*Technical Report*

UCAM-CL-TR-195
ISSN 1476-2986

Number 195

UNIVERSITY OF CAMBRIDGE

**Computer Laboratory**

# Multi-level verification of microprocessor-based systems

Jeffrey J. Joyce

May 1990

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

*https://www.cl.cam.ac.uk/*

This technical report is based on a dissertation submitted December 1989 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Pembroke College.

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

*https://www.cl.cam.ac.uk/techreports/*

ISSN 1476-2986

# Abstract

The idea of using formal logic to reason about small fragments or single layers of a software/hardware system is well-established in computer science and computer engineering. Recently, formal logic has been used to establish correctness properties for several realistic systems including a commercially-available microprocessor designed by the British Ministry of Defence for life-critical applications. A challenging area of new research is to verify a complete system by linking correctness results for multiple layers of software and hardware into a chain of logical dependencies.

This dissertation focuses specifically on the use of formal proof and mechanical proof-generation techniques to verify microprocessor-based systems. We have designed and verified a complete system consisting of a simple compiler for a hierarchically structured programming language and a simple microprocessor which executes code generated by this compiler. The main emphasis of our discussion is on the formal verification of the microprocessor. The formal verification of the compiler is described in a separate paper included as an appendix to this dissertation.

Combining correctness results for the compiler with correctness results for the microprocessor yields a precise and rigorously established link between the formal semantics of the programming language and the execution of compiled code by a model of the hardware. The formal proof also links the hardware model to the behavioural specification of an asynchronous memory interface based on a four-phase handshaking protocol.

The main ideas of this research are (1) the use of *generic specification* to filter out non-essential detail, (2) embedding *natural notations* from special-purpose formalisms such as temporal logic and denotational description, and (3) the use of *higher-order logic* as a single unifying framework for reasoning about complete systems.

Generic specification, in addition to supporting fundamental principles of modularity, abstraction and reliable re-usability, provides a mechanism for enforcing a sharp distinction between what has and what has not been formally considered in a proof of correctness. Furthermore, it is possible to create generic specifications in a pure formalism with the expressive power of higher-order logic without inventing new constructs.

Natural notations from special-purpose formalisms offer the advantage of concise and meaningful specifications when applied to particular areas of formal description. Semantic gaps between different notations are avoided by embedding them in a single logic. Special-purpose rules based on these notations can be derived as theorems with the aim of implementing more efficient proof strategies.

Finally, it is argued that the primary purpose of using mechanical proof generation techniques to reason about software and hardware is to support the intelligent participation of a human verifier in the rigorous analysis of a design at a level which supports clear thinking.

---

# Contents

Chapter 1

# Introduction

Life-critical systems are increasingly dominated by microprocessor-based electronics. To increase confidence in the design of these systems, we describe methods based on formal proof and mechanical proof-generation which can be used to link multiple levels of software and hardware description into a chain of logical dependencies.

To demonstrate the use of these methods, we have designed and verified a complete system consisting of a compiler for a hierarchically structured programming language and a microprocessor which executes code generated by this compiler. The formal proof yields a precise and rigorously established link between the formal semantics of the programming language and the execution of compiled code by a model of the hardware. The formal proof also links the hardware model to the behavioural specification of an asynchronous interface to external devices, e.g., off-chip memory, a sensor or actuator in a real-time control system.

Higher-order logic provides a single unifying framework for reasoning about diverse aspects of software and hardware description. The expressiveness of this formalism supports several powerful techniques including the use of *generic specification* and the ability to embed *natural notations* from other formalisms such as temporal logic and denotational description.

## 1.1 Motivation

Methods for reliable design encompass a large part of computer engineering, and indeed, systems engineering. The purpose of these methods is to ensure that designs meet very high standards of reliability. For instance, the FAA (*Federal Aviation Administration*) requires the probability of catastrophic failure to be less than $10^{-9}$ per 10-hour flight for a life-critical civil aviation flight control system. The reliability of these systems is vital in highly integrated systems such as the fly-by-wire flight control system of the Airbus 320 in which almost all of the mechanical and hydraulic controls in the flight deck have been replaced by microprocessor-based electronics.

There are two main approaches for achieving reliability [125]. One approach is *fault tolerance* which is concerned with building mechanisms into a design to cope with faults when they occur, e.g., component failure. The other approach is *design error exclusion* which seeks to exclude design errors to the maximum extent possible.

Conventional methods for both fault tolerance and design error exclusion have serious limitations. For instance, recent studies [92] have shown that diverse redundancy may be less effective as a measure for fault tolerance than previously thought. Design error exclusion techniques based on manual verification and validation can expose errors but non-exhaustive methods cannot possibly guarantee that a non-trivial design is free from errors.

A very different approach to design error exclusion is the use of formal proof to increase confidence in the reliability of a design. In particular, these methods can be used to show that a design is free from errors to the extent that formal descriptions of the design and its requirements are related by a formal proof.

## 1.2 Problem Definition

In this dissertation, we consider the use of formal proof as a design error exclusion technique in the design of microprocessor-based systems. These systems typically consist of embedded software running on one or more dedicated microprocessors. They are conceptually organized into multi-layered 'stacks' of software and hardware levels. A reliable design must ensure that the internal design of each layer satisfies its external design and that the external design of each layer fits properly into the stack.

We address the specific problem of building a *verified stack*[1] in which layers of this stack are linked together by formal proof to form a chain of logical dependencies from software levels down to a formal description of the hardware in terms of elementary components.

The feasibility of building verified stacks for realistic applications has already been demonstrated by a team of researchers at Computational Logic, Inc., [6,7]. Our efforts are based on a much simpler example of a verified stack, but we have dealt with many of the same basic problems. By working on a simpler example and using a more expressive formalism, we have had a greater degree of freedom to explore different approaches to solving some of these basic problems.

