

THE UNIVERSITY OF CALGARY

SECD:

The Design and Verification
of a Functional Microprocessor

BY

Brian T. Graham

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

June, 1990

© Brian T. Graham 1990



National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-61962-7

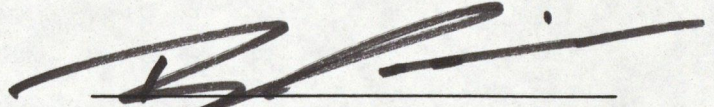
Canada

THE UNIVERSITY OF CALGARY
FACULTY OF GRADUATE STUDIES

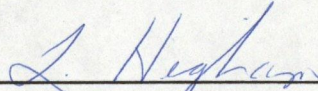
The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "*SECD: The Design and Verification of a Functional Microprocessor*" submitted by Brian Thomas Graham in partial fulfillment of the requirements for the degree of Master of Science.



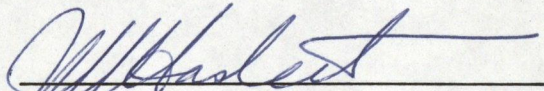
Supervisor, Dr. G.M. Birtwistle
Department of Computer Science



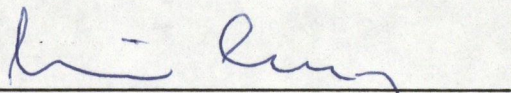
Dr. B.R. Gaines
Department of Computer Science



Dr. L. Higham
Department of Computer Science



Dr. J.W. Haslett
Department of Electrical Engineering



Dr. S.J. Curry
Bell Northern Research Ltd.

Date 1990-06-20

Abstract

The subject of this thesis is a silicon implementation of Landin's SECD machine. The starting point was an abstract specification defined by instruction transitions. Work completed includes the evolution of the design by transformation from the abstract specifications down to microcode, laying out the design in silicon, and the formal verification of its functional correctness using the HOL proof assistant.

A top level specification for the SECD system as well as an implementation level definition are generated using the HOL system. The intended operating conditions are formally defined, and installed as constraints in a machine-assisted proof of correctness stating that the computation effected by the implementation model meets the specification. The specification raises issues of the representation of S-expression data structures with destructive operation on shared structures. A solution which defines an abstract memory data type which can embed the data structures is used in the formal specification. Several issues related to the representation of temporal aspects of the chip function are analysed.

The SECD chip is one of the most complex devices subjected to formal verification to date, and is unique in combining the design and layout with the formal specification and verification of an integrated circuit. The problem size prevents presentation of either the specification or proof in their entirety, however techniques used to help manage the inherent complexity are presented in conjunction with a representative sampling of the specifications and proofs.

Acknowledgements

I have been assisted by many people throughout the years of work on this project. My greatest debt is to my supervisor, Graham Birtwistle, who has patiently supported and guided my work, and whose unwavering confidence has been a vital encouragement to strive for greater challenges. He has been instrumental in making this lab a very stimulating environment in which to work.

I owe a particular debt of gratitude to Tom Melham and Inder Dhingra, without whose gracious assistance with HOL I could not have succeeded. Thanks also to Ian Mason, whose work on the Semantics of Lisp provided key ideas for the formal SECD specification.

Many thanks to the past and present members of the VLSI and Verification group at the University of Calgary who have always been a pleasure to work with, including the original SECD team of Mark Brinsmead, Jeff Joyce, Wallace Kroeker, Breen Liblong, and Simon Williams, and past and present colleagues Tom Fukushima, Ganesh Gopalakrishnan, Mike Hermann, Rajagopal Nagarajan, Paliath Narendran, Cameron Patterson, Todd Simpson, Konrad Slind, Sue Stodart, Glen Stone, and Mark Williams.

This work could not have been completed without the support of the Alberta Microelectronics Centre, NSERC, CMC, and the Communications Research Establishment.

I especially thank my parents for their support and encouragement that gave me the opportunity to pursue this goal. Last but hardly least, I would like to thank my wife Jean for her support and patience over the years, and Timothy and Christopher, who have been a very special part of my life.

Contents

Approval Page	ii
Abstract	iii
Acknowledgements	iv
Contents	v
List of Tables	viii
List of Figures	ix
1 Introduction	3
1.1 The Impetus for Formal Methods	3
1.2 Formal Methods and Verification	5
1.3 The Nature of this Research	7
1.4 The Structure of the Thesis	10
2 Hardware Verification and Formal Methods	11
2.1 Why Higher Order Logic	12
2.2 A Brief Introduction to HOL	13
3 LispKit and the SECD Architecture	17
3.1 The Syntax of LispKit	18
3.2 The Interpretation of LispKit	21
3.3 SECD Architecture	30
3.4 Compiling LispKit to SECD Machine Code	35
3.5 Summary	39
4 SECD Architecture: Silicon Synthesis	40
4.1 Project Context	40
4.2 Levels of the Design	41
4.3 External Architecture	43
4.3.1 Abstract Machine	43
4.3.2 Abstract System: the First Refinement	44
4.3.3 The Top FSM Level	46
4.4 Internal Architecture	48
4.4.1 The Abstract Register Transfer Level	48

4.4.2	The Concrete Register Transfer Level	52
4.4.3	The Mossim Level	58
4.4.3.1	Memory Elements and Clocking	59
4.4.3.2	Control Unit	60
4.4.3.3	Datapath	63
4.4.4	Layout	64
4.4.4.1	Floorplanning	65
4.4.4.2	Design Guidelines	65
4.4.4.3	Shift Registers	66
4.4.4.4	Padframe	67
4.5	Summary and Status	67
5	Formal Specification of SECD	71
5.1	Modelling Hardware	72
5.2	The Top Level Specification	74
5.3	The Low Level Definition	79
5.4	Register Transfer Level Specification	81
5.4.1	Temporal representation	81
5.4.2	The Datapath Specification	84
5.4.3	The Control Unit Specification	90
5.4.4	The Padframe	93
5.4.5	Composing the Whole	94
5.5	Relating the Levels	95
5.5.1	Memory Abstraction	95
5.5.2	Temporal Abstraction	96
5.6	Summary	97
6	Verification of the SECD Design	100
6.1	Constraints	101
6.2	Structure of the proof	106
6.3	Unfolding the System Definition	108
6.4	Phase Level: Effect of Each Microinstruction	112
6.5	Microprogramming Level: Symbolic Execution	116
6.5.1	The initial transition	117
6.5.2	The general approach: LDF	118
6.5.3	Proving the complex sequences	124
6.6	Liveness	128
6.7	Relating the Computations over Abstraction	130
6.8	Summary	135

7	Conclusions	139
7.1	What has been accomplished	139
7.2	Putting the proof result into context	142
7.3	Retrospective Improvements	145
7.4	Hardware Verification: the future	147
	References	149

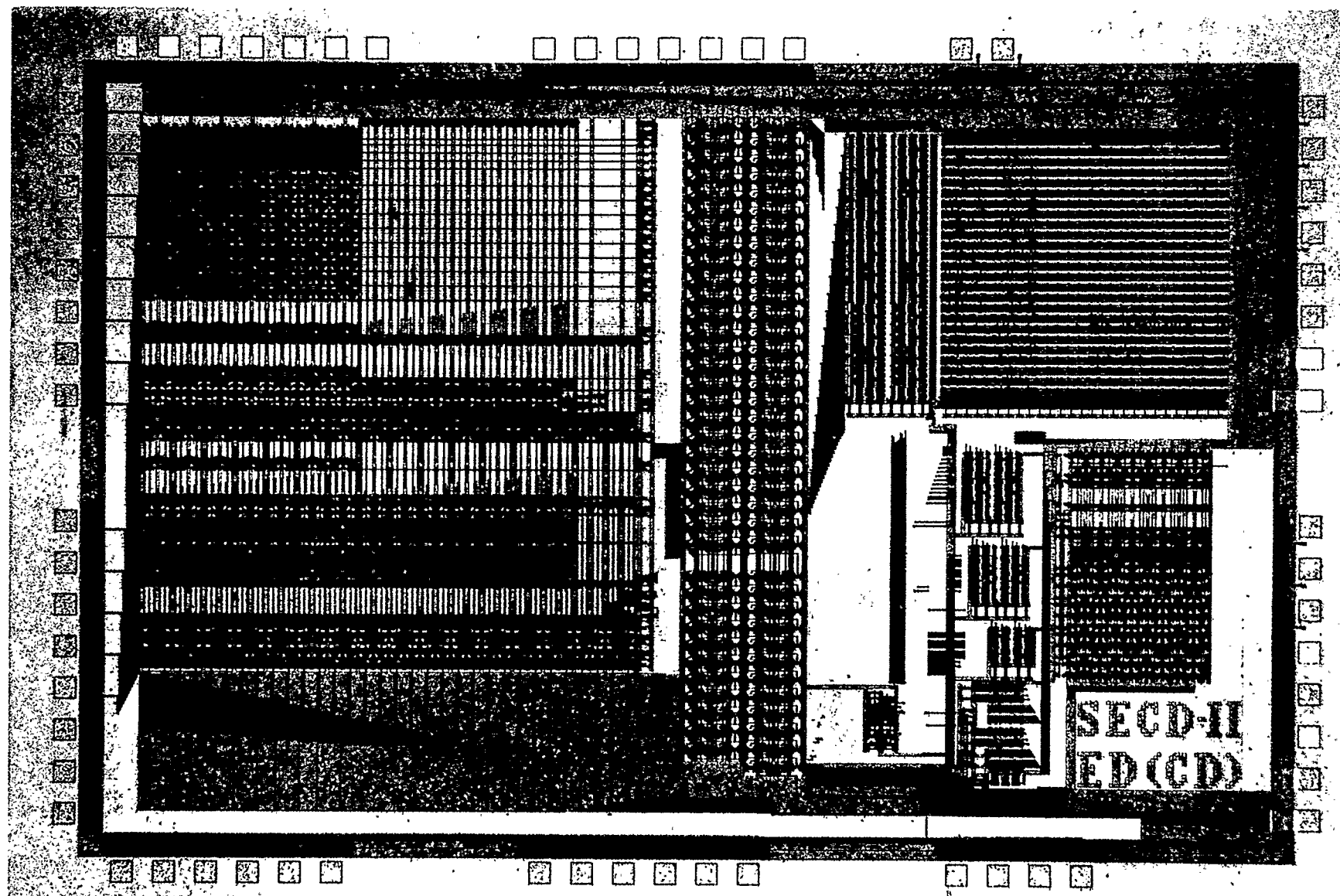
List of Tables

3.1	Well Formed LispKit Expressions	19
3.2	LispKit Interpreter Written in Franz Lisp	22
3.3	Machine Instruction Definitions	32
3.4	SECD Machine Code Generated for Well-Formed Expressions	36
3.5	LispKit to SECD Compiler Written in LispKit	38
4.1	Initial RTL Microcode Sequence for AP Instruction	51
4.2	Intermediate RTL Microcode Sequence for AP Instruction	55
4.3	Final Microcode Sequence for AP Instruction	57
4.4	Levels of Definition Summary	70
5.1	Primitive Operations on Abstract Memory Data Type	76
7.1	Comparison of Viper and SECD proofs	142

List of Figures

0.1	The SECD Chip: Microphotograph of Second Version	1
3.1	Valuelist Structure Before Destructive Operation	29
3.2	Circular Valuelist Structure After Destructive Operation	30
4.1	Top Level Finite State Machine View of SECD	46
4.2	Register Transfer Level View of SECD Machine	53
4.3	SECD Chip Major Subcomponents	68
5.1	Transition for LDF Instruction	77
5.2	Top Level Specification	78
5.3	Control Unit and Datapath Register Schematics and Definitions . . .	82
5.4	Relating low and RTL times	83
5.5	Definition of the DP Component	89
5.6	Definition of the CU Component	93
5.7	Definition of the PF Component	94
5.8	Definition of the SECD SYS Component	95
5.9	Temporal Abstraction Function Definitions	96
6.1	RTL \supset top level goal	107
6.2	Base_thm: the RTL definition simplified	111
6.3	Theorem for execution of microcode instruction at address 97	113
6.4	Microprogramming level theorems for LDF instruction	120
6.5	Microprogramming level theorems for consx1x2 subroutine	122
6.6	Decision Tree for Recursive Microprogramming Level Proof Function	123
6.7	Theorems for n iterations through loop1	126
6.8	Microprogramming level <i>State</i> theorem for LD instruction	127
6.9	Microprogramming level <i>Next</i> theorem for LD instruction	128
6.10	Abstract Memory Theorems	131
6.11	The Correctness result for the LDF instruction	134

Figure 0.1: The SECD Chip: Microphotograph of Second Version



Chapter 1

Introduction

Great strides in the technology of fabrication of physical devices in integrated circuits have created a gap between production capability and the ability to manage the complexity that such devices invite. These advances have also extended the range of integrated circuit applications, so that today any device requiring a control mechanism will almost inevitably utilize integrated circuits. The future promises continued growth in both capability of discrete components as well as the range and complexity of tasks to which they are applied. Mounting evidence of failures of integrated circuit based systems and resulting costs has focussed attention on methods of increasing their reliability. In this context, the use of formal methods has been advanced as a means of dealing with one aspect of system reliability, that of guaranteeing functional correctness of a design.

1.1 The Impetus for Formal Methods

As the smallest feature size of integrated circuit devices drops, the number of transistors that can be integrated into a single design increases. Current microprocessor designs have in excess of one million transistors on a single chip¹. Quite apart from simple transistor count, the devices are themselves increasingly complex and the development of massively parallel systems promises that this trend will continue. Reasoning about things of such complexity demands a high degree of formalism in

¹The Intel 80486 with 1.1 million and the Motorola 68040 with 1.2 million transistors were expected to be available in mid 1990, but when the 80486 was marketed in late 1989, production was suspended when an error was discovered in the floating point unit despite “hundreds of thousands of hours” of testing.

order to ensure the behaviour of the device is understood. The informal methods used for design in the past are not always adequate for today's designs.

The decrease in cost coupled with the increasing functionality of integrated circuit devices has led to a rapid growth in their range of application. Today we find them in such diverse locations as the control systems of aircraft, automobiles, ships, and trains; medical devices such as pacemakers and artificial limbs; imbedded within "smart cards" for banking, medical records, and other applications; industrial control systems including nuclear generating stations; remote sensing systems on pipelines; as well as assorted military systems.

The potential for a flawed design to cause economic disaster or loss of life is very real. Numerous documented cases of risk and actual loss connected with computer systems and related technology have appeared in the pages of the ACM SIGSOFT quarterly, *Software Engineering Notes*, spread over many issues. Peter Neumann has produced a summary of these cases with over 500 entries [Neu89]. Loss and risk categories include loss of life, potentially life-critical, loss of resources, and security/privacy/integrity problems. Behaviours to which the losses are attributed include intentional misuse; accidental misuse; misinterpretation/confusion at a man-machine interface; flaws in system concept, requirements, design, or implementation; improper maintenance/upgrade; and hardware malfunction attributable to system deficiencies, electronic or other interference, the physical environment, acts of God, etc.

The traditional method of ensuring the correctness of designs is simulation. This entails defining a model of the behaviour of the primitive components of the design, and exhaustively determining that the behaviour for each possible input condition was as desired. As the number of cases is exponential in the number of input bits for combinational circuitry, the increasing device size has made total coverage simulation too time consuming to be acceptable. Devices which maintain internal state are far

more difficult (to the point of impossible) to simulate exhaustively for each state and each input condition. Thus products are being produced today without a desirable level of assurance of behaviour over the range of all input conditions.

Formal methods, entailing the description of and reasoning about systems within a formal logic, are being applied to all levels of systems design, including both software and hardware components, in an effort:

...to increase the quality of the systems developed and to increase our confidence that the systems will behave in a predictable manner.²

1.2 Formal Methods and Verification

Formal methods generally involve representing an implementation and a specification within a formal theory. Verification compares the representations within the formal theory, reasoning that under particular constraints the implementation ensures the specification is (or is not) met.

For application to hardware, a circuit is represented in terms of primitive components, such as primitive logic gates, with levels of electrical potential abstracted to a limited number of discrete values. A primitive logic gate is expressed as a relation between its inputs and outputs. In the case of memory elements, the relation includes its state as well. Specifications are often defined in terms of more abstract data objects, in order to describe the complex behaviour in a way that may be examined for agreement with what we understand is desired. Verification involves proving that, under stated constraints representing assumptions about the operating conditions of the circuit for example, the implementation guarantees the behaviour described by the specification over all inputs and states. By using a mechanized proof system, we gain a higher degree of confidence that the proofs are indeed valid.

²Dan Craigen in [Cra89].

The use of verification must be understood in terms of its practical limitations. One obvious limitation is the accuracy of the model chosen for primitive components. As an example, consider possible transistor models. The transistor can be defined by a simple switch model, where the transistor is either on or off, depending on the gate input. This model is appropriate for CMOS, where signals are guaranteed to be strong, but will not accurately capture the behaviour of pullup transistors or transistors with weak gate signals, and could lead to an incorrect conclusion. On the other extreme, a transistor model could be as complex as the one used in SPICE. This model will give us a much higher degree of accuracy, but complexity of the proof becomes simply unmanageable for anything other than a trivial circuit. Clearly, one must choose a model appropriate for the subject, and make explicit the assumptions under which the model can capture the subject behaviour. The exceedingly complex behaviour of transistors suggests that a higher level view of the design would be more suitable for modelling. The choice of primitive logic gates and latches as the lowest level components used to model fully complementary CMOS circuits combines simplicity with accurate capture of behaviour, given the assumption that all signals have enough time to settle to stable values. Full simulation of the primitive components using such tools as SPICE can establish the detailed operating constraints under which the model will be valid.

Aside from representing primitive component behaviours, the circuit connections must themselves be correctly captured in the model. There is a need to integrate the formal model with design tools to ensure that this correspondence is maintained.

Production of a correct design requires both that the design itself meets the specification of the desired behaviour, and the desired design is actually produced in the chip. The former is the realm of verification, while the latter is in the realm of testing of the product. While the two realms can be viewed as distinct, the formalism of the design process may well offer insight in determining an appropriate test suite

for the design. Formal methods could also be used to reason about the behaviour of an assembly of exhaustively tested subcomponents.

There is necessarily a gap between the lowest level of representation and the physical hardware device, just as geometry only describes abstractions of physical objects. At the other end of the spectrum, the specification is a representation of some designer's intention, which cannot be entirely captured within a formal logic. Nor can we ever prove the validity of the specification as a representation of these intentions. These gaps are not unique to formal methods, but their presence defines the limits of what formal methods can contribute to assurance of correct design. Thus the term "partial" should be assumed whenever the words "verification" or "correctness" appear in this thesis.

1.3 The Nature of this Research

The goal of this research was to examine the application of formal methods to the design of complex integrated circuits. For this purpose, a microprocessor implementation of the functional SECD architecture, first described by Landin [Lan64] and elaborated by Henderson [Hen80], was designed, specified, and partially verified.

This work was part of a larger project on the use of formal methods in systems design at the University of Calgary. The design of hardware is only one part of producing reliable systems. It is equally important to assure the correctness of the software which will run on the system, and most important of all is the interface between the two. It is most desirable if a common formalism can be used to express both. At Calgary, the VLSI group chose to restrict its attention to functional languages running on functional architectures. It is not that the hardware is any easier to verify, but proofs of program correctness certainly are, as is the verification of the translation process. The key points in the approach are:

1. Use a functional programming language. Since they are based upon the λ -calculus they are expressive. They are also very succinct and amenable to proof.
2. Use a sugared variant of λ as the compiler target language. Since functional constructs are easy to express in terms of λ , it is relatively straightforward to express the semantics of the translation scheme and to prove its correctness.
3. Convert from λ to machine code. This step is relatively trivial if we choose a functional architecture, e.g. SECD which supports λ , or a graph reduction machine which will execute combinators.
4. Run the code on verified hardware.

Choosing λ as the common thread considerably simplifies all the above step-by-step transitions. In particular, it is possible to verify software and hardware with the same proof checker, to adopt a single proof style, and to reuse proofs.

The work is part of a long term effort in verification that started in 1985. The VLSI group chose to work with Lispkit [Hen80, HJJ83a, HJJ83b, SBGH89], and Henderson's version of Landin's abstract SECD machine [Hen80]. [Bur75, FH88, Hen80, HBGS89] explain the workings of varieties of eager and lazy SECD machines. [FH88] sketches Plotkin's [Plo75] proof of correctness of an eager SECD machine. Thus the choice of SECD was deliberate — there was much work to build on. To date the project group has

- designed, fabricated and is presently testing version II of the chip,
- constructed a rig and associated software, including compilers from Lispkit to SECD so that we can download Lispkit programs and run them on SECD,
- completed a (hand) proof that the abstract SECD machine executes Lispkit programs correctly (see [SGB89] for a full version of the proof), and

- substantially completed a machine assisted proof in HOL [Gor85, Gor88a, Gor89] of the SECD design, see [GWS89] and [BGS⁺89].

The formalism chosen for this study is a higher-order logic, which has been implemented in the HOL proof assistant by Mike Gordon of Cambridge University. The use of a higher-order logic permits specifications to be succinct and often elegant, making it easier to assure through visual inspection that the specification captures the intention. The lengthy period of time required to become adept at use of this system, combined with the availability of experienced circuit designers at the start of the project, resulted in a reversal of the intended project execution, with the chip design completed before the formal specification was prepared. This has shifted attention from the impact of formal methods on the design process to methods of specifying and verifying complex designs. Only after the fact can we speculate how revelations arising from the formal work could have contributed to a better and more reliable design.

My thesis research focuses on several aspects of the larger project:

- Development of a working integrated circuit design from a highly abstract description. The design is developed through several increasingly detailed models.
- The design of the system specification within a formal logic.
- Imbedding a model of the implementation within the same logic.
- Making the assumed operating conditions explicit within the logic.
- The proof of correctness that under the given constraints, the computation effected by the implementation matches the specified behaviour.

This work has several unique properties. The integration of circuit design and specification/verification in this project has led to a close resemblance between the formal specifications and the informal models that are used in the design process.

The choice of a functional architecture subject provided increased complexity in defining the effect of machine instruction execution, and required the representation of abstract S-expression data structures, so that the specification operates well above the level of *bits*.

One of the most significant aspects of the project has been the size and complexity of the subject system. *Because of the complexity of the SECD chip, it is not possible within the scope of a thesis to give more than an outline of most of its component specifications and the proofs: they are simply too large to be included in their entirety. All we can do is give a flavour of the work.* Despite this incompleteness, the critical concepts in designing the specification are presented in detail, and the description of the proof strategy is augmented with representative samples of results, and quite detailed descriptions of the methodology. The impact of the project size, particularly on proof management, is a recurring theme. The huge size of the proof meant that many original proof management techniques had to be developed. The achievement of the proof alone stands as a significant result. The SECD chip is one of the largest examples to date in the field of hardware verification.

1.4 The Structure of the Thesis

Chapter 2 describes other work in the field of hardware verification, and describes the HOL system which was used for the formal definition and verification in this study. In chapter 3, the SECD architecture is described, showing how it can support the execution of a Lisp-like high level language. Chapter 4 describes the evolution of the SECD design to the physical layout stage. A formal specification for the SECD system is defined in Chapter 5, as well as a lower level view closely related to the layout. The proof of correctness relating two levels is described in Chapter 6. The final chapter comprises conclusions and continuing and future work.

Chapter 2

Hardware Verification and Formal Methods

The first significant achievement in hardware verification was Gordon's machine assisted proof of the correctness of a small microprocessor with a microcoded control unit [Gor83b]. This 8 instruction machine was specified at the register transfer level, and the correctness of this model meeting a higher level specification was proved. This work was done in the LCF-LSM system [Gor83a], a predecessor of the HOL system.

Warren Hunt specified and proved the correctness of the FM8501 microprocessor [Hun85], a traditional von Neumann architecture comparable to a PDP-11 in complexity. The specification and verification was done in the first-order Boyer-Moore logic, and proven using the associated automated theorem prover.

Jeff Joyce designed the TAMARACK microprocessor based on Gordon's original example, specified and verified it in HOL [Joy88]. He has extended this work to the transistor level [Joy89b], added a configurable memory timing interface and parameterized the specification datatypes and operations [Joy89a], and has since verified the correctness of a TINY compiler generating code for TAMARACK [Joy89c]. TINY (see chapter 3 in [Gor79]) is a toy imperative language which includes assignments, conditionals, and while statements.

Perhaps the largest single verification effort to date has been the VIPER microprocessor by RSRE [Cul88] and Cohn [Coh88, Coh89b]. This work is distinguished by the fact that formal methods were applied to a commercially available product, and the considerably larger size and complexity of the device and verification effort compared to the previous examples. The chip is hard wired rather than microcode

controlled, and was defined at roughly a register transfer level with detailed implementation of data operations, at a major state level which described the operation of the chip in terms of a graph traversal, and at a more abstract top level. The correspondence of the two upper levels was fully verified, but the extension to the lowest of the three levels is incomplete, although a significant analysis of the implementation through proof techniques was accomplished.

Other significant efforts include the flooding sink local area network broadcast message eliminator by Melham [BJL⁺86], the Sobel image processing chip by Narendran and Stillman [NS89], and the Cayuga microprocessor by Sekar and Srivas [SS89]. Dhingra [Dhi88] has formalized and validated CLIC, an integrated circuit design style, in HOL.

Significant work at Computational Logic followed from Warren Hunt's work on the FM8501 described above. Bevier [Bev87] has gone on to implement and verify a multi-tasking operating system kernel for a 16-bit von Neumann architecture which includes process scheduling, response to error conditions, message passing primitives, and character I/O. Moore [Moo88] has specified the PITON language and mechanically verified its implementation on the FM8502 architecture via a compiler, assembler and linker. PITON is an assembly language designed for verified applications and includes recursive subroutine support, stack based parameter passing, and several abstract data types. Finally Young [You88] has mechanically verified a code generator for Gypsy 2.05 down to PITON.

2.1 Why Higher Order Logic

The choice of the HOL system and the use of a higher-order logic offers an expressive power that other formalisms lack. Clearly this is at some cost, as proof automation is an even more difficult problem than for a first-order logic such as the one used by

Boyer and Moore [BM79]. The expressive power is essential to capture a wide range of views of a device, and to be able to relate them in understandable ways. Without this power of expression, both the specifications and the meaning of the correctness proofs may become too obscure to be useful.

The HOL system is a very widely used proof assistant in hardware verification. Not only has it been proven a reliable tool, but there is a growing body of worked examples and a large and growing library of supporting work, including extensive sets of theorems characterising defined datatypes such as integers and bit strings. Additionally, assistance in learning HOL was readily and well provided by colleagues at Cambridge, for which the author is grateful.

2.2 A Brief Introduction to HOL

This chapter closes with a brief look at the HOL system used for the formal representation and verification of the SECD system. The following owes much of its organization to the example of [Coh89b]. A full description is beyond the scope of this work, and the reader is referred to [Cam89a, Cam89b, Cam89c] for full documentation.

HOL is a machine implementation of a conventional higher-order logic in which problems can be expressed, and interfaced to the programming language ML in which proof procedures and strategies can be encoded. The type discipline of ML ensures that the only way to creating objects of type *thm* is by the application of inference rules to other theorems or axioms. Theorems are identified by the turnstyle symbol \vdash , with assumptions to the left, and conclusion to the right.

New types, constants and axioms can be introduced by the user, and are organised in logical *theories*. Proved theorems may be saved in and retrieved from the theories,

which are organized into hierarchies in which types, constants, axioms, and theorems are inherited from ancestor theories.

The HOL system uses the ASCII characters \sim , \wedge , \vee , \Rightarrow , \Leftrightarrow , $!$, $?$, $@$, and \backslash to represent the logical symbols \neg , \wedge , \vee , \supset , \equiv , \forall , \exists , ϵ and λ respectively. Throughout this thesis, the symbols \sim , \backslash , \wedge and \vee will be replaced by the conventional logical symbols. A *term* of higher-order logic can be one of the following:

- A **variable**;
- A **constant**, including natural numbers, the boolean values T and F, etc;
- A **function application** of the form $t1\ t2$;
- An **abstraction** of the form $\lambda x.t$;
- A **negation** of the form $\neg t$;
- A **conjunction** of the form $t1 \wedge t2$;
- A **disjunction** of the form $t1 \vee t2$;
- An **equality** of the form $t1 = t2$;
- An **implication** of the form $t1 \Rightarrow t2$;
- A **universal quantification** of the form $!x.t$;
- An **existential quantification** of the form $?x.t$;
- An ϵ -**term** of the form $@x.t$,¹ expressing some arbitrary value x such that the predicate t is true;
- A **conditional** of the form $t \Rightarrow t1 | t2$, expressing *if t then $t1$ else $t2$* ;
- A **local declaration** of the form $\text{let } x = t1 \text{ in } t2$;

¹@ is a higher-order version of Hilbert's choice operator.

- A list of the form $[t_1; t_2; t_3; \dots; t_n]$ where all elements have the same *type*;
- A pair of the form (t_1, t_2) , where t_1 and t_2 may each be of any *type*.

Double quotes distinguish HOL terms in the ML interface, and a typewriter font will be used for HOL terms consistently throughout this thesis. The ML antiquotation operator \wedge permits ML identifiers bound to HOL terms to be included within HOL terms. ML comments are enclosed within % characters.

All terms in HOL have a *type*. The expression $t : ty$ means t has type ty . Built-in types include `:bool` and `:num` for *booleans* and *natural numbers*. Three type operators are `->`, `+`, and `#`, for describing function types, sum types, and product types respectively. Types may be parameterized, for example `:(bool)list` is the type of *boolean* lists. Polymorphism is allowed, and type variables are typically `*`, `**`, and so on. Nonempty new types may also be defined by mapping to an existing type within the logic. Types will consistently be shown preceded by a colon.

Many constants are built into the HOL system, including the boolean constants and natural numbers, arithmetic operators `+`, `-`, `*`, `<`, `<=`, `>`, `>=`, `DIV`, `MOD`, `EXP`, `SUC`, and `PRE`, the list operations `CONS`, `HD` and `TL`, and `FST` and `SND` selectors on pairs to name some of the more commonly used ones. The reader's attention is particularly directed to the infix function composition operator `o`, which is used repeatedly in abstraction functions.

There are two general approaches to proof within HOL: *forward* and *backward*. *Forward* proof works by applying inference rules to existing theorems and axioms to derive a desired result in the form of a new theorem. Where to start on a large complex proof is problematic, and managing the many branches involved is often-times exceedingly difficult. An alternative methodology, *backward* proof, starts with a statement of the theorem you would like proved (a *goal*), which the user incremen-

tally splits into smaller, more manageable subgoals. The HOL system manages the state of the proof on a goal stack, and when each subgoal is reduced to a theorem, it assembles the entire proof and returns the desired theorem. This methodology does not provide a distinct means of construction of *thm* type objects, rather it allows the user to generate the proof, which will use the same inference rules as the *forward* proof methodology with a top-down approach, leaving the system to manage the details. Both methods have advantages and both are used in this work. The choice of *forward* or *backward* proof will often be an important consideration in the methodology.

The HOL notation will be used for the formal definition of the SECD system, but first is a description of the abstract SECD architecture, and the development of the SECD chip design.

Chapter 3

LispKit and the SECD Architecture

This chapter introduces LispKit, a high-level programming language, defines the abstract SECD architecture, and shows how the architecture can support execution of programs written in the high-level language. The definitions of LispKit and SECD given by Henderson [Hen80] are used. Henderson also defines a LispKit interpreter, and a LispKit to SECD compiler.

LispKit is a pure functional subset of the Lisp language. A pure functional subset means that LispKit has no destructive assignment operation, so that the value of an expression is uniquely determined by the value of its constituent parts, and identical expressions always have the same value. S-expressions are defined, and then a syntax for LispKit is given. This is followed by an informal semantics for the language, defined within Franz Lisp as an interpreting function. Issues fundamental to supporting the language with a hardware system such as bindings and representation of function-valued objects are discussed.

The SECD machine architecture is described by its machine instructions and state transitions effected by each, giving a semantics for the machine language. Following this, a translation schema for well-formed LispKit expressions into SECD machine code is the basis for a LispKit compiler, and defines an operational semantics for LispKit.

To ease distinguishing between the different languages presented, different fonts will be used: *Sans Serif Italics* for LispKit expressions, *Roman Italics* for Franz Lisp expressions, and Sans Serif for SECD machine code expressions.

