possible implementation options including original ones, and then a judgement on the implementation is made through simulation or analysis. In some cases, literature results are contradicted or weakened enough to be rejected outright, and I suggest an alternative optimisation; however, many of these can be accommodated with slight modifications to allow acceptable implementation. This is especially true of those which are mutually derived and have appeared in the literature since the inception of this work, for which I have a different view of the appropriate hardware methods to be applied.

4.2 Design Philosophy

Throughout this work, I have tried to maintain a design philosophy for the TIM, in order that the changes proposed and subsequently judged for effectiveness could be fitted into a cohesive whole. There is thus a bias in the design towards those optimisations that are useful *and* can be easily integrated together. Thus, defining the design philosophy under which these determinations were to be made was key.

This presents a problem, since we can not easily draw on existing work to develop this philosophy. A quick survey of where we stand with the TIM machine would place us somewhere above the Reduced Instruction Set Computer (RISC) paradigm in complexity, but below the Complex Instruction Set Computer (CISC) definition. TIM does not reside in the continuum of systems conveniently described by the RISC and CISC extremum, since it contains elements foreign to both. Thus, the majority of Von Neumann-grounded knowledge on how to build a machine is not directly applicable. Within functional architectures, the nearest relation to TIM is the G-Machine [Joh84] (see Section 3.2.7). This older architecture implements lazy evaluation and sharing under graph reduction, while attempting to treat the inherent inefficiency problems of the graph. Unfortunately, the great failing of the various incarnations of the G-Machine is that each has a large number of instructions and contains many special-case optimisations. In comparison to TIM, the G-Machine presents a highly contrived "retrofitted" appearance, making the machine overly complicated and prone to suffer from the weight of its own instruction set and compiler requirements. To summarise, not only is the nearest competitor to TIM not very useful for developing a design model, there is very little concrete design work extant for functional architectures at all. Thus, I had to return to first principles to provide a well-integrated design philosophy for TIM.

Design Objectives The central goal, as with any architecture, is to achieve a good balance of three qualities: it should be small, efficient and "clean". The size of the architecture is important since the implementation will likely be limited by time and cost. Unnecessary design complexity should be avoided, to make the hardware amenable to verification, and reasonably easy to layout and simulate. The architecture will eventually be implemented as a single monolithic device, where a simpler, smaller design will translate to a more modest transistor budget, and directly affect the number of man-hours necessary to build the chip, and meet the predicted performance goals.

Thus, the "keep it simple" principle as used here means that more promising, more easily prototyped optimisations should be tried first over the more complex and risky variety. This will translate to more effectively pursued design and implementation. In addition, standardised design using implicitly conservative estimates on the quality of the eventual fabrication technology and of peripheral system hardware. It should be possible to use average "off-the-shelf" memory chips, interface components, and support hardware. The performance of TIM should not depend on having the fastest or densest fabrication technology, or the best memories or intelligent controllers.

The proposed changes and optimisations to TIM should contribute to the overall throughput of the architecture. This translates to higher MIPS[1] ratings, lower memory consumption, less garbage collection and so forth. Few functional architectures are competitive with commercially available Von Neumann machines, and even fewer have published MIPS ratings (an admittedly Von Neumann concept of questionable use with functional architectures). Absolute quantitative measurements are not practical without a candidate fabrication technology and at least a tentative system design, so optimisations are weighed on their ability to *compete* with each other, in the three qualities mentioned above. I use these objectives to define a good working design for TIM within the confines of an M.Sc. thesis, while the eventual implementation and comparison with conventional technologies is left to future work.

As for keeping expense and implementation difficulty low, the set of potential

[1]Millions of Instructions Per Second

improvements has been ordered according to the same criteria. For example, "paper changes" that modify the behaviour of the abstract machine come before the design of complex hardware solutions. Namely, reducing the number of update frames created by TAKE ("Conglomerate Frames") is cheaper and easier than designing a heap cache and boosting memory bandwidth.

The last quality, that of being "clean", cannot be specified precisely. Hardware designers have named this concept variously as "style", "elegance", "orthogonality", and a host of other descriptives. Essentially, it boils down to how well the architecture holds together, while minimizing waste of hardware resources, and smoothing the flow of instructions and data amongst resources. A continual emphasis on *homogeneity* throughout the design process, for the instruction set, control unit, datapath, buses and registers, can help them all work well together. The absence of loose-ends and special cases means the absence of "irritants" around which performance hits and bottlenecks may form, since the use of hardware in the time domain is more balanced.

Design Issues in TIM There are a number of implementation problems posed by TIM, but the two major ones are:

1. the wide disparity in task responsibilities amongst instructions, and

2. the very heavy memory cycle demands of each instruction.

The first problem is best characterized by comparing the TAKE instruction with for example PUSH CONST. TAKE will need tens and sometimes hundreds of micro-operations as opposed to the three or four required by to push a constant onto the stack. Instructions exhibit widely disparate execution times, types of logical tasks, and hardware and memory usage patterns. This is the case not only for these two extremes, but between different flavours of the same instruction, and for different execution contexts (marks and updates make TAKE widely variant). If the goal is to provide high throughput and the best use of the available hardware resources, how does a hardware designer balance the needs of these two unorthogonal instructions? The majority of the remaining material in this chapter is concerned with optimising the tasks TAKE performs, in order to provide a good basis for a fast implementation.

The second problem is the overriding concern with all design decisions in the TIM architecture. In all phases of execution, TIM is memory intensive (see Figure 4.1). Each operation accesses two or more logical memory partitions at least once, and in the case of all partitions but CMEM each access is four bytes wide. All of the optimisations discussed in both Chapter 4 and Chapter 5 are involved in some capacity with reducing the number of memory accesses, or improving on the speed or usefulness of each access.

Operation	CMEM	HMEM		SMEM		MMEM	
	R	R	W	R	W	R	W
PUSH ARG UNS	1	1			1		
PUSH ARG	1				1		
PUSH LABEL	1				1		
PUSH COMB	3				1		
PUSH CONST	3				1		
SELF	1			1	1		
ENTER ARG UNS	1	1					
ENTER ARG	1	1					1
ENTER COMB	3						
TAKE (no marks)	1		n	n			
TAKE (marks)	1		n+m	n		m	
RESTORE	1	r+1			r		

Table 4.1: Breakdown of Memory Accesses by Operation[2]

Philosophy of TIM Despite its being called a functional architecture, TIM is built out of the same hardware as any other computer, and must be treated as such. All design methods employed and most of the optimisations discussed herein have been used before, but applied to conventional Von Neumann machines. While the techniques are not new, they have been selected carefully to be appropriate to the structure of TIM.

Process and Tools At the initial stage, the process of choosing and exploring optimisations was a subjective one. Those optimisations which looked "good", on their own and especially in concert, were chosen for examination. All optimisations are

[2]This table mentions a new instruction "RESTORE" (see Section 5.1.2) and assumes the machine instruction format described in Section 5.4.

judged mainly from paper analysis, and argument of the pros and cons of each, concerning aspects of the implementation, effects on other optimisations and portions of the design, and so forth. This summary of trade-offs usually yielded a clear conclusion as to the efficacy of the proposed improvement. In cases where the conclusion was unclear, simulation of the architecture was used. Rather than do the thousands of simulations for a statistically solid conclusion, I sought to get an indication which side of the scale the optimisation in question falls on. Thus, the subjective analysis of the modification combined with some supporting tests, would yield the conclusion that the modification is promising, and worth further examination.

To facilitate simulations, I have designed and implemented a register-level hardware simulation tool called TIMSIM, based on the object-oriented language SIMULA and the DEMOS simulation package. TIMSIM is described in some detail in Appendix A, while Appendix B describes the TIM simulation environment and includes a sample simulation run. The results derived from simulation came in two basic varieties: qualitative and quantitative. Qualitative results about the basic behaviour of TIM are useful to substantiate the arguments for or against an optimisation or aspects of the implementation. These results would be dynamic instruction execution frequencies, memory consumption, and the like for a set of representative test programs. For instance, if there is a very effective but expensive modification to an instruction such as PUSH COMB, and this instruction has a low execution frequency, the result argues against the global usefulness of the optimisation.

Quantitative tests are used to test whether the optimisation, when fully implemented, provides better performance or not. These use the three stage process of hypothesis, experiment design, and interpretation of results. In these tests the specific change is modeled for comparison against its competitors or the unoptimised version. The comparison will always test the rationale of the optimisation, usually a potential improvement to memory consumption or reduced memory activity, and often a generic increased execution speed. Optimisations sometimes have palpable effects on the performance of other optimisations, and where these relations are not easily characterized on paper, simulation will sometimes give an indication of whether or not optimisations work well together. In some cases, there is quite a

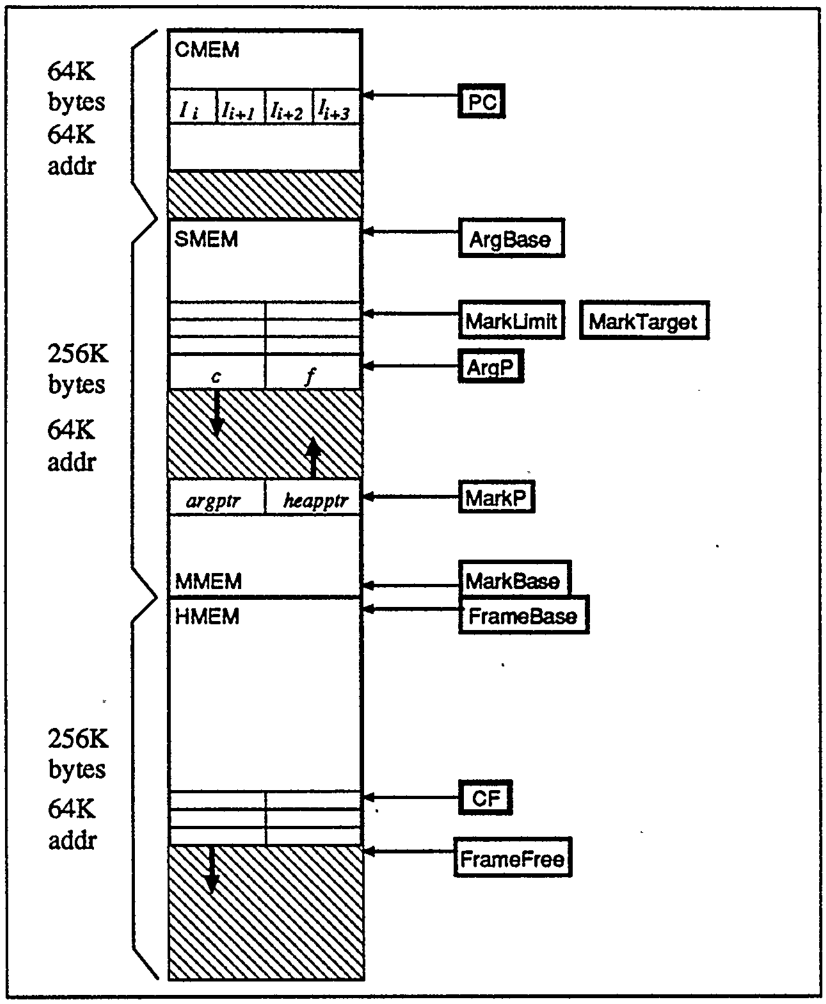


Figure 4.1: Basic Logical Memory Structure of TIM

bit of "blurring" between optimisations. These relations are treated as they arise, inevitably leading to some apparent redundancy between sections.

Throughout this and the next chapter, I have assumed that the *logical* memory partitions CMEM, HMEM, SMEM and MMEM have been assigned storage in a single *physical* memory as pictured in Figure 4.1. Each object is 32 bits wide, split into two 16 bit pointers to CMEM and HMEM. The code memory CMEM is 8 bits wide and contains 2^{16} (64K) addresses, while the frame heap HMEM is 32 bits wide and holds 2^{16} objects. The argument stack SMEM and mark stack MMEM are also both 32 bits wide, and their extent depends on the run-time placement of the stack bases.

Sec	Contents	Source	Comments
4.3	**An Alternative TAKE**	[FW87] (pg.41) [Arg89] (pg.106)	
	Stack Cache	Author	alternative solution
	ALLOC with status flag	Author	alternative implementation
4.4	**A Stack for Marks**	[FW87]	concept introduced
		[WF89] (pg.148)	concept of limit registers
		[Arg89] (pg.105)	concept mentioned
		Author	analysis and design
4.5	**Context Changes and Closures**	various	
4.6	**Conglomerate Frames**	[FW87], [WF89]	multiple suspensions held in one frame
		Author	Analysis, design, results
4.7	**Conglom. Frames and Tandem Mark Stack**	Author	Analysis, design, results

Table 4.2: Overview of Material

These object and memory widths were selected arbitrarily as the minimum needed for a practical implementation of TIM, and are easily expanded.

The contents of the remainder of this chapter and their attributions are summarised in Table 4.2.

4.3 An Alternative TAKE

Rationale This optimisation deals with "fully applied combinators", those which have all of their arguments pushed onto the stack, and are immediately applied. As defined in the literature [WF89, Arg89], the rationale is to avoid the use of the stack by pushing arguments directly into a pre-allocated frame. This is an excellent idea for avoiding the use of the stack for constructing argument lists and reducing the bandwidth demands on storage.

I can see a possible further justification in support of this optimisation, relating to the implementation of the Y operator in [FW87]. The first instruction is "TAKE 1 *and extend it to two*", which could be easily accomplished with a pre-allocated frame that can be filled in later. This avoids creating two new special flavours for the TAKE. Conversely, a largish microcode sequence for Y may be a faster and more compact solution, but is perhaps unjustified as Y is executed only once per function, and forms part of a one-time startup cost. In any event, the point is moot since it is not necessary to use Y at all in the TIM machine, as all the supercombinators have names, and the storage of interim results is automatic with the call to TAKE in each function head.

Implementation The proposed optimisation breaks the normal TAKE instruction into two pieces:

```
                 ALLOC n
                 take 1
TAKE n ⇒         :   *n   :
                 take 1
                 PTAKE
```

Figure 4.2: The reimplementation of TAKE

where the first instruction allocates a frame of size n, the last ("Preallocated TAKE") replaces the normal TAKE at the function head and simply causes CF to point to the new frame, and each singleton *take* represents one of the PUSH operations that would normally place its argument on the stack. This is in fact a new instruction (shown as HEAP PUSH in [Arg89], and PUSH ARG n INTOFRAME within the PONDER machine of [Fai86]). The compiler writer must recognise where fully applied combinators occur to substitute the ALLOC and PUSH instructions, and supply an alternate entry point to the function that invokes PTAKE to simply assume the preconstructed frame.[3]

Results In essence, as implemented in the literature this optimisation requires three new instructions be added to the repertoire, and compiler effort to utilise

[3][Arg89] overlooks the necessity of placing the PTAKE in the function body.

them. The result is a lessening of stack usage. [Arg89] states that the result is an approximate 50% saving in stack memory activity for fully applied combinators, but no results on execution speed or the effect on code memory were available. For those situations where the compiler applies the optimisation, using the instruction format of Section 5.4, Table 4.3 shows the instruction and memory cycles consumed. The notations "Xr" and "Xw" will be used frequently throughout this thesis. They represent memory read and memory write cycles respectively, where "X" may be any of C (code or CMEM), H (heap or HMEM), or S (stack or SMEM).

original	Cr	Hr	Hw	Sr	Sw	candidate	Cr	Hr	Hw	Sr	Sw
						ALLOC n	1	0	0	0	0
PUSH ARG a_n	2	1	0	0	1	HEAP PUSH a_1	2	1	1	0	0
⋮	”	”	”	”	”	⋮	”	”	”	”	”
PUSH ARG a_1	2	1	0	0	1	HEAP PUSH a_n	2	1	1	0	0
ENTER COMB f	3	0	0	0	0	ENTER COMB f	3	0	0	0	0
⋮						⋮					
f:TAKE n	1	0	n	n	0	f:PTAKE	1	0	0	0	0
n + 2 insts	2n+4	n	n	n	n	n + 3 insts	2n+5	n	n	0	0

Table 4.3: New TAKE and ALLOC costs

To judge the efficacy of this optimisation we need to determine the typical value of n, the frame size for fully applied combinators. Although time did not permit a determination of the frequency of fully applied frames in representative programs, some information can be gathered from just static analysis. Static analysis yields the result that the average frame size is 2 ($\overline{\text{size}} = 1.86$, $\sigma = 1.98$). For $n = 2$, this implies that on average four stack operations (2 read, 2 write) are saved in TAKE, while one more instruction cycle (including a CMEM access) are spent. Thus for most function closures, with no information about what proportion are fully applied, the added cost of three new instructions, and an extra entry point on each function in question, is not a profitable tradeoff compared with the number of PUSH instruction cycles saved. Note that we must have either a fourth new instruction to avoid clashing of entry points, or the inelegant fix of having PTAKE jump over whatever instruction follows.

The question which has been asked is "Does adding these new instructions help?", whereas the question should be "What is the best way to cut down memory activity preceding and during the *operation* of taking function arguments?" This task may be better much better served by a more utilitarian approach, that of using an internal stack cache to buffer arguments on-chip temporarily (see Section 6.2); this avoids new instructions completely, and serves all applications of the PUSH instruction. In any case, the optimisation as it stands above can be far better implemented using the following scheme, which uses a single new instruction and incurs less control unit complexity. ALLOC behaves as before, allocating a frame of appropriate size on the heap, *and* sets an internal status flag "HEAPDIRECT". While HEAPDIRECT is set, all PUSH instructions redirect their referents to the new heap frame, and the first call to TAKE will simply swap CF pointers and reset the flag. This is more efficient, easier to implement, and more transparent to the user.

4.4 A Stack for Marks

Rationale This idea has been suggested in the initial TIM paper [FW87], and each paper on TIM thereafter [WF89, Arg89]. To indicate that the result of a computation is shared amongst multiple contexts, a *mark* is inserted in the stack. The shared result will be a partial combinator application, and the mark delineates the partial application from regular stack parameters. On the first attempt to specialise the remaining parameters, the TAKE instruction of this combinator will detect the mark, and perform an update. Updating is performed as an *interrupt* process, creating a suspension frame which holds the stack arguments above the mark, and updating with the closure of this suspension and the current code. After completion of the interrupt, the current context resumes execution.