From these investigations, we are able to contribute some novel ideas on building verified systems. Our methods rely upon the expressive power of higher-order logic, but they are general enough to use in other formalisms of similar expressive power. Complementary work by Cohn [29,30,31] on verifying the commercially-available VIPER microprocessor (also using higher-order logic and many of same basic proof techniques) suggests that our methods could be scaled upwards to the complexity of a real system.

## 1.3 Main Ideas

This section outlines the main ideas which underlie our approach to verifying multi-layered stacks beginning with a brief overview of these main ideas.

### 1.3.1 Overview

*Generic specification* plays a dominant role in this dissertation. This is similar in concept to the 'generic mechanism' of the Ada[2] programming language which allows a subprogram or package to be parameterized by types and subprograms as well as values and objects [4]. In programming, this is a powerful technique for reliable re-use of software [51]. In the context of formal proof, genericity offers several more advantages in addition to re-usability. It can be used to filter out non-essential detail from

[1] The term 'verified stack' is due to researchers at Computational Logic, Inc., [6,7].

[2] Ada is a registered trademark of the U.S. Government - Ada Joint Program Office.

formal descriptions at each level in a design hierarchy. It also reduces the amount of special-purpose infrastructure needed to reason about particular application areas, e.g., hardware-oriented data types. While some formalisms have a built-in facility for generic specification as a primitive construct [52,142], we show how generic specifications can be created in higher-order logic with existing constructs. The key mechanisms for implementing genericity are *parameterized specifications* and the use of *uninterpreted data types* and *uninterpreted primitives* in place of defined data types and defined symbols.

Another important idea is the use of a *single unifying framework.* This is essential to avoid semantic gaps between different areas of formal description and formal reasoning. However, there is a fundamental difference of style in how a single unifying framework should be used to reason about a multi-layered stack of software and hardware layers. One approach is to re-cast diverse forms of description in one basic mold. The other approach, which we recommend, is to embed *natural notations* from well-established formalisms such as temporal logic and denotational description. We benefit from the built-in economy of these special-purpose notations when they are applied to particular areas of formal description.

To ensure that formal descriptions fit neatly into a wider context, we emphasize the importance of writing specifications which translate easily into *established notations.* At hardware levels, these established notations may be machine-readable languages such as VHDL or other conventional forms of description such as timing diagrams for interface protocols. At software levels, these established notations are mainly the natural notations of well-established formalisms such as Hoare logic and denotational semantics.

Finally, we believe that it is necessary, for all practical purposes, to use a formalism with (at least) the expressive power of *higher-order logic* to support the above recommendations. We also argue that the primary role of mechanical proof-generation is to support the intelligent participation of a human verifier in the rigorous analysis of a design at a level which supports clear thinking.

### 1.3.2 Structured Computer Organization

The structured view of a computing system as a multi-layered stack is well established in computer science and computer engineering. An early example of this concept and its usefulness was the invention of microprogramming by Wilkes in 1951 [71]. The eventual result was a drastic reduction in the complexity of the hardware which was important in the days of vacuum tube electronics. In a modern computing system, the number of levels has grown to typically include [136]:

- Problem-oriented languages
- Assembly languages
- Operating system machine
- Conventional machine
- Microprogramming
- Digital logic

Each of these levels can be thought of as a hypothetical or virtual machine which provides a complete model of computation. For instance, programs can be written in a problem-oriented language such as Ada without knowing anything about lower levels. One of the main reasons for imposing this structure on a computing system is to control the complexity of its design: each level is implemented with facilities provided by the next lower level. Another important reason for imposing this structure is to give greater independence to each layer: ideally, an Ada program should have the same result on different machines and similarly, a machine language program should have the same result on two different implementations of a particular machine architecture.

There are many ways in which this structured view of a computing system can be expanded to reveal details in specialized areas of interest. For example:

- Network models, such as the ISO Reference Model[3], can be structured into distinct layers [135,145].
- Operating systems are designed as a series of layers extending outwards from the operating system nucleus to the user level [95].
- Compilers are usually divided into a number of layers or *phases* sometimes involving intermediate languages [1].
- Architectural descriptions of a microprocessor can be presented as a hierarchy of interpretation levels which link the semantics of the instruction set to the operation of basic logic components [2].
- Instruction processing is overlapped in a pipelined microprocessor by organizing the internal architecture into several stages [91].

In many cases, the discovery or invention of layers in a complex design is based on well-conceived abstractions. For instance, Zimmerman [145] describes principles used to decide upon the seven layers of the ISO Reference Model for computer networks. Anceau [2] describes principles for the introduction of new interpretation levels in the context of microprocessor design. Katevenis [91] describes the use of extra hardware in pipelined microprocessors for managing special conditions (e.g., internal forwarding) to support higher level views of how instructions are concurrently processed by the hardware.

The fact that well-conceived abstractions are involved is significant when formal methods are employed to reason about multi-layered structures. A clean separation between abstraction levels will mean fewer special cases to consider in the formal proof and fewer *ad hoc* assumptions in the resulting correctness theorems.

We also suggest that formal methods may be used in the design process itself to demonstrate that the discovery or invention of a new layer is indeed based on a well-conceived abstraction. Birtwistle [9], Davie [40], Fourman [47,48], Hanna [69], Milne [105] and others have also suggested that formal verification may play a useful role in a *verification-driven* approach to design.

[3] *International Standards Organization* OSI (*Open Systems Interconnection*) Reference Model.

### 1.3.3 Totally Verified Systems

The idea of using logic to reason about computer programs and computer hardware is also well established in computer science and computer engineering. The use of Boolean logic as a descriptive method for logic circuits is due mainly to Shannon in the 1930's. In the 1960's, pioneering work on the application of logic to software was done by Floyd, Hoare, Landin, McCarthy, Scott, Stratchey and others. The use of logic to verify compilers and language implementations also began in the 1960's with work by McCarthy and Painter.