3.1 The Syntax of LispKit

Fundamental to the Lisp programming world is the class of objects known as symbolic expressions, or S-expressions for short. These are defined recursively as:

$$\text{S-expression} ::= \text{atom} \mid (\text{S-expression} . \text{S-expression})$$

Atoms are of two types: numeric and symbolic. A numeric atom is a possibly signed sequence of digits, which is taken as representing a decimal integer. Symbolic atoms, either constants or variables, appear as a series of letters or digits or other characters, beginning with a character. There are three special symbolic atoms: *NIL*, *T*, and *F*, which are symbolic constants and have a particular meaning attached to them.

A dotted pair is the result of a Lisp *cons* operation on two S-expressions. If *a* and *b* are S-expressions, then *(cons a b)* produces a dotted pair *(a . b)*. The dot notation used here is often replaced by the list form where possible. The rules for transforming from dot to list notation are simply:

- *(a . NIL)* may be written as *(a)*
- *(a . (b))* may be written as *(a b)*

where *a* and *b* may be any S-expressions. *NIL* is interpreted as the empty list.

LispKit provides sixteen primitive operators which are reserved symbolic constants. In order to distinguish constants from variables, constants are represented by the dotted pair whose first component is the *QUOTE* operator. Structural operators are *CONS*, *CAR*, *CDR*, and *ATOM*, which perform the standard list operations common to all Lisp variants. Arithmetic operators include *ADD*, *SUB*, *MUL*, *DIV* and *REM*. The relational operators are limited to *EQ* and *LEQ*. In addition, there is the conditional operator *IF*, the λ operator *LAMBDA*, used for defining functions, and two block defining operators, *LET* and *LETREC*, the latter being used for recursive bindings.

Well-formed expressions in LispKit form a subset of the set of S-expressions, as determined by the derivation rules of Table 3.1.

x	variable
$(QUOTE\ s)$	constant
$(ADD\ e_1\ e_2)$	} arithmetic expressions
$(SUB\ e_1\ e_2)$	
$(MUL\ e_1\ e_2)$	
$(DIV\ e_1\ e_2)$	
$(REM\ e_1\ e_2)$	
$(EQ\ e_1\ e_2)$	} relational expressions
$(LEQ\ e_1\ e_2)$	
$(CAR\ e)$	} structural expressions
$(CDR\ e)$	
$(CONS\ e_1\ e_2)$	
$(ATOM\ e)$	
$(IF\ e_1\ e_2\ e_3)$	conditional form
$(LAMBDA\ (x_1 \dots x_n)\ e)$	λ -expression
$(f\ e_1 \dots e_k)$	function call
$(LET\ e\ (x_1.e_1) \dots (x_k.e_k))$	simple block
$(LETREC\ e\ (x_1.e_1) \dots (x_k.e_k))$	recursive block

where e, e_i are well-formed expressions,
 x, x_i are symbolic atoms (variables),
 s is any S-expression,
 f is a λ -expression.

Table 3.1: Well Formed LispKit Expressions

- **Constants** may be other than just numbers, as the restriction to S-expressions implies. However, all constants must be preceded by the *QUOTE* operator. The **arithmetic** and **relational operators** are all binary operators, unlike those in many variants of Lisp. The **structural expressions** are typical of most Lisps.

- The **conditional form** takes three arguments, the first of which is the conditional expression. The second argument is evaluated only if the conditional expression evaluates to *T*, otherwise the third is evaluated.
- **Lambda expressions** are used for defining functions of one or more arguments. A list of λ -bound variables is followed by an expression which will usually include all occurrences of these variables.
- **Function calls** follow the form of primitive operators, with a function-valued expression being the first item in a list, followed by the values to be bound to its local variables.
- **Blocks** are another means of creating local bindings. The expression containing the bound variables follows immediately after the *LET* operator, followed by any number of dotted pairs of variables and expressions to be bound to the variables. The only difference between the *LET* and the *LETREC* is that the expressions to be bound to the variables in the *LETREC* may include (recursive) references to any of the bound variables, while in a *LET* they do not reference any of the locally bound variables. We shall see, in fact, that the *LET* form is unnecessary, as it is equivalent to the function call of a *LAMBDA* expression. The *LETREC*, however, is needed to define recursive functions. Furthermore, each expression e_i bound to a variable within a *LETREC* must evaluate to a function-valued object. The meaning of a function-valued object will become clear when the interpretation of LispKit is discussed below. This restriction eliminates meaningless expressions such as: $(LETREC\ x\ (x\ ADD\ (QUOTE\ 1)\ x))$ wherein an attempt is made to define x as its own successor.

Despite the derivation rules above, those expressions which will give rise to meaningful computations have not as yet been precisely defined. For the most part, the

primitive operators, as well as any functions we may define, are partial. For example, $(ADD (QUOTE 2) (QUOTE (A B C)))$ is not meaningful in LispKit, because *ADD* requires integer-valued arguments. Similarly, $(CAR (QUOTE NIL))$ is not meaningful, since the *CAR* operation is only defined on dotted pairs. For a complete denotational semantics of LispKit see [SGB89].

3.2 The Interpretation of LispKit

A full interpreter for the LispKit language adapted from [Hen80] to Franz Lisp is given in Table 3.2. A brief description of its more interesting features follows. Two sample programs give a taste of the language in use.

Bindings of variables within LispKit are represented using the concept of contexts. A context consists of a set of bindings that associate variables with values, implemented using corresponding lists of variable names and values. The value of a variable is the value located in the corresponding position in the valuelist to the location of the variable in the namelist, with the added restriction that if the variable occurs more than once, the location closest to the front of the list is used. This permits new bindings to override existing bindings within a context. For example, the namelist and valuelist:

namelist: $((x\ y)\ (z\ x))$

valuelist: $((1\ 3)\ (5\ NIL))$

represent the bindings: $x \leftrightarrow 1$, $y \leftrightarrow 3$, $z \leftrightarrow 5$. The second occurrence of *x* is rendered inaccessible in this example. This environment could have been generated by a LispKit program of the form:

```
(LET (LET (...
      (x.(QUOTE 1)) (y.(QUOTE 3)))
     (z.(QUOTE 5)) (x.(QUOTE NIL)))
```

Table 3.2: LispKit Interpreter Written in Franz Lisp

LispKit expressions will always be evaluated within some context. New bindings are added to the existing context by *LET* and *LETREC* operators. Thus, the interpretation of a variable x in a context (n, v) , is simply the value in the location in v that corresponds to the location of x in n . The interpretation of a *LET* expression is the interpretation of the expression part in the current context extended by adding the list of bound variables to the front of the variable namelist, and adding

the values, obtained by evaluating the value expressions in the existing context, to the front of the valuelist. Thus

$$(EVAL (LET e (x_1.e_1) \dots (x_k.e_k)) n v) = \\ (EVAL e ((x_1 \dots x_k).n) (((EVAL e_1 n v) \dots (EVAL e_k n v)).v))$$

Function definitions may contain both free and bound variables within the body of the lambda expression. Bindings are defined to be static, in that values bound to free variables within a λ -expression are determined from the context in which they are defined, rather than in the context in which the function is called. To facilitate this, the notion of a closure is introduced for the interpretation of a function. The closure will consist of the defining context, along with the list of λ -bound variables, and the body of the *LAMBDA* expression.

$$(EVAL (LAMBDA (x_1 \dots x_n) e) n v) = (((x_1 \dots x_n).e).(n.v))$$

The defining context consists of the namelist and valuelist. The namelist is the collection of bound variable names, and the valuelist is the corresponding collection of the values associated with each variable name. When the expression is applied to a list of arguments, the list of λ -bound variables will be added to the namelist. The evaluated arguments are added to the valuelist, creating a new context for evaluating the body of the function, in which the local variables are defined.

In a *LETREC* expression, the variables are to be bound to (possibly mutually) recursive functions, thus the context in which these functions are evaluated must include the values of the functions themselves. This is accomplished by evaluating the expressions in a context which has the values of the recursively bound variables still pending. In practice, an empty list is used as a place-holder at the beginning of the valuelist. Since each expression is required to evaluate to a function-valued object, each will evaluate to a closure, with the context part of all being identical. Thus, a single destructive *rplaca* operation can be used to alter the pending value of

the valuelist to instead point to the list of closures that are created. A circular data structure is thus created. A simplified example illustrates the idea.

$$(EVAL (LETREC e (f_1.e_1) \dots (f_k.e_k)) n v) = (EVAL e (y.n) (rplaca v' z))$$

where $y = (f_1 \dots f_k)$

where $v' = (PENDING.v)$

where $z = ((EVAL e_1 (y.n) v') \dots (EVAL e_k (y.n) v'))$

The reader should recognize that the use of a destructive Lisp operation in defining the interpretation of the LispKit expression does not conflict with the status of LispKit as a purely functional language. The LispKit programmer does not have a destructive operator to use in programming; the destructive operator is only used in creating a context to represent recursive function definitions.

To complete the interpretation, the interpretation of each well-formed LispKit expression is given.

- **Constants** are represented by a dotted pair with the atom *QUOTE* as the *car*. Regardless of its context, it will evaluate to the *cadr* of the pair. Thus *(QUOTE 2)* evaluates to 2, and *(QUOTE (a b c))* evaluates to *(a b c)*. The three special symbolic atoms, *NIL*, *T*, and *F* are mapped to the Franz Lisp values of *nil*, *t*, and *nil* respectively in the interpreter.
- The **arithmetic operators** work as one would expect. For example, *(ADD e₁ e₂)* in the context *(n,v)* will evaluate to the sum of the values of *e₁* and *e₂*, both evaluated in *(n,v)*. *SUB*, *MUL*, *DIV*, and *REM* work similarly.
- The **relational operators**, *EQ* and *LEQ*, evaluate their arguments in the same fashion as the arithmetic operators. However, *EQ* is interpreted as working the same way as the *eq* function in Franz Lisp¹, in that two S-expressions are equal

¹This differs from the definition given in [Hen80] pp. 22, 53, which defines *EQ* only when at least one of its arguments is an atom.

if they are both atoms and they evaluate to the same value, or if they are both dotted pairs and they are both pointers to the same cons cell.

- The **structural operators**, *CAR*, *CDR*, *CONS* and *ATOM* are interpreted as performing the same operations as the corresponding *car*, *cdr*, *cons* and *atom* operations in Franz Lisp, when applied to the interpreted arguments. Again, arguments are evaluated as for arithmetic operators.
- The **conditional form** evaluates its first argument in the given context, and if this evaluates to *T*, then the second argument is evaluated; otherwise the last argument is evaluated. This form evaluates only one of the two branches, permitting, for example, testing for terminating conditions of recursive function definitions.
- The **function call** is interpreted by adding the value of the function's arguments, interpreted in the current context, to the start of the valuelist in the context part of the closure. Similarly, the list of bound variables is added to the start of the namelist in the context part of the closure, and the body of the closure is evaluated in the thus extended context.

Finally, the top level program must evaluate to a function-valued object which is applied to a list of arguments. Free variables are not permitted in the top level program, so that our starting context consists of a pair of empty lists.

Example 1

The first example is a nonrecursive function that takes a function as an argument and returns a function-valued object:

```
(LET (twice double)
      (twice LAMBDA (f) (LAMBDA (x) (f (f x))))
      (double LAMBDA (x) (ADD x x)))
```

Let E represent this expression, and e_1 and e_2 the expressions to be bound to *twice* and *double* respectively.

$$\begin{aligned} & (EVAL\ E\ ()\ ()) \\ \Rightarrow & (EVAL\ (twice\ double) \\ & \quad ((twice\ double)) \\ & \quad (list\ (cons\ (EVAL\ e_1\ nil\ nil) \\ & \quad \quad (cons\ (EVAL\ e_2\ nil\ nil) \\ & \quad \quad \quad nil)))) \end{aligned}$$

The two expressions, e_1 and e_2 , evaluate to function closures as follows:

$$\begin{aligned} & (EVAL\ e_1\ nil\ nil) \\ \Rightarrow & (((f)\ LAMBDA\ (x)\ (f\ (f\ x)))\ nil) \\ \\ & (EVAL\ e_2\ nil\ nil) \\ \Rightarrow & (((x)\ ADD\ x\ x)\ nil) \end{aligned}$$

The next stage evaluates the application *(twice double)* in the context extended with the new variables and values. It is expected that the first item in the application will evaluate to a function valued object, and both it and the single argument are evaluated in the context extended with e_1 and e_2 , and then the body of the function valued object *(LAMBDA (x) (f (f x)))* is evaluated in a new context extended with the variable f and the value of *double*, producing a closure as a result.

$$\begin{aligned} \Rightarrow & (EVAL\ '(LAMBDA\ (x)\ (f\ (f\ x))) \\ & \quad '((f)) \\ & \quad '((((x)\ ADD\ x\ x)\ nil)))) \\ \Rightarrow & (((x) \\ & \quad f\ (f\ x)) \\ & \quad ((f)) \\ & \quad (((x)\ ADD\ x\ x)\ nil)))) \end{aligned}$$

A valid LispKit program requires a list of arguments to which a function is applied. The argument list (γ) will be used for this example. Thus evaluating the previous function applied to the argument list causes the body of the function to be evaluated in an environment with bindings for both x and f .

$$\begin{aligned} \Rightarrow & (EVAL (f (f x)) \\ & ((x) (f)) \\ & ((7) (((x) ADD x x) nil)))) \end{aligned}$$

The variable f is bound to a function closure, but its argument $f x$ must be evaluated. This is done in the same context, so that f is bound to the function closure of double and x to 7.

$$\begin{aligned} & (EVAL (f x) \\ & ((x) (f)) \\ & ((7) (((x) ADD x x) nil)))) \\ \Rightarrow & (EVAL (ADD x x) ((x)) ((7))) \\ \Rightarrow & (+ (EVAL x ((x)) ((7))) \\ & (EVAL x ((x)) ((7)))) \\ \Rightarrow & (+ 7 7) \\ \Rightarrow & 14 \end{aligned}$$

The result of this evaluation is installed in the context as the value to be bound to x when the outer function application is evaluated.

$$\begin{aligned} & (EVAL (f (f x)) \\ & ((x) (f)) \\ & ((7) (((x) ADD x x) nil)))) \\ \Rightarrow & (EVAL (ADD x x) ((x)) ((14))) \\ \Rightarrow & (+ (EVAL x ((x)) ((14))) \\ & (EVAL x ((x)) ((14)))) \\ \Rightarrow & (+ 14 14) \\ \Rightarrow & 28 \end{aligned}$$

Example 2

The next example is the *even* function, defined using mutually recursive functions:

$$\begin{aligned} \text{even} &= \lambda x. \text{if } (x = 0) \text{ then } TRUE \text{ else } (\text{odd } (x - 1)) \\ \text{odd} &= \lambda x. \text{if } (x = 0) \text{ then } FALSE \text{ else } (\text{even } (x - 1)) \end{aligned}$$

```

(LETREC even
  (even LAMBDA (x)
    (IF (EQ x (QUOTE 0))
      (QUOTE TRUE)
      (odd (SUB x (QUOTE 1)))))
  (odd LAMBDA (x)
    (IF (EQ x (QUOTE 0))
      (QUOTE FALSE)
      (even (SUB x (QUOTE 1))))))

```

To apply the function to the argument list *(1)*, first evaluate the *LETREC* expression. Each of the two items bound therein is evaluated in a context with the namelist augmented by *(even odd)* and the initially empty valuelist onto the start of which is *cons*'ed the special *PENDING* atom. Since both expressions are λ -expressions, they will evaluate to function closures, both containing identical contexts (pointers to the same context), with *PENDING* as the first item in the valuelists.

The body of the *LETREC*, the bound variable *even*, is then evaluated in the same context, but modified by destructively replacing the *PENDING* atom with the list of closures obtained for the λ -expressions above. This will create a circular valuelist structure, so a graphical representation will be used.

```

(EVAL example-2 nil nil)
⇒ (EVAL even
  ((even odd))
  (rplaca (PENDING)
    (((x)
      IF (EQ x (QUOTE 0))
        (QUOTE TRUE)
        (odd (SUB x (QUOTE 1)))))
    ((even odd))
    PENDING)
  (((x)
    IF (EQ x (QUOTE 0))
      (QUOTE TRUE)
      (even (SUB x (QUOTE 1)))))
  ((even odd))

```

PENDING)))

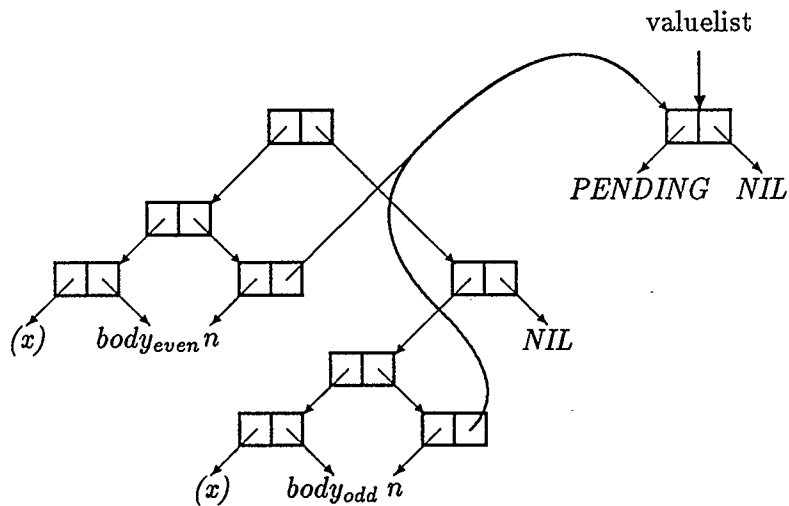


Figure 3.1: Valuelist Structure Before Destructive Operation

The important thing to note in Figure 3.1 is that the atom *PENDING* occurs in only one place, with pointers to it from several places. Thus, doing a destructive *rplaca* operation on the list with *PENDING* as the first item, will change the value of *PENDING* in **all** locations, and create a circular list structure, so that references to the recursive functions within the body of the functions, properly refer to the closure value used to represent the function.

The next step consists of evaluating the expression *even*. This returns a closure from the valuelist bound to this name. Evaluating the application of this function to the argument list *(1)*, the body of *even* is evaluated in a context augmented with the binding *1* to the lambda bound variable *x*. It should be readily apparent from the valuelist of Figure 3.2 that recursive references to *even* and *odd* are possible as the appropriate function closures are included.

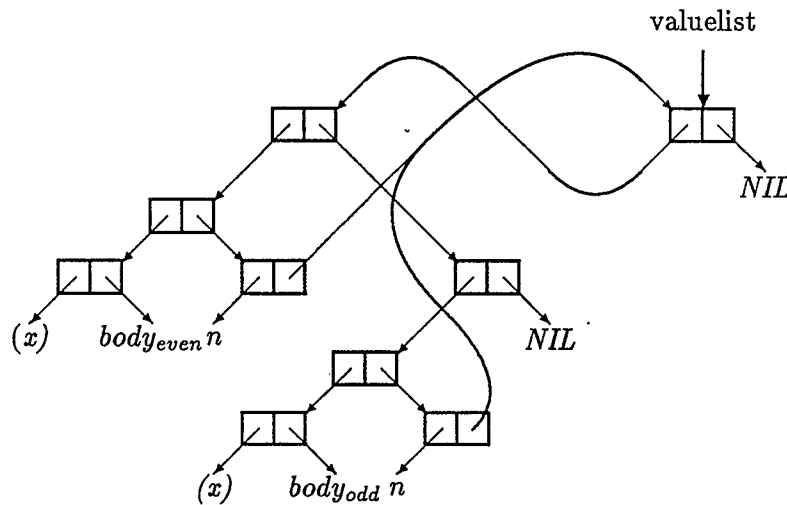


Figure 3.2: Circular Valuelist Structure After Destructive Operation

Once the circular environment structure has been established, the remainder of the function evaluation continues in a similar fashion to the previous nonrecursive example.

In summary, the LispKit language has been described, and its syntax and its semantics defined, the latter by means of an interpreter. The concept of closures to represent the value of a function object, and the use of context to define bindings, was illustrated by two examples, using both higher order and recursive functions.

3.3 SECD Architecture

Now follows a detailed description of a machine architecture, called the SECD architecture, that will support the execution of programs compiled from LispKit Lisp. A complete description of the machine instructions and state transitions effected by each will define the semantics of the machine language.

The SECD architecture, so named because of its four principal registers, was invented by Landin [Lan64] and described in detail by Henderson [Hen80]. Each of the four registers is referred to as containing an S-expression, which in the actual implementation will be a pointer to a data structure representing the expression. The term “stack” is used to refer to the data structure “within” a register. The four registers are:

S	stack	holds intermediate computation results
E	environment	holds values bound to variables during evaluation
C	control list	holds the machine-language program being executed
D	dump	saves values of other registers on calling a new function.

It is important to note that the entire state of the machine can be denoted by giving the content of its four registers. Thus instructions are defined by state changes, enabling an interpreter to be developed by pattern matching, and the use of structural induction in proofs about the machine execution.

Each instruction is defined in terms of its effect on the machine state. For example, the ADD instruction definition is:

$$(a\ b.s)\ e\ (ADD.c)\ d \rightarrow (b+a.s)\ e\ c\ d$$

This instruction expects two arguments on top of the S stack. After execution, the two arguments are replaced by their sum, which is *cons*'ed onto whatever was below the arguments on the stack. The E and D registers are unaltered, but the C stack contains the rest of the control list that followed the ADD instruction.

Similarly to the interpreter presented previously, a means of storing bindings is provided by a valuelist stored in the environment stack E, which is, as before, a list of lists. Instead of an associated namelist, variable names are replaced with a LD instruction and a pair of arguments telling it where to find the value in the

environment. The first determines which list and the second which element from that list to retrieve. Two auxiliary functions, *index* and *locate*, are defined:

$$\begin{aligned} \text{index}(n,s) &= \text{if } (n=0) \text{ then } (\text{car}(s)) \text{ else } (\text{index}(n-1, \text{cdr}(s))) \\ \text{locate}(i,e) &= \text{index } (\text{cdr}(i), \text{index}(\text{car}(i), e)) \end{aligned}$$

For example, the namelist $((x\ y)(w\ z))$ would be translated as position indices $((((0.0)(0.1))((1.0)(1.1))))$. Thus the transition for LD in Table 3.3 shows that the value loaded on the stack is the value retrieved by applying the function *locate* to the arguments and the environment list.

INITIAL STATE				TRANSFORMED STATE			
S	E	C	D	S	E	C	D
s	e	(LDC x.c)	d	→	(x.s)	e c	d
s	e	(LD (m.n).c)	d	→	(x.s)	e c	d
					where $x = \text{locate } ((m.n), e)$		
(a b.s)	e	(ADD.c)	d	→	(b+a.s)	e c	d
(a b.s)	e	(SUB.c)	d	→	(b-a.s)	e c	d
(a b.s)	e	(MUL.c)	d	→	(b*a.s)	e c	d
(a b.s)	e	(DIV.c)	d	→	(b/a.s)	e c	d
(a b.s)	e	(REM.c)	d	→	(b rem a.s)	e c	d
(a b.s)	e	(EQ.c)	d	→	(b=a.s)	e c	d
(a b.s)	e	(LEQ.c)	d	→	(b≤a.s)	e c	d
((a.b).s)	e	(CAR.c)	d	→	(a.s)	e c	d
((a.b).s)	e	(CDR.c)	d	→	(b.s)	e c	d
(a b.s)	e	(CONS.c)	d	→	((a.b).s)	e c	d
(a.s)	e	(ATOM.c)	d	→	(t.s)	e c	d
					where $t = (a \text{ is an atom})$		
(x.s)	e	(SEL $c_t c_f.c$)	d	→	s	e c_x	(c.d)
					where $c_x = \text{if } (x=T) \text{ then } c_t \text{ else } c_f$		
s	e	(JOIN)	(c.d)	→	s	e c	d
s	e	(LDF c'.c)	d	→	((c'.e).s)	e c	d
((c'.e')v.s)	e	(AP.c)	d	→	NIL	(v.e') c'	(s e c.d)
((c'.e')v.s)	(Ω.e)	(RAP.c)	d	→	NIL	rplaca(e',v) c'	(s e c.d)
(x)	e'	(RTN)	(s e c.d)	→	(x.s)	e c	d
s	e	(DUM .c)	d	→	s	(Ω.e) c	d
s	e	(STOP)	d	→	s	e (STOP)	d

Table 3.3: Machine Instruction Definitions

Loading a constant is achieved by the LDC instruction, which takes an inline constant as an argument, and places it on top of the stack.

All the arithmetic instructions work similarly to the ADD instruction described earlier: the stack is expected to have two arguments on top and they are replaced with the value obtained from applying the arithmetic operation to them. Notice the order of arguments for the noncommutative arithmetic operations SUB, DIV, and REM. The instructions effecting the structural operations CAR, CDR, CONS, and the ATOM similarly expect suitable values on top of the stack, and replace them with the result of applying the primitive structural operation.

Two commands are used to implement the conditional branch. The SEL command expects a boolean value on top of the stack S, and two code sequences as arguments. If the value on the stack is T, then the first argument is installed in the control register C, otherwise the second argument is so installed. In either case, the remainder of the control list after the two arguments is saved on the dump register D, to be restored once the selected branch control sequence is finished executing. That is the purpose of the JOIN command, which will be the last item in each of the two control list arguments to the SEL command. Thus, the execution of the code in either control sequence must leave the value stored in the D register untouched at the end of its execution.

The final set of instructions is used in implementing functions and function calls. In the interpreter for LispKit, functions are represented by a closure, containing the bound variables, the expression part of the function, and the context in which free variables are to be evaluated. In the SECD machine, the expression consists of a control list and the context is represented by an environment list. A function object is created by the LDF instruction. It takes a code sequence as an argument, and *cons*'es it onto the current environment, leaving this on top of the stack S. The function object is usually loaded into an environment and recalled with an LD instruction to apply to different arguments.

The AP instruction is used to apply a function to a list of parameters found immediately below the function object on top of the stack S. AP installs the parameters in the environment using the context part of the function object, loads the control part in the C register, empties the stack S, and saves the existing register contents for later restoration by the RTN instruction. From inside the function code, parameters are accessed by selecting items in the first list in the environment, using position indices (0.0), (0.1), etc., while variables free to the body are accessed using position indices (1.0), ..., (2.0), The RTN instruction expects the dump D to be unaltered by the execution of the control list, so that it can restore the state of the registers upon completion of the function call, with the value returned by the function call installed on top of the stack S.

Recursive functions are more complicated, since the environment must contain a copy of the function object itself, for recursive calls to the function. Thus, a circular data structure is necessary. As in the interpreter, an *rplaca* operation is used to replace the value representing the pending value of the environment, represented here by the term Ω , with the intended environment.

DUM creates the environment with the installed Ω placeholder.

The RAP instruction is similar to the AP instruction, except that in creating the environment in which the function is executed, the parameters are installed by using the *rplaca* operation on an environment whose *car* is the pending value Ω . RAP will always be executed in a state where the environment part of the function value, e' , is the same object as the contents of the environment register ($\Omega.e$). The list of parameters must consist of only function objects, and they will all contain the same environment component, so the single destructive operation creates a circular context for all the mutually recursive definitions.

3.4 Compiling LispKit to SECD Machine Code

The execution of LispKit programs on the SECD architecture requires their compilation into machine code. The compiled program, a function-valued object, loaded into the C register of the SECD machine, with an argument list loaded into the S register, should, upon completion of execution, leave a single result on the stack S, and that value should match the result of executing the LispKit program in the interpreter. The compiler will be defined by describing the translation of each well formed LispKit expression.

The machine code is generated with respect to a namelist, which is built up as the expression is compiled. This namelist is used to keep track of where values associated with variables will be found in the environment at execution time. Once again, a function is defined to extract a position for any variable in the namelist.

$$\begin{aligned} \text{location}(x, n) = & \\ & \text{if } (\text{member}(x, \text{car}(n))) \text{ then } (\text{cons}(0, \text{position}(x, \text{car}(n)))) \\ & \quad \text{else } (\text{cons}(\text{car}(z)+1, \text{cdr}(z))) \\ & \text{where } z = \text{location}(x, \text{cdr}(n)) \end{aligned}$$

Following the usage of Henderson, the definition in Table 3.2 makes use of the infix operator “|” to represent the *append* function. The namelist with respect to which the expression is being compiled is represented by “*n”.

A simple **variable** is compiled into code that loads the value from the environment list that is located in the corresponding location to the variable in the namelist. A **constant** value becomes a LDC instruction with the constant as argument.

An **arithmetic expression** compiles to a code sequence that first loads the values of the arguments, followed by the appropriate machine code instruction to execute the operation. The order of arguments is easily understood if we expect the LispKit expression (*SUB* e_1 e_2) to evaluate to $e_1 - e_2$. The code generated will cause the value of e_1 to be loaded on the top of the stack, and then the value of e_2 will be

LispKit expression	Compiled code
$x*n$	(LD i) where $i = \text{location}(x,n)$
$(\text{QUOTE } s)*n$	(LDC s)
$(\text{ADD } e_1 \ e_2)*n$	$e_1*n \mid e_2*n \mid (\text{ADD})$
$(\text{SUB } e_1 \ e_2)*n$	$e_1*n \mid e_2*n \mid (\text{SUB})$
$(\text{MUL } e_1 \ e_2)*n$	$e_1*n \mid e_2*n \mid (\text{MUL})$
$(\text{DIV } e_1 \ e_2)*n$	$e_1*n \mid e_2*n \mid (\text{DIV})$
$(\text{REM } e_1 \ e_2)*n$	$e_1*n \mid e_2*n \mid (\text{REM})$
$(\text{EQ } e_1 \ e_2)*n$	$e_1*n \mid e_2*n \mid (\text{EQ})$
$(\text{LEQ } e_1 \ e_2)*n$	$e_1*n \mid e_2*n \mid (\text{LEQ})$
$(\text{CAR } e)*n$	$e*n \mid (\text{CAR})$
$(\text{CDR } e)*n$	$e*n \mid (\text{CDR})$
$(\text{CONS } e_1 \ e_2)*n$	$e_2*n \mid e_1*n \mid (\text{CONS})$
$(\text{ATOM } e)*n$	$e*n \mid (\text{ATOM})$
$(\text{IF } e \ e_1 \ e_2)*n$	$e*n \mid (\text{SEL } e_1*n \mid (\text{JOIN}) \ e_2*n \mid (\text{JOIN}))$
$(\text{LAMBDA } (x_1 \ \dots \ x_k) \ e)*n$	(LDF $e*((x_1 \ \dots \ x_k).n) \mid (\text{RTN}))$
$(e \ e_1 \ \dots \ e_k)*n$	(LDC NIL) $\mid e_k*n \mid (\text{CONS}) \mid \dots \mid e_1*n \mid$ (CONS) $\mid e*n \mid (\text{AP})$
$(\text{LET } e \ (x_1.e_1) \ \dots \ (x_k.e_k))*n$	(LDC NIL) $\mid e_k * n \mid (\text{CONS}) \mid \dots \mid e_1*n \mid$ (CONS) $\mid (\text{LDF } e*m \mid (\text{RTN}) \ \text{AP})$ where $m = ((x_1 \ \dots \ x_k).n)$
$(\text{LETREC } e \ (x_1.e_1) \ \dots \ (x_k.e_k))*n$	(DUM LDC NIL) $\mid e_k*m \mid (\text{CONS}) \mid \dots \mid$ $e_1*m \mid (\text{CONS}) \mid (\text{LDF } e*m \mid (\text{RTN}) \ \text{RAP})$ where $m = ((x_1 \ \dots \ x_k).n)$

Table 3.4: SECD Machine Code Generated for Well-Formed Expressions

placed on top of that. Thus the stack looks like $(\text{val}(e_2) \ \text{val}(e_1).s)$, which clarifies why the machine transition definitions consider the top of stack as the second argument to the arithmetic operation. Notice that the *CONS* instruction is just the opposite, and the code sequence for the arguments is reversed in the resulting code sequence.

The **conditional expression** is compiled to a code sequence that first loads the value of the conditional part of the expression, followed by a *SEL* instruction, and

this in turn is followed by the code sequences for each of the two branches of the conditional, both of which end with a JOIN instruction.

A **function** defined by the use of *LAMBDA* is compiled into a control sequence that consists of a LDF instruction followed by the code for the function body compiled with respect to the namelist augmented with the list of locally bound variables, and with a RTN instruction appended to the end. The LDF instruction takes the control list for the function body as its argument, and creates a closure at run time by *cons*'ing this onto the current environment.