The following problems arise with processing of marks:

1. potential bandwidth saturation on the system stack and heap (CMEM and HMEM respectively),
2. quick checks for marks are impossible since they are mixed with arguments,
3. creating suspension frames quickly and efficiently is difficult when the TAKE must be interrupted.

These problems occur during the time-critical context change. We wish to accelerate the processing of marks, and achieve some measure of load-balancing to relax the demands placed on ArgP, CMEM and HMEM.

Implementation A proposed partial solution is to place marks in a separate stack of their own. This list of HMEM addresses (frame pointer + offset) would be combined with CMEM addresses that indicate mark placement amongst the arguments. The first choice for mark stack storage is in CMEM, at the end of the program code and growing upwards towards the system stack. CMEM would be addressed with two additional CPU registers MarkP and MarkBase, to hold the top and base of the mark stack. Possibly, the first mark(s) would be cached in the CPU, to save a CMEM read cycle or two, depending on the average number of marks that a TAKE must process. If this is unity, then caching only the first mark, or perhaps only the placement address is sufficient to give a speedup in most cases.

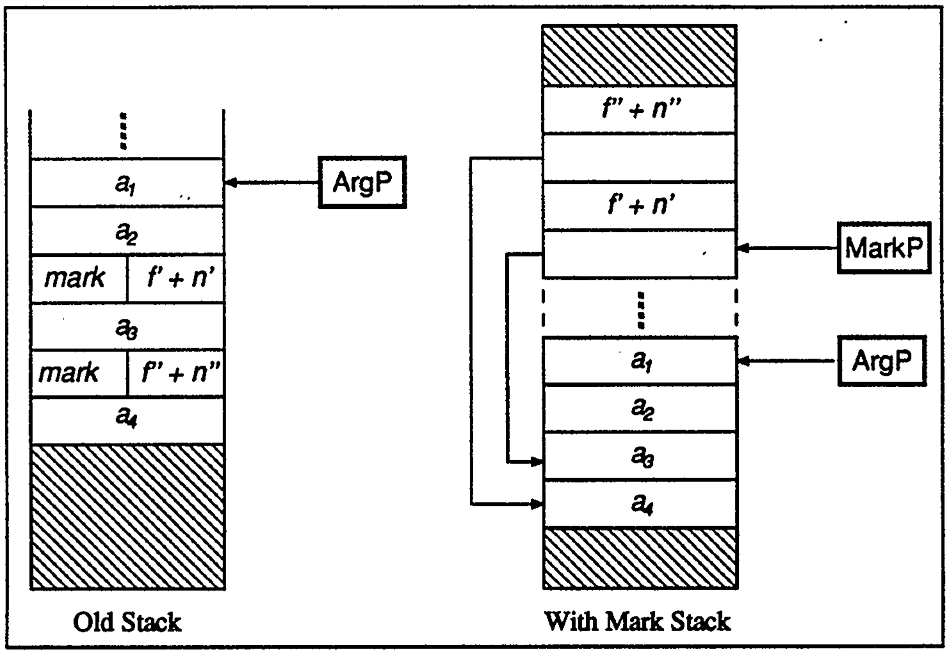


Figure 4.3: Mark Storage Methods

The utility of this optimisation depends on exactly how the suspension creations are micro. In any case, we will need to have a clear copy of the stack arguments from which to build suspensions and the TAKE frame; either we build the TAKE frame first, and construct suspensions from it as needed, or start building the TAKE frame and on encountering a mark make it a suspension frame, copying from this to restart the TAKE frame. The latter uses less overhead to keep track of things. However, both of these will use a lot of HMEM bandwidth for all the heap–heap copying. In the interests of load balancing, we would like to keep our clean copy of the arguments in CMEM, for copying to HMEM. This will be most helpful if HMEM and CMEM are physically separate memories (optimisation 6.2), but it is still useful in making the best use of the internal hardware (**ArgP**, **MarkP**, **MAR**, **MDR**) and simplifying microcode. Referring to Figure 4.3, keeping our clean copy in the old version of the stack is costly, as we have to keep reading over old marks. Trying to squeeze marks out of the arguments as we update them, or relocating the clean copy somewhere other than the heap (such as a CPU stack cache) would seem to be self-defeating in view of the average frame size and occurrence of marks. In the new version with a mark stack, repeated stack–heap copying seems obviously the way to go, starting with the smallest suspensions and working towards the complete TAKE frame.

There are the following advantages and disadvantages to this approach:

⊕ a tag bit is avoided. Mixed marks and arguments would have to be distinguished from each other. This saves address space available with each pointer, extra hardware, and micro to check the tags.

⊕ during TAKE, presense of marks in the local stack can be checked immediately, via a simple address comparison.

⊕ a simpler update algorithm, less organisation needed to keep things straight.

⊕ concurrent access of stacks and heap memory with dual physical memories. Otherwise, address generation can at least proceed concurrently.

⊖ the cost of additional CPU registers **MarkP** and **MarkBase**, the logic for address comparison, and microcode. Possibly also registers to keep the first mark **MarkTarget**, or at least its stack referent **MarkLimit** local to the CPU.

⊖ although the discussion suggests an update algorithm that will roughly balance the use of HMEM and CMEM, not much bandwidth is actually saved. We look to the use of conglomerate frames and accelerated TAKE instructions (optimisations 4.5, 4.6) to eliminate redundant use of HMEM. See Table 4.5 for a summary of the heap and stack memory cycles used when the mark stack optimisation is interrelated with use of conglomerate frames.

Results A number of simple experiments and data collection tasks are needed to determine the utility of this optimisation, and quantify its effect on performance:

1. Quantify with respect to average latency of the TAKE instruction across test suite.
2. Examine variants of update algorithm to find which best uses memory bandwidth.
3. Measurements to be made with and without the use of conglomerate frames (optimisation 4.6).
4. As to the caching of marks, find standard deviation for number of marks within individual TAKE instructions, and as a function of the argument to TAKE.
5. Also find the average separation of marks on the stack for each TAKE, and the ratio of TAKE instructions with marks to those without.

Cache size should be optimal for the average number of marks we can expect; cache cost will be optimal if we have prevalently singleton marks, and we can get away with storing only the current MarkLimit.

4.5 Context Changes and Closures

Rationale The TAKE optimisation reflects a misconception about what makes TIM a good architecture. TIM uses a processing model based on universal objects, and the unrecognised trade-off cost is a loss of processor homogeneity and greatly increased complexity in the instructions, arising from the problems of trying to fit everything into the object model. The solution to this problem is not to break instructions (and their component tasks) up into smaller units, but rather to simplify and reduce the number of tasks being performed, or in lieu of this, to reallocate tasks to machine instructions in a different way.

The vast majority of the work in TIM is performed immediately before, during and after function application, particularly in the TAKE instruction. In this section I will cover a number of ways to streamline the execution of the TAKE instruction.

Of course, the greatest cost of any memory operation is a memory access, whose latency stalls the processor. The task then is to reduce latency, or reduce memory accesses. In relation to the TAKE instruction, the first can be accomplished with parallel memory accesses or memory buffering. The application of these well-known and established techniques to TIM is covered below in Section 6.2 and Section 6.2. In this section I cover a couple of small tricks applied to the TAKE instruction. Although small, these optimisations are of a more practical and promising variety than that of Section 4.3, and provide a suitable introduction to the next section.

Implementation – common updates Two or more marks on the same stack address can be updated with the same $\boxed{c \mid f}$ suspension. This is since they reference the same frame and will wish to recover the same number of objects.

Implementation – initial TAKE (Due to [WF89]) Whereas with normal suspensions a sequence of push instructions (or micro transfers) is used to copy the suspension on top of the specialising arguments already in the stack, this is not necessary with the initial TAKE context. Regardless of the suspensions needed in the current TAKE, we can immediately pull all the arguments required (shared or otherwise) into the new frame. Marks are processed essentially independently, and the current context will be entered directly, using this frame, once the work is done.

Results For "initial TAKE", there is a minimal implementation cost involving changes to the microcode only. We save all the cycles needed to push the suspension contents back onto the stack, and re-execute the now unmarked TAKE. The first optimisation is even easier to implement in the microcode. In both cases, the optimisations are simple and effective, requiring only changes to the microcode. This is the key to their success: rather than change what the instruction is doing, the optimisations change *how they are doing it.*

4.6 Conglomerate Frames

Rationale (Due to [FW87], [WF89](pg. 149)) Multiple marks in succession can be treated optimally by creating a single update frame, and sharing this amongst all.

Implementation The previous optimisation which updated two or more redundant marks with the same suspension frame is the "base case" of this (less intuitive) optimisation. Instead of creating a new suspension frame for each marked stack location, we use a single suspension frame for all of them. In fact, we use the same frame as the current context. No confusion will arise, since marked arguments which have been updated will share successively larger slices of the same suspension frame, but cannot side-effect each other. Each updatee will only "copy out" (optimisation 5.1.2) as many objects from the suspension frame as each needs, on top of the specialising arguments already in the stack, and immediately TAKE the lot.

The following summarise the utility of this optimisation:

⊕ Heap usage is greatly reduced, and these HMEM access cycles are removed at the time-critical context change. Garbage collection is likewise more infrequent. Cost to implement the optimisation is essentially nil, requiring only changes to the microcode. A summary of the heap bandwidth savings appears below.

⊖ Conglomerate frames are at odds with another optimisation designed to save cycles at suspension restart (optimisation 5.1.2). Here, microcode uses the frame size held in the frame control word(FCW) to quickly reload the suspension (either to stack or new frame), avoiding the execution of a vector of push instructions. Obviously, this will not work unchanged when different sized suspensions coexist in the same frame.

Results The expected savings for an individual TAKE instruction are as follows; given the following definitions,

n	number of objects in subsequent TAKE
m_i	number of marks at the i^{th} object
$sgn^{-1}(x) = 1$ if $x > 0, 0$ otherwise	(inverse signum)

the cost of creating a suspension for each set of shared stack elements, measured as object write cycles to the heap, is:

$$\left(\sum_{i=1}^{n}(i+1)\cdot sgn^{-1}(m_i) + m_i\right) + n + 1$$

The sum is what each new suspension (i objects plus a frame status word) costs to build, including the update writes. The second term is the cost of the frame which we TAKE. Using the conglomerate frames gives us a cost of

$$\left(\sum_{i=1}^{n} m_i\right) + n + 1$$

cycles. In the worst case (every argument to the TAKE is marked), and factoring out the identical costs of redundant marks, we have:

original	optimized
$(\sum_{i=1}^{n}(i+1)\cdot sgn^{-1}(m_i) + m_i) + n + 1$	$(\sum_{i=1}^{n} m_i) + n + 1$
$(\sum_{i=1}^{n}(i+1)\cdot 1 + 1) + n + 1$	$(\sum_{i=1}^{n} 1) + n + 1$
$(\sum_{i=1}^{n} i) + 2n + n + 1$	$2n + 1$
$\frac{n(n+1)}{2} + 3n + 1$	$2n + 1$
$O(n^2)$	$O(n)$

Table 4.4: Worst-case heap consumption with/out conglomerate suspensions

Of course, we also save many read cycles of the heap and/or stack to copy from one suspension to another.

4.7 Conglomerate Frames and Tandem Mark Stack

Rationale The obvious usefulness of the two optimisations of a mark stack and conglomerate frames should complement each other, and provide increased efficiency.

Implementation The update mechanism is related to the method by which we store and manipulate marks. Table 4.5 summarises the memory cycle costs in CMEM and HMEM, for both read (R) and write (W) cycles. The two general update mechanisms "Heap-Heap" and "Stack-Heap" are those described along with the Mark Stack optimisation (Section 4.4), when using conglomerate frames there is only one ordering for performing updates. Note that the symbol m_i (as before) refers to a mark, which is the same size as an object (16 bits $f+n$, 16 bits CMEM address for stack limit). Thus marks, update targets, and regular arguments each take identical work for read or write cycles.

Update Mechanism		Original	Rank
Heap $\rightarrow$ Heap	W	$[\sum_{i=1}^{n}(2i$[a]$+1)\cdot sgn^{-1}(m_i)+m_i]+(n+1)$[b]	4th
	R	$(\sum_{i=1}^{n} m_i)+n$	
Stack $\rightarrow$ Heap	W	$[\sum_{i=1}^{n}(i+1)\cdot sgn^{-1}(m_i)+m_i]+(n+1)$[c]	5th
	R	$\sum_{i=1}^{n}\left(\sum_{j=1}^{i} m_j + i\cdot sgn^{-1}(m_j)\right)$	
Conglomerate Frames	W	$(\sum_{i=1}^{n} m_i)+n+1$	1st
	R	$\sum_{i=1}^{n} m_i+n$	
		Using Mark Stack	
Heap $\rightarrow$ Heap	W	$[\sum_{i=1}^{n}(2i+1)\cdot sgn^{-1}(m_i)+m_i]+(n+1)$	3rd
	R	$(\sum_{i=1}^{n} m_i)+n-1$[d]	
Stack $\rightarrow$ Heap	W	$[\sum_{i=1}^{n}(i+1)\cdot sgn^{-1}(m_i)+m_i]+(n+1)$	2nd
	R	$\sum_{i=1}^{n} i\cdot sgn^{-1}(m_i)$[e] $+\, m_i$[f]	
Conglomerate Frames	W	$\sum_{i=1}^{n} m_i+n+1$	1st
	R	$\sum_{i=1}^{n} m_i+n$	

[a] R/W cycles
[b] Sum of the stack-heap W cycles
[c] The TAKE
[d] Top mark is cached
[e] Reads for susp. frames
[f] MarkStack reads

Table 4.5: Memory Usage for Conglomerate Frames and Mark Stack

Results The conclusion from Table 4.4 seems to be that the presence of a Mark Stack is immaterial to the usefulness of conglomerate frames. Note the ranking for each update mechanism, which I have derived mainly from the cost equations but which include subjective judgement as to the efficiency and elegance of the mechanisms.

To verify the conclusions of this analysis, there is a complex experiment to perform. In addition, a useful exercise in safety would be functional simulation over the test suite, to verify that conglomerate suspensions do not side-effect each other, especially during garbage collection and further updating through child contexts. The real execution speed up can be measured in terms of the simple dynamic execution profile of our test suite, with which the cost equations can be scaled. This will not change the conclusion, only the strength of it.

4.8 Summary

In this chapter I have applied standard, well-understood and accepted arguments to a new architecture, to judge the relative usefulness of optimisations to TIM during the TAKE region of the instruction life-cycle. Building from optimisations suggested in the literature, I have provided either an alternative implementation(s) to each optimisation or an entirely new approach to accomplish the same task. I have also developed an implementation of two major optimisations, conglomerate frames and a mark stack, and analysed the behaviour of the two individually and together. I have provided a deeper analysis of optimisations than hitherto done by others, and over half of the material presented is new and original, in the form of additions, refinements and alternate approaches.

In all cases, my conclusions have been argued based on observation and subjective analysis. Most observation has taken the form of static program analysis, supported by some simulation to provide dynamic results. Since simulations are time-consuming both to design and run, it was not possible to fully cover all aspects of dynamic program behaviour. To provide statistically significant results requires huge numbers of simulations, both of my version of TIM and versions from the literature. Instead, I have settled for an indication that my arguments are correct, using a small assortment of typical programs. The results given here denote where future, more thorough work should be done.

If I were to select the "best buys" of these optimisations, I would avoid those optimisations such as splitting TAKE into multiple instructions, and concentrate my efforts on the underlying work. In almost all cases, this is the high bandwidth consumption. Rather than moving memory accesses around, it is better to reduce the numbers or shorten the extent of memory accesses. To this end, the most promising optimisation is the use of conglomerate frames, and simple optimisations to the *algorithm* of the TAKE instruction. The use of a mark stack is useful, but loses its cost vs. benefit tradeoff when implemented beside conglomerate frames.

Chapter 5

Secondary Optimisations

In this chapter, I discuss aspects of TIM at the machine-instruction and register-transfer levels. Roughly half of the material below takes the form of optimisations, differing from those in Chapter 4 only in their being further removed from the high-level TAKE instruction and context changes. Although these optimisations relate to high-level instructions, they cover low-level hardware representations of objects and implementation methods that in some instances require small changes at the instruction-level. The remainder of the material covers the implementation of TIM, including design of the instruction set format, structure and use of heap frames, and the treatment of some difficult uses of objects in the original version of TIM.

As in the previous chapter, optimisations are presented in the form of a **rationale, implementation** and **results.** In some instances, I have built on simpler optimisations previously suggested in the literature. Table 5.1 below summarises the chapter material, the majority (approximately 90%) of which is new. Of particular interest is the new heap frame model I have developed, and a new TIM instruction called RESTORE which I have designed. All of the elaborations on optimisations, new optimisations, design arguments and design specifications presented in this chapter are original work.

5.1 Frame Design

A good hardware implementation of frames is key to providing good performance in the TIM machine. In some situations, a relatively small change in the hardware at the register-level enables improvements to the way instruction-level tasks are performed. These methods are not always obvious from the instruction level of the machine, and

Sec	Contents	Source	Comments
5.1	**Frame Design**		Hardware support for frame heap and suspensions
5.1.1	**Frame Control Word**	Author	Store control/status info
5.1.2	**Self-Contained Closure Frames**		
	Suspension object stored in frame	[WF89] (pg.149)	Single generic code vector used to restart suspensions
	Combinator address stored in frame	Author	Quicker restart
	Frame status bit, new RESTORE instruction	Author	Dispense with code vector
5.2	**Marks and Updating**	Author	Improved mark handling
	"Degenerate" marks	[WF89]	Handle marks on stack top
	RESTORE method	Author	Fix flaw in [WF89]
5.3	**Objects and Direct Instructions**		Handling of special-purpose objects
5.3.1	ENTER ARG	Author	Direct mark placement
	Tagging methods	Author	Options and evaluation
5.3.2	SELF	Author	Deleting all usage
5.4	**Instruction Format Design**	Author	Constraints and treatment
	Relative Addressing	Author	Applied to PUSH LABEL
	Formats and Opcodes	Author	Instruction formats, opcodes, addressing modes

Table 5.1: Overview of Material

are most useful when they can be accomplished with changes only to the microcode and structures hidden from the user with no modifications to the instruction set necessary.

5.1.1 The Frame Control Word

Rationale The normal notion of a frame (Figure 5.1a) suffers from practical limitations. For instance, without some notion of the frame *size*, a garbage collection mechanism would not be able to properly deallocate the variable-size frames stored in the heap. This and other information pertaining to the low-level characteristics of the frame are useful to retain.