This early work was generally concerned with small fragments or single layers. However, the idea of a *totally verified system*, that is, the application of logic to every layer in a computing system was described as early as 1969 in a seminal paper by Hoare [75]:

> When the correctness of a program, its compiler, and the hardware of the computer have all been established with mathematical certainty, it will be possible to place great reliance on the results of the program, and predict their properties with a confidence limited only by the reliability of the electronics.

The idea of a *verified stack* provides a strategy for building totally verified systems. Just as the structured view of a computing system controls the complexity of its design, a structured view is also the chief means of controlling the complexity of its formal verification. In the horizontal dimension, correctness results are established for the internal design of each layer with respect to its external design. In the vertical dimension, the external design of every layer is linked together to form a chain of logical dependencies from the highest level of formal description down to the lowest level of formal description.

### 1.3.4 Generic Specification

Generic description is a powerful concept in high level language design. For instance, the 'generic mechanism' of Ada allows a subprogram or package to be parameterized by types and subprograms as well as values and objects. This feature supports fundamental principles of modularity and abstraction as well as provides a convenient mechanism for the reliable re-use of software. Generic description is also a powerful concept in the context of formal proof where modularity, abstraction and re-usability are highly desirable attributes.

In addition to these well-known advantages, generic description can be used in a formal proof to filter out non-essential detail - a modern-day Occam's razor.[4] In the case of verifying a multi-layered stack of software and hardware levels, this use of generic specification is particularly important as a mechanism for enforcing a sharp distinction between what has and what has not been formally considered in the proof.

Each layer in a multi-layered stack is a virtual machine described in terms of operations performed on data. When verifying that the internal design of a particular layer correctly implements its external design, the precise nature of these operations often

[4]The principle that the fewest possible assumptions are to be made in explaining an idea [William of Occam, English philosopher, died *circa* 1350].

turns out to be irrelevant. In such cases, defined symbols are only being used as place-holders in the correctness proof. To make it clear when a symbol is just a place-holder, we use uninterpreted primitives in place of defined symbols. For similar reasons, we use uninterpreted data types in place of defined data types.

We argue in this dissertation that verifying the internal design of individual layers is a highly localized concern which should be separated as much as possible from detail only relevant to other levels of proof. In this approach, uninterpreted primitives and uninterpreted data types only become defined when they are linked into the verified stack. This contrasts with the *closed-world* approach to formal verification where every operator and every data type is completely defined within each level.

In addition to forcing a sharp distinction between what has and what has not been formally considered at each level in a verified stack, the use of generic specification avoids much of the infra-structure required in the closed-world approach for reasoning about particular areas of application, e.g., hardware-oriented data types. Because less infra-structure is required, it is easier to reproduce correctness proofs in other formalisms which lack this infra-structure.

The primary example of generic specification in this dissertation is the formal specification of a simple microprocessor called TAMARACK-3. The basic data types and primitive data operations used in the formal specification are *uninterpreted data types* and *uninterpreted primitives* respectively.

A second example of generic specification is the specification of a behavioural model for an asynchronous memory device. We first specify a generic model which captures the essential features of how the memory device responds to a memory request and then instantiate this generic model with non-essential details. In addition to achieving a more readable specification, the generic model could be re-used for other external devices such as sensors and actuators in a real-time control system.

A third example, reported elsewhere [81], is the generic specification of a regular structure. For a large regular structure, e.g., the 31Kbit microcode ROM of the MC68000 [2], some form of genericity is essential for controlling the complexity of its structural description. This form of genericity can also be viewed as a synthesis algorithm [87].

### 1.3.5 Embedding Other Notation

A concise and meaningful specification often depends on the right choice of notation. This fact was made vividly clear during the course of our research by experimenting with different ways to specify the TAMARACK-3 memory interface which is based on a four-phase handshaking protocol.

Our first attempt (reported elsewhere [82]) used explicit time variables without the introduction of any special notation. For instance, existential quantification was used to specify the unknown length of wait states in a handshaking sequence. This first attempt achieved some of our goals, in particular, the goal of giving an independent specification for external memory (as a physically separate device). However, the specifications were cumbersome and not intuitively clear. When presenting this work, we often resorted to replacing parts of the specification with natural language phrases so that the specification might be understood. The formalization also failed to cleanly separate constraints

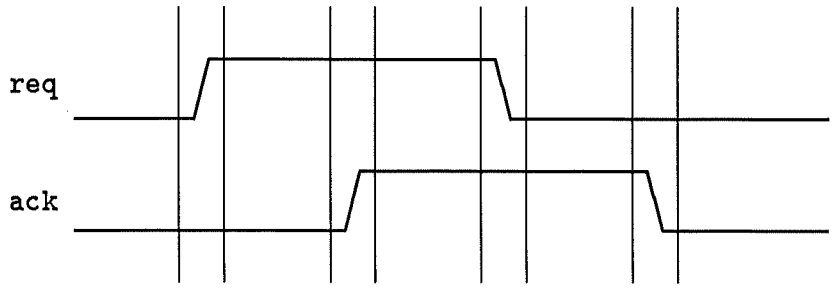



Figure 1.1: Simplified Handshaking Timing Diagram

on the handshaking signals from constraints on the accompanying data signals.

Our second attempt (and the one used in our verified stack) is based on the notation of temporal logic. The idea of using temporal logic to specify handshaking protocols had already been described by several people including Bochman [11], Dill and Clarke [44], and Fujita et al. [49]. The idea of embedding other calculi in higher-order logic had been demonstrated by Gordon [61] (for program logics) and by Hale [65] (for another form of temporal logic). The combined influence of these ideas lead us to experiment with temporal logic by defining some temporal logic operators as higher-order functions. The outcome of this experiment was far superior to our earlier attempt: the formal specifications were intuitively clear and concise. Moreover, the proof of correctness was easier and more general.

To elaborate on this point, Figure 1.1 shows a timing diagram typically found in conventional descriptions of constraints on a pair of handshaking signals, `req` ("request") and `ack` ("acknowledge"), used to synchronize data transfers between a microprocessor and an external device such as a memory chip.