The code generated for a **function application** will build a list of values of each of the arguments to the function on top of the stack S. This is followed by the code for the function object being applied. This could consist of a nameless *LAMBDA* expression generating the code just seen, or the name of the function, which would also cause the function closure to be loaded on top of the stack with a LD instruction. The last instruction is AP, which will effect the application of the function to the arguments placed on the stack. It should be noted that a *LET* expression compiles into precisely the same code sequence as an application of a *LAMBDA* expression.

The final well formed LispKit expression is the **recursive block**. An initial DUM instruction modifies the current environment by installing a placeholder Ω value. This is followed by a code sequence that builds a list of the values to be bound to each of the local variables. Each of these is expected to be a function-valued object, and will thus contain a copy of the environment in its closure. The expression is treated as a function, with a LDF instruction and the code sequence for the expression followed by a RTN instruction. The RAP instruction is used instead of the AP instruction to create the required circular environment structure.

For completeness, Henderson's LispKit to SECD code compiler is included, written in LispKit in Table 3.5. For ease in reading, all LispKit keywords are upper case, while all locally bound values have lower case labels.

```

(LETREC compile
  (compile LAMBDA (e)
    (comp e (QUOTE NIL) (QUOTE (4 21))))

  (comp LAMBDA (e n c)
    (IF (ATOM e)
      (CONS (QUOTE 1) (CONS (location e n) c))
      (IF (eq (CAR e) (QUOTE QUOTE))
        (CONS (QUOTE 2) (CONS (CAR (CDR e)) c))
        (IF (EQ (CAR e) (QUOTE ADD))
          (comp (CAR (CDR e)) n (comp (CAR (CDR (CDR e))) n (CONS (QUOTE 15) c)))
          (IF (EQ (CAR e) (QUOTE SUB))
            (comp (CAR (CDR e)) n (comp (CAR (CDR (CDR e))) n (CONS (QUOTE 16) c)))
            (IF (EQ (CAR e) (QUOTE MUL))
              (comp (CAR (CDR e)) n (comp (CAR (CDR (CDR e))) n (CONS (QUOTE 17) c)))
              (IF (EQ (CAR e) (QUOTE DIV))
                (comp (CAR (CDR e)) n (comp (CAR (CDR (CDR e))) n (CONS (QUOTE 18) c)))
                (IF (EQ (CAR e) (QUOTE REM))
                  (comp (CAR (CDR e)) n (comp (CAR (CDR (CDR e))) n (CONS (QUOTE 19) c)))
                  (IF (EQ (CAR e) (QUOTE LEQ))
                    (comp (CAR (CDR e)) n (comp (CAR (CDR (CDR e))) n (CONS (QUOTE 20) c)))
                    (IF (EQ (CAR e) (QUOTE EQ))
                      (comp (CAR (CDR e)) n (comp (CAR (CDR (CDR e))) n (CONS (QUOTE 14) c)))
                      (IF (EQ (CAR e) (QUOTE CAR))
                        (comp (CAR (CDR e)) n (CONS (QUOTE 10) c))
                        (IF (EQ (CAR e) (QUOTE CDR))
                          (comp (CAR (CDR e)) n (CONS (QUOTE 11) c))
                          (IF (EQ (CAR e) (QUOTE ATOM))
                            (comp (CAR (CDR e)) n (CONS (QUOTE 12) c))
                            (IF (EQ (CAR e) (QUOTE CONS))
                              (comp (CAR (CDR (CDR e))) n (comp (CAR (CDR e)) n (CONS (QUOTE 13) c)))
                              (IF (eq (CAR e) (QUOTE IF))
                                (LET (comp (CAR (CDR e)) n (CONS (QUOTE 8)
                                  (CONS thenpt (CONS elsept c))))
                                  (thenpt comp (CAR (CDR (CDR e))) n (QUOTE (9)))
                                  (elsept comp (CAR (CDR (CDR (CDR e)))) n (QUOTE (9)))) )
                                (IF (EQ (CAR e) (QUOTE LAMBDA))
                                  (LET (CONS (QUOTE 3) (CONS body c))
                                    (body comp (CAR (CDR (CDR e))) (CONS (CAR (CDR e)) n) (QUOTE 5)))
                                  (IF (EQ (CAR e) (QUOTE LET))
                                    (LET (LET (complis args n (CONS (QUOTE 3) (CONS body (CONS (QUOTE 4) c))))
                                      (body comp (CAR (CDR e)) m (QUOTE 5)))
                                      (m CONS (vars (CDR (CDR e))) n)
                                      (args exprs (CDR (CDR e))))
                                    (IF (EQ (CAR e) (QUOTE LETREC))
                                      (LET (LET (CONS (QUOTE 6)
                                        (complis args m (CONS (QUOTE 3) (CONS body (CONS (QUOTE 7) c))))
                                        (body comp (CAR (CDR e)) m (QUOTE 5)))
                                        (m CONS (vars (CDR (CDR e))) n)
                                        (args exprs (CDR (CDR e))))
                                      (LET (LET (CONS (QUOTE 4) c)))))))))) )
                                (complis (CDR e) n (comp (CAR e) n (CONS (QUOTE 4) c))))))))) )

    (complis LAMBDA (e n c)
      (IF (EQ e (QUOTE NIL)) (CONS (QUOTE 2) (CONS (QUOTE NIL) c))
        (complis (CDR e) n (comp (CAR e) n (CONS (QUOTE 13) c)))))

  (location LAMBDA (e n)
    (LETREC
      (IF (member e (CAR n)) (CONS (QUOTE 0) (posn e (CAR n)))
        (incor (location e (CDR n))))
      (member LAMBDA (e n)
        (IF (EQ n (QUOTE NIL)) (QUOTE F)
          (IF (EQ e (CAR n)) (QUOTE T) (member e (CDR n)))))
      (posn LAMBDA (e n)
        (IF (EQ e (CAR n)) (QUOTE 0) (ADD (QUOTE 1) (posn e (CDR n)))))
      (incor LAMBDA (l) (CONS (ADD (QUOTE 1) (CAR l)) (CDR l)))))

  (vars LAMBDA (d)
    (IF (EQ d (QUOTE NIL)) (QUOTE NIL)
      (CONS (CAR (CAR d)) (exprs (CDR d)))))

  (exprs LAMBDA (d)
    (IF (EQ d (QUOTE NIL)) (QUOTE NIL)
      (CONS (CDR (CAR d)) (exprs (CDR d)))))
)

```

Table 3.5: LispKit to SECD Compiler Written in LispKit

3.5 Summary

This chapter has described the LispKit language, and used it to illustrate the operation of the SECD architecture transitions. The LispKit syntax was given, and an informal semantics provided in the form of an interpreter function. The abstract SECD architecture was defined by giving the set of SECD language instructions, and defining state transitions for each instruction. A translation schema for LispKit into SECD code was described, and its implementation as a compiler written in LispKit supplied. This translation along with the SECD definition provides an operational semantics for LispKit. A proof of correctness of the translation and a full description of the semantics of LispKit may be found in [SGB89]. While not central to this work, the introduction of a higher level language demonstrates the operation and potential of the SECD architecture. More extensive programming examples in LispKit are available, for example [HJJ83a, HJJ83b]. Furthermore, the simple translation from the higher level language to SECD code demonstrates the suitability of the architecture for executing functional programs.

The following chapter proceeds to develop the architecture in greater detail, coming eventually to the design of a working hardware system.

Chapter 4

SECD Architecture: Silicon Synthesis

A wide variety of different implementations can satisfy the informal definition of the SECD machine given in the previous chapter. The choice of implementation was not driven entirely by the abstract specification, but evolved in concert with other criteria that constrained the design.

This chapter focuses on two major themes: development of the external architecture (the machine as seen by its users), largely influenced by external constraints and decisions, and development of the internal architecture (how the machine is physically organized), through a progressive elaboration of the system model. Both are developed within a framework of increasingly detailed levels of description of the system.

4.1 Project Context

The SECD chip arose within a larger ongoing research effort by the VLSI group at the University of Calgary. The chip was used as a vehicle to explore the use of specification to drive design synthesis. The methodology entails elaborating a design hierarchically as a tree of nodes and formally specifying the behaviour at each node. Verifying that the composition of behaviours of a node's children agrees with the node's specification assures a correct design. By deductive argument, the correctness or otherwise of a complete design can be shown. While the chip provided the team with hands-on experience with a nontrivial design, the focus of study was the design process, and this strongly influenced the dominant design criteria.

- The most important criterion was that a *correct, working device* be produced. Correctness is the primary objective of the specification-driven process.
- The next criterion was *simplicity*. Simplicity was necessary on two counts: to ensure that verification could cope with what promised to be the most complex microprocessor proof attempted to date, and secondly, to improve the likelihood of meeting the first criterion.
- *Testability* of the design was considered essential. In the event of malfunction, determination of the source of the error requires examining the state of the machine extensively. Furthermore, correct output from test problems does little to assure total design correctness. Rather, each step of the computation should be accessible for examination.
- Lastly, *utility* should be considered. It was preferable that the design could be given tractable problems, rather than be considered a toy device, incapable of all but the most trivial tasks. A particularly relevant problem would be compiling LispKit programs to SECD code.

Equal in importance to the selection of criteria is the explicit statement of items that will not be given priority. *Speed* was specifically eliminated as a determining criterion, both in terms of clocking rate, and optimality of the operation sequences, insofar as they could conflict with the simplicity criterion.

4.2 Levels of the Design

The wide gulfs between the external architecture view, captured abstractly in the top level specification, the internal architecture, expressed as an assembly of logical devices, and the layout, expressed as a set of masks, are most easily bridged by a succession of levels of description, with detail increasing at each lower level. Associated

with each level is an interpreter expressed as a simulation model. The simulation model has both a control structure, and a set of operations. The operations may be further expanded at the next lower level. There were seven major major levels of description used in the development of the SECD chip.

Abstract machine is the high level definition of the machine characterized by the contents of 4 stacks, defined by the state transitions of Table 3.3.

Abstract System level views the SECD machine as a batch mode co-processor, with external 'read' and 'print' routines to download problems and return results. The decomposition of transitions into operation sequences begins here.

Top level FSM (finite state machine) describes the control of the system in terms of major states and transitions, introducing control inputs for initialization and state transition.

Abstract RTL (register transfer level) view begins elaborating the internal architecture, adding registers, combinational logic devices, the bus, and memory, and determining data representation and word configuration.

Concrete RTL develops the control part of the internal architecture as a microcode program.

Mossim defines the design down to the transistor level, and determines the design of memory elements and the clocking scheme.

Layout level generates a set of masks, after resolving floorplanning, electrical and similar low level concerns.

The first 3 levels of description mainly develop the external architecture, while the remaining levels are concerned with developing the internal architecture.

Several programs from Gabriel [Gab85] and Henderson [Hen80] were run on all simulation models above the Mossim level. The largest test was the compilation

of the LispKit compiler from [Hen80]. A smaller LispKit program with 3 mutually recursive functions was used to test the Mossim models of the control unit and datapath.

4.3 External Architecture

The top level definition of the SECD machine given in [Hen80] (and shown in Table 3.3) defines a set of machine transitions, one for each of the 21 SECD machine instructions, in terms of contents of the four stacks. This concise specification hardly begins to define how a machine would operate. The initial development of the SECD chip filled in the external architecture definition, bridging the gap between the abstract machine and a workable specification for a hardware system.

4.3.1 Abstract Machine

In order to gain a better understanding of the way high level constructs in the source language (LispKit) were implemented by the compiled machine instructions, the abstract machine was modelled by an interpreter written in Franz Lisp ([HBGS89]), together with a LispKit compiler ([SBGH89]). Error checking on operation arguments was performed, and run-time statistics on instruction counts and environment accesses were recorded. This stage was simply a learning exercise, and did not seek to flesh out the implementation design. Thus, the interpreter did not need to define special data structures or elemental machine operations, but rather relied entirely upon the Franz Lisp data structure representation, as well as the *cons*, *car*, and *cdr* operations and the five arithmetic operations that SECD uses. Furthermore, fundamental implementation concerns were ignored, with recursive functions used to locate values in the environment list, and resource management (*i.e.* garbage collection of records) completely omitted.

4.3.2 Abstract System: the First Refinement

Realizing the SECD machine as a working system required that it be able to accept a task, compute a result, and return it. The abstract machine definition required that problems be in the form of a function to be applied to a list of arguments. The interface to permit a user to pose a problem and the machine to return a result was the first major design decision. Two major options were considered: a co-processor role and a stand-alone system.

Using the SECD as a *co-processor* to another system would permit i/o to be handled by the other system rather than the SECD chip. For instance, using SECD as a co-processor for a SUN workstation would see the SUN able to read in an S-expression, set up a memory image for the problem, signal the SECD to begin computation, receive a signal back on completion of the calculation, and print out the S-expression solution. This has the advantage of simplifying the tasks the SECD must perform.

Implementing the SECD chip as a *stand-alone system* would have required incorporating primitive read and write operations into the definition of the machine, and defining an operating system, ideally written in the higher level LispKit language. The infinite “while” loop required for an operating system could only be implemented using LETREC, but as the system was defined, this was not possible since each new level of recursion uses up more memory, so eventually system resources would be exhausted. Thus further modifications would be needed.

The co-processor option was chosen as most appropriate to the scope of the project. Major phases in the operation of the system are: 1) **load problem** into RAM (done by main processor), 2) send **start signal** to SECD system, 3) **run**, 4) stop and **signal completion** to main processor, and 5) **return result** (again, main processor task).

Representation of S-expressions (the “stuff” of programs and data in the SECD machine) is not determined at this level, but it is known that they will be stored within a finite memory, and this necessitates *garbage collection*. A simple *mark and sweep* garbage collector is used, and a “memory exhausted” error can arise. A simulation at this level (written in “C”) used external *read* and *print* routines to model the ‘load problem’ and ‘return result’ tasks of the system. These functions prepare a memory image within a finite memory data structure and extract an S-expression from a memory image respectively. The SECD system was modelled making free use of high level language constructs including complex data types to represent data records, and recursive tree traversal in the garbage collector *mark* routine.

When designing a control sequence for the simulation, a precedence (partial ordering) was established on registers for each machine instruction to prevent overwriting. As an example, consider generating a control sequence for the AP instruction transition. The abstract machine transition is given by:

$$((c'.e')v.s) \ e \ (AP.c) \ d \rightarrow \text{NIL} \ (v.e') \ c' \ (s \ e \ c.d)$$

from which the following precedence on registers is observed:

$$D \prec \begin{matrix} E \\ C \end{matrix} \prec S.$$

The control sequence then takes the following form.

$$\begin{aligned} D &= (\text{cons} (\text{cdr} (\text{cdr} S)) (\text{cons} E (\text{cons} (\text{cdr} C) D))) \\ C &= (\text{car} (\text{car} S)) \\ E &= (\text{cons} (\text{car} (\text{cdr} S)) (\text{cdr} (\text{car} S))) \\ S &= \text{NIL} \end{aligned}$$

4.3.3 The Top FSM Level

The previous abstract views of the SECD system were concerned primarily with manipulating the S-expressions in the four stacks. The finite state machine view concerns the development of a control interface for operating the system. The top level FSM has only four states, and the earlier view of the machine's state as being represented by the contents of the 4 main stacks (S, E, C, and D) is incorporated as annotations to transitions where these stack contents change (see Figure 4.1).

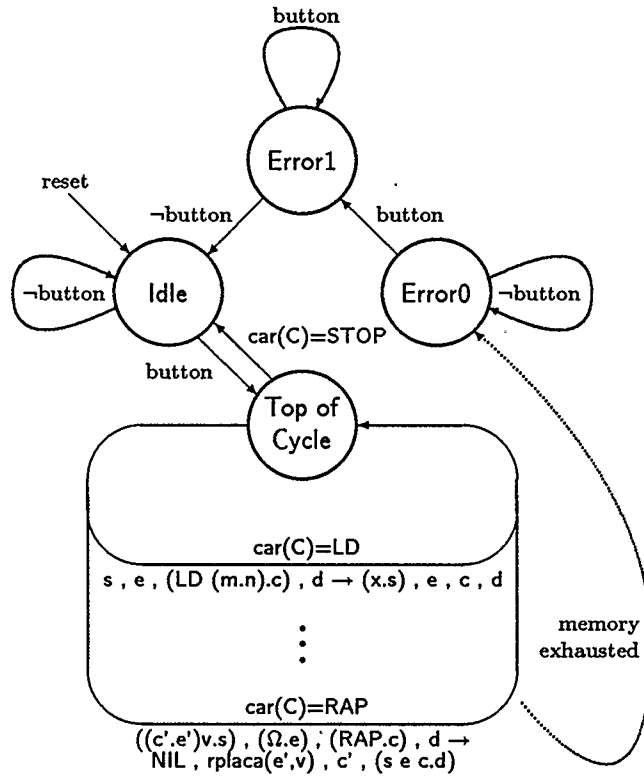


Figure 4.1: Top Level Finite State Machine View of SECD

The four states are *Idle*, *Top of Cycle*, and two *Error* states. The *Idle* state permits the chip to be turned on but prevents it from starting computation until signalled after a problem has been loaded. It also permits completion of the com-

putation to be recognizable. The concept of control state suggests the existence of state values as outputs, particularly for this purpose. The *Idle* state also introduces the idea of repeated executions, as opposed to the earlier models which permitted single computations only.

The *Top of Cycle* state corresponds to the state the machine is in at the start of execution of any SECD machine instruction. There are 21 transitions leading from this state, one for each machine instruction, forming an instruction fetch/execute style of loop. The abstract machine transition for the STOP instruction is clarified by transferring to the idle state, instead of looping infinitely. The *Error* states are only entered upon exhaustion of memory. This is a necessary error condition, consistent with the previous level view.

An external input labelled *button* controls the state transitions from the *Error* and *Idle* states. The use of separate inputs could have eliminated the need for the second error state at this level¹, but a concern over the number of pins available for inputs mandated a single signal. An external *reset* input has also been included to permit a deterministic startup of the machine. Simulations to this point have always begun with the same sequence of operations, and modelling the controller as a finite state machine similarly required selection of the startup state. The assumption of this initial state was most simply implemented by the *reset* input, along with a constraint that the reset be asserted in the initial clock cycle.

This level is not independent from the abstract system or abstract register transfer views of the SECD. Instead, it is a particular view of the system that is useful in formalising the behaviour of the system. Nor is a simulation model directly related to this level. Instead, we see the next level simulation incorporating the notion of major state by adding a state value at suitable points in the control sequence.

¹Requiring distinct values on incoming transitions from those for outgoing transitions from the state make the signal timing less critical.

4.4 Internal Architecture

The Top FSM Level describes the external architecture control interface, but leaves some other aspects of the external architecture unresolved, such as data representation, clocking, and external memory interface. Each of these are affected by decisions taken at lower levels of design, and are determined in concert with the internal architecture.

4.4.1 The Abstract Register Transfer Level

The internal architecture of the SECD chip begins development with the definition of the Register Transfer Level view. The SECD machine at this stage is seen as a set of registers, combinational logic units, a bus linking the components, and a memory. The state transformations are effected by shifting values between registers and memory, using combinational logic to perform such functions as *cons*, *car*, and *cdr*. The sequence of operations required is the model for the controller, which at this stage still retains some higher level control structures such as “if ... then ... else”, “case”, and “while”.

The S-expression data type, which is the “stuff” of SECD machine programs and data, is composed of three types of objects: numbers, symbols, and cons records. A simple mark and sweep garbage collector required the use of two bits in each record. Two additional bits indicate the “type” each record contains.

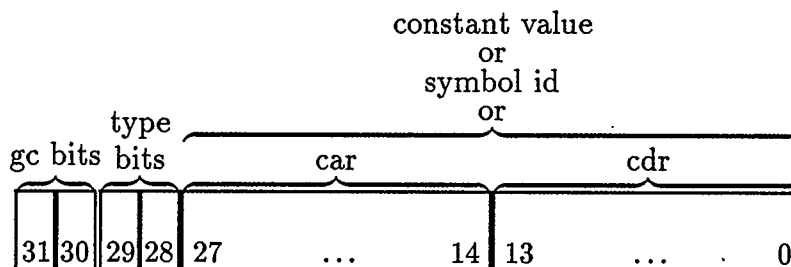
Numbers were permitted to range over integers (rather than the natural numbers), consistent with the definition of SECD.

Symbols represent atomic values which can only be tested for equality with each other. Thus, a distinct symbol identification number is a suitable representation. The “meaning” of the symbol (or its written form) is of concern only on input and output operations, and hence assignment and interpretation can

be handled entirely outside the SECD chip (by the compiler, since new symbols cannot be created in the course of executing programs). Three symbolic constants (NIL, TRUE, and FALSE) are required by the SECD chip, and were “built-in” at this stage.

Cons records represent a pairing operation of S-expressions. In typical Lisp fashion, these are implemented by pairs of pointers to other cells. The size of pointer determines the memory address space, and hence the maximum problem size that the machine can compute. As a minimum, it was felt that the SECD machine should be able to run the Lispkit compiler on Lispkit programs, and this required approximately 2^{12} words. This set a lower bound of a 28 bit word $((2 \times 12) + 4)$. The availability of memories in multiples of 8 bits made 32 bit words an appealing choice.

The final word configuration is as follows:



The S, E, C, and D stacks are implemented as 14 bit registers that contain pointers to S-expressions in memory. The free list, used to allocate unused cells as required by the computation process, is similarly implemented by a register holding a pointer to the free list in memory. Further registers were added as their need was determined. Working registers *x1* and *x2* were added, to permit computation of intermediate results as arguments to a *cons* operation. A memory address register (*mar*) was added to select memory locations. A 32 bit *arg* register was added to hold integer or symbol arguments for alu operations, and generally for holding 32

bit records read from memory, including the machine instruction codes. The output of the *alu* is connected to two 32 bit buffer registers. The 32 bit *buf1* register is necessary since the integer and symbol inputs to *alu* operations come from the *arg* register and the *bus*, and the 32 bit output must be written to some other register. The second buffer, *buf2* is used only by the *mark* routine of the garbage collector, to prevent any loss of an arithmetic result being held in *buf1*.

With each addition to the hardware, a functionality or role was determined, and thereafter this functionality was respected. Non-transparent uses of components was avoided, with the expectation that this would make the verification task more manageable. The clearest indication of this approach is the provision of separate registers: *root*, *parent*, *y1*, and *y2* (in addition to *buf2* mentioned above) for use by the garbage collector.

The description of this level is contained in a simulation characterized by data records consisting of 32 bits, 14 bit pointers, a fixed set of registers, and the modelling of memory, control and combinational logic elements by high level routines. Table 4.1 shows the code for the AP instruction for this level simulation.

The instruction sequence in the simulation was systematically derived from the previous abstract system level simulation. The bus function used for all data transfers models a single bus architecture, which was selected for simplicity. The three data structure operations are translated as follows:

car(z):	bus(mar = z); bus(x1 = carvalue(memory[mar]));
cdr(z):	bus(mar = z); bus(x1 = cdrvalue(memory[mar]));
cons(z1,z2):	bus(x2 = z2); bus(x1 = z1); consx1x2();

Five combinational logic elements are indicated. These are the ALU, the flagsunit, the consunit, and the carvalue and cdrvalue units. The latter three implement the primitive operations on records. The first two require further elaboration.

	! x2 := (cons (cdr c) d)
bus(x2=d);	
bus(mar=c);	
bus(x1=cdrvalue(memory[mar]));	
consx1x2();	! resulting cell address is in mar
bus(x2=mar)	
	! x2 = (cons e x2)
bus(x1=e);	
consx1x2();	
bus(x2=mar);	
	! d = (cons(cdr(cdr s)) x2)
bus(mar=s);	
bus(x1=cdrvalue(memory[mar]));	
bus(mar=x1);	
bus(x1=cdrvalue(memory[mar]));	
consx1x2();	
bus(d=mar);	
	! e = (cons (car(cdr s)) (cdr(car s)))
bus(mar=s);	
bus(x2=carvalue(memory[mar]));	
bus(mar=x2);	
bus(x2=cdrvalue(memory[mar]));	
bus(mar=s);	
bus(x1=cdrvalue(memory[mar]));	
bus(mar=x1);	
bus(x1=carvalue(memory[mar]));	
consx1x2();	
bus(e=mar);	
	! c = (car(car s))
bus(mar=s);	
bus(c=carvalue(memory[mar]));	
bus(mar=c);	
bus(c=carvalue(memory[mar]));	
	! s = NIL
bus(s=NIL);	

Table 4.1: Initial RTL Microcode Sequence for AP Instruction

The ALU primary function is the computation of values for the arithmetic SECD machine operations: ADD, SUB, MUL, DIV, and REM. Additional operations were required for the garbage collector, including setting and clearing of the mark and field bits, and the destructive *replcar* and *replcdr* operations used for the in-place traversal

of the data structures in memory. These operations were masked previously by recursive functions implementing the garbage collector. Lastly, there was a *decrement* operation, used in looking up values in the environment. It was also used in the “sweep” phase of garbage collecting to step through the memory address space. The high address value was built-in as a constant to provide a starting point for the sweep. The two uses had distinct data type arguments: the first used integers, while the second was applied to addresses. Thus, the 14 bit addresses had to be padded out with zeros to make 28 bit integers. For this purpose, a constant register (*the clearunit*) loads zeros onto the upper 14 bits of the bus when required.

The flagsunit returns the boolean result of predicates used both for the control of the *if ... then ... else* and the *while* structures, as well as computing the SECD machine operations EQ and LEQ. Figure 4.2 summarises the operational part of the architecture at this level, and pictures the control part as a classic finite state machine.

4.4.2 The Concrete Register Transfer Level

The next level of refinement concentrated on transforming the control sequence into the final series of microcode instructions. The transformation was accomplished in several steps, and the final resulting sequence was then compiled into a binary image used to generate a microcode ROM.

A microcode sequence was generated from the abstract register transfer level model by mechanically translating each of the 4 higher level functions into an instruction sequence as follows:

bus(z = w)	→	rw wz
bus(z = carvalue(memory[mar]))	→	rmem wcar ; rcar wz
bus(z = cdrvalue(memory[mar]))	→	rmem wz
consx1x2()	→	call(Consx1x2,\$)

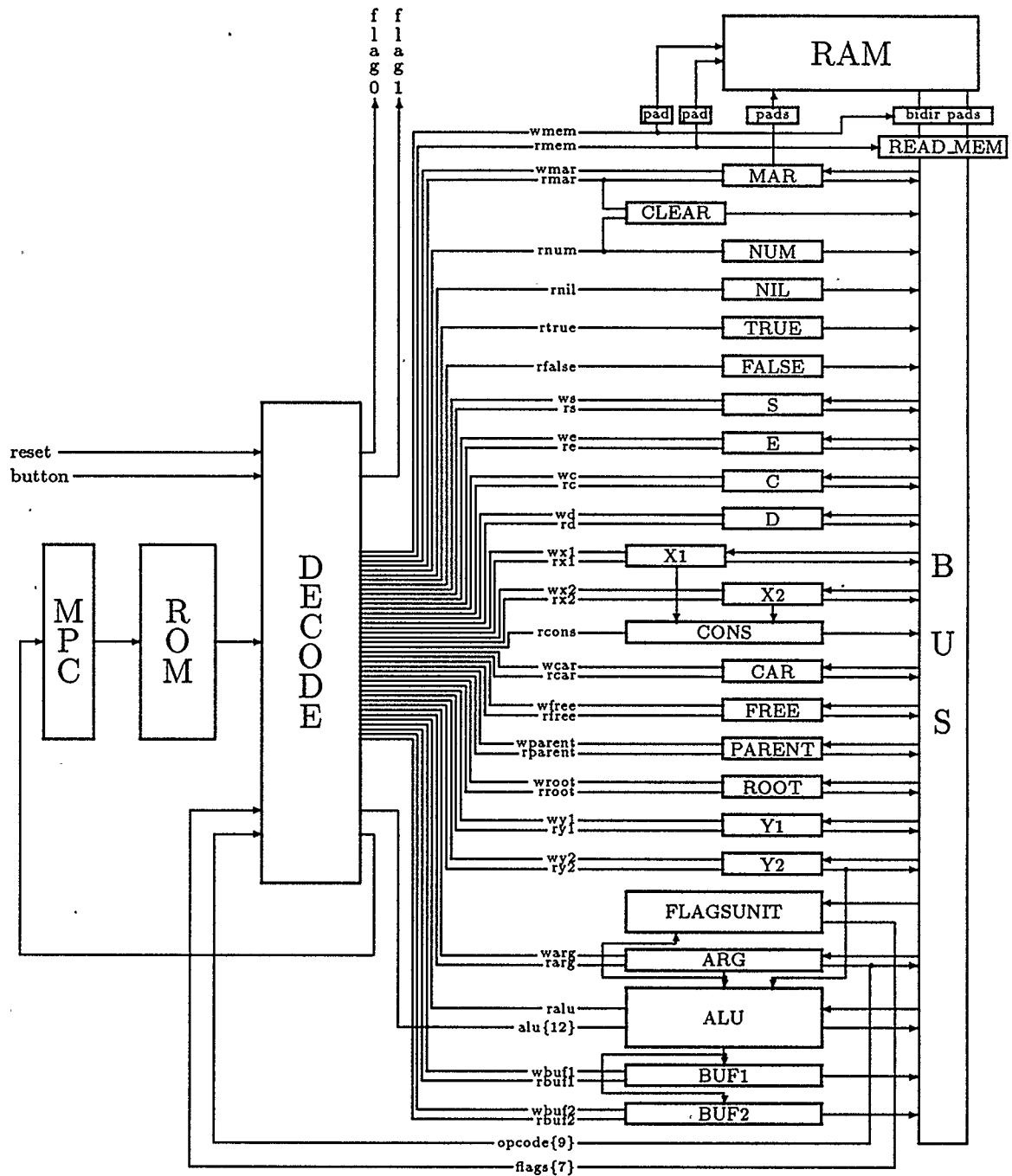


Figure 4.2: Register Transfer Level View of SECD Machine

A simple transfer of values on the bus became simultaneous read and write signals to the appropriate registers. The *car* operation required that the word be fetched from memory, and its *car* field be accessed by writing it first to the *car* register, and thence transferring it on the bus to the desired register. It was assumed that the *cdr* field could be written directly from the memory to the selected register. Finally, the *cons* operation was called from so many different locations that it was treated as a subroutine call, with the following microcode location as the second parameter to enable a return from the subroutine on completion. The new control sequence for the AP instruction is shown in Table 4.2.

The set of datapath register (and memory) control signals was now established, as were the 12 ALU operations given in the previous level. Control mechanisms for the microcode (ie. the third column in the above microcode sequence) consisted of five types: unconditional jumps, conditional jumps, subroutine calls, subroutine returns, and a jump table that uses the current machine instruction value. Of these, the conditional jumps and subroutine calls both required two addresses, while the others required a single address argument. The subroutine mechanism required a microaddress stack. Eight conditional jump instructions had conditions:

- the value of the *button* input
- is the argument an *atom* record
- the EQ operation applied to 2 arguments
- the LEQ operation applied to 2 arguments
- is record equal to the symbolic constant NIL
- is the record equal to the symbolic constant TRUE
- is the *mark* bit set
- is the *field* bit set

! D :				
112,	rd	wx2	(jump 113)	! bus(x2=d);
113,	rc	wmar	(jump 114)	! bus(mar=c);
114,	rmem	wx1	(jump 115)	! bus(x1=cdvalue(memory[mar]));
115,			(call ("Consx1x2", 116))	! consx1x2();
116,	rmar	wx2	(jump 117)	! bus(x2=mar);
117,	re	wx1	(jump 118)	! bus(x1=e);
118,			(call ("Consx1x2", 119))	! consx1x2();
119,	rmar	wx2	(jump 120)	! bus(x2=mar);
120,	rs	wmar	(jump 121)	! bus(mar=s);
121,	rmem	wx1	(jump 122)	! bus(x1=cdvalue(memory[mar]));
122,	rx1	wmar	(jump 123)	! bus(mar=x1);
123,	rmem	wx1	(jump 124)	! bus(x1=cdvalue(memory[mar]));
124,			(call ("Consx1x2", 125))	! consx1x2();
125,	rmar	wd	(jump 126)	! bus(d=mar);
! E :				
126,	rs	wmar	(jump 127)	! bus(mar=s);
127,	rmem	wcar	(jump 128)	
128,	rcar	wx2	(jump 129)	! bus(x2=carvalue(memory[mar]));
129,	rx2	wmar	(jump 130)	! bus(mar=x2);
130,	rmem	wx2	(jump 131)	! bus(x2=cdvalue(memory[mar]));
131,	rs	wmar	(jump 132)	! bus(mar=s);
132,	rmem	wx1	(jump 133)	! bus(x1=cdvalue(memory[mar]));
133,	rx1	wmar	(jump 134)	! bus(mar=x1);
134,	rmem	wcar	(jump 135)	
135,	rcar	wx1	(jump 136)	! bus(x1=carvalue(memory[mar]));
136,			(call ("Consx1x2", 137))	! consx1x2();
137,	rmar	we	(jump 138)	! bus(e=mar);
! C :				
138,	rs	wmar	(jump 139)	! bus(mar=s);
139,	rmem	wcar	(jump 140)	
140,	rcar	wc	(jump 141)	! bus(c=carvalue(memory[mar]));
141,	rc	wmar	(jump 142)	! bus(mar=c);
142,	rmem	wcar	(jump 143)	
143,	rcar	wc	(jump 144)	! bus(c=carvalue(memory[mar]));
! S :				
144,	rnil	ws	(jump 145)	! bus(s=NIL)
145,			(jump "top_of_cycle")	

Table 4.2: Intermediate RTL Microcode Sequence for AP Instruction

These values are the flagsunit outputs. Additionally, the datapath also sends the current machine instruction value to the controller. The controller thus has 12 distinct ways of selecting the next microinstruction address.