Implementation Each frame should then have an extra object-sized word associated with it to store such information as the frame size, and other status, control and garbage collection information which is outlined in following sections. This *frame control word* (FCW) will be the first object in the frame (Figure 5.1b).

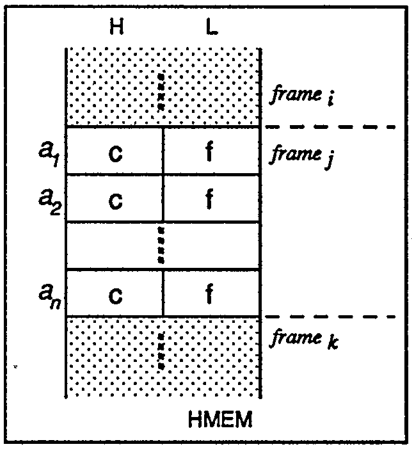


Figure 5.1a: Instruction-level frame structure

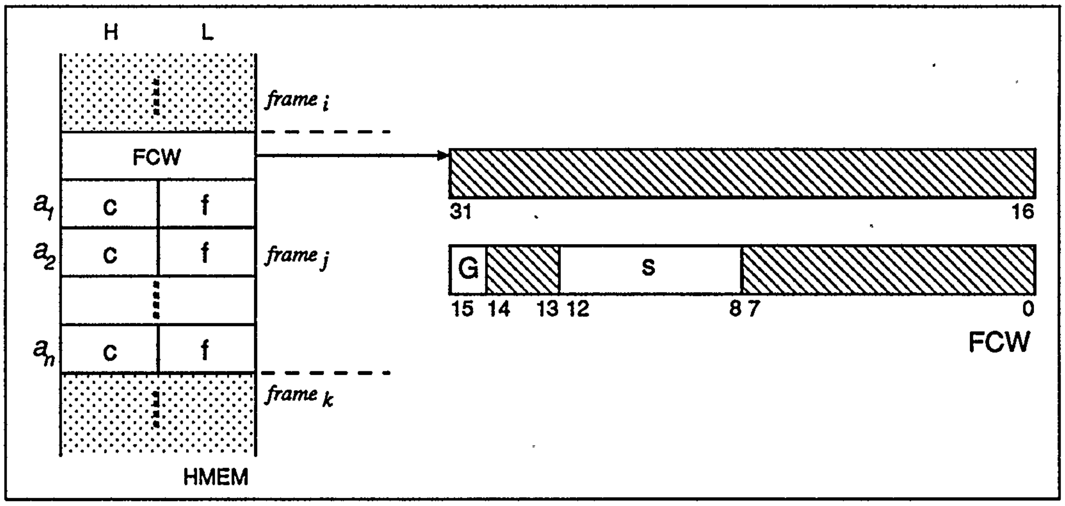


Figure 5.1b: Frame with FCW

Although an in-depth treatment of garbage collection is beyond the scope of this thesis, it is useful to mention the minimum that is required in the TIM machine by way of garbage collection bits. Fairbairn and Wray [FW87] mandate a bit pattern be stored beside each label (or rather *entry point*) in a TIM program, to specify which arguments of the current frame are used by each code fragment. Besides the GC mark bit (G) for the frame, a mark-collect scheme using these patterns would require a bit vector to specify which arguments of the frame are bereft of references, and which are active, for the collection phase. The FCW is the ideal place to store such a vector.

Results The cost of this optimisation is fairly high, considering that each frame holds an extra object and frames are typically small. However, the FCW is not accessed as frequently as are the other words of the frame, being used only for garbage collection and during other relatively infrequent operations. Furthermore, the storage costs of the FCW can be reasonably amortised over the benefits of the optimisation in the next section.

5.1.2 Self-Contained Closure Frames

Rationale When restarting suspensions, TIM relies on a sequence of PUSH ARG instructions prefixing the combinator entry, to restore the suspension frame contents to the stack on top of the arguments specialising the shared partial application. The use of a code vector for performing this task is particularly inefficient for several reasons:

1. the code vectors themselves consume memory space.
2. the PUSH ARG sequence must be executed for each re-awakened suspension.
3. the microcode for TAKE which processes marks must be able to generate CMEM addresses into the vector of PUSH instructions.

The literature has suggested two simple optimisations in this area, which I outline below. I modify the second literature optimisation to be more efficient, and then do away with PUSH ARG vectors altogether. This improved approach is made possible by the use of a FCW.

Implementation (1) The first problem is the expense of storing "restart vectors" in CMEM. Fairbairn and Wray [FW87, WF89] have suggested initially that rather than have a-vector of PUSH ARG instructions for each number of shared arguments for each combinator, TIM use instead a single vector of instructions for each combinator. For some combinator "C", the suspension and vector "P" are "created" by the TAKE instruction (Table 5.2).

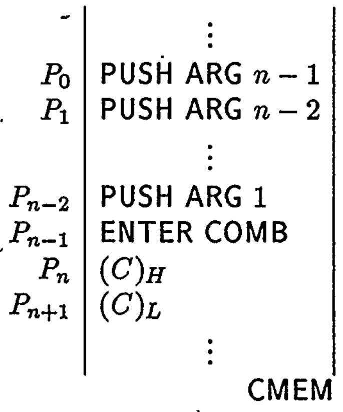
$$\boxed{\begin{array}{r}\left\{[\mathsf{TAKE}\ n;c], f_x, \left[a_1,...,a_i, \boxed{\boxed{f \mid m}}, A\right], F\left[f : \boxed{...,a_m,...}\right]\right\} \quad \text{for } 0 \le i < n \\ \Rightarrow \left\{P, f_y, A, F\left[f : \boxed{..., \boxed{P \mid f_y}, ...}, f_y : \boxed{a_1,...,a_i}\right]\right\} \\ \text{where } P = [\mathsf{PUSH\ ARG}\ i;\ldots;\mathsf{PUSH\ ARG}\ 1;\mathsf{TAKE}\ n;c]\end{array}}$$

Table 5.2: The Standard TAKE Instruction

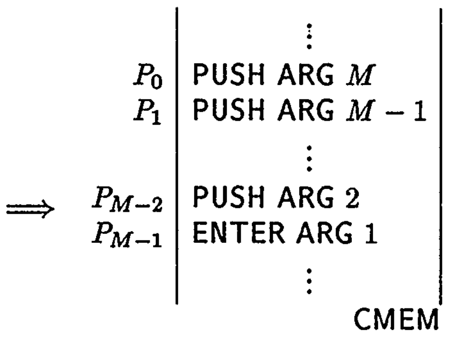
The symbol P will then be a reference into this vector, starting at the appropriate PUSH ARG instruction depending on the number of shared arguments. The vector takes the form of that in Figure 5.2a. For a combinator which takes n arguments, the maximum number of shared arguments requiring a suspension frame is $n - 1$. The vector is shown here as it would appear in CMEM starting at address P, where $(C)_H$ and $(C)_L$ indicate the high and low portions, respectively, of the address of the symbol C in CMEM. The vector consumes $(P_{n+1} - P_0 + 1 = n + 2)$ locations in CMEM. The microcode for TAKE generates the correct pointer P into the vector for installation in the updated object.

Wray [WF89] then suggests that a single vector can be used for all combinators. If each suspension stores as it's first argument the entry point of the shared combinator, the code vector in Figure 5.2b can be used to restart all suspensions. Here, M is the maximum frame size over all combinators in the code image, the first argument holds the combinator address, and the maximum number of shared arguments is $M - 1$. The literature then reduces storage use to a single vector of PUSH ARG instructions consuming $P_{M-1} - P_0 + 1 = M$ locations, and simplifies the microcode significantly by limiting address generation to a single well defined area of the program memory.

However, we can do better. For instance, a more elegant implementation is to call the entry point argument 0, and avoid the recompilation effort to slide all the arguments down one slot in the frame, using an ENTER ARG 0 to recover the context



(a): Old method



(b): Generic method

Figure 5.2: Suspension restart vectors

after restoring shared arguments to the stack. This misses the point, however, since the largest cost is not CMEM consumption, but the dynamic costs of executing these instructions at run-time.

Implementation (2, 3) The second and third problems are addressed by making further use of the frame control word. Previous approaches have not recognised that the frame pointer in the entry-point object is ignored, since the first task of the reawakened combinator is a TAKE. If only the code pointer is significant, and we have unused bits in our FCW, we store the entry point there. My reimplementation of the optimisation places the code pointer "I" in the upper half of the frame control word (only) for suspension frames, and adds a status bit called "S" which when set indicates that the frame is a suspension (Figure 5.3).

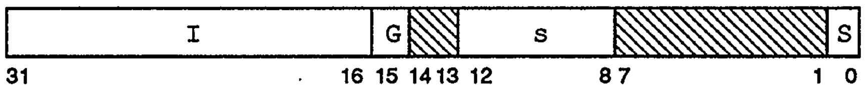


Figure 5.3: FCW for self-contained frames

This *self-contained closure* means we no longer need use a sequence of PUSH ARG instructions to put the suspension arguments on the stack. Upon executing an updated slot, the status bit S in the frame the object references is checked to see if the frame holds a suspension. The number of arguments in the frame, the arguments themselves, and the entry point of the suspended combinator are all available in the frame. A microcode subroutine can be called as an interrupt process, when the S flag is set.

This approach is very good, because we dispense completely with storing and *executing* the PUSH ARG code vector each time a suspension is restarted. Microcode generates the appropriate heap and stack addresses, and issues the memory cycle requests. Less important, but still more efficient and elegant, is that microcode is spared generating any CMEM instruction addresses whatsoever.

One problem remains, that of handling conglomerate frames. TIM must still be able to restart arbitrary portions of the suspension, ranging from no arguments up to the full suspension, without the use of the code vector to push the correct number of arguments to recreate the partial application. Notice that thus far, the contents of the code pointer in the updated object have been left unspecified, and we are not subject to any limitations in defining them to solve this problem. For s the size of the conglomerate suspension frame (number of arguments), and r the size of some suspension to be restarted ($0 \leq r \leq s$), there are a few possibilities for specifying r at runtime.

Place r in code pointer and rely on the suspension flag bit in the frame to initiate the restart. This is a poor solution, since it is a completely new use of the code pointer in objects, and would contort the decision process of the microcode. Any object being entered would have to have it's frame fetched first, so as to check the S bit is clear, before using the code pointer to fetch an instruction from CMEM (or after fetching it but before executing it).

Use a TAKE r instruction referenced by the code pointer. The TAKE would be modified to recognise when CF points to a suspension frame and modify its behaviour. Rather than retrieve arguments from the stack, it would get them from the current frame, and jump to the code pointer I in the FCW with the new (partially filled) frame. This would be good, in that it avoids use of the stack (see below) for restarting suspensions. However, this is yet another task for TAKE to perform. The TAKE instruction (TAKE n) at the top of the restarted combinator must not take it's arguments as normally, but recognise that it has been passed a partially-filled frame and take only the balance of the arguments needed (n - r). Using TAKE is a difficult and inelegant approach.

Create a new instruction for initiating suspension restarts, called RESTORE, which is used only by updated arguments pointing to conglomerate suspensions. For a restoration of r arguments from a conglomerate suspension frame f^*, the RESTORE and modified TAKE instructions appear in Figure 5.3. Here, the notation $\boxed{\boxed{\mathit{label}}, ...}$ denotes a conglomerate frame holding the CMEM address corresponding to the entry point *label*.

$$
\begin{array}{l}
\left\{[\mathsf{RESTORE}\ r], f^*, A, F\left[f^* : \boxed{\boxed{c} a_1, ..., a_s}\right]\right\} \quad \text{for } r \leq s \\
\qquad \Rightarrow \{c, f^*, [a_1, ..., a_r, A], F\} \\
\left\{[\mathsf{TAKE}\ n; c_0], f_x, \left[a_1, ..., a_i, \boxed{\boxed{f \mid m}}, A\right], F\left[f : \boxed{..., a_m, ...}\right]\right\} \quad \text{for } 0 \leq i < n \\
\qquad \Rightarrow \left\{\mathsf{RESTORE}\ i, f_y, A, F\left[\begin{array}{l} f : \boxed{..., \boxed{\mathsf{RESTORE}\ i \mid f_y{}^*}, ...} \\ f_y{}^* : \boxed{\boxed{c}\ a_1, ..., a_i} \end{array}\right]\right\} \\
\qquad \text{where } c \rightarrow [\mathsf{TAKE}\ n; c_0]
\end{array}
$$

Table 5.3: New RESTORE and TAKE instructions

Results The new instruction and frame structure, combined with the use of a frame control word streamline the restarting of suspensions. The capability of using a single suspension frame to represent multiple shared partial applications of the same combinator has been maintained, with the cost of a single instruction added to the instruction set. RESTORE simplifies the semantics of the TIM machine, while maintaining their original intent. The instruction itself is simple to implement in microcode, and has the same "boxed" format as other instructions (and is subject to the same optimisations, see Section 5.3).

There is room for further refinement of this optimisation, which minimizes the use of the stack given an appropriate memory structure (HMEM and CMEM physically independent, see Section 6.2). Identical in intent to the treatment of "fully applied combinators" in [FW87] and [Arg89] , this improvement requires that the FCW also hold the size n of the combinator frame. When executing RESTORE r, the microcode will directly allocate a new frame of size n, transfer r arguments from the suspension frame into it, and complete the frame with $n - r$ arguments from the stack. The entry point held in the FCW is made to point to the instruction following the TAKE n since it's job has been done.

5.2 Marks and Updating

Rationale The most critical cost-time locale in TIM is the context change — nearly all of the real work that goes on in the machine happens here. Inspection of static code indicates context changes occur roughly every 3-4 native machine instructions, with most of these being inexpensive continuations. The remainder involve TAKE, the more expensive and complicated variety. Apart from allocating a frame to hold the environment, marks for shared contexts must be updated, suspension frames created, garbage collection invoked, *etc.* The control unit and all logical partitions of memory are intensively used. The literature has suggested a number of inter-related optimisations of the mark updating process to save HMEM space and reduce memory bandwidth consumption during this crucial time. This section identifies them, and evaluates their use in the proposed implementation of TIM, with particular emphasis on relationship to frame design.

There are a handful of tricks to improve the processing of marks, creation of suspension frames, and the updating of share sites. This reduces to enforcing single sharing overheads for multiple recipients: we wish to stage only one marking task, rather than one for each recipient of the shared result. There are several ways to accomplish this:

1. through static analysis, encoding only the first evaluation with a PUSH ARG n instruction, leaving the rest as PUSH ARG UNS [FW87, WF89].

2. I suggest ENTER SHARED ARG check if the argument has been evaluated to

a machine value (SELF k). In this case the mark placement is aborted (and indeed the entry, see Section 5.3.2).

3. checking for marks on the top of the stack when changing contexts [FW87, WF89]. I call these "*degenerate*" marks, as they are updated with the new context itself prior to the context change.

4. A subset of this is when a SELF is being executed with a degenerate mark. I assume machine value object is written both to the stack and the mark referent prior to swapping contexts with any continuation on the stack.

All but (3) are unaffected by the design changes I have made, and the methods by which I have implemented other optimisations. The treatment of degenerate marks is flawed in the original application to TIM, but can be implemented with my *new* design for frames.

Implementation (3) "Degenerate" marks are those which require updating while they reference the first element on the stack. These occur in two special cases, both of which are handled the same way. Since no *new* suspension is required, the target of the update is overwritten directly with the code pointer and frame address of the current context $\boxed{\text{PC} \mid \text{CF}}$.

The simpler instance when the shared evaluation in question results in a single combinator C which is not a partial application. Since there are no shared arguments above the mark, it would be pointless to create a suspension frame. The microcode for TAKE can easily test for a mark on top of the stack, and update the mark target with the current context. In this case, that will be the combinator address (which location contains the current TAKE instruction) and the null frame (*ie.* $\boxed{\text{C} \mid 0}$).

The more complex instance occurs when we have recreated a suspension, only for TAKE to discover that there were no arguments above the mark. The suspension required by this mark is exactly what we began executing, and so when this situation is detected, the update target can be overwritten with the context defining the suspension. One obvious way this situation can arise is when the updated argument is reentered in the same context or some subsequent context to which it is parameter; a second shared ENTER ARG instruction places a mark before rebuilding the suspension. [WF89] talks of placing a test for marks at the top of the stack, before the regular suspension code, as in Figure 5.4.

```
                    ⋮
< test for marks and update if necessary >
PUSH ARG i
PUSH ARG i − 1
                    ⋮
PUSH ARG 1
ENTER COMB C
                    ⋮
```

Figure 5.4: Proposed test for degenerate marks

Results One problem with the solution as stated is that the exact form of the test for marks is not specified. If this is a new instruction, then each combinator would need a restart vector for *each number of arguments*, starting with the test. If it is some task installed in the microcode for the ENTER ARG instruction, then how do we detect that we are about to enter a suspension, and must test for marks at the top of the stack?

Happily, we have already dispensed with the PUSH ARG vector (see Section 5.1.2), allowing a workable implementation of a previously untenable optimisation. For TAKE, a simple test in the microcode is all that is required, as before. For suspension restarts, the old TIM was constrained to use such sequences of instructions to resume suspensions. However, with the new RESTORE instruction of Section 5.1.2, a simple microsubroutine addition to the microcode is all that is required, providing a simple and elegant implementation.

It is good that the cost of this implementation is low, as it may be that [WF89] overestimates the possibilities for this situation (*ie.* suspensions encounter degenerate marks only via their own ENTER). If this is the case, then a much better solution is to modify the microcode for the ENTER SHARED ARG instruction to test for suspensions and abort mark placement when they are discovered.

5.3 Objects and Direct Instructions

In this section I address those instructions which appear to be created at run-time, and appear to be stored within objects. These are $\boxed{\text{ENTER ARG} \mid f}$ (the lazy version) and $\boxed{\text{SELF} \mid k}$[1], produced by the TIM instructions in Table 5.4.

$\{[\text{PUSH ARG } i; C], f, A, F\}$	$\Rightarrow \left\{C, f, \left[\boxed{\text{ENTER ARG } i \mid f}, A\right], F\right\}$
$\{[\text{PUSH CONST } k; C], f, A, F\}$	$\Rightarrow \left\{C, f, \left[\boxed{\text{SELF} \mid \text{k}}, A\right], F\right\}$

Table 5.4: Sources of "Direct" Instructions

In previous implementations [FW86, FW87, WF89] these were assumed to be regular code memory addresses to instructions in the program image. I argue that these would be better left as *boxed* or *packed* instructions simply because of the frequency with which they are executed, and outline the best of several optional methods of implementation.