The corresponding description in formal notation is given by the following set of temporal logic assertions which express constraints on the pair of handshaking signals. Although these assertions involve symbols with precise mathematical meanings (in the context of formal proof), they can be informally translated into natural language by reading the operators $\sim$, $\Diamond$, $\longrightarrow$ and $\cup$ as "not", "eventually", "implies" and "until".

```
(req ⟶ (req ∪ ack))
(req ⟶ (◇ack))
(ack ⟶ (ack ∪ (∼req)))
(ack ⟶ (◇(∼req)))
((∼req) ⟶ ((∼req) ∪ (∼ack)))
((∼req) ⟶ (◇(∼ack)))
((∼ack) ⟶ ((∼ack) ∪ req))
((∼ack) ⟶ (◇req))
```

For example, the assertion,

```
(req ⟶ (req ∪ ack))
```

expresses the constraint that when `req` is true, then it must remain true until `ack` becomes true. In other words, once a request has been initiated, the request must continue until it is acknowledged. The remaining seven assertions are explained in Chapter 6.

In addition to using the notation of temporal logic to specify the asynchronous memory interface, we have used the specialized notation of denotational description [53] to specify the semantics of a structured programming language at the top of our verified stack. In a conventional framework, the semantic clause

$$C[\![\texttt{C1;C2}]\!] = C[\![\texttt{C1}]\!] \circ C[\![\texttt{C2}]\!]$$

would be used to give the denotation (or mathematical meaning) of a command sequence `C1;C2` where the semantic function $C$ is applied to syntactic objects surrounded by emphatic brackets $[\![$ and $]\!]$. We are able to closely imitate this style using relations instead of partial functions.[5]

From this and similar experiences, we believe that natural notations from well-established formalisms such as temporal logic and denotational description are valuable for specifying and reasoning about the diverse aspects of structure and behaviour in a microprocessor-based system. The ability to represent different notations in a single unified framework is one of the more essential uses of higher-order logic in our work. In addition to Gordon and Hale, several others including Camilleri [21] and Loewenstein [96] have also reported benefits of embedding other calculi in the framework of higher-order logic.

### 1.3.6 Established Notations

At the hardware level, designers use a variety of established notations for diverse aspects of structure and behaviour. Some of these notations, such as a conventional hardware description language, are machine-readable. Other forms of conventional description are not necessarily machine-readable, for instance, they could be pictorial representations of block structure and memory interface timing diagrams.

To be understood in a wider context, formal descriptions must translate easily into these established notations. In some cases, it may be possible to mechanically translate a formal description into an established notation. In other cases, this translation is more informal, e.g., understanding a set of temporal logic assertions in relation to a memory interface timing diagram. In either case, the correspondence between a formal description and the same description in an established notation should be plainly seen with only a minimum of explanation.

One area of formal description suitable for mechanical translation is the structural description of hardware. For example, Figure 1.2 shows the implementation of an AND-gate by a NAND-gate and NOT-gate.

In higher-order logic, the structure of this implementation is described by the following equation:

$$\texttt{ANDGate (i1,i2,outp)} = \exists \texttt{x. NANDGate (i1,i2,x)} \wedge \texttt{NOTGate (x,outp)}$$

[5] Gordon has also used higher-order logic to represent a denotational semantics in a similar style [61].

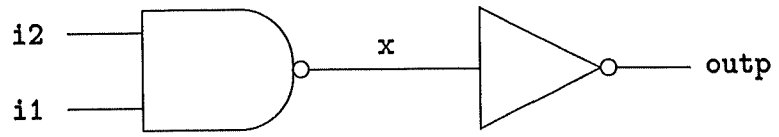



Figure 1.2: AND-Gate Implementation

In VHDL[6] this same structure is described by the following declarations:

```
entity ANDGate is
    port (i1,i2:in Bit;outp:out Bit);
end ANDGate;

architecture Structure of ANDGate is
    component NOTGate port (i:in Bit;outp:out Bit);
    component NANDGate port (i1,i2:in Bit;outp:out Bit);
    signal x : Bit;

    G1: NANDGate port map (i1,i2,x);
    G2: NOTGate port map (x,outp);
end Structure;
```

The syntactic differences between these two descriptions are mostly superficial. The VHDL declarations contain more keywords and use longer identifiers for keywords, e.g., the symbol ∃ corresponds to the keyword signal. The VHDL declarations also involve a full set of type declarations. The higher-order logic description could also be annotated with type declarations but normally this additional information is implied by context.

Overlooking minor differences of syntax, the close correspondence between formal and conventional forms of descriptions is not surprising. Both are based on the same fundamental style for describing hierarchical structure. Systems (or devices) at all levels are viewed externally as 'black boxes' connected to the external environment though a set of labelled ports, e.g, `i1`, `i2` and `outp`. Internally, they are decomposed into a set of components, e.g., `NANDGate` and `NOTGate` and internal signals, e.g., `x`. Internal connections are indicated by ports with common labels, e.g., the internal signal `x` connects the output of the NAND-gate to the input of the NOT-gate.

In general, this correspondence between higher-order logic and established notations like VHDL scales upwards for 'bigger' cases of structural description. This correspondence has been demonstrated by Van Tassel in the (mostly) mechanical translation of a VHDL specification for an earlier version of the TAMARACK-3 microprocessor into a HOL specification [140,141]. Many other formal description languages besides higher-order logic can also be used to write similar descriptions of structure which easily translate into established notations.

Other forms of conventional description are not necessarily machine-readable. For instance, asynchronous interactions between a microprocessor and external devices are

[6]VHDL is the VHSIC (*Very High Speed Integrated Circuit*) Hardware Description Language now adopted as an IEEE standard (Std 1076-1987) [3].

usually described by a mixture of natural language, flowcharts and timing diagrams. Here, we are using the term 'established notations' in a very loose sense to refer to this mixture of descriptions: though informal, they generally conform to standard conventions of style.