This evolution was suitable for mechanical rewriting, ignoring optimizations that would be obvious to the reader. The final microcode evolved through several successive refinements, aimed at reducing the ROM size. The major optimizations include:

- A peephole optimization run eliminated unnecessary bus transfers using the working *x1* or *x2* registers, when the value could be transferred directly to the required register in one operation.
- The high memory address was used to hold a pointer to a downloaded problem in memory, and to hold a pointer to the result on completion of computation. This used the high address constant built in previously for use by the garbage collector.
- An additional level of subroutining was added to share common code sequences in arithmetic and logical SECD machine instruction sequences.
- Slight revision to the ordering of conditional jump instructions was made so that the following microinstruction was always the default next address. This permitted the elimination of this default address field as an explicit value in each microcode instruction, since the only other instruction using two addresses was the subroutine call, and the return address parameter to a subroutine call was always the immediately following address as well. Thus, at most one address field was required in each microinstruction. The unconditional jump instruction was divided into remote jumps and jumps to the following microinstruction, with the latter not needing a specified argument. This increased the number of ways of choosing the next **mpc** (microprogram counter) contents to 13.

The final version of the microcode for the AP instructions is shown in Table 4.3.

Our modelling of the chip implicitly assumes an external RAM, since an outside agency is expected to download problems and upload results, and there is no provi-

```

L("AP");
/* D */
    rd      ; wx2                ; (inc ()) ;
    rc      ; wmar               ; (inc ()) ;
    rmem    ; wx1                ; (call ("Consx1x2")) ;
    rmar    ; wx2                ; (inc ()) ;
    re      ; wx1                ; (call ("Consx1x2")) ;
    rmar    ; wx2                ; (inc ()) ;
    rs      ; wmar               ; (inc ()) ;
    rmem    ; wx1                ; (inc ()) ;
    rx1     ; wmar               ; (inc ()) ;
    rmem    ; wx1                ; (call ("Consx1x2")) ;
    rmar    ; wd                 ; (inc ()) ;

/* E */
    rs      ; wmar               ; (inc ()) ;
    rmem    ; wcar               ; (inc ()) ;
    rcar    ; wmar               ; (inc ()) ;
    rmem    ; wx2                ; (inc ()) ;
    rs      ; wmar               ; (inc ()) ;
    rmem    ; wx1                ; (inc ()) ;
    rx1     ; wmar               ; (inc ()) ;
    rmem    ; wcar               ; (inc ()) ;
    rcar    ; wx1                ; (call ("Consx1x2")) ;
    rmar    ; we                 ; (inc ()) ;

/* C */
    rs      ; wmar               ; (inc ()) ;
    rmem    ; wcar               ; (inc ()) ;
    rcar    ; wmar               ; (inc ()) ;
    rmem    ; wcar               ; (inc ()) ;
    rcar    ; wc                 ; (inc ()) ;

/* S */
    rnil    ; ws                 ; (jump ("top_of_cycle")) ;

```

Table 4.3: Final Microcode Sequence for AP Instruction

sion in the model for handing control of the memory to the external agency. External RAM is consistent with the simplicity criterion, and focussed our effort on the microprocessor design, rather than the distinct concerns of RAM design. The RAM is treated as just another, though addressable, register; with read and write signals

controlling it. It was expected that the RAM would default to a read operation, and the `rmem` control line would control its gating onto the bus.

This implementation of the controller is a classical finite state machine design, with the state held in the `mpc` register (this is extended to include the microcode stack registers as well), changing with each microcode instruction executed.

4.4.3 The Mossim Level

The register transfer level defines both a control part (the microcode) and an operative part (the registers, logical units, bus, and memory). The given operations can be completed within a clock cycle, and indeed, the time quantum of the model matches the clock frequency. The next level will model the SECD down to transistors, where the design of the memory devices and the clocking scheme are the major issues.

The simulation used at this level is an implementation of Randall Bryant's Mossim simulator, written in C by Jeff Joyce, with a Common Lisp interface, called CDL, written by Breen Liblong. It models the circuit as a network of asynchronously operating switches, which settles to a stable state between changes in clock inputs. The complete CDL definition of the SECD chip is available in [GWB⁺89]. The Mossim level model was used to capture component design for simulation, and also guided the actual layout. Layout decisions determined the Mossim definition, and the Mossim model was used as a suitable form to define the components then implemented in the layout.

Previous views of the SECD divided it into two major functional components: the controller and the datapath. As described earlier, these two parts were developed somewhat independently, and it was felt that they should be independently testable. If a flaw occurred in one component, testing of the other component would still be possible. To meet this objective, and previously stated testability concerns, it was decided to add a bank of shift registers between the two components, which could

be used to trap all, or most, signals passing between them.² The largest Mossim simulation did not include the shift registers. The shift registers are intended to be transparent during normal operation, and their inclusion would have made the simulation too massive. The shift registers were specified and tested independently however.

4.4.3.1 Memory Elements and Clocking

Level-triggered latches were selected largely for reasons of space and transistor count efficiency. Level triggered latches are also in keeping with the view of circuits presented in [MC80] as a system of opening and closing valves. In some sense, level triggered latches can be viewed as falling edge triggered, although the previous state is lost at the start of the clock pulse.

Implementing the controller as a finite state machine requires buffering between current and next states. This was achieved by the use of a two-phase non-overlapping clocking scheme and paired master/slave registers, along the design style described in [MC80]. The state register in the control unit is the **mpc** register, but in a more general sense, the values on the 4-deep microcode subroutine stack are also part of the state. In the following discussion, references to the **mpc** register can be applied similarly to the stack registers. The **mpc** register is actually the slave register, the master is labelled **nextmpc**. **Nextmpc** is clocked on the ϕ_A phase, and **mpc** on the ϕ_B phase. The control unit state is considered to change on ϕ_B . Using the terminology of Anceau [Anc86], this arrangement for synchronization of the circuit uses “mixed polyphase mode”, and will be stable if a wait state is present in the clock cycle. The short time required to pass values from *nextmpc* to *mpc* along with a sufficiently low frequency ensures a wait before ϕ_B rises.

²Detailed listings of signals in the shift registers are given in [GWS89].

Particular attention was paid to possible *race* conditions. One example was the possibility of generating transient “write” signals from the ROM when the value `mpc` is changing. The solution was to delay latching of the datapath registers until after the `mpc` is latched (and the value propagated). Use of the inverse clock signal ($\overline{\phi_B}$) was still subject to race conditions³, so the ϕ_A phase was used. The clocking scheme requires that inputs to registers latching on ϕ_A be stable prior to the end of the ϕ_A pulse. The overall view of the chip now sees the control unit as changing state on ϕ_B and the datapath changing on ϕ_A .

The first layout iteration, and the Mossim simulation, really did not properly account for the operation of the external memory. It was expected that memory outputs would float unless the `rmem` signal were asserted. Capacitance induced delays of signal switching and power dissipation caused by both memory and SECD devices driving external lines simultaneously for some overlapping interval were seen as potential problems later. Thus, a more considered memory interface timing was developed (described in detail in [GWS89]), and the new control signal logic added in a design revision.

4.4.3.2 Control Unit

Several key decisions directly affected the design of the control unit:

- the encoding of the microcode
- the interface between the control unit and datapath.
- the choice of signals trapped by shift registers.

A simple view of the ROM has a fully horizontal microcode, with discrete outputs for each read, write, and alu control signal (23, 17, and 12 signals respectively).

³One could devise a scenario where the $\overline{\phi_B}$ signal overlaps the start of the ϕ_B pulse, and thus random write signals may be generated.

Further, the address field required by the goto, subroutine call, and conditional jump instructions was nine bits (since the microcode length was approximately 400 instructions). Lastly, the method of selection of the next microinstruction has 13 possibilities. A fully horizontal microcode ROM would be $9 \times 400 \times 74$. With a square pitch of 12.5λ for the ROM layout, in the 3μ process, the ROM size was approximately 2.06×7.5 mm, excluding routing to and from, and buffering of inputs and outputs. While this size ROM might fit on the chip, it was felt that it could be reduced considerably, and in the process line capacitance would also be lowered, thereby providing a higher probability of reliable operation.

Microcode characteristics were thus re-examined in the search for an encoding scheme. The mutually exclusive assertion of individual read, write, alu, and test signals during any cycle suggested these signals should be encoded. The number of distinct combinations of control signals in a microinstruction numbered approximately 120, while the number of distinct combinations of read and write signals numbered approximately 86. Further, the address and alu fields are sparse in the microcode; alu instructions appeared 17 times in total, while the address field was used approximately 118 times. The simple encoding of read, write, alu control, and test fields to microinstruction fields was selected, since it was a natural way of breaking up the signals, permitting easier examination for error detection, and would be a simple encoding to verify later on. This reduced the microinstruction word length to 27 bits.

Physical arrangements for the ROM layout were also considered. The sparse address field, and the correspondence between the use of this field and the test field selecting other than the next instruction, suggested using two ROM devices, one for the read, write, and alu fields, and the other for the address and test fields. The two devices would be $9 \times 400 \times 14$ and $9 \times 115 \times 13$ respectively. The alu fields could be generated directly from the address decoder outputs, and reordering

the decoder outputs (to something conceptually closer to a PLA structure than a ROM) could enable sharing of a single decoder between the 2 devices. This scheme was abandoned, largely because of its complexity, and the implicitly inconsistent treatment of outputs, and the line lengths resulting from the need for the full 9×400 decoder. Also, the savings generated by the simple encoding already brought the ROM into an acceptable size. Reduction of the length of internal lines in the ROM was achieved by the transformation to a $7 \times 100 \times 104$ ROM, with two bit column decode. This produced a nearly square device, and considerable flexibility in the control unit layout. The ROM layout was generated automatically from a bit pattern produced from the microcode simulation of the previous level. The decoder component is a fully complementary CMOS device, while the 'OR' is implemented in a pseudo NMOS design, using pullup transistors in each column, and n-type devices exclusively in the plane.

Since deciding to have the ROM output encoded signals, decoders were required. It was possible to decode fully within the control unit, or permit the datapath to decode. The decoders were included in the control unit because they did not easily fit with the bitslice dominated layout approach of the datapath. Further, if the decoded signals were routed through the shift registers, more flexible control was available in debug operation mode. The same automated ROM/PLA generator was used to produce the layout of all 3 decoders.

The 13 alternative methods of selecting the next microcode instruction select from only 4 possible values:

- the address following the current one in the microcode,
- the address supplied as the A field in the microcode,
- the SECD machine instruction code (opcode), and
- the value on top of the microcode subroutine stack.

A 4×1 MUX gates these values to `nextmpc` register. The mux control signals are generated by a *test* field of the microcode, in combination with the value of the flag and button inputs. The logic is implemented in a PLA, again generated automatically.

A bit-slice approach is used for the `mpc` and `nextmpc` registers, the microcode stack, and the logic to implement the selection of the next microinstruction. The required control signals for this are generated from a single row of random logic. Other signals, such as the control for the bidirectional i/o pads, were generated from random logic that was located with the PLA device.

4.4.3.3 Datapath

While *symbol* and *cons* record representation was fixed at an earlier stage, the representation of integers was left until the design of the datapath. The use of the `alu` to decrement addresses as well as integers required that the mapping of 14 bit addresses to 28 bit integers (accomplished by clearing the upper 14 bits) should be consistent with the representation of integers. While both two's complement and sign magnitude conformed to this constraint, the use of two's complement representation produced a simpler `alu` implementation.

The datapath was designed around a 32 bit bus, connecting all the registers and combinational logic devices. Most registers are simply 14 bits, connected to the *cdr* field of the bus, aside from the `alu` output `buffer` registers and the `arg` register which are all 32 bits. The `car` register inputs are connected to the upper 14 bit address field of the bus, while its outputs are connected to the lower (*cdr*) field. The `x1` register inputs and outputs both connect to the *cdr* field of the bus, but the output additionally connects to the *car* field inputs of the `consunit`. The `clearunit` sets the upper (*car*) field of the bus to zeros when the `mar` or `num` registers are read, because these are the sources of addresses that are decremented by the `alu`. This

operation effectively maps a 14 bit address to a 28 bit integer. The **alu** was simplified by omitting the three most complex (in terms of area) operations: **mul**, **div**, and **rem**. Their opcodes were not eliminated however, and the implementation defaults to the **dec** operation.

Registers are grouped into subcomponents: **regs-14-misc**, **regs-14-car**, **regs-14-y2**, **regs-32-arg**, and **regs-32-bufs**. No error checking of type bits is implemented for any operations in the datapath. Logical **alu** operations maintain the (unaffected) bits, while the arithmetic operations produce 32 bit output with the type bits set to integer, and both mark and field bits cleared. Similarly, the **consunit** outputs a 32 bit value with record bits set to *cons* and mark and field bits cleared.

Once the padframe was designed, it was found necessary to add one additional unit to the datapath. This **read-mem** unit allows the input values from the bidirectional pads to be passed onto the bus only when the **rmem** signal is high. This is necessary since the bidirectional pads were designed as write-enabled, and default to input mode. The busgates prevent the pads from writing onto the bus when not reading from memory.

4.4.4 Layout

A full custom design was selected in preference to gate array or semi-custom using a standard cell library. The design was too large for available LSI gate array dies at the time, and no semi-custom layout tools were at hand. Our team had the expertise to undertake full custom design, and it also provided a better learning experience than the other options. Further, non-custom designs suffered from constraints, including the number of gates in gate array, and the availability of suitable cell libraries. Full custom fabrication was available through MOSIS, with a clear and concise set of

scalable design rules. The layout design was completed using the ‘Electric’ system ([Rub87]).

4.4.4.1 Floorplanning

Mosis offered a 3μ double metal p-well CMOS process, in dies that permitted maximum project sizes of: 2.3×3.4 mm, 4.6×6.8 mm, 6.9×6.8 mm, and 7.9×9.2 mm. Initial estimates indicated the largest size would be required, and this size became a fixed constraint. A lengthy delay between the completion of the layout (late 1986) and the chip fabrication (fall 1988) permitted us to use the 2μ process introduced by Mosis in the interim. The use of scalable design rules was vital in enabling us to take advantage of the new technology *without any redesign*.

The major functional components, namely the datapath and controller, from the register transfer level were maintained as major floorplan elements. The shift register block was located between these two, and all were surrounded by a padframe.

4.4.4.2 Design Guidelines

The project team had already completed a microprocessor layout (the Tamarack⁴) and thus had some experience in layout. The design would use a cell library, with guidelines rigidly controlling the cell designs. Power and ground rails occurred at the top and the bottom of cells in metal-2, and bit slices were arranged so one rail was shared between two adjacent slices. Data generally flowed horizontally through the cells in metal-1, while control signals and clock lines run vertically in polysilicon. The use of metal-2 was restricted within cells to the rails, so that it could be freely used for horizontal interconnect running over the cells without any design rule violations.

Cell height was selected based on the example of past work, and by building sample cells using different heights. An optimal value of 86λ was selected. Each

⁴A second implementation of the Tamarack chip, an implementation of Gordon’s toy computer [Gor83b], was undertaken by the VLSI group as a warm-up for the SECD project.

cell was to be self-sufficient, so all required well/substrate contacts were internal. Port locations and boundary clearances were standardized, and multiple instances of ports was encouraged to ease cell composition.

All library cells were defined and exhaustively simulated in Mossim. Layout cells used the same root name for ports as the Mossim definition, and a one-to-one correspondence between the two definitions was attempted through all levels in the design hierarchy. The XOR cell was an exception, since Mossim could not correctly model the 6 transistor design actually used.

4.4.4.3 Shift Registers

The shift register block is a simple device used in the previous Tamarack2 design. It uses separate controls and clocks⁵, and is used to take a “snapshot” of the state of the chip, or to enter a vector for testing. Every signal that was considered reasonably useful for testing was routed through the shift registers. This included all read, write (except the write memory signal which was initially expected to be exported directly), and alu signals, the status flags from the datapath, and additionally, the mpc contents (the lines between the mpc register and the ROM), and the control signals for the 4×1 MUX feeding the `nextmpc` register, and controlling the microcode stack. In total, 72 bits are trapped by the shift registers. The only value passing between these components that was not trapped was the machine instruction code. Pass through lines were provided in the shift registers for this signal. Additionally, the control signal for the bidirectional pads, which was added at a late stage in the design, was not trapped. This made examining datapath register contents more difficult, as described in [GWS89], so the *wmem* signal was also routed through the shift registers in a later version.

⁵The use of distinct clocks for system and shift registers was chosen to simplify the logic design and improve the probability of obtaining working subcomponents on the chip by minimizing operational dependencies.

4.4.4.4 Padframe

The original die size and package had a limit of 64 pins. Bidirectional pins (32) were used for the bus to memory link. The MAR output required 14 pins, system clocks 2 pins, control inputs 2 pins, state output signals 2 pins, shift register controls, input, and output a total of 4 pins. 2 pins were used for the separate shift register clocks, instead of using one pin for a clock control input. Lastly, 2 power and 2 ground pins were used. A pair of power and ground pins (called *dirty power* and *dirty ground*) drive the pads only, to reduce noise on the supply lines to the chip, while the other pair drives the rest of the chip. In the final layout, the distribution of the pads around the chip perimeter was constrained by the number of bonding fingers along each cavity edge of the package, and a maximum 45 degree angle of bonding wires.

Simple input and output pads contain no logic, aside from buffering outputs, but the design of the bidirectional pads required more effort. When used in output mode, the pad had to increase the drive strength of the signal. A step-up buffer was used for this, but it must not drive in input mode. Thus the circuitry turns off both n and p-transistors by providing 0 and 5 volt gate inputs respectively. The designs were simulated using SPICE, and switching times in the 20 nanosecond range were achieved using a load capacitance of 50 pf. This speed was quite acceptable, given the constraints stated earlier.

4.5 Summary and Status

The preceding description of the design process used on the SECD gives a general context in which the decisions affecting the design were made. The hierarchical levels of description are summarised in Table 4.4. The scope of the project described was considerable, and involved a team of up to 9 individuals at various times.

The author's involvement in this process included:

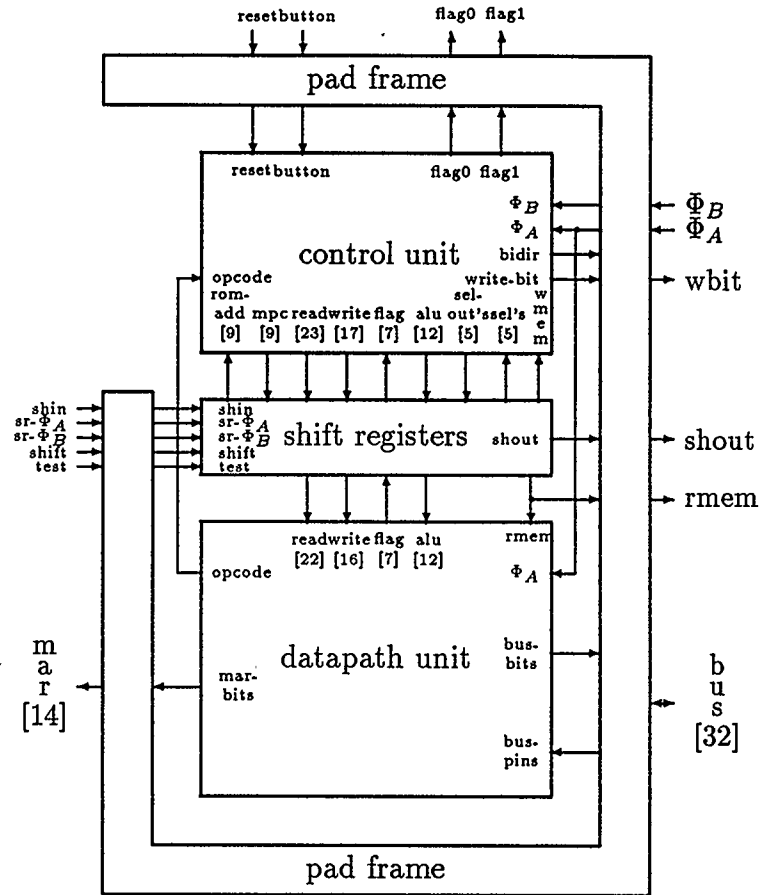


Figure 4.3: SECD Chip Major Subcomponents

- the design of the Mossim definition of the control unit [GWB⁺89],
- floorplanning design of the control unit,
- responsibility for the layout of the control unit component of the chip [BGJ⁺89],
- preparation of the operating specification for the chip along with a preliminary test methodology [GWS89], and

- continued involvement in coordinating later modifications and revised specifications for the fabricated design.

The detailed description of the design given will aid the reader in understanding the formal definition of the system presented in the following chapter. Of particular importance is the timing scheme, and the relation between the levels which are formally described.

Two versions of the SECD chip have been fabricated. The first had a design error in the shift register component which made the chip entirely nonfunctional. This was purely a layout error, where the layout did not correspond to the Mossim model. The second design also has a flaw in a primitive gate, and once again, the cell layout did not correspond to the Mossim description of the circuit. Testing of this version of the chip is presently continuing, in concert with developing software for a chip controller board described in [Wil89].

Level	time quantum	Simulation model	Introduced concerns/ design decisions
Abstract Machine	SECD instruction cycle	SECD interpreter written in Franz Lisp: lisp data structures and operations (car, cdr, cons); recursive functions	learning exercise only
Abstract System	SECD instruction cycle	written in C: high level language constructs, data types; external read and print routines;	finite memory, garbage collection, memory exhausted error
Top Level FSM	SECD instruction cycle	none	chip control and control inputs, states, repeated computations
Abstract RTL	varied number of clock cycles	C simulation: higher level constructs (if-then-else, case, while); high level routines for cons, car, cdr	operative part of internal architecture — logic components, bus, registers, memory; record types, data representation, word configuration; in-place garbage collection
Concrete RTL	clock cycle	C simulation: fully developed microcode and operative part of architecture	implemented cons, car, cdr; flow control; reserved memory locations; memory interface
Mossim	instants	asynchronous switch level model of transistors; clock phases permit settling of all subcircuits	clocking (timing architecture); memory element design; testability
Layout		none	floorplanning, electrical characteristics

Table 4.4: Levels of Definition Summary

Chapter 5

Formal Specification of SECD

The SECD example differs from previous hardware verification subjects in the functional nature of the machine. Its support for procedures and recursion provide more complex instruction and memory state transitions than do traditional architectures. The chip is also significantly larger than many previous examples. These concerns combined with the complicated nature of the transitions made the management of complexity a larger issue.

Because of the complexity of the SECD chip, it is not possible within the scope of a thesis to give more than an outline of most of its component specifications: they are simply too large to be included in their entirety. All we can do is give a flavour of the work.

A full HOL specification of the SECD system is given in [Gra89a]. The definitions alone comprise some 70 pages (4000 lines) of HOL source.

The formal specification of the SECD chip consists of three levels of description:

- the **top level** specification, which defines the programming level model,
- the **register transfer level** definition, which defines the internal architecture in terms of multi-bit registers and functional components, and
- the **low level** definition, consisting of primitive logic gates, single bit latches, and transistor networks for regular structures such as ROM's and PLA's.

The lowest level corresponds closely to the layout and Mossim hierarchies, although the definitions of simple logic gates do not go all the way down to the transistor

level.¹ This is the model we wish to relate to the top level specification, but the problem size demands at least one intermediate level to abstract out some of the complexity. The intermediate level chosen corresponds very closely to the Concrete RTL view of the design development, and is the highest level which still expresses the internal architecture structure and control.

This chapter focuses on the RTL and top levels, limiting discussion of the low level to a few key elements that impact the sequel. It begins with a description of how hardware is modelled in HOL, and the data types and primitive operations common to all levels.

5.1 Modelling Hardware

This research is concerned with bistate logic and logic devices, and uses *boolean* values to represent levels of electrical charge, with T and F representing 5v and 0v respectively. Signals that vary over time are represented by functions from discrete time, represented as `:num` values, to values of the appropriate type. For example, a single bit signal will be of type `:num->bool`. More complex datatypes may be built from single bits, particularly bit vectors. At the lowest level, we may describe a group of signals from *time* (i.e.:`num`) to `:bool`, while at a higher level it may be preferable to describe a signal from *time* to a group of `:bool` values, and relate the two with an abstraction function.

Primitive combinational hardware devices are represented in HOL by predicates which express logical relations on time-dependent input and output signals. For example, a 2 input nand gate would be expressed by the relation:

$$\text{NAND_spec } a \ b \ c = !t:\text{num}. c \ t = \sim(a \ t \ \wedge \ b \ t)$$

¹[SBGS89] contains a complete library of combinational logic gates down to a simple transistor model.

This expresses the fact that the output c at time t will be the value expressed by the *nand* relation on a and b at that same time. Universal quantification over the explicit time parameter indicates the relation holds at all points of time. Relations expressing devices with a memory component include an internal state signal as well, and relate signals at two different points of discrete time. For example, a simple *D-type* latch can be expressed by the relation:

D-type clk in out = !t:num. out(t+1) = clk t => in t | out t.

The composition of devices is expressed by conjunction with connected ports sharing the same parameter. Hidden nodes are expressed using existential quantification. For example, a 4 NAND gate implementation of an *exclusive-or* function may be defined by the circuit:

```
XOR_imp a b c =
? n1 n2 n3:num->bool.
  (NAND_spec a b n1) ^
  (NAND_spec a n1 n2) ^
  (NAND_spec n1 b n3) ^
  (NAND_spec n2 n3 c )
```

Extensive use is made of signals representing fixed numbers of bits, referred to as *word_n* types.² This type is defined in terms of specified width *buses* of discrete signals, where objects of type **bus* are defined using the constructors *Wire* for the base case (discrete signal), and *Bus* for the addition of each subsequent signal. They resemble nonempty lists. A specified *word_n* type object, such as a *:word14* value used to represent an address, is created by applying the constant *Word14* to a *:(bool)bus* of Width 14. *Word14* is an abstraction function from the *representing* type *:(bool)bus*. The corresponding *abstraction* function *:Bits14* maps objects of type *:word14* back to *:(bool)bus* type objects. Constants of *word_n* type may be

²This data type definition was implemented by T. Melham.

written as binary strings: `#00000000000011` represents a 14 bit word that would be interpreted as the number 3.

A simple memory can be defined as a function of type `:word14->word32`. Since new values can be written to memory, it will be described as a function from discrete time: `:num->word14->word32`. Other examples of nonprimitive data types include tuples and lists. [Gor86] gives a general description of describing hardware in the HOL system.

As the unit of discrete time is different for each level of definition, a convention is adopted of subscripting the explicit time parameters to indicate the granularity of time intended, low level (fine grain) time t_f , RTL (medium grain) time t_m , and top level (coarse grain) time t_c .

5.2 The Top Level Specification

At the most abstract level, the SECD machine is defined in terms of transformations to S-expressions in the 4 stacks, as shown in Table 3.3. A formal specification of the top level behaviour is ideally defined in terms of transformations to an S-expression data type that closely resembles this elegant definition. The closer the resemblance the better we are assured that the HOL specification captures the intent.

The method used by SECD for implementing recursive function definitions as closures with a circular environment component raises the complexity of the data representation problem considerably. Such circular S-expression lists, created by a destructive operation, cannot be mapped to a simple recursive data type. Further, structure is shared by S-expressions, particularly the environment component of closures. Each mutually recursive function closure references the same environment, which is also in the E stack. When a destructive replace operation is performed (by

executing a RAP instruction) to create the circular list structure, the change affects the common component of all closures simultaneously.

Thus, a much more primitive representation has been chosen to describe the top level specification, following on the work of Mason [Mas86, Mas88] on the semantics of Lisp. Rather than directly defining transformations on structures of an S-expression data type, an abstract memory type is defined which can contain representations of S-expressions. Further, a set of primitive operations upon the memory is defined, corresponding to the operations on S-expressions, namely *cons*, *car*, *cdr*, *atom*, *rplaca*, *eq*, *leq*, *add*, and *sub*. The 4 state registers contain pointers to the appropriate S-expression representation. Finally, an additional *free* pointer to the free list structure is needed to define the *cons* operation. The state of the machine is then defined by a tuple:

$$(S, E, C, D, Free, memory, FSM_state).$$

where the *FSM_state* is one of the 4 major states of the top level finite state machine view of the machine (Figure 4.1).

The abstract memory type μ is basically a function type: $\mu = \delta \rightarrow (\delta \times \delta \cup \alpha)$ where δ is the domain of the function and α is the set of atoms: $\alpha = integers \cup symbols$. The set of symbols includes the symbolic constants: T, F, and NIL. The domain of the function is chosen to be the type of 14 bit words (*:word14*), matching the type that is used by the lower level definition. The definition of memory is extended to incorporate garbage collection features by adding *mark* and *field* bits to each cell:

$$\mu = \delta \rightarrow ((bool \times bool) \times (\delta \times \delta \cup \alpha)).$$

Additionally, the garbage collection operations *replacd*, *setf*, and *setm* are included, as well as a *garbage_collect* function, which was left undefined for this proof effort. Extractor functions for the *mark* and *field* bits, and the *integer* and *atom* fields are

provided for the values returned by the μ function. The relevant built-in functions and their types are summarized in Table 5.1. To distinguish these abstract memory functions, the function names uniformly begin with “M_”.