5.3.1 Reimplementing ENTER ARG instructions

When using PUSH to pass a shared context argument to another context, it is not the actual context argument that is pushed, but an object with an indirect reference to it. This indirection takes the form of an ENTER SHARED ARG instruction combined with the same frame specification. When this special object is later entered in some other context, the ENTER instruction inside will push a mark to indicate the argument is shared, and then enter the argument as usual. The indirection to a *shared* ENTER is necessary so that *both* the argument *and* its original location are available at the time the mark must be laid on the stack.

In previous implementations, the deferred ENTER instruction is stored in CMEM and the object references this instruction with the appropriate address. The compiler

[1]and possibly RESTORE n (see Section 5.1.2), although its inclusion is not necessary to this discussion.

generates all possible applications of deferred ENTER instructions, and places them in a reserved portion of the program object code image, against their potential use at run-time. There are a few problems with this implementation of the marking scheme:

- The microcode of the PUSH instruction must calculate the address of the correct deferred ENTER at runtime,
- Without a loadable microstore, all programs must have a certain portion of CMEM reserved for these instructions, imposing maximum limits on arguments and frame sizes,
- The full instruction cycle used and the CMEM storage used to store these instructions is wasteful, especially considering that every shared argument must use a deferred entry.

Implementation There are two ways we can go to address these problems, *(i)* we can try and adapt the hardware design to make the current method of marking more efficient, and *(ii)* we can rethink the implementation of marking.

To make marking more efficient *(i)*, the immediate approach to dealing with the indirect instruction is for microcode to accommodate a hashing algorithm that converts a PUSH ARG n instruction and the size of the referenced frame into a CMEM address. Depending on the program, less than 1K of memory in TIM is reserved for the ENTER instructions, and thus a particular argument n in a frame of m objects would be firm-wired to a particular memory location. The microcode to calculate the appropriate address would not be excessively complex, especially if the ENTER SHARED ARG instruction fits in a single byte (see Section 5.4), implying a 1–1 mapping of instructions to addresses. The lowest bytes in code memory could be filled with the ENTER instructions, and the address is directly calculated from the argument n and inserted into the pushed object.

However, if we are to rethink the marking scheme *(ii)*, the goal is to avoid the instruction cycle while limiting added complexity. There are two approaches to consider:

boxed we retain the use of the ENTER instruction, but as a boxed instruction within the object, or

"shared" tag we dispense with the indirection, pushing the original argument on the stack, and use some sort of signal to indicate that a mark should be deposited on the stack when we next encounter this object.

The shared tag option as stated is invalid, since it does not contain the argument *location* information needed to construct a mark. Any method that includes this information is functionally equivalent to the boxed instruction option, and may as well make use of the self-same instruction and its microcode. In any event, the shared tag method would require a respectable amount of recompilation effort that is not justifiable within the scope of this thesis, and changes to the original TIM laziness instructions that are regressive.

Settling on the boxed instruction option, it is obvious that a tag is required here as well, to distinguish the *direct* type of object from the normal code-pointer variety. Upon execution, the boxed option skips the CMEM instruction fetch, and decodes the contents of the code-pointer portion of the object directly, containing an ENTER, SELF or other instruction. The signal is given through the use of a tag bit within the object.

Potential tagging methods in TIM A system designer can consider a tag bit as one more bit that we could be using for addressing. Whether we add tags to the existing storage width, or we allocate them from within the existing word width, each additional tag has to be stored and represents a halving of the useful address space. For the sake of argument, I assume we allocate a bit "D" from the existing object format. Tag bit set indicates the code pointer portion of the object should be sent directly to the instruction register (IR) for decoding.

On the other hand, we can use a range of addresses in the code pointer (*ie.* `0xFFFX` = `0xFFF0`↔`0xFFFF`) to denote a direct instruction. The instruction fetch register (IFR) would perform a test for addresses in this range, and abort the instruction fetch when a match occurs. The problem here is that we need at least 8 bits for the instruction in question (ENTER SHARED ARG). The number of direct instructions (with arguments) we need consumes bits in the pointer in competition with the memory address consumption. For example, allocating the CMEM addresses in the range 0xFFE0↔0xFFFF will double the potential instruction codes to 32, and consume an additional 16 addresses of storage. Explicitly mapping a subset of the CMEM addresses to our small selection of special instructions is inefficient and expensive to test at run-time. Reallocating any portion of the 16 bit pointer for identifying bits indicates three possible approaches, none usable:

- the ID is a vector index meaning we have to go out to memory anyway for the instruction,
- the ID is a code for the instruction, meaning we must execute some form of costly hashing algorithm on it to yield the desired instruction, or
- the ID is an 8-bit instruction itself, which means that the instruction format and opcodes **must** use the same bit patterns as the addresses for the reserved region of memory. This wastes memory, and contorts the instruction format.

Returning to the use of tag bits, there are a few options for placing the tag, depending on the design of objects and memory. If the logical memory CMEM shares a physical memory with one or more logical memory partitions, then part of every code address is ignored. For instance, if we assume the physical memory is split between code in the lower half, and stack/heap in the upper half, then only code addresses in the range 0x0000↔0x7FFF are valid. Thus, the most significant bit of the code pointer in every object can be used as a tag with impunity (Figure 5.5a), which when set indicates the object is a direct instruction. The other way we can go is to use a bit from the frame address. The most obvious place is the least significant bit of the frame pointer. Considering that all heap frames are at least two objects in length (a FCW and one argument), using the lsb of frame addresses as a tag bit implies that all frames must begin on an even address boundary. Figure 5.5b illustrates the scheme. The disadvantage here is that the heap may suffer from significant external fragmentation if the majority of frames are of *odd* length.

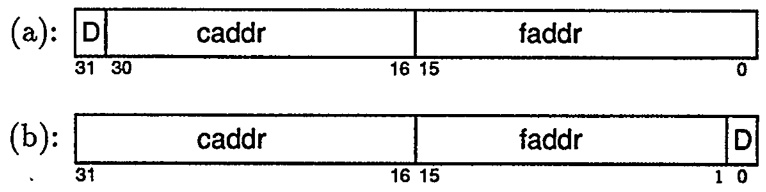


Figure 5.5: Options for tag bit placement

Results It is not clear whether the effort to eliminate extra instruction cycles for ENTER ARG is worth the benefit, when using tag bits. Performing a direct decode of the instruction address pointer as an interrupt process based on the pointer contents is a poor competitor, since it is inelegant and will cost almost as much as it saves. More importantly, it will not mesh well with the approach taken for the SELF instruction in the next section.

5.3.2 Eliminating the SELF Instruction

Rationale In TIM, the instruction SELF is used to represent and manipulate machine values, and its implementation is intended to maintain consistency with the universal object philosophy. There are a number of improvements to the way TIM handles SELF, which all involve avoiding having SELF appear in the instruction register by interpreting SELF not as an instruction but some variety of *object marker*. These optimisations indicate a need for a better representation of machine values in TIM since they all attempt to "design around" the current implementation. The central issue is to provide a consistent, efficient way to distinguish constants from regular objects. TIM will still hold values within objects, but a SELF *instruction* is not strictly necessary.

Implementation Machine values are encountered in only three ways: *(i)* as found on the stack as an argument to a machine operation, *(ii)* at the end of a machine operation, where it is necessary to change contexts to a closure on the stack, or *(iii)* as a frame argument which is entered. In each case, there is no need to explicitly execute a SELF, only to recognise the object as a machine value and perform the appropriate action.

One way to avoid executing SELF is to use strictness analysis. Normally, machine operations have supercombinator wrappers that ENTER each argument in insure the operation is applied to reduced machine values. [FW87] and [WF89] have stated that while strictness analysis generally applied does not yield significant improvements, the benefits are there for expressions over machine values that have been found to be strict. For these, only the arguments to the strict *expression* are entered and placed on the stack. More conventional code consisting of straight machine operators *sans* wrappers operates exclusively on the contents of the stack until the strict expression is reduced. This is a good optimisation, but is a compiler technology outside the scope of this work; I am concerned with those cases where we are not sure if the argument has been evaluated or not.

For *(ii)* above, the situation is very simple. All ground type operations currently evaluate their arguments, and place the result k in the PC and CF as $\boxed{\mathsf{SELF} \mid k}$. Afterwards, the SELF performs the a context change by swapping itself with (hopefully) the continuation on top of the stack. The optimisation is then to have all ground type operations perform the context swap themselves directly. The machine operation does a microcode jump to the segment for SELF, which is modified to locate the constant in the accumulator or alu, and to construct the SELF machine value object. Note that this scheme works whether the machine operations are being used strictly or not.

For *(i)* and *(iii)* above, the situation is the same, since machine operations enter their arguments just as any other combinator. The approach we should use is to have the microcode for ENTER check the argument it references after the fetch, to see if it is a machine value. If so, the context change is aborted and the object stored in a temporary register. Normally, the SELF would be decoded after the context change, and cause a swap of contexts with the top of the stack:

$$\boxed{\left\{\mathsf{SELF}, k, \left[\boxed{c_x \mid f_x}, A\right], F\right\} \Rightarrow \left\{c_x, f_x, \left[\boxed{\mathsf{SELF} \mid k}, A\right], F\right\}}$$

The interrupt process would perform the same task, placing the top element of the stack into the PC and CF, and overwriting it with the contents of the temporary register. In other words, ENTER ARG jumps to the microcode for SELF when it detects a machine value, again slightly modified as mentioned above.

Thus, the SELF formalism is retained only as an object label. The only problem remaining is how to "mark" an object as one holding a machine value. We can not use the best placement of the tag bit in Section 5.3.1, since this consumes a bit of the frame pointer, used here as precision for the value. So, we must resort to some manipulation of the CMEM code address. In the unadorned TIM, a convenient location for the SELF instruction would be CMEM[0x0000] = SELF, and all machine value objects would point to the first byte location in CMEM. To avoid the fetch, we can test for the address equal to 0x0000, indicating the object holds a machine value, and should be treated as shown above. It is very easy to do a "test for zero" in hardware very quickly, in parallel with the instruction fetch stage. This interpretation of the object format appears in Figure 5.6a.

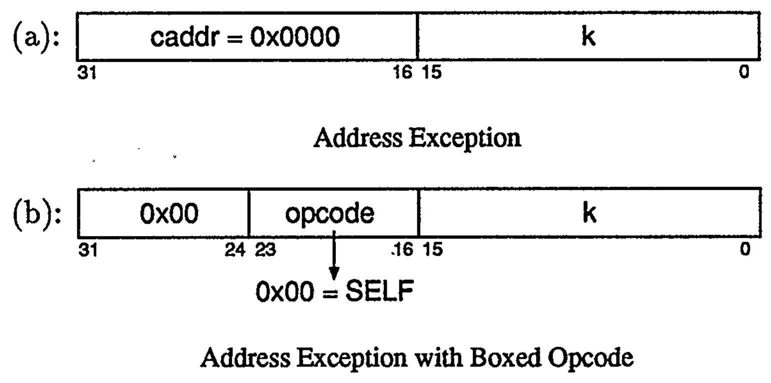


Figure 5.6: Forms of the machine value object (SELF)

For consistency, a possible additional change is to allocate the 8-bit opcode 0x00 to the SELF instruction; this leaves the microcode open to use the code pointer as an address, an identifier for machine values, or the actual instruction to be executed.

Results Using the scheme above, all interactions with machine values avoid not only an instruction cycle, but the decode of the "boxed" instruction as well.

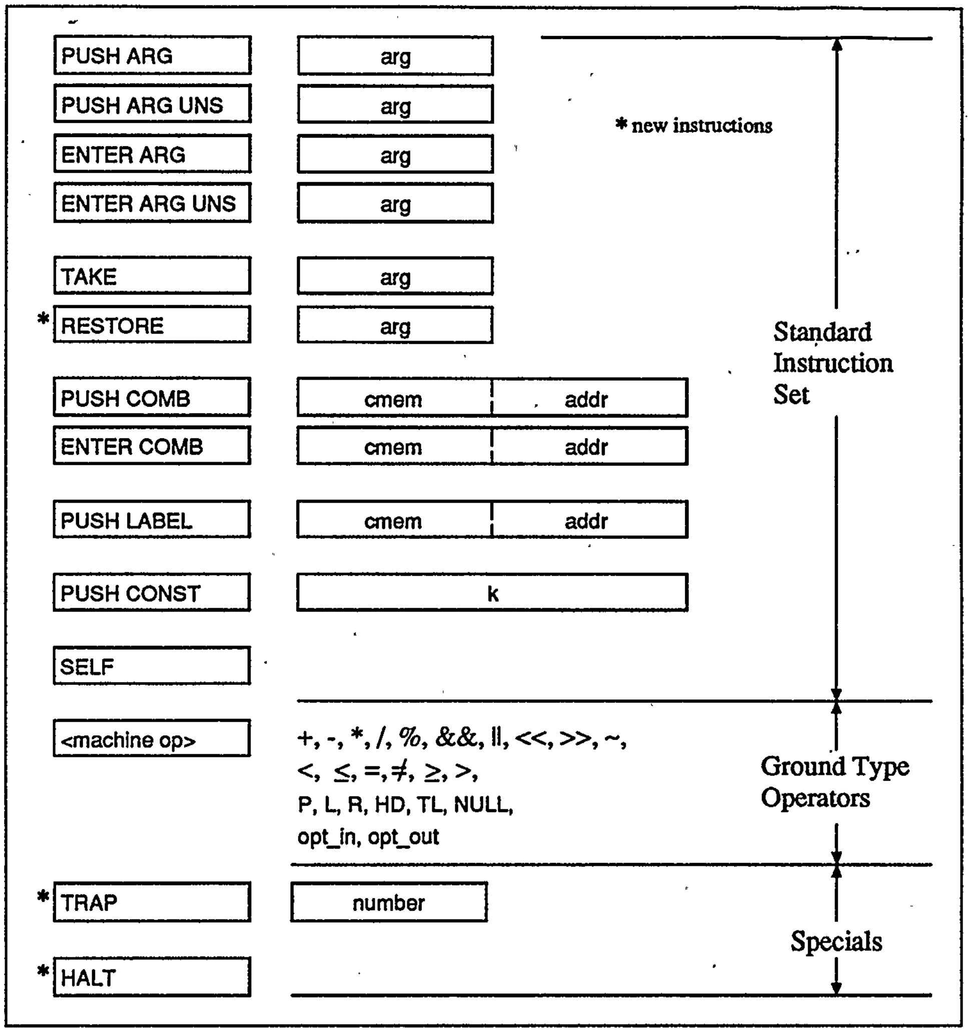


Figure 5.7: The TIM instructions

5.4 Instruction Format Design

This section will discuss the decisions I have made to minimize the size of the physical instructions in TIM, and optimise the mapping of instructions to opcodes. The overriding goal is to use as little CMEM space as possible to represent each instruction, while meeting the requirements of the machine as laid out by the optimisations. The original concept of TIM implies an instruction roughly the size of an object, with byte- or word-sized arguments. The logical appearance of the TIM instruction set is as in Figure 5.7.

The two initial constraints I have imposed is that the instruction opcode should fit in one byte, and that arguments should be placed within the byte if at all possible. This is not unreasonable, as there are only 37 instructions to be represented, and only a handful require a full-word argument. The following observations drove the design of the instruction formats:

- full words (16 bits) are required for combinator addresses and constants, as this is half-object size we are using.
- all of the argument/frame handling instructions should have the same *format* (PUSH ARG, ENTER ARG, TAKE and RESTORE).
- argument numbers for frame handling instructions do not need 16 or even 8 bits of storage. Static analysis of program code from the PONDER [Fai86] environment demonstrated that the maximum frame size ever used contained 24 arguments, and the average size was only 2. This indicates a very liberal number of arguments can be represented using $\lceil log_2 24 \rceil = 5$ bits.
- it would be advantageous if the opcodes can be assigned so that 5 bit arguments are reserved for these instructions within the 1 byte opcode.
- although PUSH LABEL appears to need a 16 bit full-word combinator address, the labels referenced in these instructions are spatially local to the instruction address (PC) and always at a higher address. A single byte would yield a PC-relative offset of 256 forward CMEM addresses, which is sufficient for all programs observed. A 5-bit offset stored within the instruction byte would give 32 forward CMEM references, which may be sufficient.

The complete list of instructions and opcodes appears below in Figure 5.8.