In this case, we have aimed to write formal descriptions which can be understood in relation to conventional forms of description with a minimum of explanation. Earlier, we described how a set of temporal logic assertions can be informally translated into a natural language description which directly corresponds to the timing diagram in Figure 1.1 Hence, natural notations such as temporal logic, besides offering the advantage of built-in economy, can also be successful as machine-readable versions of established notations such as flowcharts and timing diagrams.

## 1.4 The TAMARACK Stack

The verified stack described in this dissertation is based on a compiler for a very simple programming language called IMP and a very simple microprocessor called TAMARACK-3. The compiler and the microprocessor are both organized into a series of layers. The complete stack (for the purposes of this dissertation) is shown in Figure 1.3.

- Compiler
  - IMP language (hierarchically structured)
  - SM code (flat intermediate form)
  - TM code (target machine)
- Microprocessor
  - programming level
  - microprogramming level
  - phase level (register-transfer level structure)

Figure 1.3: The TAMARACK Stack

This stack can be extended both upwards and downwards. A paper by Gordon [61] shows how to extend this stack upwards by deriving Hoare proof rules for reasoning about IMP programs from a denotational semantics similar to one we have given for IMP. In work reported elsewhere [80,81,84,87,88], we have extended this stack downwards to the transistor level for an earlier version of the TAMARACK-3 microprocessor. In collaboration with researchers at SRI (Menlo Park), we have also begun to consider how to bridge the semantic gap between the bottom layer in the TAMARACK stack and the engineering models used to describe components in the library of a commercial silicon compiler [89].

### 1.4.1 A Very Simple Compiler

The IMP programming language is a hierarchically structured language with only a few basic constructs, e.g., expressions, assignment statements, while-loops. A semantics for this language is given in a typical denotational style by semantic functions which satisfy a set of semantics clauses. The main difference from the semantics given for IMP by Gordon [61] is the use of modular arithmetic to model the finite size of machine words and memory.

The compiler for this language is implemented by two layers or 'phases'.[7] The first phase compiles the hierarchically structured program into a flat intermediate form called SM (Simple Machine) code. The second phase assembles SM code into TM (Target Machine) code. These two compilation phases are shown in Figure 1.4 where a simple IMP program is first compiled into SM and then assembled into TM. Operational semantics are given for both SM code and TM code.

The semantics and compiler for IMP are not parameterized to the same extent as the underlying hardware model. This is partly because this represents an earlier stage in our research and partly because a few more computational details are needed to verify the compiler. Nevertheless, the compiler specification is parameterized by the number of bits in a full-size word and, indirectly, by the size of memory.

This part of the verified stack is described in a separate paper included as an appendix to this dissertation; the paper is based on a technical report [85] which gives full details of the compiler and its formal verification. Both the paper and technical report are based on an earlier version of TAMARACK-3 but only slight modifications were needed to adapt this work to the current version of the hardware.

### 1.4.2 A Very Simple Microprocessor

The main emphasis in this dissertation is on the TAMARACK-3 microprocessor which occupies the bottom half of this verified stack. This microprocessor has eight programming level instructions and a single addressing mode. The only kind of hardware exception is a single level, non-vectored hardware interrupt. The microprocessor can be interfaced to external memory to operate in one of three possible modes: fully synchronous, fully asynchronous, and extended cycle mode. All I/O is memory-mapped. Figure 1.5 shows a functional diagram for the externally available signals of TAMARACK-3 (excluding clock signals, reset signal and voltage sources). These signals would be pins or groups of pins in a microchip implementation of this design.

As we have shown in Figure 1.3, the microprocessor part of the verified stack is separated into three layers. The programming level model is a description of its operation as seen by a machine language programmer. This includes both the instruction set semantics and the processing of interrupts. The next layer down is the microprogramming level which describes the sequential execution of microcode. The phase level, at the very bottom of the verified stack, reveals the structural organization of the internal architecture in terms of register-transfer level components.

The original version of this simple microprocessor was described by Gordon [55,58]

[7] A compiler phase is an entirely different idea than the phase level of the microprocessor architecture. Both uses of the term 'phase' are standard terminology [1,2].