Operation	Type
primitive operations	
M_Car, M_Cdr	$(\delta \times \mu \times \delta) \rightarrow (\delta)$
M_Cons	$(\delta \times \delta \times \mu \times \delta) \rightarrow (\delta \times \mu \times \delta)$
M_Eq, M_Leq	$(\delta \times \delta \times \mu \times \delta) \rightarrow bool$
M_Add, M_Sub	$(\delta \times \delta \times \mu \times \delta) \rightarrow (\delta \times \mu \times \delta)$
M_Replaca, M_Replacd	$(\delta \times \delta \times \mu \times \delta) \rightarrow (\delta \times \mu \times \delta)$
M_setm, M_setf	$(bool \times \delta \times \mu \times \delta) \rightarrow (\delta \times \mu \times \delta)$
extractor functions	
M_mark, M_field	$\delta \rightarrow \mu \rightarrow bool$
M_Int_of	$(\delta \times \mu \times \delta) \rightarrow integer$
M_Atom_of	$(\delta \times \mu \times \delta) \rightarrow \alpha$
predicates	
M_Atom	$(\delta \times \mu \times \delta) \rightarrow bool$
M_is_cons	$(\delta \times \mu \times \delta) \rightarrow bool$
M_is_T	$(\delta \times \mu \times \delta) \rightarrow bool$
auxiliary functions	
M_garbage_collect	$(\mu \times \delta) \rightarrow (\mu \times \delta)$
M_CAR, M_CDR	$(\delta \times \mu \times \delta) \rightarrow (\delta \times \mu \times \delta)$
M_Cons_tr	$(\delta \times \delta \times \mu \times \delta) \rightarrow (\delta \times \mu \times \delta)$

Table 5.1: Primitive Operations on Abstract Memory Data Type

For consistency, all functions take both a memory (μ) and the free list pointer (δ) as the last items in the tuple argument, although the free pointer is not always used. Operations such as M_Car, M_Cdr, M_Eq, M_Leq, etc. do not alter memory, while M_Cons, M_Add, M_Sub, M_setm, M_setf, M_Replaca, M_Replacd do alter one cell

in the memory, and thus must return the new memory. In order to permit composition of the primitive memory operations, the `M_CAR` and `M_CDR` functions are provided, returning the unaltered memory and free pointer. For example, to access an argument to a `LD` command, we can write:

```
let m = M_Int_of(M_CAR(M_CAR(M_CDR(c, MEM, free)))).
```

The function `M_Cons_tr` simply transposes the first 2 arguments of the `M_Cons` function, to permit composition where the memory and free pointer arguments are handed on from the computation of the first *cons* argument instead of the second.

```
LDF_trans (s:δ,e:δ,c:δ,d:δ,free:δ,MEM:μ) =
  let cell_mem_free_1 = M_Cons_tr(e,M_CAR(M_CDR(c, MEM, free)))
  in
  let cell_mem_free_2 = M_Cons_tr(s, cell_mem_free_1)
  in
  (cell_of cell_mem_free_2,          % s %
   e,                               % e %
   M_Cdr(M_CDR(c, mem_free_of cell_mem_free_2)), % c %
   d,                               % d %
   free_of cell_mem_free_2,         % free %
   mem_of cell_mem_free_2,          % memory %
   top_of_cycle)                   % state %
```

Figure 5.1: Transition for LDF Instruction

Figure 5.1 gives the top level transition specification for the LDF instruction. Following this, in Figure 5.2 the next state of the machine is defined for each instruction, using the set of 21 such transitions, and the the top level specification for the SECD is given. The top level function, `SYS_spec`, closely resembles the top level finite state machine of Figure 4.1, with 4 states, 2 possible transitions from 3 of them, and 18 transitions from the *top_of_cycle* state, one for each (implemented) machine instruction. The system must be assured of starting out in the *idle* state.

```

NEXT (s: $\delta$ , e: $\delta$ , c: $\delta$ , d: $\delta$ , free: $\delta$ , MEM: $\mu$ ) =
  let instr = M_int_of(M_CAR(c, MEM, free))
  in
    ( (instr = LD)   =>   (LD_trans   (s, e, c, d, free, MEM))
    | (instr = LDC)  =>   (LDC_trans  (s, e, c, d, free, MEM))
    | (instr = LDF)  =>   (LDF_trans  (s, e, c, d, free, MEM))
    | ...
    | (instr = LEQ)  =>   (LEQ_trans  (s, e, c, d, free, MEM))
    | % (instr = STOP) % (STOP_trans (s, e, c, d, free, MEM))

```

```

SYS_spec (s: $\delta_c$ ) (e: $\delta_c$ ) (c: $\delta_c$ ) (d: $\delta_c$ )
         (free: $\delta_c$ ) (MEM: $\mu_c$ )
         (button:bool $_c$ ) (state:state $_c$ ) =

  (state 0 $_c$  = idle) ^

  ! t $_c$ .
  ((s(t $_c$ +1), e(t $_c$ +1), c(t $_c$ +1), d(t $_c$ +1),
    free(t $_c$ +1), MEM(t $_c$ +1), state(t $_c$ +1)) =
    ((state t $_c$  = idle)
     => (button_pin t $_c$ 
        => (M_Cdr(M_CAR(NUM_addr, MEM t $_c$ , free t $_c$ )),
            NIL_addr,
            M_Car(M_CAR(NUM_addr, MEM t $_c$ , free t $_c$ )),
            NIL_addr,
            M_Cdr(NUM_addr, MEM t $_c$ , free t $_c$ ),
            MEM t $_c$ , top_of_cycle)
        | (s t $_c$ , e t $_c$ , c t $_c$ , d t $_c$ , free t $_c$ , MEM t $_c$ , idle))
    | (state t $_c$  = error0)
     => (button_pin t $_c$ 
        => (s t $_c$ , e t $_c$ , c t $_c$ , d t $_c$ , free t $_c$ , MEM t $_c$ , error1)
        | (s t $_c$ , e t $_c$ , c t $_c$ , d t $_c$ , free t $_c$ , MEM t $_c$ , error0))
    | (state t $_c$  = error1)
     => (button_pin t $_c$ 
        => (s t $_c$ , e t $_c$ , c t $_c$ , d t $_c$ , free t $_c$ , MEM t $_c$ , error1)
        | (s t $_c$ , e t $_c$ , c t $_c$ , d t $_c$ , free t $_c$ , MEM t $_c$ , idle))
    | % (state t $_c$  = top_of_cycle) %
    (NEXT (s t $_c$ , e t $_c$ , c t $_c$ , d t $_c$ , free t $_c$ , MEM t $_c$ )))

```

Figure 5.2: Top Level Specification

The given types of all system parameters have the subscript “ c ”. This identifies them as signals sampled at a *coarse* grain of time.

The type α was defined in HOL as the type `:atom`, and the type μ was defined as the abstract data type `:(word14,atom)mfsexp_mem`. Both types have associated REP and ABS functions to map between the abstract and representing types. For example, the term `REP_mfsexp_mem(memory:(word14,atom)mfsexp_mem)` has the type:

`:word14->((bool # bool) # (word14 # word14 + atom)).`

The type `:atom` is defined by the simple grammar: `atom = Int integer | Symb num`, where `Int` and `Symb` are type constructors.

5.3 The Low Level Definition

The low level definition uses simple gates as base components, typically with simple functions such as AND, OR, XOR, etc., and bears a one-to-one relation to the layout of gates in the SECD chip.

The representation of time uses the coarsest granularity that captures the behaviour of the clock phases: using 4 points per clock cycle, one for each phase and for the points between phases. The presence of two separate clocks, one for the normal mode of chip operation, and the other operating the shiftregisters for test mode, confused the representation. A full set of constraints were defined to describe the operation of the clocks [GB89].

- A clock cycle always consists of nonoverlapping assertions of ϕ_A followed by ϕ_B , for either clock.
- The separate clocks do not cycle simultaneously. Once started each clock always completes its cycle uninterrupted by the other clock.
- The system clock phase ϕ_B is asserted at powerup. The reset input is asserted at this time as well, forcing the chip into a known initial state (see 5.4).

- Lastly, since the verification will only concern normal mode of operation, the system clock will be the only clock that cycles.

Although restricting the chip to normal mode of operation, these constraints made it clear that intervals of the fine granularity of time correspond to advancing either clock.

The behaviour of level triggered latches at this time granularity would be captured adequately by the D-type definition given in section 5.1, with a clock phase line as its first argument. However, the standard design regimen for 2-phase non-overlapping clocking, that of linking memory devices clocked on opposite clock phases with combinational logic, was violated in one part of the SECD chip design. Two registers clocked on the same clock phase were wired in series, with the output of the first feeding the input of the second through strictly combinational logic. This odd circuit feature degrades chip performance, but at sufficiently slow clock rates the chip can still function properly since the output of a level triggered latch changes at the start of the clock pulse and no circular feedback path exists. This single piece of questionable design affected the entire chip specification and added an obligation to prove the impossibility of circular feedback. A modified definition of the primitive latch was required to express this behaviour appropriately, expressing the present state as a function of the clock signal and input at the present moment instead of the previous moment.

```
latch clk inp state =  
  !t. state t = (clk t) => inp t | state (PRE t)
```

The regular structures, such as the microcode ROM, decoders, and a PLA can be defined as logical functions, but their complexity is quite different from the rest of the primitive gates. These regular structures were produced by automated layout tools, and the transistor network produced was specified in HOL. The use of a pseudo

nMOS OR plane in the PLA and ROM required the use of a multi level logic, to capture the floating value which will “pull up” to 5v in the actual device. The output of the device will always be at a fully restored level, so the interface converts back to a boolean data type. This work was similar to that reported in [Joy88].

5.4 Register Transfer Level Specification

The Register Transfer Level (RTL) model of the chip is similar in component hierarchy to the low level view, but does not have the same depth. The lowest items in the RTL hierarchy include devices such as multi-bit registers, which are non primitive components of the low level hierarchy. The chip has 3 major components: the control unit, the datapath, and the pad frame (CU, DP, and PF respectively). Generally, the primitive components are behavioural specifications for the corresponding components at the low level.

Notably absent from this view is the shift register component. Under normal operation, these are intended to be transparent to the operation of the chip, and thus constraining their controls and the clocks appropriately at the lower level permitted abstracting them out of this level of description completely. However, for future compatibility with an extended specification that can include debug mode, a clock signal `SYS_Clocked` was included to represent the cycling of the system clock lines. This signal parameterizes every memory device in this level view.

5.4.1 Temporal representation

The time granularity at this level corresponds to clock cycles. Selecting which point of the cycle to use for this time grain was complicated by the outputs of registers clocked on different clock phases multiplexing into a register input (*i.e.* this is the case when two registers clocked on the same phase are wired in series). The part

of the clock cycle over which the output of the register is stable is determined by the phase clocking the register. In order to have a uniform temporal abstraction for the time parameter for each input type, the point of the clock phase selected for sampling at the RTL (medium) time granularity was during the assertion of the first clock phase (ϕ_A) at the finer grain.

Two different types of registers were used. The control unit used pairs of latches clocked on opposite clock phases, while the datapath registers clock write operations on ϕ_A to prevent transitory spikes on state transitions from causing unwanted writes. The low level circuit and the RTL specification for each are shown in Figure 5.3.

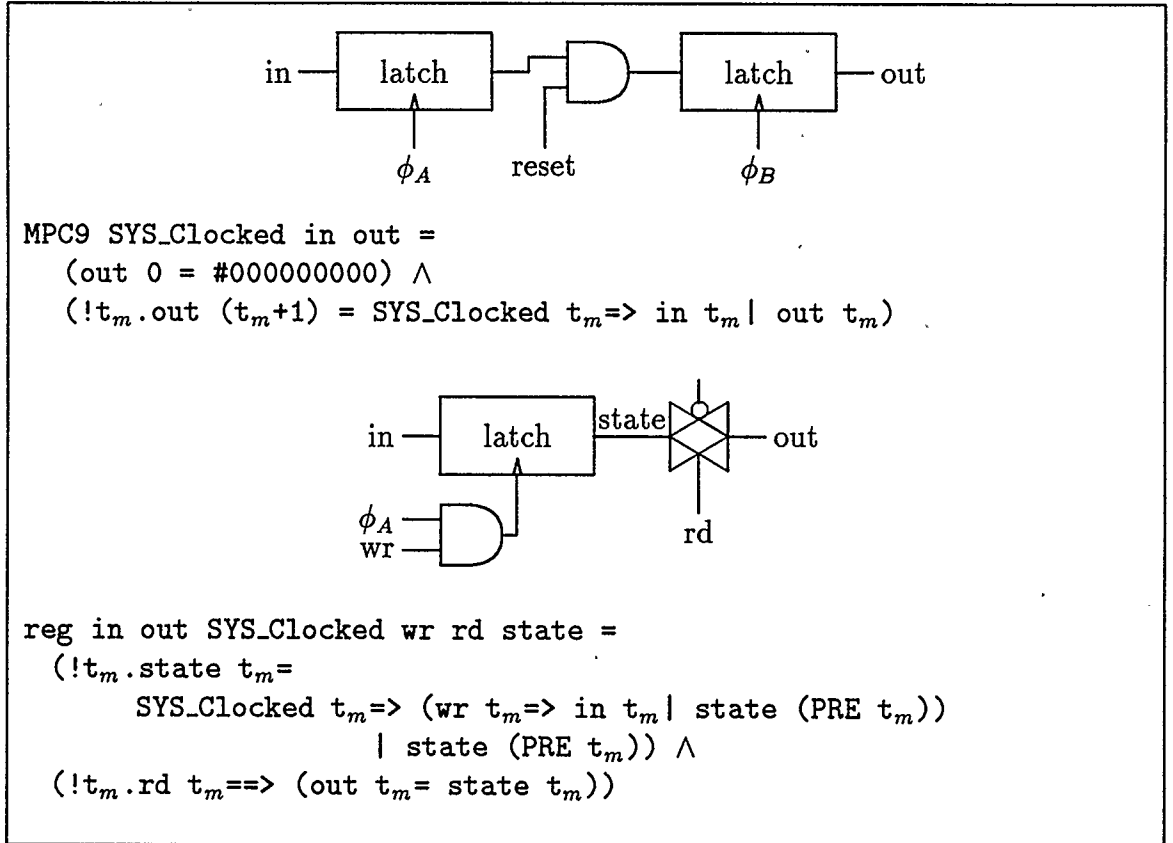


Figure 5.3: Control Unit and Datapath Register Schematics and Definitions

In both definitions, the clock signal `SYS_Clocked` is an abstraction of the two distinct clock phases (ϕ_A and ϕ_B), and is asserted when the system clock cycles. (Under the lower level clock constraints, the system clock always cycles.) Note that the MPC9 register specification does not include the *reset* input explicitly. However, the lower level constraints on the *reset* signal force an initialisation at power up, and this appears as a determined value in the register at time $t_m = 0$. The typical datapath register has a gated output, and is clocked by both the system clock phase and the specific write signal. The clock signal was included here to permit eventual extension of the verification to the test mode of chip operation. The difference in the definition of the registers and the choice of point on the time cycle can be seen from the timing diagram in Figure 5.4.

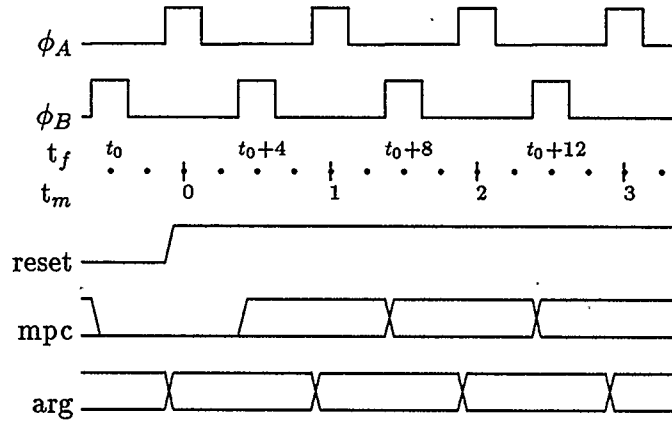


Figure 5.4: Relating low and RTL times

The *reset* signal is held low at the start of the clock operation. The MPC9 register content (*i.e.* *mpc*) at t_m+1 is a function of the inputs at time t_m . If the ARG register (a typical datapath register) content at time t_m feeds into the first MPC9 latch while ϕ_A is asserted, this relation holds, as it also does if the MPC9 content itself feeds back into this latch. Sampling one point of the fine granularity earlier would have

required distinct temporal abstractions for the signals arising from registers clocked on different clock phases.

5.4.2 The Datapath Specification

The definition of the datapath begins with itemizing the data types introduced, specified constants, and definitions of primitive operation. Component definitions and the composition of the whole follow. Refer to Figure 4.2 for a visual representation of all the datapath components and their organization.

Four fixed length word types are needed for the datapath, :word32 for representing words from memory, :word2 for record type and garbage collection bits fields of words, :word14 for memory addresses, and :word28 for atom value fields. Several *word_n* constants are defined below.

T_addr	= #00000000000001	F_addr	= #00000000000010
NIL_addr	= #00000000000000	NUM_addr	= #11111111111111
ZEROS14	= #00000000000000	ZERO28	= #000000000000000000000000000000
RT_SYMBOL	= #10	RT_NUMBER	= #11
		RT_CONS	= #00

The first 4 are the addresses of reserved words in memory, ZEROS14 is used to pad a 14 bit address to a 28 bit number, ZERO28 is the representation of the integer 0, and the last 3 are the record type identifiers.

Field extractor functions extract fields or single bits from words: *garbage_bits*, *mark_bit*, *field_bit*, *rec_type_bits*, *car_bits*, *cdr_bits*, and *atom_bits*. The *word_n* types offer straightforward extractor definitions, without the need for obscure conversions to different datatypes typical of an earlier *word_n* implementation.

```

car_bits (Word32(Bus b32 (Bus b31 (Bus b30 (Bus b29
    (Bus b28 (Bus b27 (Bus b26 (Bus b25
    (Bus b24 (Bus b23 (Bus b22 (Bus b21
    (Bus b20 (Bus b19 (Bus b18 (Bus b17
    (Bus b16 (Bus b15 (Bus b14 (Bus b13
    (Bus b12 (Bus b11 (Bus b10 (Bus b9
    (Bus b8 (Bus b7 (Bus b6 (Bus b5
    (Bus b4 (Bus b3 (Bus b2 (Wire (b1:bool)
    )))))))))))))))) =
Word14(Bus b28 (Bus b27 (Bus b26 (Bus b25 (Bus b24 (Bus b23
    (Bus b22 (Bus b21 (Bus b20 (Bus b19 (Bus b18 (Bus b17
    (Bus b16 (Wire b15))))))))))

```

Recognizer functions compare the record type field of a word with the record type constants: `is_symbol`, `is_number`, `is_cons`, and `is_atom`. Conversion functions to map bit values to numbers and integers are defined.

```

bv x = x => 1 | 0

(val n (Wire w)      = 2 * n + (bv w)          ) ^
(val n (Bus b bus) = val (2 * n + (bv b)) bus)

Val = val 0

(iVal (Wire w)      = neg (INT (bv w))          ) ^
(iVal (Bus b bus) =
    INT (val 0 bus) minus INT ((2 EXP (Width bus)) * bv(b)) )

```

Lastly, **constructor functions** for each record type are defined, using the `Concat` function to join *buses* together.

```

bus32_cons_append a b c d =
    Word32(Concat(Bits2 a)(Concat(Bits2 b)(Concat(Bits14 c)(Bits14 d))))

bus32_num_append a =
    Word32(Concat(Bits2 #00)(Concat(Bits2 RT_NUMBER)(Bits28 a)))

bus32_symb_append a =
    Word32(Concat(Bits2 #00)(Concat (Bits2 RT_SYMBOL)(Bits28 a)))

gc_bus32_append a b c =
    Word32 (Bus a (Bus b (Tl_bus (Tl_bus (Bits32 c))))))

```

The 4 constant “registers”: NUM, Nil, TRUE, and FALSE, as well as the *clearunit* that pads a 14 bit address with zeros to 28 bits, are all implemented with busgates. A busgate is simply a set of transmission gates, controlled by a *read* signal. The 14 bit version is defined as:

```
busgate14 in_val rd out =
  !tm. (rd tm) ==> (out tm = in_val tm)
```

When used to gate a constant value onto the bus, the predicate is applied to an abstraction of the form $(\lambda t_m. \text{NIL_addr})$. The READ_MEM component is also a busgate, which connects the bus to the bidirectional i/o pad output. This is required to isolate the pad output from the bus when a read memory instruction is not asserted, since the pads default to input mode of operation.

The form of the register definition was described earlier. Versions for 2, 14, and 32 bit are defined, as are further variants with input and output connected to the same node, which is typical for most registers connected to the bus. The definitions for 14 bit versions follow.

```
reg14 in_sig out_sig clocked wr rd st =
  (!tm. st tm = clocked tm ==> (wr tm ==> in_sig tm | (st (PRE tm)))
    | (st (PRE tm))) ^
  (!tm. (rd tm) ==> ((out_sig tm) = (st tm)))

register14 sgl = reg14 sgl sgl
```

The Cons unit is simply implemented with 4 busgates: two 14 bit busgates connect the car and cdr fields inputs, while two 2 bit busgates are connected to constants for the garbage bits and record type bit fields. The output is a 32 bit value, with appropriate field selectors applied to select the connection for each busgate.

The FLAGSUNIT performs various tests on data values and returns 7 status flags for use by the control unit. Only the LEQ operation needs definition; the others test particular bits or compare all bits. LEQ compares the integer values represented by the words being compared.

```

LEQ_prim (x:word32) (y:word32) =
  let ival_x = (iVal o Bits28 o atom_bits) x
  in
  let ival_y = (iVal o Bits28 o atom_bits) y
  in
  ((ival_x below ival_y) ∨ (ival_x = ival_y))

FLAGSUNIT bus arg
  atomflag bit30flag bit31flag zeroflag nilflag eqflag leqflag =
!tm.(atomflag  tm = is_atom (bus tm))           ^
    (bit30flag tm = field_bit (bus tm))         ^
    (bit31flag tm = mark_bit (bus tm))          ^
    (zeroflag  tm = (atom_bits (bus tm) = ZERO28) ) ^
    (nilflag   tm = (cdr_bits (bus tm) = NIL_addr)) ^
    (eqflag    tm = (bus tm = arg tm))         ^
    (leqflag   tm = LEQ_prim (arg tm) (bus tm))

```

The ALU can perform 9 distinct operations. Six of these are used exclusively in garbage collection: *replcar*, *replcdr*, *setbit31*, *setbit30*, *resetbit31*, and *resetbit30*. The remaining operations are the arithmetic operations: *add*, *sub*, and *dec*. The *mul*, *div*, and *rem* operations were not implemented, and selecting any of these defaults to the *dec* operation. All 6 of the garbage collection operations are destructive operations, replacing a bit or field of the operand word.

```

REPLCAR (x:word32) (y:word14) =
  bus32_cons_append (garbage_bits x) (rec_type_bits x) y (cdr_bits x)

REPLCDR (x:word32) (y:word14) =
  bus32_cons_append (garbage_bits x) (rec_type_bits x) (car_bits x) y

SETBIT30 (x:word32) = gc_bus32_append (mark_bit x) T x
SETBIT31 (x:word32) = gc_bus32_append T (field_bit x) x
RESETBIT30 (x:word32) = gc_bus32_append (mark_bit x) F x
RESETBIT31 (x:word32) = gc_bus32_append F (field_bit x) x

```

The output of the ALU is gated onto the bus under the control of the *ralu* signal. When no ALU operation is selected, the value computed is not significant. The

definition uses implication to define the computed output only when one of the alu control signals is asserted. If none of the control signals is asserted, the value of alu is undefined. Likewise, if more than one control line is asserted, the computed output is also unimportant. This is expressed in the specification by the use of a `one_asserted` predicate on the control lines which implies the desired behaviour. This predicate states that if one of the signals is asserted, the remainder are not.

```

ALU replcar replcdr sub add dec mul div rem
  setbit30 setbit31 resetbit30 resetbit31
  ralu arg y2 bus alu =
(one_asserted_12 replcar replcdr sub add dec mul div rem
  setbit30 setbit31 resetbit30 resetbit31 ==>
(!tm. let bus28 = (atom_bits (bus tm))
  in
    let arg28 = (atom_bits (arg tm))
  in
    ((replcar tm    ==> (alu tm = REPLCAR (arg tm) (y2 tm))) ^
     (replcdr tm    ==> (alu tm = REPLCDR (arg tm) (y2 tm))) ^
     (sub tm        ==> (alu tm = SUB28   arg28   bus28)) ^
     (add tm        ==> (alu tm = ADD28   arg28   bus28)) ^
     (dec tm        ==> (alu tm = DEC28   arg28)) ^
     (mul tm        ==> (alu tm = DEC28   arg28)) ^
     (div tm        ==> (alu tm = DEC28   arg28)) ^
     (rem tm        ==> (alu tm = DEC28   arg28)) ^
     (setbit31 tm    ==> (alu tm = SETBIT31 (arg tm))) ^
     (setbit30 tm    ==> (alu tm = SETBIT30 (arg tm))) ^
     (resetbit31 tm ==> (alu tm = RESETBIT31 (arg tm))) ^
     (resetbit30 tm ==> (alu tm = RESETBIT30 (arg tm))))))
^
(!tm. ralu tm ==> (bus tm = alu tm)))

```

(The SUB28 and ADD28 functions are the specifications for the operations performed by the low level component, and match what is specified at the top level.)

The datapath DP, defined in Figure 5.5, consists of the conjunction of subcomponents, closely resembling the diagram in Figure 4.2. The bus is expressed as the wiring together of the components. The CAR register is unique in having the input connected to the *car* field of the bus, and the output to the *cdr* field. All other 14

```

DP bus_bits mem_bits SYS_Clocked rmem mar wmar rmar rnum rnil rtrue rfalse
s ws rs e we re c wc rc d wd rd free wfree rfree parent wparent rparent
root wroot rroot y1 wy1 ry1 x1 wx1 rx1 x2 wx2 rx2 y2 wy2 ry2 rcons
car wcar rcar atomflag bit30flag bit31flag zeroflag nilflag eqflag leqflag
arg warg rarg buf1 wbuf1 rbuf1 buf2 wbuf2 rbuf2 replcar replcdr
sub add dec mul div rem setbit30 setbit31 resetbit30 resetbit31 ralu
=
? alu.
  let a_bus = (λ tm. car_bits (bus_bits t))
  and d_bus = (λ tm. cdr_bits (bus_bits t))
  in
    ( (READ_MEM mem_bits rmem bus_bits)                                ^
      (MAR a_bus d_bus SYS_Clocked wmar      rmar      mar)          ^
      (NUM  rnum  a_bus d_bus)                                         ^
      (NIL  rnil  d_bus)                                               ^
      (TRUE rtrue d_bus)                                               ^
      (FALSE rfalse d_bus)                                             ^
      (S_reg d_bus SYS_Clocked ws      rs      s)                    ^
      (E_reg d_bus SYS_Clocked we      re      e)                    ^
      (C_reg d_bus SYS_Clocked wc      rc      c)                    ^
      (D_reg d_bus SYS_Clocked wd      rd      d)                    ^
      (FREE_reg d_bus SYS_Clocked wfree  rfree  free)                ^
      (PARENT_reg d_bus SYS_Clocked wparent rparent parent)          ^
      (ROOT_reg d_bus SYS_Clocked wroot  rroot  root)                ^
      (Y1_reg d_bus SYS_Clocked wy1     ry1     y1)                  ^
      (X1_reg d_bus SYS_Clocked wx1     rx1     x1)                  ^
      (X2_reg d_bus SYS_Clocked wx2     rx2     x2)                  ^
      (Cons x1 x2 rcons bus_bits)                                      ^
      (CAR_reg a_bus d_bus SYS_Clocked wcar  rcar  car)              ^
      (FLAGSUNIT bus_bits arg atomflag bit30flag bit31flag
        zeroflag nilflag eqflag leqflag)                             ^
      (Y2_reg d_bus SYS_Clocked wy2     ry2     y2)                  ^
      (ARG_reg bus_bits SYS_Clocked warg  rarg  arg)                  ^
      (ALU replcar replcdr sub add dec mul div rem
        setbit30 setbit31 resetbit30 resetbit31
        ralu arg y2 bus_bits alu)                                     ^
      (BUF1_reg alu bus_bits SYS_Clocked wbuf1 rbuf1 buf1)           ^
      (BUF2_reg alu bus_bits SYS_Clocked wbuf2 rbuf2 buf2))          ^

```

Figure 5.5: Definition of the DP Component

bit registers have inputs and outputs connected to the *cdr* field of the bus (as well as other connections in some cases).

5.4.3 The Control Unit Specification

The CU differed most from the lower level. The layout hierarchy was replaced by one that better reflected its functionality, using three components for its definition: a *state register*, a microcode *ROM*, and a *decode* section. Conjoining the definitions of these 3 parts gives a typical finite state machine behaviour, defining the next state and current output values as a function of the current state and current input values.

The control unit utilizes several fixed length word types: `:word27` for the microinstruction word (*ROM* output), `:word4` for test and alu fields of a microinstruction, `:word5` for read and write fields, and `:word9` for the A field and the micro pc (*ROM* input). Field extractor functions include: `Read_field`, `Write_field`, `Alu_field`, `Test_field`, and `A_field`. An increment function for `:word9` values is used to find the next address value for sequential microcode execution. It is defined in 2 parts, the `inc` function operates on boolean buses, and presents a very implementation-like view. `Inc9` works on values of type `:word9`, first extracting the *bus* value, then applying the `inc` function, taking only the bus part of the returned value, and casting it as a `:word9` result.

```
(inc (Wire b)      = (Wire (~b),b)) ^
(inc (Bus b bus) = (let (bs,fg) = (inc bus)
                    in (Bus (fg => ~b | b) bs, (fg ^ b))))
```

```
Inc9 = Word9 o FST o inc o Bits9
```

The state register component for the control unit includes not only the *mpc* register, but a 4-deep *stack* for microcode subroutine calls as well. The definition of these registers differs slightly, lacking any provision for resetting. The clock signal fed to these latches is the same system clock, and the load signal is the OR of the *pop* and *push* control signals, so latching only occurs when a stack command is issued.

```

Stack_latch9 Clocked load in_sig out_sig =
  !tm. out_sig (tm+1) =
    Clocked tm => (load tm => in_sig tm | out_sig tm) | out_sig tm

STATE_REG Clocked load
  (next_mpc,next_s0,next_s1,next_s2,next_s3)
  (   mpc,      s0,      s1,      s2,      s3) =
(MPC9      Clocked      next_mpc mpc) ^
(S_latch9 Clocked load  next_s0 s0) ^
(S_latch9 Clocked load  next_s1 s1) ^
(S_latch9 Clocked load  next_s2 s2) ^
(S_latch9 Clocked load  next_s3 s3)

```

The microcode ROM was defined by a set of introduced theorems giving the value of the ROM function output for every relevant input. The theorems were generated from the intermediate form of the microcode that was used to produce a binary image for the ROM layout generator. A typical ROM theorem is:

```

|- ROM_fun #0000000000 = #0000101100001000000000000000

```

This reliance on introduced theorems is unfortunate, as any such introduced theorems cast suspicion on the security of the results. In this case, however, an argument may be made that the addition of these theorems does no more than define a new function, and is a *conservative extension* to the HOL theory. A *conservative extension* means that there are no results provable in HOL after the extension that were unprovable before, aside from theorems which contain the introduced term. This property must hold whenever a new constant is defined in HOL.

It was necessary to specify the ROM outputs for approximately 400 different :word9 inputs. This serves as a specification for the ROM, which should be verified from a transistor level model of its implementation. However, the lack of regularity of the function, aside from its expression of the microprogram, meant each distinct output would have to be listed in the definition of the function, and the data structure that resulted was simply *too large* for the system to manipulate. So instead of *defining*

a new constant using the normal HOL system definition functions, a new constant ROM_fun was *declared*, and separate theorems defining the value of the function for each of 400 input values were created. Although the function is not totally defined, the constraints under which the system operates limit the ROM inputs to those for which the output is defined.

```

CU_DECODE
  button mpc s0 s1 s2 s3 rom_out opcode atomflag ... leqflag
  flag0 flag1 nextmpc next_s0 ... next_s3 push_or_pop
  ralu ... rcons write_bit ... wy2 dec ... resetbit30
  =
!tm. let mpc_plus_1 = Inc9 (mpc tm) in
      let read_bits = Read_field (rom_out tm)
      in ...
      let idle_state = mpc tm = #000010110
      in ...
      let selOp = (test = #0010)
      in
        ((flag0 tm = idle_state ∨ error_state) ∧
         (nextmpc tm = (selA) => A_address |
          ((pop) => s0 tm |
           ((selOp) => opcode tm | mpc_plus_1))) ∧
         ((next_s0 tm, next_s1 tm, next_s2 tm, next_s3 tm) =
          push => (mpc_plus_1, s0 tm, s1 tm, s2 tm) |
          pop => (s1 tm, s2 tm, s3 tm, #000000000) |
          (s0 tm, s1 tm, s2 tm, s3 tm)) ∧
        ...
        (ralu tm = (read_bits = #00001)) ∧
        ...
        (resetbit30 tm = (alu_bits = #1100)))

```

The CU_DECODE component has 61 outputs: 12 ALU control signals, 18 write signals, one of which controls the bidirectional i/o pads, 23 read signals, 2 state flags, a stack control signal (push_or_pop), the input for the MPC9 state register, and inputs for each of the 4 stack registers; 15 inputs: the button input, the current mpc value, the value in each stack cell, the 27 bit ROM output, the machine instruction input, and 7 status flags from the datapath. There are no hidden lines as such, but the definition makes extensive use of let bindings to simplify the term. The let bound terms represent the incremented mpc value, the 5 fields of the ROM output,

flags for each of the 3 states identified by the state flags, and complex logical values that control the next state outputs for the MPC9 and stack. The entire control unit definition is given in Figure 5.6.

```

CU SYS_Clocked button mpc s0 s1 s2 s3 opcode
atomflag bit30flag bit31flag zeroflag nilflag eqflag leqflag
flag0 flag1 ralu rmem rarg rbuf1 rbuf2 rcar rs re rc rd rmar
rx1 rx2 rfree rparent rroot ry1 ry2 rnum rn timer rtrue rfalse rcons
write_bit bidir warg wbuf1 wbuf2 wcar ws we wc wd wmar wx1 wx2
wfree wparent wroot wy1 wy2 dec add sub mul div rem
setbit30 setbit31 resetbit31 replcar replcdr resetbit30
=
? rom_out nextmpc next_s0 next_s1 next_s2 next_s3 push_or_pop.
  (STATE_REG SYS_Clocked push_or_pop
    (nextmpc, next_s0, next_s1, next_s2, next_s3)
    ( mpc,      s0,      s1,      s2,      s3))      ^
  (ROM_t mpc rom_out)                                ^
  (CU_DECODE button mpc s0 s1 s2 s3
    ...
    setbit30 setbit31 resetbit31 replcar replcdr resetbit30)

```

Figure 5.6: Definition of the CU Component

5.4.4 The Padframe

The padframe definition includes input, output, and bidirectional pads. The input and output pads are defined by stating the equality of the pairs of nodes connected by the pads. The bidirectional pads off-chip connections are bus_pins, bus_bits are the bus nodes, and mem_bits are the nodes linking the pads to the input of the READ_MEM unit described earlier. The component parts of the pads are not defined separately however, but the definition of the functionality appears in the PF definition, shown in Figure 5.7.

```

PF
    button      flag0    flag1  bidir  write_bit    % chip side %
    bus_bits    mem_bits mar_bits    rmem

    button_pin  flag0_pin flag1_pin    write_bit_pin rmem_pin
    bus_pins    mar_pins

    =
    !tm.(button t          = button_pin t) ^
        (flag0_pin t       = flag0 t)      ^
        (flag1_pin t       = flag1 t)      ^
        (write_bit_pin t   = write_bit t)   ^
        (rmem_pin t        = rmem t)       ^
        (bidir t => (mem_bits t = bus_pins t) % read from memory %
          | (bus_pins t = bus_bits t)) ^ % write to memory %
        (mar_pins t        = mar_bits t)

```

Figure 5.7: Definition of the PF Component

5.4.5 Composing the Whole

The SECD chip is the composition of the CU, DP, and PF. The parameters to the definition include the `SYS_Clocked` signal; control unit state values `mpc`, `s0`, `s1`, `s2` and `s3`; datapath state values `s`, `e`, `c`, `d`, `free`, `x1`, `x2`, `y1`, `y2`, `car`, `root`, `parent`, `buf1`, `buf2` and `arg`; input `button_pin`; state outputs `flag0_pin` and `flag1_pin`; memory interface outputs `write_bit_pin`, `rmem_pin` and `mar_pins`; and the bidirectional memory bus `bus_pins`. A multitude of signals are hidden, including the *read*, *write* and *alu* control signals generated by the CU, status flag signals, and on-chip signals connected through the pads, including `button`, `flag0`, `flag1`, `bus_bits`, `mem_bits` and `mar_bits`.