Following the same philosophy as that for Huffman coding, the design of the instruction set format should be tailored around the most frequently used instructions. In this way, the representation of instructions is allowed to emphasise those instructions which require advantages in shorter formats, reduced decode times, and simpler execution. I thus concentrated on the PUSH ARG, ENTER ARG, TAKE and

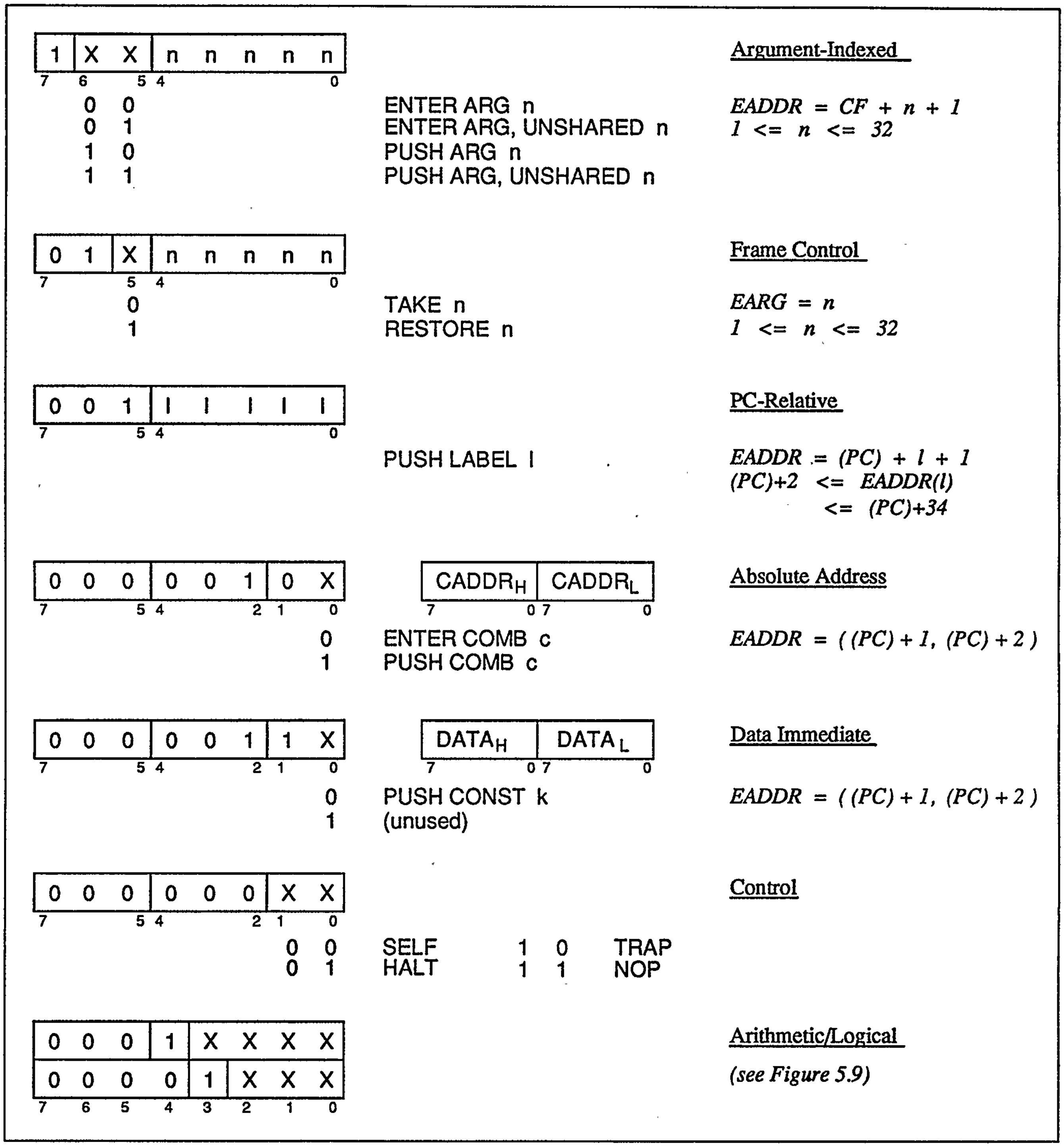
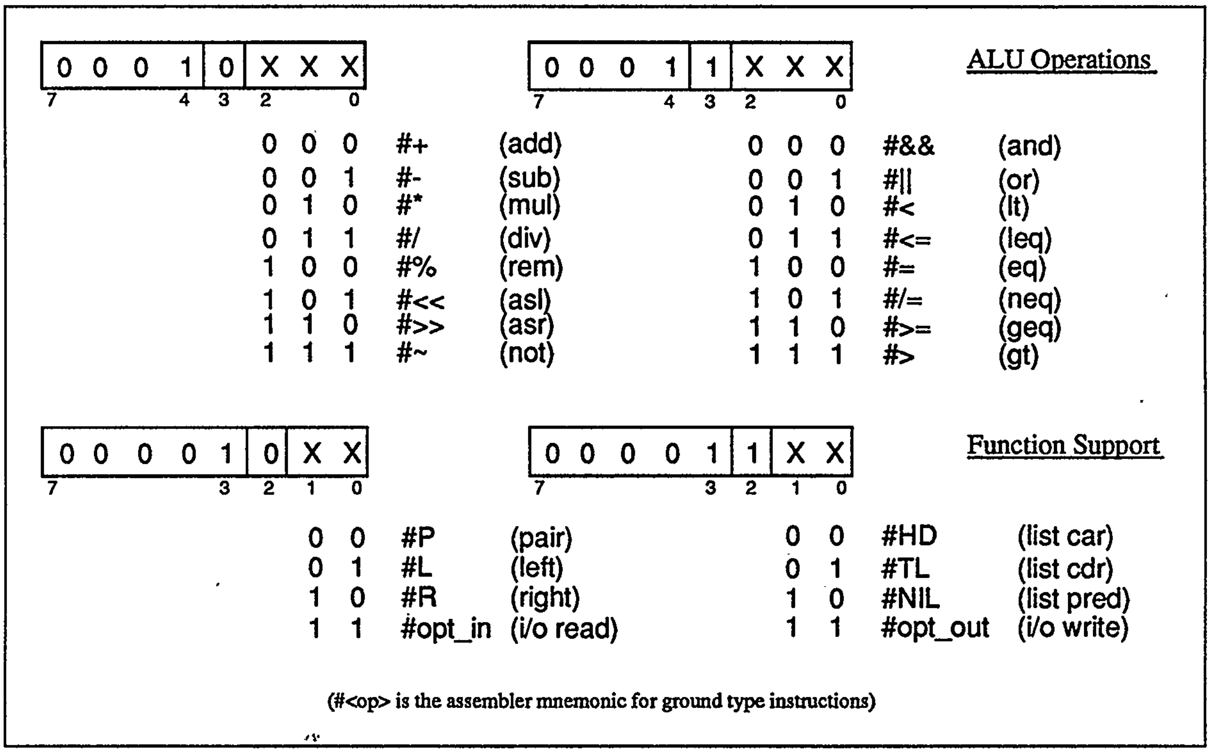


Figure 5.8: Main TIM instruction formats and opcodes

RESTORE instructions. Each of these can fit its argument specification within 5 bits, and together (with shared and unshared versions) there are 6 instructions. I separated the PUSH and ENTER argument instructions from the frame handling instructions. The first use the equivalent of register-indexed addressing mode, which I call *ARGUMENT INDEXED.* The second uses the argument as a frame size specification, which I class as *FRAME CONTROL.*

In the PUSH LABEL l instruction, the label is encoded as a 5 bit offset in an instruction classed *PC-RELATIVE.* Note that at run-time following the decode, the PC holds the CMEM address pointing after the instruction (PC + 1. Furthermore, since labels never reference the instruction immediately following the PUSH LABEL, and all labels are forward references, we can get one extra address of offset. Thus the labels which are encoded fall into the range $(\mathsf{PC})+2 \leq l \leq (\mathsf{PC})+34$. At run-time, the address which is pushed is thus $(\mathsf{PC})+\text{offset}\ +1$.



0 0 0	#+	(add)	0 0 0	#&&	(and)
0 0 1	#-	(sub)	0 0 1	#||	(or)
0 1 0	#*	(mul)	0 1 0	#<	(lt)
0 1 1	#/	(div)	0 1 1	#<=	(leq)
1 0 0	#%	(rem)	1 0 0	#=	(eq)
1 0 1	#<<	(asl)	1 0 1	#/=	(neq)
1 1 0	#>>	(asr)	1 1 0	#>=	(geq)
1 1 1	#~	(not)	1 1 1	#>	(gt)

0 0	#P	(pair)	0 0	#HD	(list car)
0 1	#L	(left)	0 1	#TL	(list cdr)
1 0	#R	(right)	1 0	#NIL	(list pred)
1 1	#opt_in	(i/o read)	1 1	#opt_out	(i/o write)

(#<op> is the assembler mnemonic for ground type instructions)

Figure 5.9: TIM ALU instruction opcodes

For the ENTER COMB and PUSH COMB instructions, a full word follows the instruction containing the absolute CMEM address of the combinator. The *ABSOLUTE ADDRESS* mode instructions fetch the address and change contexts immediately.

The PUSH CONST *k* instruction is called *DATA IMMEDIATE*, and behaves similarly to *ABSOLUTE ADDRESS*, fetching the full-word operand following the instruction.

The *CONTROL* instructions are SELF, HALT, TRAP and NOP, and consume a single byte, having no operands.

The *ALU* instructions consume the remaining 24 available opcodes, and are listed in Figure 5.9.

5.5 Summary

In this chapter I have suggested a number of optimisations to the instruction and register level of the TIM machine. I have created a new model for heap frames that allows a shared partial combinator application, or suspension, to be stored so that it can be restarted much more easily and quickly than in previous implementations. I have also introduced a new instruction called RESTORE to facilitate this optimisation, while maintaining the semantics and flavour of TIM. I have covered a number of small optimisations to the placement and processing of marks, to ensure these are still possible with the different designs specified in Chapter 4 and Chapter 5, and have found that in one instance ("degenerate" marks) the RESTORE instruction simplifies the optimisation. I have suggested a range of approaches to removing or reducing the effect of "direct" instructions, using tags or other markers to make the operations needed implicit to the object, and found it is not worth the effort, except with the SELF instruction. Finally, I have analysed the needs of the architecture for a physical instruction set and developed a set of compact instruction formats. A PC-relative mode for TIM combinator labels was introduced, and the opcodes allocated to the main, ALU, and special instructions of TIM were summarised.

Chapter 6

Summary and Conclusions

6.1 Contributions of the Thesis

In this thesis, I have provided some new approaches to the TIM implementation problem, and extended some of the existing work. The improvements and optimisations have covered a range of areas in the TIM architecture. Chapter 4 dealt with those improvements closely related to the construction and storage of heap frames, while the lower-level firmware, hardware and microcode optimisations were confined to Chapter 5. Chapter 5 also outlines a new heap frame model I have devised to simplify the usage of shared results, and a new TIM instruction called RESTORE to be used with the frame model. To test the optimisations and designs presented, an instrumented hardware simulation package called TIMSIM was developed at an early stage of the research (Appendices A, B).

Following the introduction, the next chapter (Chapter 2) provided an in-depth introduction to functional languages and architectures. The discussion of functional programming notations leading to SuperCombinators was to give the reader a clear understanding of the environment upon which TIM operates. Particular emphasis was placed on those elements of the functional architectures which are common with the Three Instruction Machine, presented in order to give the reader a notion of the "evolutionary path" in functional programming that has led to TIM.

A significant portion of the text was dedicated to a thorough description of the TIM abstract definition (Chapter 3), with emphasis on relating the TIM instructions and structures back to their higher-level functional language counterparts. In this way, the reader was to be given an intuitive understanding of the TIM abstract architecture, and the usefulness of the TIM approach.

The core of the thesis began with a discussion of the design philosophy I have applied to the TIM architecture, emphasising the importance of task optimisation and treatment of memory bandwidth demands. Chapter 4 was dedicated to analyzing three major and two minor optimisations to the context change and update mechanisms in TIM. Operating upon the major firmware structures of TIM, these optimisations have been suggested in the literature. My contribution has been extension of the previous work to specify a practical design, and analysis of the usefulness of the optimisations. I determined that efforts to break the TAKE instruction (Section 4.3) up are useful, but not as rewarding as simple changes to the way that TAKE operates (Section 4.5). An implementation of a stack for update marks was proposed, and analysis of its costs and savings (Section 4.4) performed. The use of a single frame conglomerate of shared combinator suspensions was discussed, and found to be a useful and essentially costless optimisation to the TIM machine. Lastly, I analysed the effects of implementing the two most promising optimisations in tandem, and determined that the effort is not warranted, as the presense of a mark stack is immaterial to the usefulness of conglomerate frames.

In Chapter 5, I concentrated on a number of original optimisations to the TIM abstract design, and specified design information for a number of areas of TIM. I have specified a new model for frames that includes status and control information (Section 5.1.1). The use of the new self-contained frame model facilitates a new optimisation to restarting shared combinator suspensions, which allows such suspensions to avoid over half of the previous memory accesses (Section 5.1.2). I have defined a new machine instruction called RESTORE to implement the new restart procedure, which is simple and easy to add to the TIM assembly code and to machine microcode. I verified that the new frame design and restart mechanism works with certain useful optimisations to mark creation and mark updating from the literature (Section 5.2). Certain usages of TIM instructions are inefficient and make inelegant use of microcode and TIM objects; I analyse several ways of reimplementing the ENTER ARG instruction as used in marking operations (Section 5.3.1), and the SELF instruction used to represent machine values and initiate context changes (Section 5.3.2). Finally, I present the design for a compact and efficient instruction format, including a new relative addressing mode, and several special instructions (Section 5.4).

6.2 Future Work

There are a large number of potential paths for further research on the TIM architecture. This thesis has suggested some "good bets" for optimisations and implementation details. A direct continuation of my thesis work would be to greatly extend and refine the simulations of TIM to gather statistically significant evidence to support or refute my conclusions with greater certainty. This would be followed by a first draft design of a VLSI chip and subsequent fabrication, so as to construct a real Three Instruction Machine system. The eventual goal is to benchmark TIM against other real functional architectures, providing empirical evidence of the merits and shortfalls of TIM.

Alternatively, the scope of this exploration could be widened to include memory design, advanced implementation techniques, and so forth. I have done some initial exploration in these areas, which is outlined below:

Storage Design The demands on storage in TIM mandate high memory speed and bus bandwidth, to achieve respectable performance. TAKE in particular makes intensive use of both stack and heap, but all operations involve accesses to two or more logical memory partitions.

[FW87, WF89] suggest splitting the heap from the stack, motivated by the suggestion of direct memory transfers during the TAKE operation. There are a number of additional reasons to assign separate physical memories to the logical memory partitions that I have used. The fact that heap storage is an allocatable heap, while all other memories are simple storage, means support for garbage collection is wasted the partitions reside in common storage. If interleaving and caching are applied to TIM, each partition may require different cache line widths and bandwidth. The initial implementation assumes that all logical partitions exist in one physical memory, as in Figure 4.1. The next option is to isolate the heap HMEM partition from others, and the most promising (and expensive) arrangement is to assign a separate physical partition for code, heap, and stacks.

Interrupt Processing Because TIM will operate in the real world, it must be able to deal with real-time events and exceptions. Against this potentiality I have done the initial work to develop an interrupt-processing mechanism. This would be used for input/output operations (through the opt-in and opt-out supercombinators used in code from the PONDER environment [Fai86]), control unit exceptions, direct memory↔memory transfers as described in this section, and any other interface applications. The typical implementation would have traps hardwired to a jump vector loaded with trap handler entry points.

Advanced Implementation Techniques Hardware accelerators and bandwidth-enhancing memory design techniques can be applied to TIM as well as any other architecture, with the observation that the common understanding of bottlenecks will likely not apply. All of the following techniques will require deeper modeling of TIM to obtain information to tailor the application of these techniques to TIM. The required information at a minimum includes extensive measurements of the spatial and temporal locality of each type of memory reference, knowledge of the size and lifetime of supercombinator and suspension contexts, and a number of other dynamic execution parameters. The techniques I suggest are listed starting with the least complex and most promising.

Instruction Prefetch With a bus 4 times the width of the basic instruction unit, it would make sense to fetch all 4 consecutive bytes of memory and hold them within the CPU, to speed instruction fetches. To simplify the microcode control of the fetching, the buffer could be made 8 bytes long (or twice the bus width), with valid program code being read through each half alternately.

Frame Buffer Frame heap memory write accesses exhibit a higher degree of spatial and temporal locality than heap reads, due to TAKE and suspension frame creations. Subsequent read accesses (excluding suspension restarts) are more spread out. This observation leads to the potentially useful addition of a frame buffer, not to be confused with a frame cache, but which is an interim approach.

Two benefits are derived: *(i)* frame creations occur in the frame buffer first, and the actual heap write cycles can be spread out over subsequent instructions which do not make use of the heap, *(ii)* reads of the current context arguments do not have to go to heap memory, useful when one considers that arguments are accessed immediately after creating a new current frame. A change to a new context would flush any remaining portion of the frame buffer as an interrupt process.

Stack Buffer As with the last modification, this reduces bandwidth consumption by retaining some number of the top elements of the stack in the CPU. This requires heavy simulation to determine the appropriate size of the buffer and quantify it's benefits, and some complexity is involved in optimistically fetching into and writing out the buffer when it is necessary. This modification would be of benefit to most machine instructions, and certainly to speeding up argument list stacking and subsequent TAKE's. A stack buffer would form an interim approach prior to using smart memory controllers and direct memory↔memory transfers as suggested in [FW87].

Frame Heap Cache Analogous to multiple register sets in RISC [Pat85, Tab87] technology, the frame buffer modification is extended to a full caching scheme, so that contexts may be retained for reading and writing within the CPU. I suspect that 4-8 cache slots would be mandated by simulations, but that the benefits derived would not be much greater than with a simpler 1-2 slot frame buffer, as above.

In addition memory interleaving would make more bandwidth available by speeding up all of the high-spatial locality operations, particularly when used in conjunction with the buffering schemes suggested above.

6.3 Final Comments

I have proposed a design for TIM encompassing the essential elements of a practical implementation. The design techniques and hardware tools that I have applied to

TIM are well known and understood, derived from the vast body of knowledge arising from the study and practice of conventional computer architecture. TIM must compete not only with other functional machines, but also with the established conventional technologies (as do functional languages). It may be that abstract functional machines are ultimately best implemented by programming their operations into conventional high speed microprocessors, to make use of the knowledge and economy of scale advantages in an established industry.

For the present time, TIM is a promising architecture for the efficient execution of functional languages. This practical design combined with realistic performance goals should provide a simple and fast implementation of TIM, when fabricated as a concrete machine in VLSI silicon.

Bibliography

[Abd74] Syed Kamal Abdali. *A Combinatory Logic Model of Programming Languages.* PhD thesis, Computer Science, University of Wisconsin, April 1974.

[Abd76] S. K. Abdali. An abstraction algorithm for combinatory logic. *Journal of Symbolic Logic*, 41(1):222–224, March 1976.

[AJ89] Lennart Augustsson and Thomas Johnsson. Parallel graph reduction with the $\langle \nu, G \rangle$-machine. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 202–213, Imperial College, London, September 1989. ACM, Addison-Wesley.

[AP81] L. Aiello and G. Prini. An efficient interpreter for the lambda-calculus. *Journal of Computer and System Sciences*, 23:383–424, 1981.

[Arg88] Guy Argo. The G-TIM: a refined Three Instruction Machine. In Thomas Johnsson et al., editors, *Proceedings of the Workshop on Implementation of Lazy Functional Languages*, number 53 in Programming Methodology Group Technical Reports, Aspenås, Sweden, September 5-8 1988. Chalmers University of Technology and University of Göteborg.

[Arg89] Guy Argo. Improving the Three Instruction Machine. In *Proceedings of the Fourth Conference on Functional Programming and Computer Architecture*, pages 100–115, Imperial College, London, September 11–13 1989. ACM.