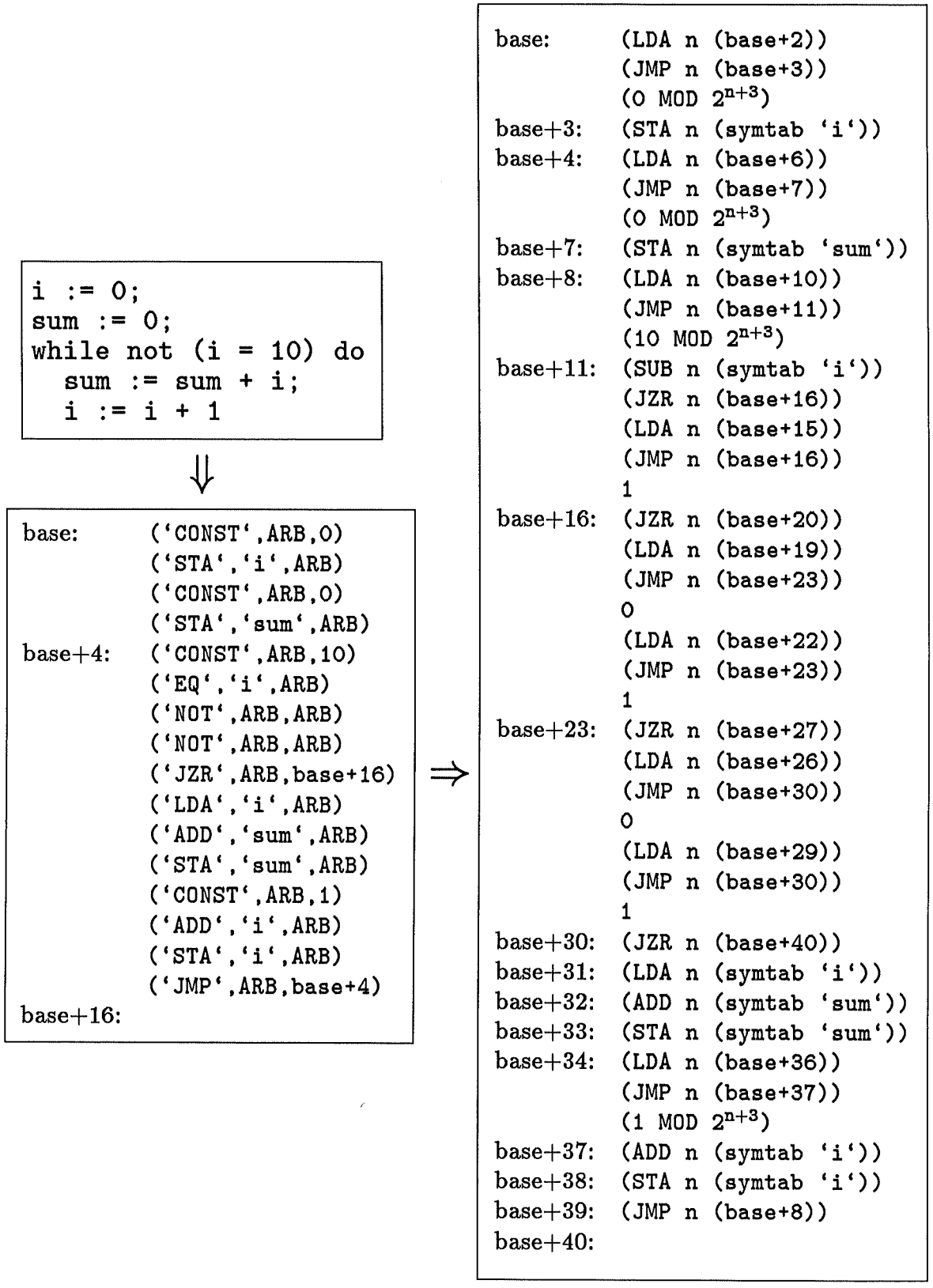
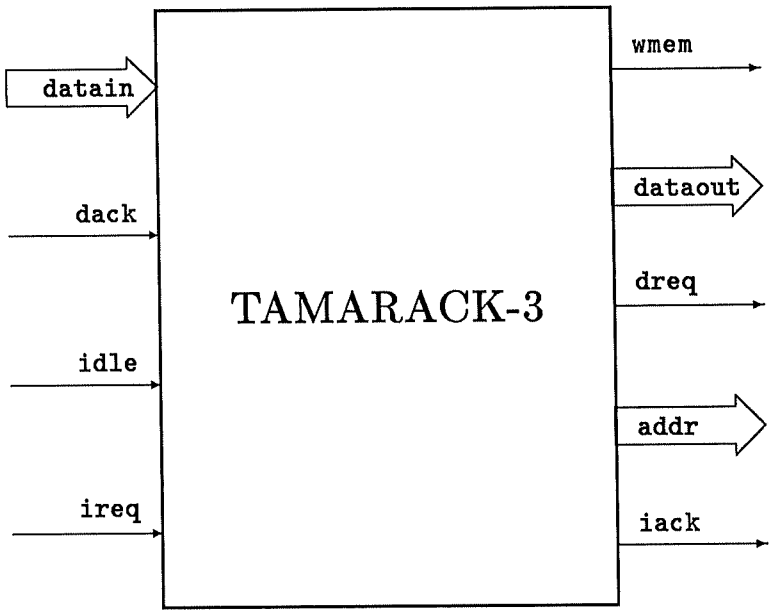



Figure 1.4: Two Phase Compilation of an IMP Program




| | | | |
|---|---|---|---|
| `datain` | - data from memory | `wmem` | - read/write select |
| `dack` | - data acknowledge | `dataout` | - data to memory |
| `idle` | - extended cycle mode | `dreq` | - data request |
| `ireq` | - interrupt request | `addr` | - address to memory |
| | | `iack` | - interrupt acknowledge |

Figure 1.5: Functional View of the TAMARACK-3 Microprocessor.

for a predecessor of the HOL system called LCF_LSM [57]. This example was re-done in HOL, implemented as a CMOS microchip (as an exercise in verification-driven design), and given the name TAMARACK-1 [78,79,88]. The design was then streamlined in TAMARACK-2 by eliminating some non-essential control features (and used as the target machine for an earlier version of the IMP compiler described in [85]). The version described here, TAMARACK-3, is distinguished from earlier designs by the addition of a hardware interrupt mechanism and the ability to interact asynchronously with external devices using handshaking signals.

Because TAMARACK-3 was designed as a verification example, it is not seriously intended for practical applications.[8] We have deliberately avoided some forms of complexity found in 'real designs' which do not necessarily contribute interesting verification problems aside from the very important problem of managing the sheer size of such proofs. For instance, multiple addressing modes are not provided because the problem of establishing correctness results for a particular instruction in different addressing modes is largely a matter of repeating the same proof strategy with slight variations for each case (as Cohn [30] has reported for the verification of the commercially-available VIPER microprocessor).

Although we have purged as much repetitious complexity as possible from the design of TAMARACK-3, the formal verification of this design is *not* a trivial problem. In place of repetitious complexity, we have introduced features which give rise to some intrinsically complex verification problems. In particular, the use of handshaking signals for data exchanges with external memory has lead us to consider the problem of reasoning about asynchronous interactions between a microprocessor system and external devices. By avoiding repetitious complexity, we have been left with a great deal of flexibility to consider different approaches to this particular problem, and more generally, to consider different strategies and techniques for structuring a formal proof into several layers.