The external memory is represented only at the top level, and the conjunction of the chip description and a simple memory with Fetch and Store commands constitutes the definition of the system shown in Figure 5.8.

```

Fetch14 mar bus (mem:word14->word32) = (bus = mem mar)

Store14 mar bus (mem:word14->word32) = ( $\lambda$  a. (a = mar) => bus | mem a)

SRAM mem W_bar G_bar address_in in_out =
(!tm. (mem(SUC t) = (( $\neg$  W_bar t) => Store14(address_in t)(in_out t)(mem t)
      | mem t))  $\wedge$ 
      (W_bar t  $\wedge$  G_bar t ==> Fetch14 (address_in t)(in_out t)(mem t))))

```

```

SYS memory SYS_Clocked mpc s0 s1 s2 s3
  button_pin flag0_pin flag1_pin
  write_bit_pin rmem_pin bus_pins mar_pins
  s e c d free x1 x2 y1 y2 car root parent
  buf1 buf2 arg
=
(SECD SYS_Clocked mpc s0 s1 s2 s3
  button_pin flag0_pin flag1_pin
  write_bit_pin rmem_pin bus_pins mar_pins
  s e c d free x1 x2 y1 y2 car root parent
  buf1 buf2 arg)  $\wedge$ 
(SRAM memory write_bit_pin rmem_pin mar_pins bus_pins)

```

Figure 5.8: Definition of the SECD SYS Component

5.5 Relating the Levels

5.5.1 Memory Abstraction

Abstracting from the SRAM memory of the RTL level to the abstract memory type maintains the mark and field bits unchanged, and maps the 28 bit field to the appropriate *cons*, *integer*, or *symbol* record based on the record type bits. The memory abstraction function is defined in two steps: first a function maps records in the range of the SRAM memory function into the range of the representative type for abstract memories, and then `ABS_mfsexp_mem` is applied to this function composed with the SRAM memory:

```

Mem_Range_Abs:word32->((bool#bool)#((word14#word14)+atom))w =
  ((mark_bit w),(field_bit w)),
  ((is_symbol w) => INR(Symb(Val(Bits28(atom_bits w))))
  |(is_number w) => INR(Int(iVal(Bits28(atom_bits w))))
  | INL(car_bits w, cdr_bits w))

mem_abs (M:word14->word32) = ABS_mfsexp_mem(Mem_Range_Abs o M)

```

The `Mem_Range_Abs` function is total, and the unused record type (`#01`) gets mapped to the *cons* type record of the abstract memory. This ensures that the result of composing it with an implementation memory always returns an object in the representative type of the abstract memory, and thus applying `REP_mfsexp_mem` to `(mem_abs M)` at some value *v* will be `Mem_Range_Abs (M v)`. It further ensures the desirable property that every non-atomic record is a *cons* type record.

5.5.2 Temporal Abstraction

The coarsest grain of time used to describe the system corresponds to the points when the system is in a major state of the top-level FSM (*Idle*, *Error0*, *Error1*, and *Top_of_Cycle*). We map from this coarse granularity to the medium grain points of time when specific addresses are in the MPC9 register. The mapping is not a linear

```

⊢ Next t1 t2 f = (t1 < t2) ∧ (f t2) ∧
                  ! t.(t1 < t) ∧ (t < t2) ==> ¬ f t

⊢ (IsTimeOf 0 f t = f t ∧ !t'.(t'<t) ==> ¬ f t') ∧
  (IsTimeOf (SUC n) f t = ? t'. IsTimeOf n f t' ∧ Next t' t f)

⊢ TimeOf f n = @t.IsTimeOf n f t

⊢ when (s:num->*) (p:num->bool) = λn.s (TimeOf p n)

```

Figure 5.9: Temporal Abstraction Function Definitions

function, since the number of cycles needed to execute any machine instruction varies, and can vary between executions of the same instruction. The latter differences arise due to garbage collection calls during instruction execution, as well as varying search distances required to load values from the environment. The definitions of the required mapping functions are well described in [Mel88].

5.6 Summary

This chapter has presented two levels of formal specification of the SECD system, the top level specification and the register transfer level implementation. Important issues at the lowest level that impact the higher levels were discussed. Lastly, two important abstractions used to relate the different levels were presented.

The top level specification needed to express transformations to S-expression data objects. Their representation was complicated by the destructive operation used to create circular environment components, and the fact that the expressions could be shared by the different registers, so that a destructive operation to the E stack could also affect the S stack. An abstract memory data type was defined in which the data structures could be imbedded. A set of operations on abstract memory data objects expressed transformations to the data structures. This technique has provided a clearer specification than would have been possible by using a lower level view of a memory and system state. Additional motivation was the complexity of the machine instruction transitions, which involve multiple updates to memory, giving rise to considerably more complexity than previous microprocessor specification and verification examples, which typically were limited to single memory updates in any transition. It is also a step in the right direction to providing a suitable specification to interface with a software system.

The discussion of the low level specification focussed on fundamental issues of clock operation and initialization, and the definition of primitive memory devices. The notion of fine grain time integrated the operation of two distinct clocks, where units of time relate to the advance of either clock.

The register transfer level is an appropriate level of representation for VLSI designs. Most of the primitive components used should become available as verified library components, as the field of verification matures. The control part of the design is expressed more succinctly than the lower level, but relating the two representations is relatively simple. This representation closely resembles the informal Concrete RTL model created in designing the chip. The closeness suggests that formal specification may integrate into the design process quite naturally, providing a useful tool for expressing the information that the designers actually use.

The absence of a definition for the garbage collection function illustrates how a formal specification could be incrementally developed in a top down manner, when not all parts have been fully elaborated. The precise nature of the garbage collection function was not defined, but its *type* could be established, knowing simply that it would make some changes to a memory. By defining a constant of the appropriate type, filling in the detail could be deferred.

One of the more important and difficult parts of the specification concerned the temporal representations, and abstractions relating the different time granularities. Considerable time was spent defining precisely how the clocks should operate, and determining how to represent the behaviour at the next higher level. These issues were fundamental to the operating specification for the SECD chip [GWS89], which was the basis for the design of the chip controller [Wil89].

A great deal of computational effort resulted from the use of a well-defined *word_n* type. Previous HOL proofs relied on insecurely introduced constants ([Coh88, Joy88]), or infinite vectors of which values beyond a certain size were as-

signed an “arbitrary” value ([Joy89a]). The advantage of the well defined types used herein include the higher confidence level from avoiding the arbitrary introduction of new axioms, and the more natural definition of subfield functions on $word_n$ values. This latter advantage is not insignificant, since the resulting specifications are much more clear, and less prone to error.

The sheer size of this system made the specification a daunting task. In retrospect, some of the most challenging aspects of the project involved the development of clear and concise definitions in the specification. Definitions were repeatedly revised, as the design itself, and issues in verifying the design became better understood. The simplicity of many definitions belie the amount of care taken in their design.

The organization of the HOL theory hierarchy in which the system is defined bears mentioning. Whenever a definition is altered, all theories that inherit the definition need rebuilding. A change to a data type theory high in the hierarchy could require many hours of updating of HOL theories. The maintenance was aided considerably by keeping a well documented makefile, and rebuilding overnight. Careful design of the theory hierarchy and separation of dependencies can save a lot of time. It should be noted that even at the last stages of the proof, changes high in the theory hierarchy were made, as clumsy definitions done at the early stages of the project were replaced.

The introduction of new datatypes and function definitions was often accompanied by at least a partial axiomatisation. Thus statistics on primitive inferences for the specifications include both. Data type definitions took over 47,000 primitive inferences, and associated theorems over 37,000 primitive inferences. These high numbers result from recursive definitions on the $word_n$ datatypes. Specifications required over 1,700 primitive inferences, while proofs about the memory abstractions added up to over 28,000 primitive inferences.

Chapter 6

Verification of the SECD Design

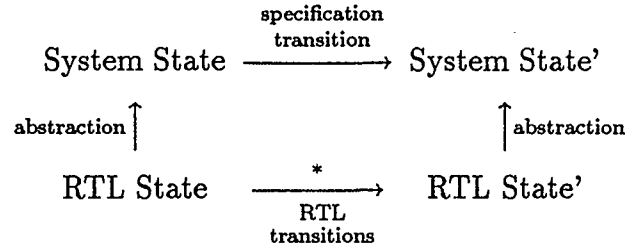
A proof of correctness relates two levels of description of a design, proving that, subject to stated constraints, the behaviour of the lower level of description ensures the behaviour specified at the higher level. Parameters to the higher level description are abstractions of the lower level ones, with regard to temporal granularity and data type. The lower level can be considered an *implementation* of the higher level *specification*. Using this terminology, the goal of a proof has the form:

$$\begin{array}{l} \text{constraints} \supset \\ \text{implementation (state) (inputs) (outputs)} \supset \\ \text{specification (abs o state) (abs o inputs) (abs o outputs)} \end{array}$$

It is desirable that a series of such correctness proofs relate the lowest level description (that closest to the physical device), to the highest level (that which comes closest to the designer's intention).

This chapter will describe the proof of correctness relating the top and register transfer levels, under normal mode of operation, and exclusive of garbage collection. The abstractions on signal and state values must map between two granularities of time, as well as different data types, particularly for the memories. The register transfer definition includes a memory function with simple Fetch and Store operations only, while the top level uses the abstract memory data type. The task of the verification is to show that the sequence of operations performed at the register

transfer level commutes with the specification transition at the abstracted top level.



This level of the proof is largely concerned with control operation, almost completely ignoring arithmetic and logical comparison operation semantics. A proof of correctness relating the lowest level and register transfer level must address these issues, defining precisely the modulo arithmetic operations. While this lowest level proof has not been completed, and is not described herein, certain logical subcomponents, for example the random logic generating the state flag signals and the bidirectional i/o pads, were proved to correctly implement the register transfer level specification, as the most straightforward and reliable way of checking their design.

The constraints on the scope of the proof are addressed first, then the form of the correctness goal and the approach used in achieving the proof are given. Following that, each major stage in the proof is outlined, using the LDF transition as the running example.

6.1 Constraints

Constraints limiting the scope of the proof to normal mode of operation and excluding garbage collection need to be translated into the formalism. Additionally, the proof is limited to valid programs, which expresses the pattern matching in the left side of transitions defining the informal machine (Table 3.3), and to properly configured free lists. Lastly, two assumptions about the decrement operation are added. These two assumptions describe things that the lowest level definition must ensure, and

may thus be eliminated when the lower level proof is undertaken. The constraints are as follows.

clock_constraint The chip is running in normal operating mode (*i.e.* the shift registers are not operating).

```
clock_constraint SYS_Clocked = ! tm. SYS_Clocked tm
```

reserved_words_constraint There are three reserved words in memory that contain the symbolic constants for NIL, T, and F.

```
reserved_words_constraint mpc memory =
!tm.(state_abs (mpc tm) = top_of_cycle) ==>
((memory tm NIL_addr = bus32_symb_append #0000000000000000000000000000) ^
 (memory tm T_addr = bus32_symb_append #0000000000000000000000000001) ^
 (memory tm F_addr = bus32_symb_append #0000000000000000000000000010))
```

well_formed_free_list Informally, the free list must be a linear *cdr* linked list, containing only *cons* cells, aside from the last cell which is NIL. Only cells that are not used in any data structure that is part of the computation (*i.e.* cells accessible from the *s*, *e*, *c*, or *d* registers), and cells that are not reserved words should appear in the free list. For the formal specification, there must be enough cells in the free list for the maximum number of *cons* operations in any SECD instruction: the AP instruction performs the operation four times. The definition of this constraint makes use of a path function which traces through an S-expression data structure in a memory, taking the *car* or *cdr* at each step according to whether the head of the `:(bool)list` argument is F or T respectively. The predicate `all_cdr_path` holds when every element in the `:(bool)list` argument will cause the *cdr* direction to be selected.

```

(path (mem:word14->word32) (v:word14) [] = v) ∧
(path mem v (CONS 1 L) =
  (is_cons (mem v)) => 1 => (path mem(cdr_bits(mem v))L)
    | (path mem(car_bits(mem v))L)
    | v)

(all_cdr_path [] = T) ∧
(all_cdr_path (CONS h t1) = h ∧ (all_cdr_path t1))

```

Using these two functions, the desired properties of the free list are defined. The `linear_free_list` function requires that if any two different path arguments return the same address, then that address is the address of the `NIL` reserved word in memory. The `not_in_free_list` predicate requires that a particular address not appear in the free list. This will apply to the high address reserved word, which is used to hold a pointer to the computation result upon executing the `STOP` instruction. The `nonintersecting` predicate states that no path in the free list leads to the same address as a path from a given cell. Lastly, the predicate `n_cells_in_free_list` requires that the first `n` addresses following `cdr`'s from the *free* address all point to *cons* cells in memory. The function `nth` applies its 2^{nd} argument n times to the last argument (*i.e.* $nth\ n\ f\ b = f^n b$).

```

linear_free_list (mem:word14->word32) (free:word14) =
  !l1 l2. (all_cdr_path l1) ∧ (all_cdr_path l2) ==>
    ¬(l1 = l2) ==>
      (path mem free l1 = path mem free l2) ==>
        (path mem free l1 = NIL_addr)

not_in_free_list (mem:word14->word32) (free:word14) (v:word14) =
  !l. (all_cdr_path l) ==> ¬(path mem free l = v)

nonintersecting (mem:word14->word32) (free:word14) (v:word14) =
  !l1 l2. (all_cdr_path l2) ==>
    ¬(path mem free l2 = NIL_addr) ==>
      ¬(path mem v l1 = path mem free l2)

n_cells_in_free_list (mem:word14->word32)(free:word14) (n:num) =
  !n'. (n' < n) ==> (is_cons (nth n' (mem o cdr_bits)(mem free)))

```

The full constraint conjoins the above predicates applied to the appropriate values. From this and the `reserved_words_constraint`, theorems are derived which assert that the first four addresses in the free list are not `NIL_addr`, and that these same addresses are all distinct.

```

well_formed_free_list memory mpc free s e c d =
  !tm. (state_abs (mpc tm) = top_of_cycle) ==>
    (n_cells_in_free_list (memory tm) (free tm) 4) ∧
    (linear_free_list (memory tm) (free tm)) ∧
    (let nonintersecting_with_free_list =
      (nonintersecting (memory tm) (free tm))
    in
      (nonintersecting_with_free_list (s tm) ∧
       nonintersecting_with_free_list (e tm) ∧
       nonintersecting_with_free_list (c tm) ∧
       nonintersecting_with_free_list (d tm)) ∧
      (not_in_free_list (memory tm) (free tm) NUM_addr)

```

valid_program_constraint The state of the machine must pattern match with the left side of the abstract machine transition for one of the 18 implemented machine instructions, or have a problem loaded in memory when the machine is directed to start computation. The constraint gives a quite detailed specification of the type and in some cases the value of each record in memory involved in the pattern match. The constraint on record types is necessary for the abstraction from simple to abstract memory types. This `valid_program_constraint` must be assured by the correctness of the compilation process. As with the `free_list_constraint` it is restricted to times after the machine is initialised.

The component constraining the arguments for the LD instruction requires further explanation. First it is required that the argument is a cons cell, and each branch points to a number cell. The integer values represented by these cells are nonnegative, and furthermore, the environment has a value in the position indicated; *i.e.* it is composed of at least m lists, and the m^{th} list has at least n elements.

```

valid_program_constraint memory mpc button_pin s e c d =
!tm.
  (((state_abs(mpc tm) = idle) ∧ button_pin tm) ==>
    (is_cons(memory tm NUM_addr)) ∧
    (is_cons(memory tm(car_bits(memory tm NUM_addr)))) ∧
  ((state_abs(mpc tm) = top_of_cycle) ==>
    let head_c = memory tm(c tm)
    in
    ((is_cons head_c) ∧
    let instr' = memory tm(car_bits head_c)
    and next_c = cdr_bits head_c
    in
    ((is_number instr') ∧
    let instr = atom_bits instr'
    in
    (((instr = ^LD_instr28) ∧
    (is_cons(memory tm next_c)) ∧
    let arg_cons_cell = memory tm(car_bits(memory tm next_c))
    in
    ((is_cons arg_cons_cell) ∧
    let m_cell = memory tm(car_bits arg_cons_cell)
    and n_cell = memory tm(cdr_bits arg_cons_cell)
    in
    ((is_number m_cell) ∧ (is_number n_cell) ∧
    let m = (iVal(Bits28(atom_bits m_cell)))
    and n = (iVal(Bits28(atom_bits n_cell)))
    in
    ((~NEG m) ∧ (~NEG n) ∧
    (!m'.(m'<=(pos_num_of m)) ==>
      (is_cons(nth m'((memory tm) o cdr_bits)
      (memory tm(e tm)))))) ∧
    (!n'.(n'<=(pos_num_of n)) ==>
      (is_cons(nth n'((memory tm) o cdr_bits)
      (memory tm(car_bits
      (nth (pos_num_of m)
      ((memory tm) o cdr_bits)
      (memory tm(e tm))))))
    )))))))) ∨ ...
    ((instr = ^LDF_instr28) ∧
    (is_cons(memory tm next_c)) ) ∨ ...

```

DEC28 assumptions The first assumption states that the :num equivalent obtained by decrementing a 28-bit value that represents a positive integer is the same value obtained by taking the predecessor of the :num equivalent represented by the original value. The second states that if the 28-bit value rep-

resents a positive integer, the result of applying DEC28 will produce a value that represents a nonnegative integer. Both of these properties are what is expected of a decrement operation, and must be assured by the lower level implementation. The properties enable us to assure termination of LD instruction sequences retrieving values from the environment.

```

DEC28_assum1 =
  !w28. PRE(pos_num_of(iVal(Bits28 w28))) =
    pos_num_of(iVal(Bits28((atom_bits o DEC28)w28)))

DEC28_assum2 =
  !w28. (POS(iVal(Bits28 w28)))==>
    ~(NEG(iVal(Bits28((atom_bits o DEC28)w28))))

```

6.2 Structure of the proof

The goal for the proof of correctness is shown in Figure 6.1. Most of the signals are simply abstracted from the medium to the coarse time granularity. Additionally, the memory and state are abstracted to the appropriate data type. The temporal abstraction of the single input signal argument to the top level specification (*i.e.* the `button_pin`) should be carefully considered. It can be shown that this input affects the next state of the machine at the medium time granularity in only 3 places in the microcode, and these 3 places correspond to points in coarse grain time, since they are all *major states*.

Although the lower (RTL) level definition of the system for this proof consists of the wiring together of high-level components defined behaviourally, it is not feasible to undertake the proof of the goal in Figure 6.1 directly. The size of terms generated alone makes this impossible to manage. The proof is instead undertaken in several stages.

First, a simplified, flattened specification for the register transfer level definition was obtained. Second, this specification was used to derive theorems for the change

```

((clock_constraint SYS_Clocked)                ^
 (reserved_words_constraint mpc memory)         ^
 (well_formed_free_list memory mpc free s e c d) ^
 DEC28_assum1 ^ DEC28_assum2                    ^
 (valid_program_constraint memory mpc button-pin s e c d)) ==>
  (SYS memory SYS_Clocked
   mpc s0 s1 s2 s3
   button_pin
   flag0_pin flag1_pin write_bit_pin rmem_pin
   bus_pins mar_pins
   s e c d free
   x1 x2 y1 y2 car root parent buf1 buf2 arg) ==>
  SYS_spec ((mem_abs o memory) when (is_major_state mpc))
            (s when (is_major_state mpc))
            (e when (is_major_state mpc))
            (c when (is_major_state mpc))
            (d when (is_major_state mpc))
            (free when (is_major_state mpc))
            (button_pin when (is_major_state mpc))
            ((state_abs o mpc) when (is_major_state mpc))

```

Figure 6.1: RTL \supset top level goal

of state effected by each microcode instruction. These theorems represent the cumulative effect of asynchronous events through the phases of one full clock cycle, and roughly correspond to the “Phase level” of description of Anceau [Anc86]. Third, a sort of microcode level simulation used the microinstruction theorems to step through the microcode to produce theorems that summarized the behaviour of instruction sequences. This corresponds to the “Microprogramming level” of Anceau. Fourth, the sequence theorems were used to prove a liveness property for the system. And last, the state transitions resulting from the register transfer level definition were proved to correspond to those defined at the top level, for all transitions possible under the constraints.

At the time of this writing, the first four stages are fully completed. The last stage has been analysed, the top level goal was split into subgoals for each transition, as well as the initial state, and a sample proof of one transition, that for the LDF instruction, has been constructed. The completion of the proof foresees no major obstacles, but requires, as has each preceding stage, considerable investment in time spent managing the proof. As with the specification of the system in the previous chapter, the size of theorems prevents inclusion in their entirety. Using the HOL pretty printer, a complete listing of the theorems is estimated at over 15,000 lines (250 pages!), three times as much as the definitions. A complete listing of the HOL proof is presently under preparation in [Gra90b]. A description of each stage of the proof follows.

6.3 Unfolding the System Definition

The register transfer level of description consists of a conjunction of behavioural descriptions of major components. For example, the control unit consists of a *state register* (with five fields), a *ROM* and a *decode* unit. A specification for the conjunction of these three subcomponents is not appreciably simpler than the conjunction of the component behaviours individually. It is only when the whole system is assembled that simplifications become possible. A good example involves the *one_asserted* property of the ALU control lines. The property is a constraining part of the definition of the ALU, but it is provable from the control unit definition that it holds for the ALU control lines generated by the control unit definition of the control unit. This constraint can be eliminated when the control unit and datapath are conjoined.

Thus the simple hierarchical approach, where both implementations and specifications are defined independently for each component of the hierarchy and a correctness proof of the top component is achieved by using the correctness results of

subcomponent parts, was abandoned. Instead starting with the definitions of the major subcomponents of the chip, each was simplified by expanding all definitions, and UNWIND'ing¹ existentially quantified variables (hidden wires) where possible. When the only remaining occurrence of the existentially quantified variable is the left side of an equation, it may be PRUNE'd, or eliminated from the expression. This was continued up to the top of the hierarchy where more substantial simplifications could be made. The process of creating these proofs was relatively straightforward, complicated only by the relatively large size of terms involved. However, the large term size forced extremely careful management of the proof process. Only primitive rules and tactics were usable, the more powerful tactics would easily exhaust memory, and thus the problem was compounded.

Simplifying the datapath and control unit was undertaken first. Rather than using a *forward proof* approach, since it was easier to prove two terms equivalent than to “massage” a term into a particular desired form, considerable time was spent defining goals for tactical proofs. The goal for the datapath introduced two antecedents: the `one_asserted` property applied to the ALU control lines, and the `clocked_constraint`, while the consequent was the equality of the DP and its expanded and simplified version. The proof expanded all definitions and let expressions, rewrote with the two antecedents to simplify the resulting expressions, flattened and reordered all conjuncts, and moved the universally quantified time parameter out to enclose all conjuncts. The single existentially quantified value `alu` remained, outside the level of (and enclosing) the time parameter in the theorem.

The control unit was treated similarly. The goal was a simple equality of the CU expression and its expanded and simplified version. All definitions were expanded and let expressions unfolded, the expression for the next state values of the five

¹UNWIND is a HOL conversion which unfolds the equations for existentially quantified variables that occur as conjuncts in an expression.

field state register was split into distinct expressions for each field, the time parameter in moved out to enclose all conjuncts, the seven existentially quantified values: `rom_out`, `nextmpc`, `next_s0`, `next_s1`, `next_s2`, `next_s3`, and `push_or_pop` were unwound and pruned, and a few logical simplifications were made.

A theorem showing that the ALU control outputs from the control unit have a `one_asserted` property was proved. The intensive computation required to prove inequality of two constants of a specified *word_n* type made it desirable to prove an exhaustive set of theorems for all possible values of each subfield of the microcode ROM output. Rather than repeatedly proving the cases for each of the 400 ROM addresses, this required 68 theorems² all told. These theorems were of the form:

```

Alu_base_0001 =
⊢ (Alu_field(ROM_fun(mpc tm)) = #0001) ==>
  ((Alu_field(ROM_fun(mpc tm)) = #0001) = T) ∧
  ((Alu_field(ROM_fun(mpc tm)) = #0010) = F) ∧
  ...
  ((Alu_field(ROM_fun(mpc tm)) = #1011) = F) ∧
  ((Alu_field(ROM_fun(mpc tm)) = #1100) = F)

```

In addition, theorems for the value of the `Inc9` function, used to calculate the next mpc address for sequential microcode, for each of 400 addresses were proved, typified by the theorem for the lowest address.

```

⊢ Inc9 #000000000 = #000000001

```

At the next level, SECD is simplified by expanding the PF and DP components, but not the CU. The `one_asserted` constraint is eliminated, the `clocked_constraint` is included, hidden lines connected directly to input or output pads are unwound and pruned, including `button`, `flag0`, `flag1`, `write_bit`, and `mar.bits`, and all remaining existentially quantified variables are moved to the outermost level.

²A theorem for each possible value is required, including the no-op value, hence there are 13 possible values for the alu and test fields, 18 for the write field, and 24 for the read field.

```

Base_thm =
[clock_constraint SYS_Clocked; ^SYS_imp]
⊢ ? bus_bits_t mem_bits_t alu_t.
  (mpc(SUC tm) =
    (((Test_field(ROM_fun(mpc tm)) = #0001) ∨
      (Test_field(ROM_fun(mpc tm)) = #0011) ∧
        field_bit bus_bits_t
          ∨
      ...
      (Test_field(ROM_fun(mpc tm)) = #1011))
    => A_field(ROM_fun(mpc tm))
    |((Test_field(ROM_fun(mpc tm)) = #1100)
    => s0 tm
    |((Test_field(ROM_fun(mpc tm)) = #0010)
    => Opcode arg tm
      | Inc9(mpc tm)))))) ∨
  ...
  (memory(SUC tm) =
    ((Write_field(ROM_fun(mpc tm)) = #00001)
    => Store14(mar_pins tm)(bus_pins tm)(memory tm)
      | memory tm)) ∨
    ((Read_field(ROM_fun(mpc tm)) = #00010) ==>
      (bus_bits_t = mem_bits_t)) ∨
    ...
    ((Alu_field(ROM_fun(mpc tm)) = #0001) ==>
      (alu_t = DEC28(atom_bits(arg tm)))) ∨
    ...
    (¬(Write_field(ROM_fun(mpc tm)) = #00001) ∧
      (Read_field(ROM_fun(mpc tm)) = #00010) ==>
        (bus_pins tm = memory tm(mar_pins tm))) ∨
    ...
    (s tm = ((Write_field(ROM_fun(mpc tm)) = #00110)
      => cdr_bits bus_bits_t | s(PRE tm))) ∨
    ...
    (flag1_pin tm = (mpc tm = #000101011) ∨ (mpc tm = #000011000))

```

Figure 6.2: Base_thm: the RTL definition simplified

At the top of the definition hierarchy, the static RAM definition of memory was added in, and the CU and SECD simplifications were applied to create one flattened expression for the SYS implementation. All internal read and write control lines

were unwound and pruned. The only remaining existentially quantified variables, `bus_bits`, `mem_bits`, and `alu`, were quantified over all conjuncts. The most important step involved moving the time parameter outside the existentially quantified internal lines, and replacing these time varying lines with static values. The equivalence of this transformation was proved giving the following theorem.

$$\vdash (! (t:\text{num}). \quad ?(a_t:*) . P \ t \ a_t) =$$

$$(? (a:\text{num} \rightarrow *) . ! (t:\text{num}). P \ t(a \ t))$$

A set of specialised conversions was designed to pinpoint precisely where in the goal term to apply the theorem, resulting in the theorem of Figure 6.2. Importantly, the time parameter can be generalized over the entire theorem conclusion, so that the expression can be evaluated when constraining some value at a given time t_m ; in fact the simplified expression for the system description was evaluated repeatedly under the constraint `mpc tm = x`, where `x` is one of the 400 microcode addresses. From this stage onward, equivalence was no longer attempted, so the final form of the theorem has the `clock_constraint` and the `SYS` definition³ as assumptions, and the conclusion is the simplified expression for the system, with the three existentially quantified variables. Also the conjunct giving the initial value of the `mpc` (*i.e.* at time $t_m = 0$) was dropped from the expression. Representative samples of each type of conjunct in the resulting theorem are shown in Figure 6.2. Generating this stage of the proof required over 500,000 primitive inferences, with just about half this number devoted to the proofs for the values of the `Inc9` function.

6.4 Phase Level: Effect of Each Microinstruction

The simplified expression for the system was next utilised to produce theorems giving the effect of each microinstruction execution on the system state. A sample

³This constraint will be abbreviated as `SYS_imp` in the following material.

theorem for the system when the value #001100001 is in the MPC9 register is shown in figure 6.3. This is the first instruction of the microcode sequence for the LDF instruction, and effects a transfer of the content of the E_reg register to the X2_reg register.

```

SYS_lemma_97 =
[clock_constraint SYS_Clocked; ^SYS_imp]
⊢ (mpc tm = #001100001) ==>
  (mpc(SUC tm) = #001100010)      ∧
  (s0(SUC tm) = s0 tm)            ∧
  (s1(SUC tm) = s1 tm)            ∧
  (s2(SUC tm) = s2 tm)            ∧
  (s3(SUC tm) = s3 tm)            ∧
  (memory(SUC tm) = memory tm)    ∧
  (x2 tm = e(PRE tm))            ∧
  (rmem_pin tm = F)                ∧
  (buf1 tm = buf1(PRE tm))        ∧
  (buf2 tm = buf2(PRE tm))        ∧
  (mar_pins tm = mar_pins(PRE tm)) ∧
  (s tm = s(PRE tm))              ∧
  (e tm = e(PRE tm))              ∧
  (c tm = c(PRE tm))              ∧
  (d tm = d(PRE tm))              ∧
  (free tm = free(PRE tm))        ∧
  (x1 tm = x1(PRE tm))            ∧
  (car tm = car(PRE tm))          ∧
  (arg tm = arg(PRE tm))          ∧
  (parent tm = parent(PRE tm))    ∧
  (root tm = root(PRE tm))        ∧
  (y1 tm = y1(PRE tm))            ∧
  (y2 tm = y2(PRE tm))            ∧
  (write_bit_pin tm = T)            ∧
  (flag0_pin tm = F)                ∧
  (flag1_pin tm = F)                ∧

```

Figure 6.3: Theorem for execution of microcode instruction at address 97

The large number of such theorems (337 excluding garbage collection sequences) required that they be generated without direct user intervention. A single proof function was designed for this purpose, capable of generating the required theorems

in a forward proof manner, without prior statement of the specific content of each. Although the verifier did not know in advance what each theorem would state, this could have been determined. Specific samples were worked out for help with developing the proof function. As failures or unsatisfactory theorems were observed, the proof function was revised. Generating the set of theorems took approximately 36 hours for each attempt, running on a dedicated Sun 3/60 workstation with 16 megabyte memory, and required in excess of 4.6 million primitive inferences. The theorems were divided among seven theories, each containing adjacent instruction code sequences, to reduce search times for theorems at the next proof stage.