[Arm89] James R. Armstrong. *Chip-Level Modeling with VHDL.* Prentice-Hall, Englewood Cliffs NJ, 1989.

[Aug88] Lennart Augustsson. The ν-G-machine. In Thomas Johnsson et al., editors, *Proceedings of the Workshop on Implementation of Lazy Functional Languages*, number 53 in Programming Methodology Group Technical Reports, Aspenås, Sweden, September 5-8 1988. Chalmers University of Technology and University of Göteborg.

[Bac78] J. W. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[Bar81a] Mario R. Barbacci. Instruction Set Processor Specifications (ISPS): The notation and its applications. *IEEE Trans. on Computers*, C–30(1):24–40, January 1981.

[Bar81b] H. P. Barendregt. *The Lambda Calculus - Its Syntax and Semantics.* North-Holland, Amsterdam, 1981.

[Bar82] Mario R. Barbacci. An introduction to ISPS. In Daniel P. Siewiorek, C. Gordon Bell, and Allen Newell, editors, *Computer Structures: Principles and Examples*, chapter 4, pages 23–38. McGraw-Hill, New York, 1982.

[BdHP86] Alan Bundy, Den du Boulay, Jim Howe, and Gordon Plotkin. The researchers' bible. DAI Teaching Paper 4, University of Edinburgh, September 1986.

[BGJ$^+$89] G. Birtwistle, B. Graham, J. Joyce, S. Williams, M. Brinsmead, M. Keefe, W. Kroeker, B. Liblong, and W. Vollmerhaus. The SECD Machine on a Chip. In *Int. Conf. on CAD and CG*, Beijing, 1989.

[Bir79] G. M. Birtwistle. *Discrete Event Modelling on Simula.* Macmillan Press, London, 1979.

[BMS80] R. M. Burstall, D. B. MacQueen, and D. T. Sannella. HOPE: An experimental applicative language. In *LISP Conference Record*, pages 136–143, Stanford CA, 1980. Stanford University.

[BPR88] Geoffrey L. Burn, Simon L. Peyton Jones, and J. D. Robson. The spineless G-machine. In *Proceedings of the Conference on LISP and Functional Programming*, Snowbird, Utah, July 25-27 1988. ACM.

[Bur88] Geoffrey L. Burn. A shared memory parallel G-machine based on the evaluation transformer model of computation. In Thomas Johnsson et al., editors, *Proceedings of the Workshop on Implementation of Lazy Functional Languages*, number 53 in Programming Methodology Group Technical Reports, Aspenås, Sweden, September 5-8 1988. Chalmers University of Technology and University of Göteborg.

[BW88] Richard Bird and Phillip Wadler. *Introduction to Functional Programming.* Series in Computer Science (C. A. R. Hoare ed.). Prentice-Hall, London, 1988.

[Car83] Luca Cardelli. The Functional Abstract Machine. Technical Report TR-107, AT&T Bell Laboratories, Murray Hill NJ 07974, 1983.

[Car84] Luca Cardelli. Compiling a functional language. In *Proceedings of the Conference on LISP and Functional Programming*, pages 208–217, Austin, Texas, August 6-8 1984. ACM.

[CCM87] G. Cousineau, Pierre-Louis Curien, and Michel Mauny. The Categorical Abstract Machine. *Science of Computer Programming*, 8:173–202, 1987.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic*, volume I. North-Holland, Amsterdam, 1958.

[CGMN80] T. J. W. Clarke, P. J. S. Gladstone, C. D. MacLean, and A. C. Norman. SKIM — the S,K,I Reduction Machine. In *LISP Conference Records*, pages 128–135, Stanford, CA, 1980. Stanford University.

[Che84] Marina C. Chen. A methodology for hierarchical simulation of VLSI systems. Research Report YALEU/DCS/RR-325, Yale University, Computer Science Department, August 1984.

[CHS72] H. B. Curry, J. R. Hindley, and J. P. Seldin. *Combinatory Logic*, volume II. North-Holland, Amsterdam, 1972.

[Chu36] A. Church. An unsolvable problem of elementary number theory. *Amer. J. Math.*, 58:345–363, 1936.

[Chu41] A. Church. The calculi of lambda-conversion. In *Annals of Mathematics Studies*, volume 6. Princeton University Press, NJ, 1941.

[CJH89] Shiu-Kai Chin, Damir A. Jamsek, and Paul R. Humenn. 5th generation logic programming architectures. 24 month report, CASE Center, Syracuse University, Syracuse NY 13244, 1989.

[Coe89] David Coelho. *The VHDL Handbook.* Kluwer Academic Publishers, Norwell MA, 1989.

[CR36] A. Church and J. B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39:472–482, 1936.

[Cur29] H. B. Curry. An analysis of logical substitution. *Amer. J. Math.*, 51:363–384, 1929.

[DeB72] N. DeBruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Math.*, 34:381–392, 1972.

[Fai86] Jon Fairbairn. Making form follow function — an exercise in functional programming style. Technical Report 89, University of Cambridge Computer Laboratory, Corn Exchange Street, Cambridge, England CB2 3QG, June 1986.

[FC88] Richard M. Fujimoto and William B. Campbell. Efficient instruction level simulation of computers. *Transactions of the Society for Computer Simulation*, 5(2):109–124, April 1988.

[FH88] Anthony J. Field and Peter G. Harrison. *Functional Programming.* International Computer Science Series. Addison-Wesley, 1988.

[FW86] J. Fairbairn and S. C. Wray. Code generation techniques for functional languages. In *Proceedings of the Conference on LISP and Functional Programming*, pages 94–104, Cambridge, MA, August 1986. ACM.

[FW87] Jon Fairbairn and Stuart C. Wray. TIM: A simple, lazy abstract machine to execute supercombinators. In *Proceedings of the Conference on Functional Programming and Computer Architecture*, number 274 in Lecture Notes in Computer Science, pages 34–45, Portland, OR, September 1987. Springer-Verlag.

[Geo89] Lal George. An abstract machine for parallel graph reduction. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 214–229, Imperial College, London, September 1989. ACM, Addison-Wesley.

[Gor88] Michael J. C. Gordon. *Programming Language Theory and its Implementation.* Series in Computer Science (C. A. R. Hoare ed.). Prentice-Hall, London, 1988.

[GWB+89] B. Graham, S. Williams, G. Birtwistle, J. Joyce, and B. Liblong. The CDL/Mossim for Henderson's SECD machine. Research Report 89/341/03, Computer Science Department, University of Calgary, 1989.

[Har86] Robert Harper. Introduction to Standard ML. LFCS Report Series ECS-LFCS-86-14, Department of Computer Science, University of Edinburgh, November 1986.

[HB84] Kai Hwang and Fayé A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, New York, 1984.

[HBGS89] M. J. Hermann, G. Birtwistle, B. Graham, and T. Simpson. The Architecture of Henderson's SECD Machine. Research Report 89/340/02, Computer Science Department, University of Calgary, Alberta, Canada T2N-1N4, January 1989. Prepared under Contract No. W2213-8-6362/01-SS with the Department of National Defence, 78 pages.

[Hen80] P. Henderson. *Functional Programming – Applications and Implementation.* Prentice–Hall, London, 1980.

[Her87] Mike Hermann. An Exploration of Functional Architectures. Undergraduate research project report, Computer Science Department, University of Calgary, Alberta, Canada T2N 1N4, April 8 1987.

[HJJ83a] P. Henderson, G. A. Jones, and S. B. Jones. The LispKit manual, volume 1. Technical Monograph PRG-32(1), Oxford University Computing Laboratory, 1983. See also [HJJ83b].

[HJJ83b] P. Henderson, G. A. Jones, and S. B. Jones. The LispKit manual, volume 2. Technical Monograph PRG-32(2), Oxford University Computing Laboratory, 1983.

[HLS72] J. R. Hindley, B. Lercher, and J. P. Seldin. *Introduction to Combinatory Logic.* Cambridge University Press, London, 1972.

[HMM86] Robert Harper, David MacQueen, and Robin Milner. Standard ML. LFCS Report Series ECS-LFCS-86-2, Department of Computer Science, University of Edinburgh, March 1986.

[HS86] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and the λ-calculus.* Cambridge University Press, London, 1986.

[Hud89] Paul Hudak. Conception, evolution and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[Hug84] R. J. M. Hughes. Why functional programming matters. Technical Report 16, Programming Methodology Group, Dept. of Computer Science, University of Göteborg, S-412 96 Göteborg, Sweden, November 1984.

[Hug89] R. J. M. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

[JGCH89] Damir Jamsek, Kevin J. Greene, Shiu-Kai Chin, and Paul R. Humenn. WINTER: WAMs in TIM expression reduction. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logical Programming*, volume 2, pages 1013–1029. MIT Press, 1989.

[Joh84] Thomas Johnsson. Efficient compilation of lazy evaluation. *ACM SIGPLAN Notices*, 19(6):58–69, June 1984.

[Joh87] Thomas Johnsson. *Compiling Lazy Functional Languages.* PhD thesis, Department of Computer Sciences, Chalmers University of Technology, Göteborg, Sweden, 1987.

[Joy84] M. S. Joy. *On the Efficient Implementation of Combinators.* PhD thesis, University of East Anglia, September 1984.

[Ken82] J. R. Kennaway. A new combinator set. Technical report, University of East Anglia, 1982.

[Kie85] Richard B. Kieburtz. The G-machine: A fast, graph-reduction evaluator. In *Conference on Functional Programming Language and Computer Architecture*, pages 400–413, Nancy, France, 1985. IFIP.

[Kle81] Stephen C. Kleene. Origins of recursive function theory. *Annals of the History of Computing*, 3(1):52–67, January 1981.

[Knu86] Donald E. Knuth. *The TEXbook.* Addison-Wesley, Reading, MA, 1986.

[Lam86] Leslie Lamport. LATEX*: A Document Preparation System.* Addison-Wesley, Reading, MA, 1986.

[Lan64] Peter J. Landin. The mechanical evaluation of languages. *Computer Journal*, 6(4):308–320, January 1964.

[Lan65a] P. J. Landin. The correspondence between ALGOL60 and Church's Lambda Calculus, Part 1. *Communications of the ACM*, 8(2):89–101, February 1965.

[Lan65b] P. J. Landin. The correspondence between ALGOL60 and Church's Lambda Calculus, Part 2. *Communications of the ACM*, 8(3):158–165, March 1965.

[Leh85] Axel Lehmann. Hybrid and hierarchical simulation of computer systems. In A. Jávor, editor, *Simulation in Research and Development*, pages 217–221. North-Holland, Amsterdam, 1985.

[Mos75] Peter D. Mosses. *Mathematical Semantics and Compiler Generation.* PhD thesis, University of Oxford, 1975.

[Mye78] G. J. Myers. Storage concepts in Software-Reliability-Directed computer architecture. In *5th Annual Symposium on Computer Architecture*, pages 107–113. ACM SIGARCH, 1978.

[Pat85] D. A. Patterson. Reduced-Instruction Set Computers. *Comm. ACM*, pages 8–21, January 1985.

[Pau87] Lawrence C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF.* Number 2 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1987.

[PCSH87] Simon L. Peyton Jones, Chris Clack, Jon Salkild, and Mark Hardie. GRIP — a high performance architecture for parallel graph reduction. In *Proceedings of the Conference on Functional Programming and Computer Architecture*, number 274 in Lecture Notes in Computer Science, pages 98–112, Portland, OR, September 1987. Springer-Verlag.

[Pet70] G. W. Petznick. *Combinatory Programming.* PhD thesis, University of Wisconsin, 1970.

[Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Int'l Series in Computer Science. Prentice-Hall, London, 1987.

[Pey88] Simon L. Peyton Jones. The spineless tagless G-machine. In Thomas Johnsson et al., editors, *Proceedings of the Workshop on Implementation of Lazy Functional Languages*, number 53 in Programming Methodology Group Technical Reports, pages 145–156, Aspenås, Sweden, September 5-8 1988. Chalmers University of Technology and University of Göteborg.

[Poo87] R. J. Pooley. *An Introduction to Programming in SIMULA.* Computer Science Texts. Blackwell Scientific Publications, Oxford, 1987.

[Ram86] John D. Ramsdell. The CURRY chip. In *Proceedings of the Conference on LISP and Functional Programming*, pages 122–131, Cambridge, MA, August 1986. ACM.

[Ros84] J. Barkley Rosser. Highlights of the history of the lambda-calculus. *Annals of the History of Computing*, 6(4):337–349, October 1984.

[SBGH89] T. Simpson, G. Birtwistle, B. Graham, and M. Hermann. A Compiler for LispKit Targeted at Henderson's SECD Machine. Research Report 89/339/01, Computer Science Department, University of Calgary, Alberta, Canada T2N-1N4, January 1989. Prepared under Contract No. W2213-8-6362/01-SS with the Department of National Defence, 80 pages.

[SBN82] Daniel P. Siewiorek, C. Gordon Bell, and Allen Newell. *Computer Structures: Principles and Examples.* McGraw-Hill, New York, 1982.

[Sch24] Moses Schönfinkel. Über die bausteine der mathematischen logik. *Math. Annalen*, 92:305–316, 1924. (Translated to English in [Sch67]).

[Sch67] Moses Schönfinkel. On the building blocks of mathematical logic. In Jean van Heijenoort, editor, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*, pages 355–366. Harvard University Press, Cambridge, MA, 1967.

[Sch86] M. Scheevel. NORMA: A graph reduction processor. In *Conference on LISP and Functional Programming*, pages 212–219, Cambridge, MA, August 4–6 1986. ACM.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[Sto83] W. Stoye. The SKIM microprogrammer's guide. Technical Report 40, University of Cambridge Computer Laboratory, Corn Exchange Street, Cambridge, England CB2 3QG, October 1983.

[Sto85] William Stoye. *The Implementation of Functional Languages using Custom Hardware.* PhD thesis, Cambridge University (Magdalene College), Cambridge, UK, May 1985. Available as TechReport 81, University of Cambridge Computer Laboratory, Corn Exchange Street, Cambridge, England CB2 3QG.

[Tab87] Daniel Tabak. *Reduced Instruction Set Computer – RISC – Architecture.* Industrial Control, Computers and Communications Series. John Wiley & Sons, 1987.

[Tra85] K. R. Traub. An abstract parallel graph reduction machine. In *Proceedings of the* 12th *International Symposium on Computer Architecture*, volume 12, pages 333–341, Boston, MA, June 17–19 1985. IEEE.

[Tur79a] D. A. Turner. Another algorithm for bracket abstraction. *Journal of Symbolic Logic*, 44(6):267–270, June 1979.

[Tur79b] D. A. Turner. A new implementation technique for applicative languages. In *Software — Practice and Experience*, volume 9, pages 31–49. John Wiley and Sons, September 1979.

[Tur84] D. A. Turner. Combinator reduction machines. In *International Workshop on High Level Computer Architecture*, pages 1–13, Los Angeles, CA, 1984.

[Tur85] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16, Nancy, France, September 1985. Springer-Verlag.

[Veg84] S. R. Vegdahl. A survey of proposed architectures for the execution of functional languages. *IEEE Transactions on Computers*, 33(12):1050–1071, December 1984.

[VHD87] *Standard VHDL Language Reference Manual*, #1076 1987. $40, IEEE Service Centre, B1331, Piscataway NJ, 08855. Ph. 201–981–1391. IEEE Standard Sales: Ph. 800–678–IEEE.

[Wad71] C. P. Wadsworth. *Semantics and Pragmatics of the Lambda-calculus.* PhD thesis, Oxford University, 1971.

[WF89] S. C. Wray and J. Fairbairn. Non-strict languages — programming and implementation. *Computer Journal*, 32(2):142–151, February 1989.

[WSC89] Ronald Waxman, Larry Saunders, and Harold Carter. VHDL links design, test, and maintenance. *IEEE Spectrum*, 26(5):40–44, May 1989.

Appendix A

Simulation Tool: TIMSIM

When I began investigating TIM, I realised that the designs I derived for comparison would have to be simulated extensively, to evaluate each for efficiency and the utility of the individual changes incorporated in previous designs. Simulation data would provide a direct feedback on the development of the architecture, and the intent was that the gradual specification of each of the sub-components would make it a "simulation-driven" design process. Simulation has been called upon to perform data-gathering throughout the different phases of my thesis research. In this section I describe the general nature of the simulation tasks involved in TIM, and how these defined the desired capabilities of the simulator and its final implementation form. I continue with an outline of the structure of the simulator package and its important components. I conclude with a simple example simulation, and some suggestions for improvements to the package.

A.1 Application to the TIM

The first phase of my thesis research is the evaluation of numerous optimisations to the abstract machine, both my own and those proposed in the literature. The majority of these can be characterised as modifying the capabilities of instructions, or adding a new stack to handle objects previously stored elsewhere. To evaluate one optimisation over another, or provide "before-and-after" information needs only a high-level architectural simulation. Here, almost all detail of implementation is abstracted away, in the interests of speed and simplicity. The type of data collected is coarse: examples are memory and stack consumption rates, raw numbers of function calls, profiles of instruction execution, and total use of instruction cycles. The essential task is one of deciding whether the optimisation makes the machine quicker or not, and the nature of the *primary mechanism* behind any improvement in efficiency. Lastly, the estimated cost of implementation is factored into any judgement on the usefulness of the optimisation. In some cases, there may be ambiguous results or several subtle mechanisms cooperating behind a demonstrated improvement. Here, the instrumentation is extended and/or the design model detail is expanded, perhaps in several ways, to better discriminate the effects of the optimisation on the machine.

The second phase of my research is built on the abstract machine definition that has been derived, a collection of optimisations providing a more or less viable TIM. This "settled" machine definition is ready to be made concrete as a paper hardware design. Here I am concerned with design details of the CPU control section structure, microcode engine, memory structure and interface, and even such things as the firmware-defined structure of objects and heap frames. Each piece of hardware must be tested for efficiency, demonstrated correct as a realisation of the abstract design, and qualified as to how well it fits with its companions. I expand the existing simulations to encompass this detail by partitioning the design into its major hardware blocks and signal paths, and an iterative process is used to find a "best-fit" hardware design for the abstract definition. Not only are options in design evaluated against each other, but an implicit second evaluation of the abstract definition and its component optimisations occurs. I may find that portions of the abstract specification are not amenable to hardware implementation, or that optimisations which "look good on the blackboard" are not so beneficial once subjected to the test of practicality. There is an important design feedback from implementation to the abstract definition.