### 1.4.3 Linking the Compiler to the Microprocessor

The target machine of the IMP compiler is an instance of the generic TAMARACK-3 programming level model. The latter is more general partly because uninterpreted data types and uninterpreted primitives are used in the formal specification of TAMARACK-3 in place of defined data types and defined symbols. To link the top part of the verified stack with the bottom part, an instance of the TAMARACK-3 programming level model is created to satisfy the specification of the target machine, i.e., the operational semantics of TM code.

This is partly achieved by associating defined data types and defined symbols mentioned in the IMP semantics with uninterpreted data types and uninterpreted primitives that appear in the formal specification of the TAMARACK-3 microprocessor. For example, the defined symbol +, used in combination with the modulus function to specify the semantics of an IMP plus-expression, is associated with an uninterpreted primitive called add which is used to specify the semantics of a TAMARACK-3 ADD instruction.

[8] But with a few more ALU functions, more kinds of conditional branches, and an alternative to absolute addressing, it is possible to imagine the use of a TAMARACK-like microprocessor in a very simple real-time control application. As Turner et al. [138] remarked with regard to railroad signalling, "the most complex interlock arrangement requires a simple combination of Boolean and time-based sequential logic, all well within the capability of the arithmetic unit of a small microprocessor".

The TAMARACK-3 programming level model is also more general because normal program flow can be interrupted by an external interrupt request. Even though interrupts are fully supported in the bottom half of the verified stack, it is currently necessary to disable interrupts (by assuming that the interrupt request pin `ireq` is wired to 'ground') in order to link the compiler to the microprocessor. This limitation is due to the way that we have defined the semantics of the IMP language: it is not a limitation of the microprocessor verification. However, it should be possible in future work to allow interrupts by modifying the semantics of IMP either using a continuation semantics [53,132] (and thus, remaining in a denotational framework) or else using a structured, or Plotkin-style, operational semantics [117].

Establishing that the target machine of the IMP compiler is an instance of the TAMARACK-3 microprocessor is the main illustration in this dissertation of how individual layers dealing with widely separated concerns can be linked together to form a chain of dependencies from the highest level of formal description down to the lowest level of formal description.

## 1.5 Related Work

Our higher-order logic approach is directly inspired by Gordon [59]. The idea of using higher-order logic to specify and reason about hardware was first advocated by Hanna [67]. This section describes other work related specifically to our main areas of interest: microprocessor verification, verified synthesis, compiler verification and verified systems. There is a great deal of important work by others in the more general area of verifying hardware; we mention some examples of this work elsewhere in this dissertation as it relates to specific points of interest.

### 1.5.1 Microprocessor Verification

Gordon's [58] verification of a simple computer using the LCF_LSM system was an early example of how formal proof and mechanical proof-generation could be used to reason about the design of a microprocessor. In addition to our work on TAMARACK-3, Gordon's example has been used to illustrate several other approaches to animating and verifying microprocessor hardware. This includes work by Barrow [5], Camilleri [19,20], Curzon [39], Davie [40], Richards [121], Van Tassel [140,141], and Weise [143].

The diversity of structure and behaviour in a typical microprocessor provides a rich source of verification problems. Currently, the field of microprocessor verification is dominated by two main examples: Hunt's verification of FM8501 [76] and Cohn's verification of VIPER [29,30,31]. These two examples are widely seen as the current state-of-the-art in microprocessor verification. Hunt was the first to consider the problem of reasoning about the implementation of a handshaking protocol in a microprocessor system. The VIPER project shows how the design of a microprocessor can be subjected to formal analysis in a series of decreasingly abstract levels.

#### 1.5.1.1 FM8501

FM8501 is a 16-bit microprocessor similar in complexity to a PDP-11. The instruction set of this microprocessor is rich enough to support realistic applications.

The programming level model of FM8501 and its internal architecture are formally specified in pure Lisp. The Boyer-Moore theorem-prover [13] was used to generate a formal proof of correctness which bridged the gap between the semantics of the FM8501 instruction set and a gate-level description of its implementation.

A particularly interesting aspect of the FM8501 example is the asynchronous memory interface which uses handshaking signals to synchronize data exchanges between the microprocessor and external memory. In the absence of existential quantification (which is not provided in Boyer-Moore logic), Hunt used an oracle to 'guess' the length of wait states in handshaking interactions. The correctness proof established that the implementation of the microprocessor is correct for all possible oracles.

The FM8501 also served as a prototype for the FM8502 which occupies the bottom layer of the verified stack developed by researchers at Computational Logic, Inc. [6,7].

#### 1.5.1.2 VIPER

The other well-known state-of-the-art example is Cohn's [29,30,31] verification of the VIPER microprocessor. This commercially-available microprocessor was designed by the British Ministry of Defence for use in life-critical applications [36,37]. Cullyer, one of the designers of VIPER, produced an informal paper-and-pencil correctness proof which related the top-level specification to the next lower level of description called the 'major state machine'. This level of proof was then re-done by Cohn using the HOL system and later extended down to an even lower level of description called the 'block level description'.

The first level of proof showed that the major state machine, with corrections, faithfully implements the top level specification of VIPER. This level of proof was exclusively concerned with flow of control and not with arithmetic or logical computations.

The second level of proof dealt with the block level description of VIPER which directly relates to the circuit design. In addition to flow of control, this level of proof was concerned with arithmetic and logical computations performed by functional units. At the block level, the proof only considered the operation of VIPER under normal conditions. These conditions appear explicitly in the correctness results as stated assumptions. Under these conditions, a limited set of correctness results were obtained which "amount to an analysis of the block machine under all circumstances covered by the specification" [30]. One of the reasons that made it impractical to carry out a complete proof of the block level was the current lack of support in the HOL system for reasoning about bit-level operations.