The proof function works as follows:

1. Fetch the theorem defining the value of the microcode function ROM_fun for the given address, add the assumption that $\text{mpc } t_m$ is equal to the address, and substitute this term for the address in the theorem. The resulting theorem has the form:

. $\vdash \text{ROM_fun}(\text{mpc } t_m) = \#000000000000000000110001000$

In this case and similarly in all following theorems, the dot to the left of the turnstile symbol represents the assumption $\text{mpc } t_m = \#001100001$.

2. Apply a conversion to transform the 27 bit constant value output to the equivalent form expressed as Word27 applied to a 27 bit $:(\text{bool})\text{bus}$, in this case $\text{Word27}(\text{Bus } F(\text{Bus } F \dots (\text{Bus } T(\text{Bus } F(\text{Bus } F(\text{Wire } F))))))$. For each field of the ROM output, apply the field selector function, expand its definition, and convert the resulting *bus* form of the field value back into the *word_n* constant representation. Five theorems, one for each field, are produced, typically of the form:

. $\vdash \text{Read_field}(\text{ROM_fun}(\text{mpc } t_m)) = \#01000$

. $\vdash \text{A_field}(\text{ROM_fun}(\text{mpc } t_m)) = \#000000000.$

3. The four theorems for the `Test_`, `Alu_`, `Write_`, and `Read_field` values are then each resolved with the control unit theorem for the appropriate constant and field, and split into conjuncts, giving a set of theorems each having the form:

. $\vdash (\text{Read_field}(\text{ROM_fun}(\text{mpc } t_m)) = \#00001) = F$, or
 . $\vdash (\text{Read_field}(\text{ROM_fun}(\text{mpc } t_m)) = \#01000) = T$.

4. The relevant `Inc9` theorem is retrieved from `Inc9_proofs` and specialised:

. $\vdash \text{Inc9}(\text{mpc } t_m) = \#001100010$.

5. The expressions for state flag values are evaluated with the `mpc` value substituted in, to generate a theorem of the form:

. $\vdash (\text{mpc } t_m = \#000010110) \vee (\text{mpc } t_m = \#000011000) = F$.

6. The set of all the preceeding theorems is assembled as a list, and substituted into the `Base_thm` from the previous stage, using the primitive inference rule `SUBST` and a template. A series of rewrites reduces constant boolean expressions and removes eliminated existentially quantified variables, and is followed by a `PRUNE`'ing of remaining existentially quantified variables.

7. Any remaining existentially quantified variables will not have been removed because they occur in a conjunct in which only one field is defined. This occurs when a 14 bit value is transferred over the bus, for example. They are eliminated by a specialised rule that uses theorems about the existence of fields of an existentially quantified variable:

`car_bits_thm` = $\vdash !y. ?x. \text{car_bits } x = y$
`cdr_bits_thm` = $\vdash !y. ?x. \text{cdr_bits } x = y$.

8. Discharge the original assumption of the value of the `mpc`.

Only one of the 337 microinstruction theorems required additional simplification. The last microinstruction (308) of the sequence for the `SECD STOP` instruction

stores a pointer to the result of the computation in the *cdr* field of the highest memory address. This is effected by reading the *S_reg* register and writing to memory. Only the *cdr* field of the bus is driven, so only that field of the memory record has a determinable value, and even the record type is indeterminate. This is clearly an example of imprecise specification that formal methods will bring to our attention. It is a moot point whether this could, in practice, present any difficulty in the system operation. However, the problem here was the inability of the proof function to eliminate the existentially quantified value *bus_bits_t*, leaving the subterm:

```
. ⊢ ... (?bus_bits_t.
      (memory(SUC tm)=Store14(mar_pins(PRE tm))bus_bits_t(memory tm)) ∧
      (cdr_bits bus_bits_t = s(PRE tm)) ∧ ...
```

To resolve this problem, the expression for the state of memory was changed to:

```
(!a.((a = mar_pins(PRE tm))
    => (cdr_bits(memory(SUC tm)a) = s(PRE tm))
    | (memory(SUC tm)a = memory tma))).
```

This expression only defines part of the memory output at the subject address, and allowed the expression for *bus_bits_t* to be substituted in the memory expression, leaving a single occurrence of *bus_bits_t* in the theorem, which could be pruned.

6.5 Microprogramming Level: Symbolic Execution

Each one of the 337 lemmas about the RTL description gave the immediately following state in terms of the present state. These results were next combined to give the state after a number of steps from some starting state.

The top level system definition, *SYS_spec*, describes a four state finite state machine, with two transitions from each of three states controlled by the button input, an initial transition from a deterministic startup state, and 18 possible transitions,

one for each machine instruction code, from the fourth state. Further, four instruction sequences have a branch conditional upon some function of the state, and one (the sequence for the LD instruction) contains two loops. Thus there are minimally 29 paths to consider. Additionally, several instruction sequences call subroutines, and subroutine calls are nested. Under the `well_formed_free_list` constraint, the subroutines have a deterministic execution time.

The proofs generated in this section required a total of 1,144,000 primitive inferences. The number of inferences for each proof was in quite direct proportion to the number of microcode instructions in the sequence.

This section begins with a simple proof for the initial transition, and continues with a description of the general approach to developing the longer sequence proofs, using the LDF instruction as a typical example. Following this is a description of the more complicated LD sequence proof, and the application of the developed methodology to the proofs for the most complex AP and RAP instruction sequences.

6.5.1 The initial transition

The first proof stage produced an expanded definition of the system which included a conjunct giving the initial mpc value. Taking this first conjunct gives the theorem:

$$\dots \vdash \text{mpc } 0_m = \#000000000.$$

Applying *modus ponens* to `SYS_lemma_0` and this theorem, and taking the first conjunct, gives the theorem:

$$\dots \vdash \text{mpc}(\text{SUC } 0_m) = \#000010110.$$

Applying the predicate *is_major_state* to *mpc* at both time 0_m and `SUC 0_m` , then expanding all definitions produces the two theorems:

$$\dots \vdash \neg \text{is_major_state } \text{mpc } 0_m$$

$$\dots \vdash \text{is_major_state } \text{mpc}(\text{SUC } 0_m).$$

The final theorem appears deceptively simple:

$$\dots \vdash \text{TimeOf}(\text{is_major_state } \text{mpc}) 0_m = \text{SUC } 0_m.$$

The proof begins by expanding the definition of `TimeOf`, but this introduces the `SELECT` operator `@` used in its definition. It is necessary to show that there is a unique value which satisfies the body of the operator, and a specially devised tactic `SELECT_UNIQUE_TAC` [Gra90a] does just that, splitting the goal into two parts. The first requires a proof that the value `SUC 0m` satisfies the predicate, while the second must show that the satisfying value is unique. This property is provable from the definition of `IsTimeOf`, and is captured by the theorem `IsTimeOf_IDENTITY`.

$$\vdash !n \ f \ t1 \ t2. \text{IsTimeOf } n \ f \ t1 \wedge \text{IsTimeOf } n \ f \ t2 \implies (t1 = t2)$$

6.5.2 The general approach: LDF

The proof for the initial transition was simply concerned with the time the machine gets to a *major state*, and what that state is. For most of the remaining transitions, the state of the rest of system, comprising the external memory as well as the on-chip registers, is of equal concern.

Central to the method is the idea of symbolic execution. Given a base theorem expressing the accumulated change to the system state after execution of some microcode sequence, one conjunct states the value of `mpc` at the next point in time. The next microinstruction is symbolically executed by applying `MATCH_MP` (a HOL variant of Modus Ponens) to the appropriate microinstruction lemma from the previous stage and the conjunct from the base lemma, and rewriting the resulting theorem with the base theorem conjuncts to include the previous accumulated computation. It was also necessary (and convenient) to prove whether or not the system was in a *major state* at each step. This second result was accumulated in a theorem giving a defined interval in which the system was not in a *major state*. The sequence

proof would end when a *major state* was reached, whereupon a specific value for the Next time the system is in a *major state* is determined.

Once again, the number of theorems demanded minimizing user intervention. A proof function was designed to effect the single step computation described above, producing a new pair of theorems from a theorem pair argument. A higher level function calls it recursively to effect a series of steps, producing a pair of theorems summarizing the computation of a sequence of microinstructions. The sequence for a machine execution, in this case the LDF instruction, is summarized in the two theorems in Figure 6.4.

The mpc value at the end of the sequence corresponds to the `top_of_cycle` state. The memory has had two cells altered: the first cell has pointers to the code argument to the LDF instruction, and the current environment `e`, effectively representing a closure. The second cell has pointers to the first cell, and the original `s` value. The `s` pointer is updated to point to the latter cell, effectively storing the closure on top of the `s` stack. The `c` pointer now points to the rest of the control list following the LDF instruction and its argument. Notice that the `cdr` operation is performed on different memories: the first on the original memory, and the second after the memory is updated with the two rewritten cells. The `free` pointer has similarly been updated to the the third cell in the original free list. The content of the other registers is irrelevant to the abstracted state, and they are removed in the final theorem shown. The expression for the updated memories appears several times, so they have been bound in `let` expressions to simplify reading.

The first sequences proved were the subroutines. Base theorems were of the form (this is for the *consx1x2* subroutine):

$$\begin{aligned} \dots \vdash & (\text{mpc } t_m = \#101000101) \wedge ((\text{free}(\text{PRE } t_m) = \text{NIL_addr}) = F), \text{ and} \\ & \vdash !t_m".((\text{PRE } t_m) < t_m") \wedge (t_m < t_m) ==> \neg \text{is_major_state mpc } t_m". \end{aligned}$$

with the assumptions of the first matching the conjuncts of its conclusion. The

```

LDF_state =
..... ⊢ let mem1 =
      Store14 (free tm)
        (bus32_cons_append
          #00 RT_CONS
          (car_bits(memory tm(cdr_bits(memory tm(c tm))))
          (e tm))
          (memory tm)
    in
    let mem2 = Store14 (cdr_bits(memory tm(free tm)))
      (bus32_cons_append #00 RT_CONS(free tm)(s tm))
      mem1
    in
    ((mpc(tm+26) = #000101011) ∧
     (memory(tm+26) = mem2) ∧
     (s(tm+26) = cdr_bits(memory tm(free tm))) ∧
     (e(tm+26) = e tm) ∧
     (c(tm+26) = cdr_bits(mem2(cdr_bits(mem2(c tm)))) ∧
     (d(tm+26) = d tm) ∧
     (free(tm+26) = cdr_bits(mem1(cdr_bits(memory tm(free tm))))))

LDF_Next = ..... ⊢ Next tm(tm+26)(is_major_state mpc)

The assumptions are:
[ clock_constraint SYS_Clocked
; ~SYS_imp
; reserved_words_constraint mpc memory
; well_formed_free_list memory mpc free s e c d
; mpc tm = #000101011
; opcode_bits(memory tm(car_bits(memory tm(c tm))) = #000000011
]

```

Figure 6.4: Microprogramming level theorems for LDF instruction

second theorem is vacuously true, since it describes the property over an empty interval.

The first theorem evolves at each step as described earlier. The second uses the theorem `Next_step`:

$$\vdash !ts \text{ tf } f. (\neg f \text{ tf}) \wedge$$

$$\begin{aligned}
& (!t. (ts < t) \wedge (t < tf) ==> (\neg f \ t)) ==> \\
& (!t. (ts < t) \wedge (t < (SUC \ tf))) ==> (\neg f \ t)
\end{aligned}$$

Once the value of the mpc at the given time is used to prove the system is *not* in a *major state*, it is combined with the accumulated interval theorem and *Next_step* to give a new theorem, covering an interval one time unit longer. The results for the *consx1x2* subroutine are shown in Figure 6.5.

The proof strategy for the instruction transition sequences saw the completion of theorems for each of the subroutines exclusive of garbage collection. Next, the sequence common to all instructions, essentially a fetch instruction operation starting at *top_of_cycle* state, was proved up to where the control stream branches to each individual instruction code sequence. For each of the 18 instructions, an assumption about the value of the next SECD instruction was added to the base theorem, and one more step proved. The results were used as arguments to the proof function, which advanced through the microinstruction sequence until a *major state* was encountered.

The starting theorem for the proof function in these cases was slightly different from that used for the subroutines. The final theorem had to express the final state for all state variables at the same time, in terms of the initial values at time t_m , unlike the form of the microinstruction theorems typified by the example in Figure 6.3. Thus, only the conjuncts describing the next state of mpc, s0, s1, s2, s3, and memory from the microinstruction lemma for the *top_of_cycle* address form the base theorem. The starting theorem for the range in which it is not in a *major state* differs as well:

$$\vdash !t_m. (t_m < t_m) \wedge (t_m < (SUC \ t_m)) ==> \neg \text{is_major_state mpc } t_m.$$

The proof function is composed of a recursive function applied to a function which can prove single steps. The decision tree for the recursive function is shown in Figure 6.6. When the next instruction is not a simple constant, it must be either a conditional branch or a case split on the SECD instruction code. In the latter case, user

```

Consx1x2_state =
.... ⊢ (mpc(SUC(SUC(SUC(SUC tm)))) = s0 tm) ∧
      (s0(SUC(SUC(SUC(SUC tm)))) = s1 tm) ∧
      (s1(SUC(SUC(SUC(SUC tm)))) = s2 tm) ∧
      (s2(SUC(SUC(SUC(SUC tm)))) = s3 tm) ∧
      (s3(SUC(SUC(SUC(SUC tm)))) = #000000000) ∧
      (memory(SUC(SUC(SUC(SUC tm)))) =
        Store14 (free(PRE tm))
          (bus32_cons_append #00 RT_CONS(x1(PRE tm))(x2(PRE tm)))
            (memory tm)) ∧
      (bus_pins(SUC(SUC(SUC tm))) =
        bus32_cons_append #00 RT_CONS(x1(PRE tm))(x2(PRE tm))) ∧
      (rmem_pin(SUC(SUC(SUC tm))) = F) ∧
      (buf1(SUC(SUC(SUC tm))) = buf1(PRE tm)) ∧
      (buf2(SUC(SUC(SUC tm))) = buf2(PRE tm)) ∧
      (mar_pins(SUC(SUC(SUC tm))) = free(PRE tm)) ∧
      (s(SUC(SUC(SUC tm))) = s(PRE tm)) ∧
      (e(SUC(SUC(SUC tm))) = e(PRE tm)) ∧
      (c(SUC(SUC(SUC tm))) = c(PRE tm)) ∧
      (d(SUC(SUC(SUC tm))) = d(PRE tm)) ∧
      (free(SUC(SUC(SUC tm))) = cdr_bits(memory tm(free(PRE tm)))) ∧
      (x1(SUC(SUC(SUC tm))) = x1(PRE tm)) ∧
      (x2(SUC(SUC(SUC tm))) = x2(PRE tm)) ∧
      (car(SUC(SUC(SUC tm))) = car(PRE tm)) ∧
      (arg(SUC(SUC(SUC tm))) = arg(PRE tm)) ∧
      (parent(SUC(SUC(SUC tm))) = parent(PRE tm)) ∧
      (root(SUC(SUC(SUC tm))) = root(PRE tm)) ∧
      (y1(SUC(SUC(SUC tm))) = y1(PRE tm)) ∧
      (y2(SUC(SUC(SUC tm))) = y2(PRE tm)) ∧
      (write_bit_pin(SUC(SUC(SUC tm))) = F) ∧
      (flag0_pin(SUC(SUC(SUC tm))) = F) ∧
      (flag1_pin(SUC(SUC(SUC tm))) = F)

```

```

Consx1x2_nonmajor =
.... ⊢ !t'm. (PRE tm) < t'm ∧ t'm < (SUC(SUC(SUC(SUC tm)))) ==>
      ¬ is_major_state mpc t'm

```

The assumptions are:

```

[ clock_constraint SYS_Clocked
; ~SYS_imp
; (free(PRE tm) = NIL_addr) = F
; mpc tm = #101000101
]

```

Figure 6.5: Microprogramming level theorems for consx1x2 subroutine

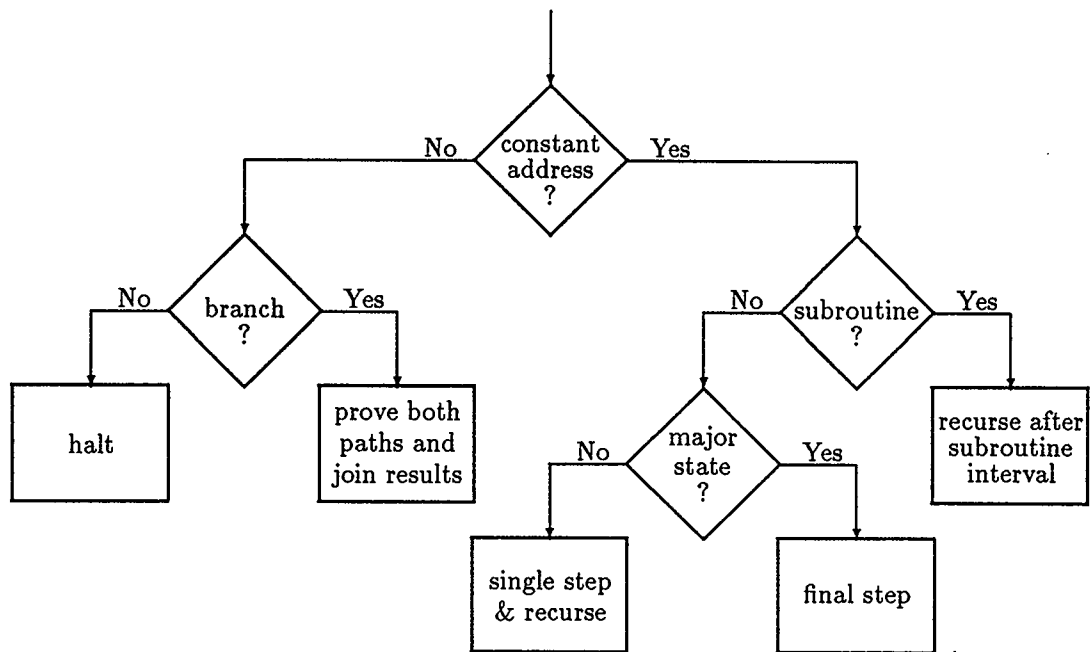


Figure 6.6: Decision Tree for Recursive Microprogramming Level Proof Function

intervention is required to add an appropriate assumption determining which instruction code is to be executed, as well as any other assumptions related to that instruction. These assumptions are taken directly from the `valid_program_constraint`. In the case of a simple branch, both legs of the proof are followed, one by adding the conditional expression as an assumption, and the other by adding the negation. Subroutine calls encountered in the computation sequence use the subroutine theorems to accumulate the effect of the subroutine execution, using an interval proof function. Most instructions will cause a recursive call on the result of executing one more microinstruction. When a *major state* is encountered execution must stop, but only after the step function completes updating the state of the datapath registers. This is required since these registers are not part of the starting state theorem, and their final value needs to catch up one step. The state values that are not relevant

to the next level of the proof are culled out at this point. At the last step, the time parameters are changed to a sum of t_m and a constant. A further simplification binds the intermediate values of memory in `let` expressions to produce a theorem as in Figure 6.4.

6.5.3 Proving the complex sequences

Three sequences were impossible to prove using the proof function described. Two of them, the sequences describing the effect of executing an AP or RAP instruction, produced terms which were simply too large, and memory faults occurred. Two aspects of the term were excessive in size: expressing the time parameter for the resulting state in terms of repeatedly applying SUC to the start time t_m , and the repeated occurrence of the expression for the state of memory after successive store operations. Both of these are reduced in the sample LDF theorem of Figure 6.4, but the reduction takes place after the theorem proof is essentially complete. As an example, the completely unfolded (*i.e.* with all `let` expressions removed) version of the state theorem for the RAP instruction is 2,436 lines long, compared to 50 lines for the form containing `let` expressions. It is considerably more difficult, if it is indeed possible, to effect the proof without unfolding `let` bindings in intermediate results. The overhead of converting time parameters to simple addition of with a constant is reduced by performing it once only, at the end of the proof.

The proof for the LD sequence had to demonstrate termination of execution of the microcoded “while” loops, and the accumulated effect of executing these loops was proven first. This part of the proof method provided an approach which was later successfully applied to the previously described sequences. The major difference was that segments of the microcode were proven just as for the subroutine sequences, the time parameter was converted to a sum with a constant, and then the intervals were assembled much as had been done for the subroutines in the earlier proofs. A

complex conversion from a time parameter consisting of a sum of several constants was required once only, and the proofs were successfully completed. While this is not significant in terms the logical formalism, it is typical of the problems encountered when applying formal methods to substantial systems. Much time was spent managing complexity. Creating a proof is often not in itself a problem, creating a large proof within fixed resource limits is.

The proof of termination of the microcoded “while” loops required proving that under the two DEC28 assumptions, the n^{th} application of DEC28 to the 28 bit value that represents n gives the value ZERO28, and for any fewer than n applications, the value is not ZERO28.

```

loop_terminates_lemma =
..  $\vdash !w28.$ 
     $\neg \text{NEG}(\text{iVal}(\text{Bits28 } w28)) \implies$ 
       $(\text{nth}(\text{pos\_num\_of}(\text{iVal}(\text{Bits28 } w28))) (\text{atom\_bits } \circ \text{DEC28}) w28 = \text{ZERO28})$ 

nth_DEC_NOT_ZERO =
..  $\vdash !n \ w28.$ 
     $\neg \text{NEG}(\text{iVal}(\text{Bits28 } w28)) \implies$ 
       $n < (\text{pos\_num\_of}(\text{iVal}(\text{Bits28 } w28))) \implies$ 
       $\neg (\text{nth } n (\text{atom\_bits } \circ \text{DEC28}) w28 = \text{ZERO28})$ 

```

A pair of theorems summarizing the accumulated effect of one pass through the while loop, with the assumption that the termination condition is false at the start, were proved. Next, using the `nth_DEC_NOT_ZERO` theorem and the one pass theorems, the accumulated computation effect of up to n iterations through the loop was captured in the pair of theorems in Figure 6.7. The proof proceeded by induction on n .

Finally, `loop_terminates_lemma` was used to derive that the exit condition held after n iterations. The theorems for each sequence, consisting of the two loop sequences, and the sequences between, and before and after the loops, were assembled and simplified to give the final theorems shown in Figures 6.8 and 6.9. Aside from the

```

loop1_nth_state =
.... |- !n t_m.
  ~NEG(iVal(Bits28(atom_bits(arg(PRE t_m))))) ==>
  n <= (pos_num_of(iVal(Bits28(atom_bits(arg(PRE t_m))))) ==>
  (mpc t_m = #000111000) ==>
  (mpc(t_m + (5 * n)) = #000111000) ^
  (s0(t_m + (5 * n)) = s0 t_m) ^
  (s1(t_m + (5 * n)) = s1 t_m) ^
  (s2(t_m + (5 * n)) = s2 t_m) ^
  (s3(t_m + (5 * n)) = s3 t_m) ^
  (memory(t_m + (5 * n)) = memory t_m) ^
  (arg(PRE(t_m + (5 * n))) = nth n(DEC28 o atom_bits)(arg(PRE t_m))) ^
  (buf2(PRE(t_m + (5 * n))) = buf2(PRE t_m)) ^
  (s'(PRE(t_m + (5 * n))) = s'(PRE t_m)) ^
  (e'(PRE(t_m + (5 * n))) = e'(PRE t_m)) ^
  (c'(PRE(t_m + (5 * n))) = c'(PRE t_m)) ^
  (d'(PRE(t_m + (5 * n))) = d'(PRE t_m)) ^
  (free(PRE(t_m + (5 * n))) = free(PRE t_m)) ^
  (x1(PRE(t_m + (5 * n))) =
    nth n(cdr_bits o (memory t_m))(x1(PRE t_m))) ^
  (x2(PRE(t_m + (5 * n))) = x2(PRE t_m)) ^
  (car(PRE(t_m + (5 * n))) = car(PRE t_m)) ^
  (parent(PRE(t_m + (5 * n))) = parent(PRE t_m)) ^
  (root(PRE(t_m + (5 * n))) = root(PRE t_m)) ^
  (y1(PRE(t_m + (5 * n))) = y1(PRE t_m)) ^
  (y2(PRE(t_m + (5 * n))) = y2(PRE t_m))

loop1_nth_nonmajor =
.... |- !n t_m.
  ~NEG(iVal(Bits28(atom_bits(arg(PRE t_m))))) ==>
  n <= (pos_num_of(iVal(Bits28(atom_bits(arg(PRE t_m))))) ==>
  (mpc t_m = #000111000) ==>
  (!t'.
    (PRE t_m) < t' ^ t' < (t_m + (5 * n)) ==> ~is_major_state mpc t')

```

Figure 6.7: Theorems for n iterations through loop1

assumptions typical of each instruction proof, there are the two DEC28 assumptions, as well as the assumptions that the parameters to the LD instruction both represent nonnegative integers.

```

LD_State =
..... ⊢ let m =
      pos_num_of
      (iVal
      (Bits28
      (atom_bits
      (memory tm
      (car_bits
      (memory tm
      (car_bits(memory tm(cdr_bits(memory tm(c tm))))))))))
in
  let n =
    pos_num_of
    (iVal
    (Bits28
    (atom_bits
    (memory tm
    (cdr_bits
    (memory tm
    (car_bits(memory tm(cdr_bits(memory tm(c tm))))))))))
in
  let mem1 =
    Store14 (free tm)
      (bus32_cons_append #00 RT_CONS
      (car_bits
      (memory tm
      (nth n
      (cdr_bits o(memory tm))
      (car_bits
      (memory tm(nth m(cdr_bits o(memory tm))(e tm))))))
      (s tm))
      (memory tm)
  in
    ((mpc(tm + (40 + (5 * (m + n)))) = #000101011) ∧
    (memory(tm + (40 + (5 * (m + n)))) = mem1) ∧
    (s(tm + (40 + (5 * (m + n)))) = free tm) ∧
    (e(tm + (40 + (5 * (m + n)))) = e tm) ∧
    (c(tm + (40 + (5 * (m + n)))) =
    cdr_bits(mem1(cdr_bits(mem1(c tm)))) ∧
    (d(tm + (40 + (5 * (m + n)))) = d tm) ∧
    (free(tm + (40 + (5 * (m + n)))) = cdr_bits(memory tm(free tm)))

```

Figure 6.8: Microprogramming level *State* theorem for LD instruction

```

LD_Next =
..... ⊢ Next
      tm
      (tm + (40 +
        (5 *
          ((pos_num_of
            (iVal
              (Bits28
                (atom_bits
                  (memory tm
                    (car_bits
                      (memory tm
                        (car_bits(memory tm(cdr_bits(memory tm(c tm)))))))))) +
            (pos_num_of
              (iVal
                (Bits28
                  (atom_bits
                    (memory tm
                      (cdr_bits
                        (memory tm
                          (car_bits(memory tm(cdr_bits(memory tm(c tm))))))))))
              (is_major_state mpc)

```

Figure 6.9: Microprogramming level *Next* theorem for LD instruction

6.6 Liveness

If the SECD system initially reaches a *major state*, and if every time it starts in a *major state* all possible paths eventually return it to a *major state*, then the temporal abstraction function from the coarse granularity of time to the medium granularity is total. This important property is essential for the last stage of the proof.

The function `state_abs` is well-defined only when one of four values is in the MPC9 register, and these are precisely the four values for which the temporal abstraction predicate `is_major_state mpc` is true. Unfortunately, one cannot prove that the abstraction predicate holds at the abstracted time, i.e. `(is_major_state`

`mpc`) when $(\text{is_major_state } \text{mpc}))t_c$, or in general $(P \text{ when } P) \ t$ (which is definitionally equivalent to $P(\text{TimeOf } P \ t)$). This limitation arises from the use of the SELECT operator \textcircled{Q} in defining the temporal abstraction function TimeOf^4 , given in Figure 5.9. This obstacle was overcome by proving a “liveness” property for the predicate used for the abstraction $(\text{is_major_state } \text{mpc})$. Liveness as defined by Inf states that the predicate is true infinitely often.

$$\vdash \text{Inf } f = !t. ?t'. (t < t') \wedge (f \ t')$$

The proof of this property is simplified by using the following theorems.

```

Inf_thm =
  ⊢ !f. (?t'. 0 < t' ∧ f t') ∧
    (!t. f t ==> (?t'. t < t' ∧ f t')) ==> Inf f

Next_exists_thm =
  ⊢ !t t1 f. Next t t1 f ==> (?t'. t < t' ∧ f t')

```

The first limits the proof requirement to showing that a *major state* is reached initially, and every time the machine starts in a *major state* it will reach a *major state* again in the future. This reduces the number of starting points to four, instead of every possible machine state (consisting of 400+ values that `mpc` can have). The microprogramming level theorems defining the `Next` times the temporal abstraction function holds are used for each branch of the proof. The second theorem expresses that part of the definition of `Next` that satisfies the proof requirement for times starting in *major states*.

⁴From the defining axiom for the SELECT operator $\text{SELECT_AX: } \vdash \forall(P: * \rightarrow \text{bool}) \ (x: *). \ P \ x \ ==> \ P(\textcircled{Q} \ P)$, it is necessary to show that P holds at some value x in order to derive that $P(\textcircled{Q} \ P)$ holds. See [Gra90a] for further description.

```

liveness =
[ clock_constraint SYS_Clocked
; ^SYS_imp
; valid_program_constraint memory mpc button_pin s e c d
; reserved_words_constraint mpc memory
; well_formed_free_list memory mpc free s e c d
; DEC28_assum1
; DEC28_assum2
]
⊢ Inf(is_major_state mpc)

```

The proof is achieved by a series of case splits: first using `Inf_thm` it is split into time 0_c and times t_c such that `is_major_state mpc t_c` . The latter is split into the four *major states* using the definition of `is_major_state`, and finally `valid_program_constraint` divides the possible transitions from `top_of_cycle` state into 18 cases, each of which is solved by the appropriate `*_Next` theorem from the previous stage. This was the simplest stage of the proof, requiring only about 1,000 primitive inferences.

6.7 Relating the Computations over Abstraction

The microprogramming stage provided a set of theorems that defined the lower level view of the effect of computation of each instruction, giving new values for each of the `s`, `e`, `c`, `d`, and `free` state variables and the low level memory in terms of low level operations on the memory and register contents.

The *specification* for each transition defines the new values for the same state variables and the *abstract* memory in terms of abstract memory operations on the previous state values and abstract memory. To complete the verification, it must be proved that each of these low level computations correspond to the transition on the abstracted state specified by the top level specification. A few of the theorems relating abstract memory operations to RTL level operations are given in Figure 6.10.

```

car_cdr_mem_abs_lemma =
  ⊢ is_cons(memory v) ==>
    !x. (M_Car(v,mem_abs memory, x) = car_bits (memory v)) ∧
        (M_Cdr(v,mem_abs memory, x) = cdr_bits (memory v))

number_mem_abs_lemma =
  ⊢ is_number(memory v) ==>
    !x. M_int_of(v,mem_abs memory, x) =
        iVal(Bits28(atom_bits (memory v)))

opcode_mem_abs_lemma =
  ⊢ is_cons(memory v) ==>
    is_number(memory(car_bits(memory v))) ==>
    !x. M_int_of(M_CAR(v,mem_abs memory,x)) =
        iVal(Bits28(atom_bits(memory(car_bits(memory v)))))

cons_unfold_1_lemma =
  ⊢ is_cons(memory free) ==>
    !v w. M_Cons(v,w,mem_abs memory,free) =
        (free,
         mem_abs(Store14 free(bus32_cons_append #00 RT_CONS v w)memory),
         cdr_bits(memory free))

cons_unfold_2_lemma =
  ⊢ ¬ (free = cdr_bits(memory free)) ==>
    is_cons(memory(cdr_bits(memory free))) ==>
    !v w z.
      M_Cons(v,w,mem_abs(Store14 free z memory),cdr_bits(memory free)) =
        (cdr_bits(memory free),
         mem_abs(Store14 (cdr_bits(memory free))
                       (bus32_cons_append #00 RT_CONS v w)
                       (Store14 free z memory)),
         cdr_bits(Store14 free z memory(cdr_bits(memory free))))

```

Figure 6.10: Abstract Memory Theorems

The antecedents in each theorem express conditions under which the low level operations correspond to well-defined operations on abstract memory type objects. An initial obstacle to the proof is that the function from the representing type for abstract memories to the abstract memory type is only well-defined for appropriate

representing memories. It was necessary to show that the function `Mem_Range_Abs` applied to a low level memory would return an object in the representing type for abstract memories. With this result, the composition of `REP_mfsexp_mem` with `ABS_mfsext_mem` was the identity when applied to `Mem_Range_Abs o memory`. This result is expressed in the the following theorems:

```
Mem_Range_Abs_lemma =
  ⊢ !memory. IS_mfsexp_mem(Mem_Range_Abs o memory)

REP_ABS_Mem_Range_Abs =
  ⊢ REP_mfsexp_mem(ABS_mfsext_mem(Mem_Range_Abs o memory)) =
    Mem_Range_Abs o memory
```

The proofs of the theorems in Figure 6.10 are achieved by expanding definitions of the operations and the `mem_abs` function, applying the above theorems to eliminate the `REP` and `ABS` functions, expanding the definition of `Mem_Range_Abs`, and using the distinctness of record types. The theorem `opcode_mem_abs_lemma` combines the two previous theorem results for the machine instruction code value evaluation. The last theorem, `cons_unfold_2_lemma`, is used when two `M_Cons` operations are performed on a memory.