In essence, the simulator is used only for modeling designs at three of the several traditional simulation "levels" of hardware: the Architectural, Instruction, and Register-Transfer levels (there are also Logic, Switch, and Device or Analogue levels). The variety of data I collect is used primarily to derive qualitative aspects of the design, where pattern is more important than precision. At the higher levels of abstraction, information on simple quantities such as instruction frequency or memory access localities tells me before I start where the most effort on instruction design should be spent, and how the memory hierarchy might best be laid out. At lower abstraction, more refined measurement is used to pick out performance bottlenecks or "hot spots" in the overall design, using information such as percentage utilisation of busses, hardware contention, and so forth.

One of the simplifying assumptions I make is that precision in time modeling is of secondary importance, and that quantities such as propagation delay, setup and hold times, and drive capabilities can be ignored. This is reasonable in view of the type of information I am interested in, and also that the goal is to make a paper design from an abstract one. It would be pointless to model technology-dependent quantities or electrical properties, without being at the layout stage armed with a target technology and fabrication process. Thus to concentrate on measures of throughput and utilisation, it is useful and valid to 1) model communications paths as having no delay, and 2) use a time resolution that is no finer than a major or minor machine cycle.

A.2 TIMSIM Package Characteristics

Once the tasks of the simulator were defined, I went on to devise some of the general capabilities I wanted to include. The majority of simulations would differ from each other only in a few small details of design or instrumentation. This high degree of design *locality* implied that one of the primary capabilities should be very quick incremental modifications between simulation runs. The portions of a simulation, in order of most frequently modified, are:

1. instrumentation; tracing and data collection, and the TIM programs being executed.

2. machine design; the major blocks of hardware, main communications paths and control lines, and their interconnections.

3. basic building blocks; definitions of the logic blocks used above, including ALU, registers, memories, latches, control store, *etc.*

4. basic support package; all of the entities on which logic and communications models are built.

The structure of the simulation package reflects the ordering of these domains (see Section A.3.6).

At the outset, I was not sure how complex the simulations or how elaborate the data collection would be, and so I wanted to ensure that TIMSIM was both *extensible* and *maintainable*. As my ideas of what was wanted grew with experience, this would ease the necessary changes to the package. To this end the simulator was written in a very general form, with many hooks for extra packages, procedures, logic classes and so forth. Further, package debugging hooks (using the Trace Facility, Section A.3.5), were left in place during development, to aid maintenance and upgrades, and detect errors in package use.

A more complicated aspect of the rapid prototyping goal was the inclusion of automatic design debugging. Again, implemented through the Trace Facility, this capability exists primarily in the modeling of communications links, and is by no means complete or even comprehensive. The motivation was to discourage simple design errors, such as a mismatch between bus and logic port widths, or bus contention during simultaneous writes from logic blocks. TIMSIM was endowed with a few key checks to catch a majority of the simple microcode or connection errors that are easily made.

An essential capability of TIMSIM was to *focus attention only on details of interest*. This was justified by concerns for faster model development, and secondarily more efficient simulations. Models and instrumentation are emplaced only for portions of the machine design under study; the remainder is abstracted away and its function emulated using more efficient means. Simulations may thus span several regions of interest concurrently, perhaps at different simulation levels (or "resolutions"). As an illustration, consider a comparison of two designs based on their efficiency during a context change. For the particular enabling native instructions, it would be useful to trace that portion of the program at the instruction level, while observing the behaviour of the microcode engine at the register transfer level, and collecting data to summarise memory access patterns at the architectural level. All other detail of the ALU, busses, control section and so forth can be abstracted away into amorphous logic-blocks that use Simula [Poo87] to emulate external functional behaviour. Models can be extended both to encompass more detail, and to examine detail at finer resolutions, in a recursive fashion. This process can be extended as far as necessary, fracturing logic blocks and communications links into more and more modeling primitives, until each register and wire is simulated independently, if desired.

A.3 Structure of TIMSIM

At this point I knew enough to select a simulator package. I had knowledge of various public-domain tools, dealing in functional-, register- and transistor-level simulation. Several commercial tools were also available, such as "ISP′" from Endot Inc. (now DATA I/O) based on ISPS [Bar81a, Bar82], but these are quite expensive (*ie.* a commercial license for ISP′ was $US50 K in mid-1988). I had difficulty finding information on the university-developed tools, and commercial tools were prohibitively costly. VHDL [VHD87, WSC89, Arm89, Coe89] was as yet unavailable. In addition, it seemed I did not need many of the advanced capabilities of this software, only the upper part of any mixed-mode facilities provided, and would be satisfied with a fairly efficient and extensible workbench. For both these reasons and academic enthusiasm (the opportunity to learn something about logic simulations and how they are put together) I decided to write my own simulator package. The decision was influenced by the fact that our site already had a simulation development package, and local expertise in its use.

The TIMSIM package was built using the tools in the Discrete Event-based Modeling on Simula package (DEMOS) [Bir79]. DEMOS is itself based on Simula, and augments the simulation primitives there by providing more convenient processes, synchronisation and data collection facilities, in a fashion analogous to the improvement of LaTeX upon TeX [Lam86, Knu86].

In its simpler use, the programmer develops a simulation model of the real world, and breaks this into "active" and "inactive" agents that become known as *entities* and *resources*. Entities become scheduled coroutines which perform actions, while resources are their consumables. A number of intercommunication, synchronisation and scheduling schemes are provided for controlling both. Entities may queue for resources or the attention of each other, and follow a number of different protocols, variants of either entity-resource synchronisations or entity-entity cooperations. Using the object-oriented philosophy, it is relatively easy to build on the DEMOS package in any way or to any complexity necessary, simply by defining additional classes of simulation entities on top of existing ones.

The hardware model I devised for my simulator is a simple one. There are similarities to a VLSI simulation model described in [Che84]; to be fair, that previous work is highly sophisticated and far better developed in comparison. I began with the idea that synchronous circuits can be broken into blocks of logic which do all the "work", connected together by communications paths (from busses to single control lines) which only transport data. The consumable resources are generic signals transmitted on the busses and manipulated within logic blocks, forming the medium of exchange for data.

Within an arbitrary synchronous system, each logic block will follow the same basic pattern: sample the input ports, spend some finite time working, and assert new signals on the output ports. Clock "ticks" control these actions, and logic blocks as discrete simulation entities are scheduled to execute on clock tick events. Communications paths accept signal assertions from writers, order them by time and signal type, check for conflicts, and return the appropriate signal for each strobe performed by a reader. Logic blocks are thus primary entities, and communications paths are slaved to their requests.

A.3.1 Resource: Num

One of the most important simulation resources is Num, since it is used everywhere to represent machine values. They have several characteristics:

- used to hold the values of signals during transmission amongst logic blocks.

- provides the simulation entity underneath any type of storage cell within a logic block or elsewhere.
- implements any arithmetic or logical operation desired within a logic block under simulation, or for direct modeling.
- contains its own conversion routines for I/O with control files and tracing.
- can represent any arbitrary magnitude of binary value (1–64 bits)
- model valid binary values, as well as *undefined* and *high-impedance* (for bus models) conditions.

Not only are they versatile, but a lot of effort went into making them efficient. Each Num contains three separate internal representations of its value, using whichever as needed. To implement logical binary operations such as shifting or a bitwise XOR, a binary format is used. For faster arithmetic operations, a standard integer format is used (when possible). For I/O and tracing functions, there is a text format. A Num will use whichever is needed under the circumstances, and keep track of which formats are currently valid using "dirty bits". The text format in particular can use any of bases 2, 8, 10, or 16, for either displaying information, or accepting data from external sources.

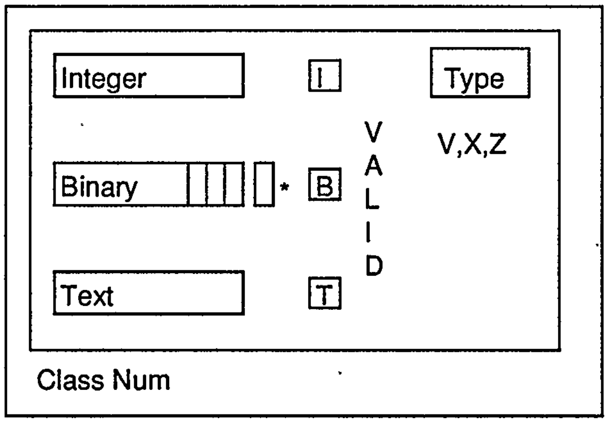


Figure A.1: class Num

class Num(size):	object class, functions
Num.write(int)	load Num with an integer value
Num.SIGsetZ	load Num as high-impedance
Num.SIGsetX	load Num as undefined
Num.cp(othernum)	copy some other Num value
Num.equal(othernum)	compare values
Num.SIGis*	predicates; is number valid (V), high-Z (Z) or undefined (X)?
Num.SIG*v	predicates; which of integer, boolean and text internal formats is valid?
Num.size	binary representation size
Num.int	integer representation
Num.txt	text representation

	library support functions
numnewcp(srcnum)	func, create new copy of srcnum
notbin(srcnum)	binary not
incbin(N,C)	binary increment with carry
negbin(N)	2's complement negation

Table A.1: class Num interface

A.3.2 Resource: Signal

A signal is the basic unit of communication for buses and wires modeled with class Comlink. It contains a Num to hold an asserted signal or group of signals, as well as information about which logic block asserted the signal, and at what simulation time. Local routines provide access to this information while protecting it from modification. A signal is modeled within DEMOS as a linked list element, so it may be queued and sorted with its neighbours within a ComLink.

class Signal(author, authorID, signal, timestamp):	object class, functions
Signal.who:	textname of author of this signal
Signal.whosn:	ID of author
Signal.what:	value of the signal (Num)
Signal.whn:	time of signal assertion

Table A.2: class Signal interface

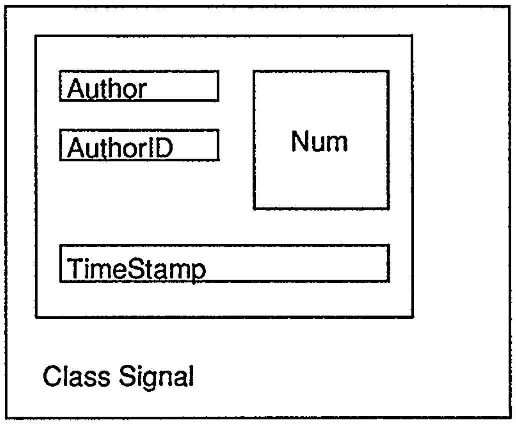


Figure A.2: class Signal

A.3.3 Entity: ComLink

A ComLink (communications link) models an abstracted signal path used to transmit binary values of any arbitrary width amongst logic blocks. There is no restriction on the number of logic blocks connected to a Comlink, and each connection "port" may be transparently used for input, output, or bidirectional transmission.

The task of the ComLink is threefold: 1) accept asserted signals from writers, 2) return appropriate signals to readers which "strobe" the communications path, 3) time-order and resolve the asserted signals to supply strobes and tracing information. Here it is obvious why a Signal (and component Num) may be a real asserted value (Num = V), a notification of *de*-assertion (Num = Z), or a undefined value (Num = X) to signify an undefined state in a logic block. Thus we can detect when a bus is undriven and ready to accept an asserted signal (or to flag an erroneous strobe), flag a contention due to two or more conflicting assertions, and *be able to propagate undefined conditions* to other parts of the circuit (useful for detecting more transient varieties of design errors). Table A.3 shows the varieties of signals pairs which may occur, and how they are resolved to decide on real bus contents at the simulation time of a logic block strobe. Note the combinations that result in contention, and that actually all but two pairs result in a "bad strobe".

There are two simple interface procedures for ComLink, predictably named read and write. Both deal only in simple values (Num) and identifying information, hiding

On Bus	Asserted	Strobe	
X	X	X	Contention!
X	Z	X	
X	V	X	Contention!
Z	X	X	
Z	Z	Z	Bad strobe
Z	V	V	
V	X	X	Contention!
V	Z	V	
V	V	X	Contention!

Legend
X = Undefined signal
Z = Hi-Impedance
V = Binary value: [0 | 1]*

Table A.3: Signal Resolution Rules

most details of transmission from logic blocks.

class ComLink:	object class, functions
ComLink.write(args):	author (authorID) asserts a Num value.
ComLink.read:	strobe the contents of the bus

Table A.4: class ComLink interface

Each ComLink maintains two separate queueing systems, a two-stage signal queue, and a queue for strobe requests. The first signal queue SigQ immediately accepts asserted signals and holds them for later processing when the ComLink is scheduled. The second signal queue MemQ contains the results of signal resolution performed on SigQ. Once older signals are discarded, newer ones are sorted in by time and type, and error checked. The result is a *local memory* for the ComLink, holding the communications history located immediately around the current simulation time (CTS).

The strobe queue holds logic blocks which have requested to read the ComLink. This queue is a DEMOS construct that provides entity-entity cooperation, through two function calls q.coopt and q.wait. Normally, this provides a pairwise synchronisation between entities which are *waiting* and those which wish to *cooperate*. In my use, only the single ComLink entity itself cooperates, and it does so in a busy loop. Logic blocks, through a call to ComLink.read, indicate they are waiting for service. When the ComLink becomes the current "active" process, it services each waiting logic block with the resolved signal for the CTS, and re-schedules them with

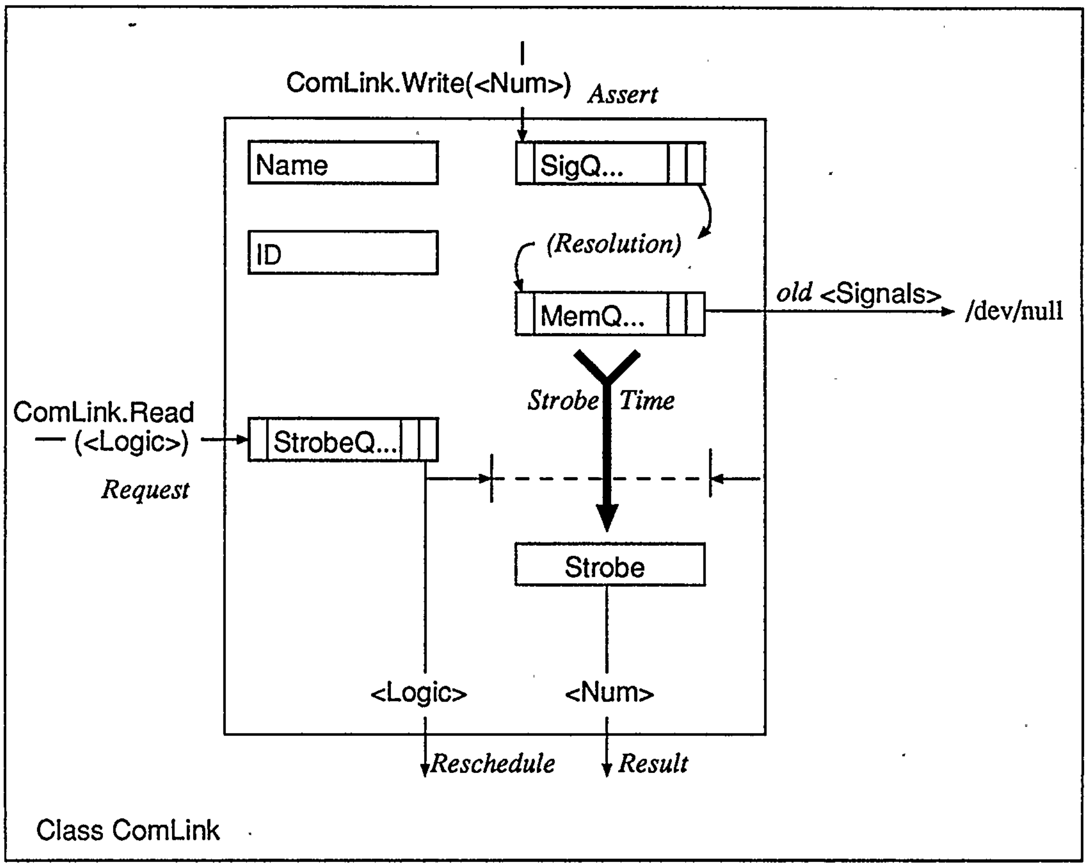


Figure A.3: class ComLink

no delay (at the CTS). In effect, a reader observes a strobe to occur in zero time.

Signal resolution is used to provide an accurate result for the latest signal strobe. To be accurate, all signal assertions *up to and including* those occurring at the simulation time of this strobe must be "in hand". This is guaranteed by the following conditions:

1. signal assertions occur without a context change, and are time-stamped to the CTS.

2. every inactive ComLink (those without waiting requests) stays outside the main scheduling queue.

3. when a ComLink obtains a read request it is scheduled at the CTS, but behind any other entities scheduled for the CTS (FIFO queue). *ie.* any entity who

might_still write a signal at this time will do so prior to *any* ComLink becoming active.

4. an active ComLink services *all* waiting logic blocks before cooperating again.
5. after service, logic blocks are re-scheduled for the CTS. With multiple scheduled ComLinks, these will normally become active prior to the re-scheduled blocks, which potentially write more signals. I use a simple trick to once again force each ComLink behind the logic blocks scheduled for the CTS.

With the safety measures above, and "instantaneous" service, arbitrarily many concurrent signal strobes and signal assertions may be accurately processed. Thus, logic with multiple ports can be represented as a single block, with read/write delays modeled individually or collectively, according to choice.

A.3.4 Entity: Logic

This class is the generic logic block. Initially featureless, the user adds internal Simula code and DEMOS directives to implement the functional behaviour of a register, ALU, or whatnot. To reflect reality, logic blocks may be connected to arbitrarily many communications paths (input, output, and bidirectional) or clock signals. Similar to other simulation languages, clocks and signal paths are represented as arguments to the new class. In the "outside" world, unused ports are simply left *dis*-connected using null; internally, required ports are checked before use.

As to modeling time within the logic block between two actions, a logic block calls the DEMOS hold() function to reschedule itself at the time that the next action should begin. This method is used to implement both the behaviour triggered by clock signals and that represented by internal propagation delays. Thus the core of a logic block looks very much as in Figure A.4.

Currently, the basic logic block is very simple, existing mostly to keep hardware modules under the same roof in the simulation; there are only two interface routines. Strb_time is for use by a ComLink to ensure that a read request from this logic block had indeed been serviced in zero simulation time. Strobe is used by a servicing ComLink to return the result of a read operation. Check_overdrive determines if a

```
logic class dtype(init,warmstart,clock,D,Q,QN);
ref (Num) init; ref(rdist) warmstart, clock; ref(comlink) D,Q,QN;
begin
  ref (Num) state;

  state := init;             !** initialise the logic;
  hold(warmstart.sample);
loop:                        !** top of cycle;
  edge_time := time;
  vporttime := time;         !** read input port;
  D.read;
  state := vport;
  hold(T_sub_P.sample);      !** propagation delay before outputs asserted;
  Q.write(title,id,state);   !** assert on output ports;
  QN.write(title,id,state.negate);
  check_overdrive(clock.sample,0.0); !** used up more time than given?;
  hold(clock.sample);        !** schedule at the next clock event;
  repeat;
end-class-dtype;
```

Figure A.4: Sample Logic block

clock signal is overdriving the logic; prior to rescheduling, it compares elapsed time since the last clock event with the tentative time for the next one, to ensure that the "next" event wouldn't already have occurred.

class Logic:	object class, functions
Logic.strobe(signal):	return signal strobe.
Logic.strb_time:	return signal strobe time
Logic.check_overdrive:	verify clock consistency

Table A.5: class Logic interface

Internally, the normal name and serial number are maintained. Edge_time is used to store the simulation time at "top-of-cycle" for calculation of elapsed time by check_overdrive. Vport and vport_time represent a "virtual" port, used as interface to comlinks we are reading by routines strobe and strb_time.

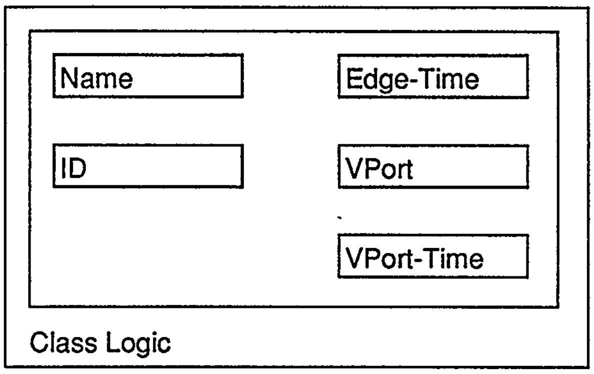


Figure A.5: class Logic

A.3.5 Trace Facility

This is a special class that provides a generic trace facility. Each is a boolean flag that holds a title, ID, and type information, and can be enabled and disabled as needed. There is also a count of times the flag has been "successfully" used, useful for threshold checks and the like.

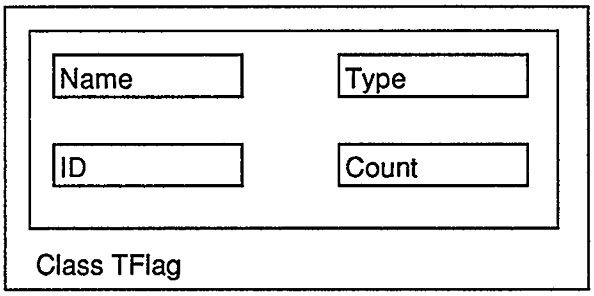


Figure A.6: class TFlag

These are used in conjunction with tracing routines that accept a condition and a text string as parameters. Thus, if the flag is enabled, then the condition is tested, and if true, the text message is dumped to a listing. For easy parsing of simulation results, each such message is prefixed by the flag name, ID, and type, and the simulation time.

Each trace flag also has a virtual routine, executed on a "successful" flag; this can be defined to implement a special reporting function or perform some other task. Also, virtual routines and sub-classes can be made to behave differently based on the flag type. The types of trace flags and their intended meanings are as follows:

class TFlag(Title,ID,type,initial):	object class, functions
TFlag.enabled	is flag up?
TFlag.enable/disable	obvious
TFlag.TF(condition,string)	test flag and condition, report string
TFlag.TFE(condition,string)	as above, return condition result
TFlag.extra	the virtual procedure hook

Table A.6: class TFlag

Type	Meaning
F	Fatal error, usually TIMSIM package bugs
E	Normal error causing immediate exit
W	Warning
I	Informational message (*ie.* not serious)
T	Development trace
D	Debugging trace flag, not normally enabled.
U1-4	User-defined

Table A.7: Varieties of Trace Flags

A.3.6 User Interface

The simulator itself is laid out into several libraries of code. At the bottom is the DEMOS/Simula foundation for timsim, on which several varieties of support routines and classes are built, including the general purpose math class Num. In tfsim, the instrumentation tools that augment DEMOS data collection routines are defined. The core of the simulator exists in rtlsim, where Logic, ComLink and other hardware primitives are defined. The two libraries timcell and timsim represent all of the simulation-specific information, which may expand into several libraries. Timcell initially holds only a few common circuit fragments, which are later augmented by user-defined registers, memories, ALU's, *etc.*. Timsim contains the actual design and instrumentation for a particular simulation.

The task of the user is fairly simple. First, the varieties of logic blocks that will be used are designed, and specified in the file timcell.sim (or as many files at this level as desired). Once the machine design has been parsed into the interesting bits, each is declared as a pre-defined comlink or logic block in the file timsim.sim, and any desired system clocks or trace routines are specified at this time. The actual

timsim	configuration
timcell	component designs
rtlsim	Signal, Comlink, and Logic architecture modeling primitives
tfsim	the trace facility
stdsim	I/O and support functions
numconv	internal conversions, I/O for Num type
numtype	definition and math operations for 3-part Num type
stdmath	support functions
stdio	support functions
DEMOS	
Simula	

Table A.8: TIMSIM library package hierarchy

design and interconnections are specified when each declaration is instantiated, as the signal paths and clocks are all passed as parameters to the appropriate logic blocks. The last phase involves setting all of the Trace Facility switches to observe what is wanted, and specifying where to find the contents for any internal ROM (*ie.* microcode) or RAM blocks (*ie.* a (TIM) machine-language program to execute).

Appendix B

Simulation Environment

In this appendix I give a brief description of the TIM workbench environment, describing the tools, utilities, sources, test programs and their interdependencies. The following are the major elements:

Simulator The TIMSIM simulation support package.

Assembler The PONASM assembler, accepting TIM code from Ponder environment.

Standard Libraries The standard library of TIM code functions, implementing the ground type operations.

Instruction Set Definition This definition of opcodes, fields and object code templates.

Microcode Definition The microcode definitions of ground type instructions.

Pre-assembler Performs pre-assembly editing of TIM code for compatibility.

TIM Code The TIM macrocode source programs.

Utilities Smaller utilities that perform a variety of useful functions.

The flow of information between these elements is shown in Figure B.1, for a sample program `testprog`. Solid lines represent the production and consumption of source, listing and object files. Dashed lines represent transfers that have not as yet been automated and require editing of SIMULA code. The boxes are the executables in the environment, while the bubbles represent the definition files and standard library sources.

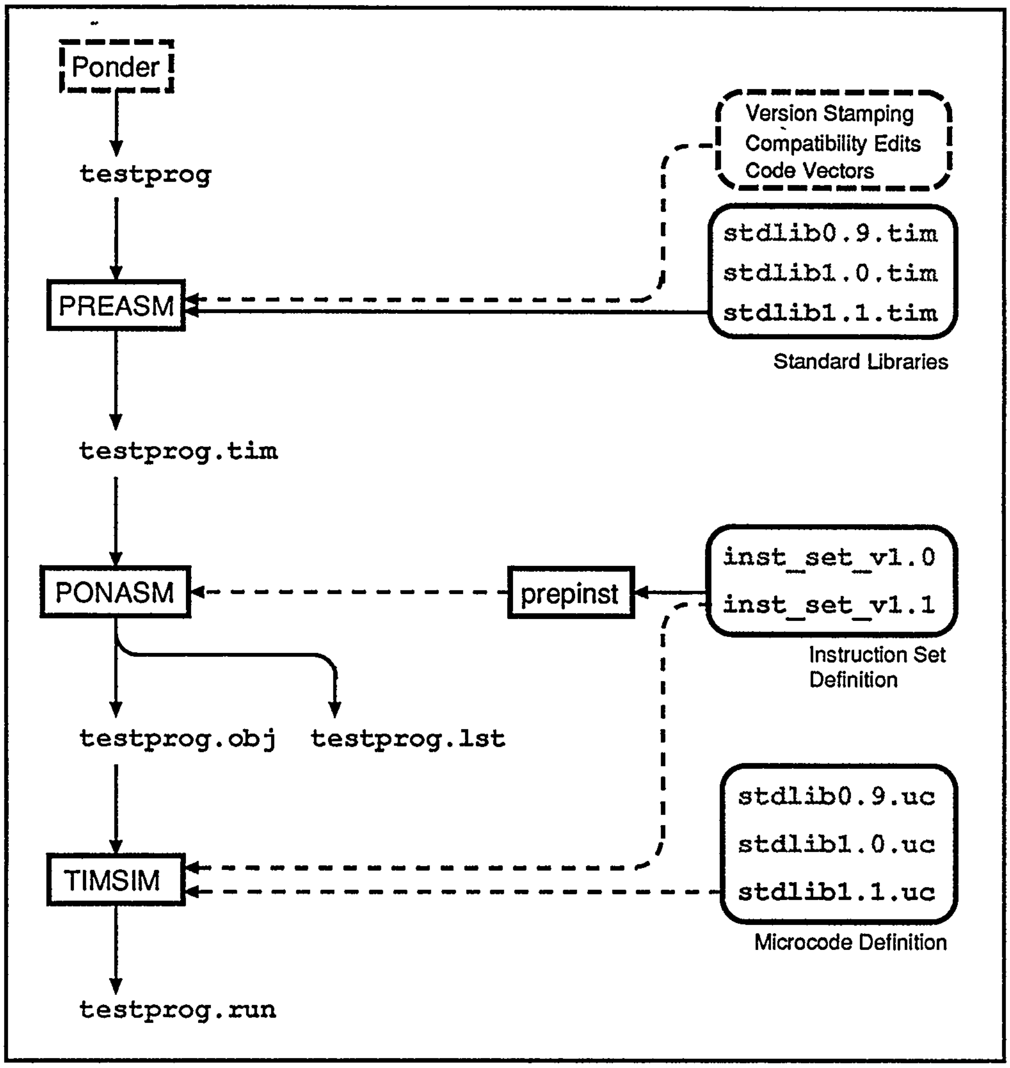


Figure B.1: Overview of Environment

Simulator This simulator is called TIMSIM, and is used to execute TIM binaries produced by the assembler ponasm. The simulator internals are kept synchronised with those of the assembler through use of the information held in the instruction set and microcode definitions (see below). The instruction set definitions are encoded in `timsimdef.sim`, while the microcode resides in `timcell.sim`. The simulator is almost fully compatible with the TIM source code produced by the Ponder environment. The support package for hardware simulations has been described in Appendix A. It should be restated that the top 2-3 files of the package are specialised to implement the TIM architecture. TIMSIM is currently at version 2.0, and consists of the files below which comprise approximately 2900 lines of SIMULA code.

timsim.sim	TIM simulation top level, configuration file.
timcell.sim	TIM-specific hardware modules, firmware internals.
timsimdef.sim	TIM-specific firmware and hardware logical structures.
rtlsim.sim	register transfer level simulator support package.
tfsim.sim	simulation tracing facilities.
stdsim.sim	generic simulation support (user interface).
numconv.sim	improved variable radix number interface.
numtype.sim	improved variable radix number structure.
stdmath.sim	simulator math function support.
stdio.sim	general I/O routines.

Assembler The assembler is called PONASM, and is a two-pass binary assembler which translates TIM macro code into an "object" consisting of binary opcodes, addresses and data. Although meant to take Ponder TIM code as input, the assembler is usable for any handwritten program. To this end the assembler includes full debugging support, a user interface, list-generation with assembled code appearing beside source macrocode, and a symbol table output. The instruction set definition which specifies opcodes and masks resides in `pondef.sim`, and is formatted for inclusion by the utility tt prepinst (see Utilities below). The binaries are produced as ascii-readable octal numbers, prefixed with object size and entry point address. In future, the object file will be made to hold the symbol table, enabling TIMSIM to produce a much more readable execution trace. PONASM is currently at version 3.0, and consists of the files below, comprising approximately 2500 lines of SIMULA code.

ponasm.sim	Ponder TIM code Machine Assembler toplevel.
ponparse.sim	token parser.
poncode.sim	machine code generation routines.
ponscan.sim	lexical scanner.
pondef.sim	definitions for TIM machine, and assembler internals.
symtab.sim	generic assembler support for symbol tables.
stdasm.sim	generic assembler support, user interface.
numconv.sim	variable radix number interface.
numtype.sim	variable radix number structure.
stdmath.sim	math functions to supplement SIMULA.
stdio.sim	general interface.

Standard Libraries The standard libraries contain TIM code functions implementing the ground type operations of the TIM architecture. Each of the functions is a TIM code "wrapper" implementing the argument evaluation and other operations necessary for the machine instructions to function correctly. The utility `preasm` (see below) is used to splice the standard library currently in use into the TIM code program. At assembly time `ponasm` will link the functions with their respective calls in the TIM code program. Some operations have several code variants, depending on the manner in which the machine instruction is defined to operate. The alternative function definitions have been included and documented, but are currently commented out. There are three versions of the standard libraries, the most current being `stdlib1.1.tim`, which contains 134 functions, and consists of 670 lines of TIM macrocode.

Microcode Definition The microcode definitions encode the operations for all TIM ground type machine instructions, in the form of register transfer language (RTL). These instructions are used only by functions in the standard libraries, and there is a high degree of interdependency between the function definitions in the standard library and the function of the corresponding machine instructions. Although not used directly to configure the simulator, these definitions are used to code the simulator internals, and are maintained as a master list of operations to be matched against their uses in the standard libraries. Eventually, an RTL parser could be used to load and interpret the microcode directly. There are three versions of the microcode definitions, that currently in use being `stdlib1.1.uc`. This file comprises 254 lines of microcode transfer statements, defining the microcode routines for 35 ground type and built-in machine instructions (arithmetic, logical, list, pair, input/output and specials).

Pre-assembler The utility `preasm` is used to prepare TIM source code from the Ponder environment for use in the TIMSIM environment. This includes *(i)* a standard library be spliced into the source code, *(ii)* any necessary vectors of PUSH ARG and ENTER ARG instructions are added to the source, and *(iii)* that obsolete or Ponder-specific forms of the TIM instructions are edited out. `preasm` also ensures that the source is marked with information specifying the origin of the source program, the standard library included, and the version of `preasm` performing the editing tasks.

This is necessary to keep track of the several versions of executables, TIM sources, standard libraries and instruction set definitions which are available. To maintain compatibility amongst all elements, a method of version tracking was implemented that labels all files with their origins. The TIM code programs, instruction set definition and standard libraries all contain version information identifying their creation date, pathname and in some instances last modified date. Each executable (not just `preasm`) recognises, maintains and passes on this information so that the exact content of any test program is known at run time. This information is held at the beginning of each file in strings prefixed with the key "IDStamp". For example, the object-code output of the assembler for the test program `fibonacci.tim` appears in Figure B.2, as the first few lines of the file `fibonacci.obj`. This specifies version, date, and pathname information for the original Ponder source, the run of the pre-assembler, the standard library used, the resultant TIM source `fibonacci.sim`, and the `PONASM` assembly.

TIM Code The amount of compiler effort expended to produce TIM macrocode sources from a higher-level functional language is large. The compiler must incorpo-

```
!IDStamp /home/vlsi/hermann/vaxponder/Tim/fibonacci (Oct 4 12:54, Aug 21 1989)
!IDStamp preasm v1.1 (Dec 1 21:51)
!IDStamp /tmp_mnt/fsg/fsg.usera/vlsi/hermann/Shop/run/Trials/fibonacci.tim(91.\
12.01:21:51:36)
!IDStamp stdlib1.1.tim (v1.1:91.11.30)
!IDStamp PONASM v3.1 (MJH 91.10.16)   1991-12-01 21:51:51
003341
000000
100 000
040 003
004 002706
005 002542
040 003
 :   :  :
```

Figure B.2: Version control information

rate a functional language intermediate code (FLIC), a λ-lifting algorithm to produce supercombinators, sharing analysis, strictness analysis, and a TIM macrocode generator. It was decided early on that the design of a compiler to provide TIM code would be beyond the scope of the thesis. To this end, the PONDER environment of Fairbairn and Wray [Fai86] was used to provide TIM macrocode. This package compiles the "Ponder" functional language into TIM code in a four stage process. An additional reason to use this source code was that in addition to providing sharing- and strictness-analysed code, the use of the same compiler output would improve the usefulness of any TIM performance comparisons, by placing the Ponder TIM implementation and mine on an even source code "footing".

Due to system software upgrades, the local Ponder installation soon broke, preventing any new Ponder programmes from being compiled and tested. The list of available test programs is 78 long, of which nearly one quarter are too large to be practically useful for testing. A small selection of hand-written programs was also created for testing and debugging purposes, but these lack any practical application in performance testing. The Ponder-derived sources can be hand-patched to change the original program arguments and internal function arguments. This would allow testing of the same program with a range of arguments, but the hand-patching process is difficult and tedious.

The entire list forms the large test suite, used for static code analysis, and statistical study on program size and instruction arguments.

Utilities One utility that has been mentioned is `prepinst`, used to prepare instruction set definitions for inclusion in the assembler. In addition, there are a number of useful utilities which automate the pre-assembly and assembly process, ensure that versions of code, libraries and executables do not clash, and so forth.

Summary If the reader is interested in obtaining the TIMSIM environment, the author can be contacted at the address below, or through internet e-mail addressed to `hermann@cpsc.ucalgary.ca`.

Mike Hermann
c/o Computer Science Department
University of Calgary
Calgary, Alberta,
Canada T2N 1N4
Phone: (403) 220 7691
FAX: (403) 284 4707

The software requires Lund Software Standard Simula (revision 4.10 or greater) or its equivalent, and currently operates under SunOS UNIX (release 4.1.1), on Sun 3 and SPARC architectures. The DEMOS (Discrete Event MOdelling under Simula) package source (written in Simula) is also required, and can be obtained from Dr. Graham Birtwistle (`graham@cpsc.ucalgary.ca`) at the address above.