The top goal for the final proof (Figure 6.1) splits into two parts: the state when the machine first reaches a *major state*, and the state of the machine when it is next in a *major state*, in terms of its state in the previous *major state* (i.e. at time t_c). The first goal is quite simply solved by using `phase_lemma_0` to show that at time $SUC\ 0_m$ a *major state* is reached, and that at all times before that (namely time $t_m = 0$), the system is not in a *major state*. The required theorem has the form:

```
⊢ clock_constraint SYS_Clocked ==>
  ^SYS_imp ==>
    (((state_abs o mpc) when (is_major_state mpc))0_c = idle)
```

The second goal may be split into a set of subgoals, corresponding to each transition defined in the `SYS_spec`. The transitions are determined by the state at time

t_c , which is a function of the mpc at $\text{TimeOf}(\text{is_major_state mpc})t_c$. Using the liveness theorem from the previous stage, with the theorem TimeOf_TRUE :

$\vdash !f. \text{Inf } f \implies (!n. f (\text{TimeOf } f \ n)),$

to obtain the result that $\text{is_major_state mpc}$ is true at all points of the coarser granularity of time. With this result, the state abstraction function is well defined, and mpc has one of four values at all points in the coarser granularity of time.

Theorems for the correctness of each top level transition are typified by the LDF transition theorem in Figure 6.11. The last 4 assumptions in the theorem match the applicable portion of $\text{valid_program_constraint}$ for the LDF instruction branch. The major steps in proving this theorem follow.

- Using the fifth assumption, derive the assumption that the system is in *top_of_cycle* state at time t .
- derive the value of the lowest 9 bits of the 28 bit instruction code from the bottom assumption, for resolving with the less specific assumption of LDF_state and LDF_Next ,
- Expand the definition of when , and perform a β -reduction.
- Add the assumption obtained by resolving LDF_Next with the other assumptions.
- Resolve the previous assumption with TimeOf_Next_lemma and rewrite with the resulting theorem. This substitutes $\text{TimeOf}(\text{is_major_state mpc})t_c+26$ for $\text{TimeOf}(\text{is_major_state mpc})(\text{SUC } t_c)$.
- Unfold the definition of LDF_trans .
- Using the abstract memory theorems of Figure 6.10 resolved with the given assumptions rewrite the right side of the goal, producing expressions for the next state in the terminology of the lower level definition.

[illegible]

Figure 6.11: The Correctness result for the LDF instruction

- Rewrite with the triplet selector function definitions (`cell_of`, `mem_of`, `free_of`, `mem_free_of`, etc.) used in the definition of `LDF_trans`.
- Unfold the left side of the goal equation by rewriting with the `LDF_state` theorem.

- Split into separate subgoals for each of the `s`, `e`, `c`, `d`, `free`, `memory`, and `state` components. The remaining subgoals are trivially solved for all but the `c` values, which require use of a derived theorem to establish that the cells accessible from the `c` pointer have not been altered by the writing of two records to the memory. This is necessary since the new value for `c` is computed after the records are written.

Splitting the top goal (Figure 6.1) on the `state`, and for `top_of_cycle` state, on the instruction codes permitted by the `valid_program_constraint`, reduces the problem to cases solved by similar theorems for each top level transition.

As this stage was not completed at the time of this writing, statistics for the entire stage are lacking. However, we can extrapolate from the LDF example, which required approximately 20,000 primitive inferences. This sample is of average complexity, so the entire proof should be achievable with just under a half a million primitive inferences.

6.8 Summary

This chapter has presented the proof of the correctness theorem relating the RTL and top level specifications of the SECD system. Constraints limiting the scope restricted the chip to normal mode of operation and a properly configured memory. This latter component involved the values in reserved memory locations, the form of the free list, and the permissible machine codes and system state associated with each. The latter constraint effectively defines the pattern matching inherent in the informal state transitions defining the abstract machine in Table 3.3. The constraints on the free list concern both its structure, and its separation from data structures in the memory. By requiring a minimum number of records in the free list, consideration of garbage collector correctness is deferred. Lastly, two assumptions about

the decrement operation will eventually be discharged upon completion of the lower level proof.

The design of the constraints evolved as the proof proceeded. What in retrospect appears as an obvious collection of conditions was the result of incremental extensions, reflecting a strong desire to minimize the constraints to those necessary to achieve the proof. The final versions capture complicated conditions quite elegantly, and express some of the properties central to a specification for the garbage collector.

The proof presented in this chapter compares in size with the largest of previous efforts. The problem size relates largely to the complexity of individual instruction semantics, as well as the system size.

The staged approach to the proof owes much to previous efforts in microprocessor verification, particularly those of Gordon [Gor83b], Cohn [Coh88, Coh89b], and Joyce [Joy88, Joy89a], as well as the work of Melham [Mel88] and Dhingra [Dhi88] on temporal abstraction. The methodology at each stage differs mainly from prior work in the sheer size of proof effort required, demanding much less user intervention at each level, and the generality of the approach.

Problem size dominated the proof strategy at each stage. The first stage of proof had enormous terms, which were many times larger than could be displayed on a workstation screen. Term size also limited the use of powerful tactics such as rewriting, since intermediate steps in the tactic could exhaust available memory, and cause a failure. Limiting the tools to the most primitive rules and tactics made otherwise trivial proofs both tedious and time consuming.

The phase proof stage had to cope with both large size terms and large numbers of theorems. The high overhead required to prove inequality of $word_n$ constants led to exhaustive case theorems for the possible values, which were then used repeatedly as required by other proofs. User intervention was minimized by use of a proof function, although many hours of work went into the latter's design. Any alteration

to the proof function required regenerating all theorems, impeding progress for a day and a half. The theorems had to be divided among several HOL theories, as once again available memory would be exhausted. This had the advantage of more efficient theorem retrieval at later stages.

The microprogramming stage offered some of the more interesting proof challenges. The number of theorems generated was still large enough to demand minimizing user intervention, but the stronger motivation was the length of time required for each proof, much of which was spent retrieving phase theorems. Timing information about the execution time for each sequence is part of the proof result, rather than input required of the user.

Term size became a factor as the number of records written to the memory during a single machine instruction transition rose. The expression size grew exponentially in this number, so that some sequences could not be proved with the original proof function. While the final theorem could be simplified by the use of let expressions, the intermediate terms generated during the proof were so large and complex as to be completely incomprehensible, and only the machine based proof assistant could reliably cope.

The proof of termination of the microcoded loops has not appeared in previous microprocessor proofs in the form shown here⁵, although such proofs have been part of the domain of software verification ([Gor88b]).

The liveness stage of proof was made trivial by the design of the constraints and the careful design of the theorems deriving from the microprogramming stage. Only the simple initial transition caused some difficulty, forcing the development of techniques for dealing with the \mathcal{Q} operator used in the definition of the temporal

⁵Joyce [Joy89a] uses a form of temporal logic in a microprocessor proof with an asynchronous memory interface.

abstraction function. This work resulted in a technical report [Gra90a] which has had wide distribution within the community of HOL users worldwide.

The current incompleteness of the final proof stage leaves some uncertainty about the completeness of the proof development so far. However, the completion of the sample proof for the LDF transition, and the analysis of the remaining problems, gives a high degree of confidence that there are no serious obstacles remaining.

Development of the proof often revealed failings or inadequacies in the definition of components, datatypes, and functions, and even of the top level specification. In many cases, most particularly for the constraints, the increased understanding of the problem distilled the definitions to their essence. However, any such changes had to propagate through the theory hierarchy, which took several days to accomplish at the late stages of the proof.

The use of different datatypes to define state transitions at each level contributes a useful case study in the use of data abstraction. While the benefits in this particular abstract memory data type may be argued, the concept of higher level data structures with a limited number of primitive operations, representing much more complex lower level manipulations, can clearly be useful in presenting information more clearly and concisely. The added proof effort is minimal, since proof of the correspondence between operations must be done only once for each operation, and these results may be used repeatedly.

Chapter 7

Conclusions

7.1 What has been accomplished

Work described in this thesis has covered the whole spectrum from abstract architecture through VLSI design and layout, to formal specification and formal proof of correctness. The thesis has

- described the abstract SECD architecture and shown how it supports execution of a high level functional language,
- described the evolution of a realized system specification and hardware implementation,
- presented the formal definitions of the specification and an implementation view of the realized system,
- formalized the constraints under which the system is designed to function, and
- shown the method of achieving a machine proof of correctness that the lower level correctly implements the top level specification.

The end result of the work described will be the production of a *partially* verified hardware implementation of a functional architecture. Although many gaps remain between the hardware device and the formal description, some advances in linking formal methods with hardware design have been achieved. The path taken, beginning with the physical design followed by formal representation and partial verification, was a direct result of availability of talents and the long and sometimes painful process of becoming adept at the use of the HOL system. The intimate knowledge of

design issues was necessary in the formalization of representations, but the physical layout did not benefit from the use of formal methods in the design evolution. In several aspects of the design, this lack was apparent at later stages, particularly the odd *write-through* of the ARG register when fetching a machine instruction, and also the storing of a pointer to the computation result in executing the STOP instruction. The former most likely arose from incremental modifications, where originally the instruction was fetched but not stored in a datapath register. The implications of such incremental changes are not always obvious, whereas creating the formal specification drew attention to this feature.

In all fairness, the points made above are minor compared to the other problems discovered in the physical layout: the wiring errors in the *shift registers*, and the wiring error in an *XOR* gate. Rather than arguing against the use of formal methods, they show up the limitations of their use. In representing a physical device, we are proposing that a formal description captures the abstract essence of the device. We need to ensure that the formal description accurately represents the circuit, which in both cases it did not. The use of tools to automatically generate layout, such as described in [SBHS89] that transforms HOL specifications into *LSI Logic* gate array net-lists, or tools to check consistency of layout with a formal representation, could have avoided the problems we encountered. Indeed, even the *Electric* layout tool has a *Mossim* circuit extractor which could have been used to compare with the *Mossim* simulation model. An argument for verification of such tools themselves parallels the argument for the use of formal methods in specifying the design.

Aside from the scope of the work including actual chip design, some innovations have added to previous work on microprocessor verification. The use of an abstract data type and operations thereon to capture the top level behaviour and the resulting proof of the correspondence of the abstract and more primitive operations has provided a more understandable specification at the top level. It is essential that

this level be understandable if we are to realistically relate it to the *intention* of the designer.

The top level specification also explicitly includes an initial state requirement, which necessitates lower level constraints on clocking and the *reset* input. Such explicit treatment of constraints is a step forward in providing meaningful information to designers using a product. In hand with the initial state specification, the next state transition specification has been defined more generally, with the explicit time parameter generalized internally in the definition, unlike the single step specification used in [Coh88, Coh89b]. This produces a clearer correctness statement than the examples cited, particularly in that the temporal relation between the granularity of time in the two levels is expressed entirely in terms of the *when* function applied to the predicate identifying points of time in the finer grain that correspond to points at the coarser grain. This avoids the need for an explicit function to give the time between synchronization points relating the two time granularities. These values in fact were never supplied, but were generated by the proof process, which is sensible when the system complexity is considered.

The proof of the effect of computation of the microcoded *while* loops presents techniques which may be useful in similar control sequences. The approach shows how the principles of proof of while loop constructs in Floyd-Hoare logic are adaptable to hardware, when the explicit time of completion of the loop execution is also a factor in the proof.

One other difference has been the use of a well-defined multi-bit word data type. This data type has advantages in representing constants of specific types and simplicity of defining typical operations such as subfield extraction. Some examples of low level recursive component definitions based upon this type required explicit constraints of equality of word widths of parameters to the *:(bool)bus* level definition, and some involved tactics to provide the same data representation format for each

parameter through the inductive proof of correctness. This low level definition is easily overlayed with a $word_n$ instantiation, where the fact that the parameters are of the same type ensures the word width constraint is fulfilled, and thus the correctness statement simply states equality of the implementation and specification.

7.2 Putting the proof result into context

The completion of the single level of correctness proof of such a complex device is a substantial accomplishment. Comparison with the proof of the Viper microprocessor [Coh89b], the most substantial previous example, indicates the proofs are of the same magnitude, using the number of primitive inferences as a measure. The Viper is a fairly simple 32-bit microprocessor designed for safety critical applications, and is commercially available. It is hard-wired rather than microcoded, and consists of approximately 5000 gates.

proof stage	primitive inferences	
	Viper	SECD
definition	95,000	199,000
expanding definitions	5,130,000	506,000
phase proofs		4,600,000
microprogramming proofs		1,144,000
liveness proof		1,000
total	6,253,000	6,440,000

Table 7.1: Comparison of Viper and SECD proofs

The proof organizations for the two chips differed, and the results in Table 7.1¹ assemble Viper results into roughly corresponding stages of the SECD proof. The results for both omit the final stage of the proof. The Viper proof included proofs of correctness of the implementation of arithmetic operations, which the SECD proof did not.

The HOL system has proved a reliable platform for such a project, although forcing very careful management of proofs at times. One observation has arisen repeatedly through the work: the difficult problem is not the proof of specific theorems, but rather the creation of useful and correct specifications and constraints, and the formulation of a strategy to achieve a result which entails a series of theorems. This highlights the need for specification tools such as Camilleri's tools for executable specification [Cam88]. Becoming sufficiently competent in the use of the HOL system to undertake significant proofs takes considerable time and dedicated effort, but in many cases the management of the proof complexity was far more challenging than the management of the thread of the proof, which was often quite mechanical. Where difficulties arose in completing a proof, it was more often a flaw in the definitions or the constraints, and thus the proof process fed back to the specification and constraints constantly.

The scope of this project has been extended by other works. An analysis and informal proof of correctness of the compilation algorithm by Simpson [SGB89] is a significant step towards the verified system ideal: verified software, compiled by a verified compiler, being executed on a verified hardware system. One hopes some day that such a system may be used to implement a proof environment as well. More important perhaps than the continuity of trustworthiness at each level gained by verification would be the formal specification interfacing each, so that the abstract

¹Statistics for Viper were taken from [Coh89b]

domain of algorithms can be related to the finite world of hardware in an explicit manner.

Several omissions limit the accomplishment. No attempt was made to verify that garbage collection was correctly implemented. While some provision was included in the model for this later extension, the meaning of what the garbage collector does has not been formalized. A suitable datatype to represent this function may be found in set theory, where a *state* set may consist of all cells reachable from the state registers, disjoint from the set of cells reachable from the free register. A *cons* operation on the abstract memory moves a cell from the *free* set to the *state* set. The garbage collect function causes the universe of cells to be equal to the disjoint union of the two sets, while not altering the state set. It may be possible to define such sets using the path function, but this has not been explored.

The constraints also limit the verification to normal mode of operation, omitting the test mode use of the shift registers. Once again, the low level model and clocking constraints were designed for this possible extension. In other constraints, such as the initializing of the MPC9 register by constraining the clocks and `reset` input, we are trading off completeness of the specification against utility and simplicity of the specification. We really are unconcerned with how the device may operate under circumstances when these constraints are not in force. The action of the chip when illegal instruction codes are encountered is perhaps an exception. A predictable and traceable recovery may be desirable, but this omission is at least clearly discernable from the `valid_program_constraint` in the correctness result.

The limitation of the verification to the two upper levels of description of the system has resulted from a limitation of time rather than any difficulty in the problem itself. The extension of the proof to the lower level is expected to be considerably more simple in many respects, since it will be feasible to treat components such as registers and the ALU individually. The datapath composition of these components

matches the *RTL* level view almost identically, hence the composition itself would not require much consideration. Given a library of datapath components, the *RTL* level is an appropriate level at which to stop in VLSI specification. Similarly, a verification that the low level definition of the *control unit* ROM correctly implements the *RTL* ROM should derive from a library ROM model, tailored with a particular transistor layout. The remainder of the *control unit* hierarchy differs between the two levels, and will require a more complex proof effort, but not one which raises new problems.

7.3 Retrospective Improvements

Looking back, there is much that could be improved, particularly in the design of the chip. The lack of concern with speed of operation was explicit, but some aspects of the design could be improved dramatically without increasing complexity. Among the most inefficient features is the memory fetch operation. With an external RAM, the memory interface will be the determining timing constraint, and the timing is further delayed by passing it through the *ARG* register, which delays its propagation by part of a clock phase. The addition of an extra cycle for this operation could well be made up by a higher clock rate.

The imbalance of the clock phases, with only a single logic element separating the output of the latch clocked on ϕ_A from the input of the latch clocked on ϕ_B , versus the entire rest of the chip and memory logic for the other phase, means much more computation occurs during one half of the cycle than the other half. Separating the pair of latches holding the controller state, placing the ϕ_A triggered latch after the ROM for example, could even out this imbalance considerably, and perhaps contribute to a better overall clocking scheme. The datapath registers could be clocked on ϕ_B , thus having both *datapath* and *control unit* changing on the same clock phase.

A modification driven by the formal specification would separate the functionality of the decrement operations of numbers and addresses. The added complexity from duplicating functional components is mitigated by avoiding the devices needed to pad address values with zero's.

In the formal representation of the system, the relationship of the formal model to the layout deserved far more careful attention. At the other extreme, if starting over again, I would reconsider the top level representation, and try to use a datatype that could more readily encompass the garbage collection operation. The top level specification could also be relaxed somewhat. As presented here, it specifies the sequence of operations even at times when the sequence is immaterial to the effect of the computation. It would be preferable to specify the *properties* of the new values in each of the *s*, *e*, *c*, and *d* registers, rather than the result of specific abstract memory operations. It is quite possible to define a more abstract level specification that has this property, and prove that the present top level ensures the specified behaviours.

The SECD chip project is continuing, and the verification relating the top two levels will be completed shortly. The lower level specification and verification relative to the *RTL* definition will be at least partly completed as well. A third version of the chip will be submitted for fabrication after a full check of correspondence of the extracted circuit and the formal model.

The SECD architecture may well be superceded as the choice for future projects, as it is somewhat more complex and inefficient than others developed later, such as *CAM* [Cur86] or *TIM* [FW87] machines. However, as a study of the verification of a complex design, it will be useful for those who follow. The specification and verification methodology are quite general and will provide useful guidelines.

7.4 Hardware Verification: the future

So where does this leave hardware verification as a contributor to the development of complex systems? First, we have shown that a formal specification can not only express the behaviour of a moderately complex system, but indeed can help clarify what the behaviour is or should be. Second, using formal inference methods, we can relate two levels of description of the system, and gain a high degree of assurance that the lower level model correctly implements the behaviour of the high level model. By extending this process to a low enough level, say the model extracted from the circuit layout, or by automated transformation of formal definitions to layout, we may decrease the likelihood of wiring errors in the layout. By extending the process to the compiler and software levels, we may be able to better relate algorithms and their execution on hardware.

There remains a vital point. Verification cannot in any way ensure that any hardware device is “correct”. At best we deal only with abstract models of things, rather than the actual devices themselves. Even if our model is “correct” in terms of bearing a one to one correspondence to the circuit in the device, the abstraction loses much relevant information. We are constrained immediately by how accurately our formal model captures the behaviour of the device. Errors that occur in the fabrication of the device destroy any accurate relationship in any event. At the other end of the range, we have formal models of some abstract ideas in the head of the designer. The relation between the two is not something that we can ever be sure is accurate. Confidence in the correctness of a specification can be gained by exercising it with a high level (perhaps symbolic) simulator and by subjecting it to public scrutiny (hence the requirement for readability and succinctness).

Of course, these same limitations apply equally to the use of simulation to “verify” design correctness. Simulation at the switch level and above uses the same under-

lying model as formal specification. The difference lies in *how* the model is used. Simulations run many individual tests to cover all input possibilities. Formal proofs take the same model and the same specifications as are used in the simulation model and manipulate the latter formally to prove the correctness of the design elaboration. Full coverage, that is correctness over all input values, comes automatically. Further, most VLSI designs are regular or contain regular subsystems (e.g. the n -bit adder is a row of 1-bit full adders). Regular sub-systems can be verified using *induction*. Inductive proofs split into two cases; the base case and the inductive case (show the correctness of a sub-system of size $n+1$ assuming the correctness of a sub-system of size n). So proofs of regular systems do not balloon in length with n , whereas the number of simulation runs required does. Whilst it must be admitted that carrying out a proof is much harder than writing a simulation program, we should also remember that the simulation program will be unproven.

Adopting verification techniques does not impose a new design methodology. Information already present is used in a much more formal way to guarantee the correctness of a design elaboration. Thus verification techniques should embed well into CAD systems.

The methodology, tools, and experience are not yet there, i.e. the subject is still in its infancy. There is a strong need for libraries of specifications to be established, and for large case studies to be published so work can proceed towards establishing a robust and reliable technology and automating (part of) it. Despite these current drawbacks, it remains an approach with a promising future. Formal verification should be considered as another weapon in the armoury of hardware designers, particularly useful for showing the correctness of regular systems and for conducting proofs of functionality at the sub-system level and above.

References

- [Anc86] François Anceau. *The Architecture of Microprocessors*. Addison-Wesley Publishing Company, 1986.
- [Bev87] W. R. Bevier. A Verified Operating Systems Kernel. Technical Report CLI-11, Computational Logic Inc, Austin, Texas, 1987.
- [BG90] G. Birtwistle and B. Graham. Verifying SECD in HOL. In J. Staunstrup, editor, *Formal Methods for VLSI Design*. Proceedings of the 1990 IFIP WG 10.5 Summer School to be held at Lyngby, Denmark, North Holland, 1990.
- [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. also University of Calgary, Computer Science Department, Research Report 89/354/16.
- [BGMS88] G. Birtwistle, B. Graham, T. Melham, and R. Schediwy. Hardware Verification by Formal Proof. In Vijay K. Bhargava, editor, *Canadian Conference on Electrical and Computer Engineering*, pages 379–384. Canadian Society for Electrical Engineering, 1988.
- [BGS⁺89] G. Birtwistle, B. Graham, T. Simpson, K. Slind, M. Williams, and S. Williams. Verifying an SECD Chip in HOL. In L. J. M. Claesen, editor, *Proceedings of the IFIP TC10/WG10.5 Workshop on Applied Formal Methods for Correct VLSI Design, Leuven, November 13–16, 1989*, pages 149–158, Amsterdam, 1989. North Holland.
- [BJL⁺86] G. Birtwistle, J. Joyce, B. Liblong, T. Melham, and R. Schediwy. Specification and VLSI design. In G. J. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 83–97, Amsterdam, 1986. North Holland.
- [BM79] R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, New York, 1979.
- [Bur75] W. Burge. *Recursive Programming Techniques*. Addison-Wesley, New York, 1975.
- [Cam88] A.J. Camilleri. *Executing Behavioural Definitions in Higher Order Logic*. PhD thesis, University of Cambridge Computer Laboratory, 1988.

- [Cam89a] Cambridge Research Center, SRI International, Cambridge, England. *The HOL System: Description*, 1989.
- [Cam89b] Cambridge Research Center, SRI International, Cambridge, England. *The HOL System: Reference Manual*, 1989.
- [Cam89c] Cambridge Research Center, SRI International, Cambridge, England. *The HOL System: Tutorial*, 1989.
- [Coh88] A. J. Cohn. A Proof of Correctness of the VIPER Microprocessor: The First Level. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 27–71, Norwell, Massachusetts, 1988. Kluwer. Also University of Cambridge, Computer Laboratory, Tech. Report No. 104.
- [Coh89a] A. J. Cohn. The Notion of Proof in Hardware Verification. *Journal of Automated Reasoning*, 5:127–13, 1989.
- [Coh89b] A. J. Cohn. A Proof of Correctness of the VIPER Microprocessors: The Second Level. In G. Birtwistle and P. A. Subrahmanyam, editors, *Trends in Hardware Verification and Automated Theorem Proving*, pages 1–91, New York, 1989. Springer Verlag.
- [Cra89] Dan Craigen. Position Paper for FM89. Submitted to FM89, Conference on the Use of Formal Methods in Systems Design, Halifax, July 1989.
- [Cul88] W. J. Cullyer. Implementing Safety Critical Systems: The VIPER Microprocessor. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 1–26, Norwell, Massachusetts, 1988. Kluwer.
- [Cur86] P-L. Curien. *Categorical Combinators, Sequential Algorithms, and Functional Programming*. Pitman, London, 1986.
- [Dhi88] I. S. Dhingra. *Formal Validation of an Integrated Circuit Design Style*. PhD thesis, University of Cambridge Computer Laboratory, 1988.
- [FH88] A. J. Field and P. G. Harrison. *Functional programming*. Addison-Wesley, New York, 1988.
- [FW87] J. Fairbairn and S. Wray. TIM: A simple, lazy abstract machine to execute supercombinators. In G. Kahn, editor, *Functional Languages and Computer Architecture*, pages 34–45. Springer Verlag, 1987.
- [Gab85] P. Gabriel. *Performance and Evaluation of LISP Systems*. MIT Press, Boston, 1985.

- [GB89] B. Graham and G. Birtwistle. Formalising the Design of an SECD chip. In M. Leeser and G. Brown, editors, *Proceedings of the Cornell Workshop on Hardware Specification, Verification, and Synthesis: Mathematical Aspects*, pages 40–66, New York, 1989. Springer-Verlag.
- [Gor79] M. J. C. Gordon. *The Denotational Description of Programming Languages*. Springer Verlag, London, 1979.
- [Gor83a] M. J. C. Gordon. LCF-LSM: A System for Specifying and Verifying Hardware. Technical Report 41, Computing Laboratory, University of Cambridge, 1983.
- [Gor83b] M. J. C. Gordon. Proving a Computer Correct using the LCF-LSM Hardware Description Language. Technical Report 42, Computing Laboratory, University of Cambridge, September 1983.
- [Gor85] M. J. C. Gordon. HOL: A machine oriented formulation of higher order logic. Technical report 68, Computing Laboratory, University of Cambridge, 1985.
- [Gor86] M. J. C. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. Milne and P.A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 153–177, Amsterdam, 1986. North-Holland.
- [Gor88a] M. J. C. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128, Norwell, Massachusetts, 1988. Kluwer.
- [Gor88b] M. J. C. Gordon. *Programming language theory and its implementation*. Prentice Hall, London, 1988.
- [Gor89] M. J. C. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Trends in Hardware Verification and Automated Theorem Proving*, pages 387–439, New York, 1989. Springer Verlag.
- [Gra89a] B. Graham. Formal Specification of the SECD Chip, Top and Register Transfer Levels. Research Report 89/370/32, Computer Science Department, University of Calgary, 1989.
- [Gra89b] B. Graham. SECD: Design Issues. Research Report 89/369/31, Computer Science Department, University of Calgary, 1989.
- [Gra90a] B. Graham. Dealing with the Choice Operator in HOL88. Research Report 90/382/06, Computer Science Department, University of Calgary, 1990.

- [Gra90b] B. Graham. Formal Proof of the SECD Chip, Top and Register Transfer Levels. Research Report, Computer Science Department, University of Calgary, 1990. in preparation.
- [GWB⁺89] B. Graham, S. Williams, G. Birtwistle, J. Joyce, and B. Liblong. The Mossim Specification of the SECD DESIGN. Research Report 89/341/03, Computer Science Department, University of Calgary, 1989.
- [GWS89] B. Graham, S. Williams, and G. Stone. Operating Specification for the SECD Chip. Research Report 89/353/15, Computer Science Department, University of Calgary, 1989.
- [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, 1989.
- [Hen80] P. Henderson. *Functional programming; applications and implementation*. Prentice Hall, London, 1980.
- [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.
- [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.
- [Hun85] W. A. Hunt. FM8501: a Verified Microprocessor. Technical Report 47, Computer Science Department, University of Austin at Texas, Austin, Texas, 1985.
- [Joy88] J. Joyce. Formal verification and implementation of a microprocessor. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 129–157, Norwell, Massachusetts, 1988. Kluwer.
- [Joy89a] J. Joyce. Case Study: Microprocessor Systems, 1989. In *The HOL System: Tutorial*, pages 115–232.
- [Joy89b] J. Joyce. Multi-Level Verification of a Simple Microprocessor. Progress Report, December 1989.
- [Joy89c] J. Joyce. A Verified Compiler for a Verified Microprocessor. Technical Report 167, University of Cambridge Computer Laboratory, 1989.
- [Lan64] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

- [Mas86] Ian A. Mason. *The Semantics of Destructive LISP*. Center for the Study of Language and Information, 1986.
- [Mas88] Ian A. Mason. Verification of Programs that Destructively Manipulate Data. *Science of Computer Programming*, 10:177–210, 1988.
- [MC80] Carver Mead and Lynn Conway. *Introduction to VLSI Systems*. Addison-Wesley Publishing Company, 1980.
- [Mel88] T. F. Melham. Abstraction mechanisms for hardware verification. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 267–291, Norwell, Massachusetts, 1988. Kluwer.
- [Moo88] J. Strother Moore. PITON: A Verified Assembly Level Language. Technical Report CLI-22, Computational Logic Inc, Austin, Texas, 1988.
- [Neu89] Peter G. Neumann. Illustrative risks to the public in the use of computer systems and related technology. Position paper submitted to FM89, Conference on the Use of Formal Methods in Systems Design, Halifax, July 1989.
- [NS89] P. Narendran and J. Stillman. Formal Verification of the Sobel Image Processing Chip. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 92–127, New York, 1989. Springer Verlag.
- [Plo75] G. D. Plotkin. Call-by-name, call-by-value, and the lambda calculus. *Theoretical Computer Science*, 1(1):125–159, 1975.
- [Rub87] S.M. Rubin. *Computer Aids to VLSI Design*. Addison-Wesley, Reading, MA, 1987.
- [SBGH89] T. Simpson, G. Birtwistle, B. Graham, and M. J. Hermann. A Compiler for Lispkit Targetted at Henderson's SECD machine. Research Report 89/339/01, Computer Science Department, University of Calgary, 1989.
- [SBGS89] S. Stodart, G. Birtwistle, B. Graham, and K. Slind. Chrestomathy of HOL Specifications and Proofs. Technical Report, Computer Science Department, University of Calgary, 1989. Prepared under Contract No. W2213-8-6362/01-SS with the Department of National Defence, 146 pages.
- [SBHS89] K. Slind, G. Birtwistle, M. Hermann, and T. Simpson. From Specification to Layout: Transforming HOL Specifications into Gate Array Net-Lists. In *Proceedings of Canadian Conference on Electrical and Computer Engineering*, 1989.

- [SGB89] T. Simpson, B. Graham, and G. Birtwistle. From LispKit to SECD Chip: Some Steps on the way to a Verified System. In *Proceedings of the Third Banff Verification Workshop*, 1989. Submitted for publication.
- [SS89] R. C. Sekar and M. K. Srivas. Formal Verification of a Microprocessor Using Equational Techniques. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 171–217, New York, 1989. Springer Verlag.
- [Wil89] M. Williams. SECD Controller Board Implementation. Technical Report #89/359/21, Computer Science Department, University of Calgary, Calgary, 1989.
- [You88] W. D. Young. A Mechanically Verified Code Generator. Technical Report CLI-37, Computational Logic Inc, Austin, Texas, 1988.