A major source of complexity in the VIPER proof is the problem of managing the sheer size of the correctness proof for a real design. There are 120 sequences of major state transitions to consider and each of these is implemented by a sequence of minor state transitions. However, the verification task would have been much harder if not for the assumption (as a condition of normal operation) that every memory request is satisfied in a fixed and minimal number of clock cycles.[9] Consequently, state transitions

[9]Although the VIPER design supports other protocols, the high-level specification given by VIPER

in the top level specification correspond to a finite set of fixed sequences at lower levels.

A particularly interesting outcome of the VIPER project is that it revealed weaknesses in the links between designer, verifier and manufacturer. One problem faced by Cohn was to derive a formal description of the block level from a mixture of engineering documents, partly pictorial and partly textual, supplied by the VIPER designers. Errors were found in both the top level specification of VIPER and in the major state machine, but these errors were not propagated down to the fabricated chips. Drawing from this experience, Cohn has contributed a sound appreciation of the scope and limitations of using formal proof to verify microprocessor systems.

Cullyer [38] describes plans for using VIPER in a railroad signalling application which also involves use of the HOL system to verify software against a formalization of well-established rules of railroad signalling. Gordon [63] has proposed to verify a compiling algorithm for a structured assembly language called VISTA which is targetted to VIPER.

#### 1.5.1.3 Other Work

Crocker et al. [35] describe a re-verification of the FM8501 using SDVS (*State Delta Verification System*). The semantics of SDVS were well-suited to specifying the external memory as a separate process from the FM8501. In previous work, this group verified a microcoded packet switch called the C/30 used in the US Defense Data Network; however, this work is not yet publicly documented.

Another example of verifying an asynchronous memory interface is described by Sekar and Srivas [128]. Bickford and Srivas [8] have begun work on verifying a 3-stage instruction pipelined RISC processor with plans to eventually extend this example with interrupts and an asynchronous memory interface.

Herbert [72] verified a network interface chip in the Cambridge Fast Ring using both the LCF_LSM and HOL systems. The formal description of this chip was mechanically generated from the designer's original specification written in Modula-2.

Narendran and Stillman [114] hand-translated the VHDL description of an image processing chip into first-order logic and used RRL (*Rewrite Rule Laboratory*) to generate correctness results. Formal verification revealed several errors unknown to the chip designers.

Birtwistle and Graham [10,64] describe work on formalizing the design of a functional language co-processor based on Landin's SECD machine. The formal verification is being undertaken in the HOL system with plans to verify a complete system based on the fabricated chip.

Bowen [12] has used Z, a specification language developed at Oxford University, to specify the entire MC6800 instruction set including interrupts and memory configurations.

Leonard [94] is investigating techniques for specifying computer architecture in higher-order logic. This work is particularly concerned with multi-processor systems which interact through shared memory. This work also considers techniques for mapping architectural specifications to implementations.

Rushby and von Henke [124] have used the SRI EHDM (*Enhanced Hierarchical Design Methodology*) system to verify the Interactive Convergence Clock Synchronization Al-

designers to Cohn ignored these other protocols, and hence, the proof had to also ignore them [32].

gorithm of Lamport and Melliar-Smith. Clock synchronization is fundamental to fault tolerance mechanisms in life-critical systems implemented by a set of communicating microprocessors. This work demonstrated the value of mechanical proof-generation with the discovery of several technical flaws in previously published hand-proofs.

### 1.5.2 Verified Synthesis

Early work on verified synthesis by Milne [104] proved the correctness of a very simple silicon compiler in a process algebra called CIRCAL.

May and Shepherd [130] used algebraic laws of the OCCAM programming language [122] to transform a high-level specification of the INMOS T800 Transputer floating-point unit into a microcode level description.

Martin et al. [97] have used synthesis techniques amenable to formal verification to generate the first entirely asynchronous (also called self-timed or delay-insensitive) microprocessor from a high-level specification based on CSP. The surprising robustness of the fabricated chips to variations in temperature and VDD voltage values [98] may also be significant for life-critical applications.

Brown and Leeser [15] describe work on compiling programs into application specific chips generating a microcoded controller and datapath as an intermediate stage in the synthesis process. The synthesis procedures are being developed using the NUPRL theorem-prover [34].

Brock and Hunt [14] describe techniques for verifying circuit generation functions formally specified by a list of constants in the Boyer-Moore logic [13]. They have verified a family of ALU's in the Boyer-Moore theorem prover using these techniques.

Fourman et al. [47,48] have combined CAD technology with a rule-manipulation system based on a higher-order polymorphic predicate calculus of partial terms. Their system, called LAMBDA, supports the interactive refinement of designs directed by the user through a graphical interface but constrained by the underlying logic.

Hanna et al. [69] use the VERITAS$^+$ system in a goal-directed manner to interactively generate a design and a correctness proof. The synthesized design is then translated into the MODEL hardware description language.

Johnson et al. [77] have considered the interplay between design verification and design synthesis. They have developed a transformation system DDD (*Digital Design Derivation*) based on functional algebra which has been used to reduce the FM8501 programming level model to a gate level description.

### 1.5.3 Compiler Verification

The earliest example of compiler correctness (that we are aware of) was described more than twenty years ago by McCarthy and Painter [99]. They verified an algorithm for compiling arithmetic expressions into code for an abstract machine. This early work established a paradigm for subsequent work on compiler correctness (as summarized by Cohn [28]):

- Abstract syntax.
- Idealized hardware.

- Abstract specification of the compiler.
- Denotational source language semantics.
- Operational target machine semantics.
- Correctness stated as a relationship between the denotation of a program and the execution of its compiled form.
- Proofs by induction on the structure of source language expressions.

In a separate report [85], we give a detailed history of subsequent developments in this area. This includes work described by: Kaplan [90]; Burstall and Landin [17]; Milner and Weyhrauch [107]; Morris [110,111]; Chirica [24]; Milne and Strachey [106]; Goguen et al. [50]; Russell [126]; Cohn [28]; Polak [118,119]; Thatcher et al. [137]; Chirica and Martin [25]; Despeyroux [42] and Collier [33]. These developments include the use of algebraic methods and domain theory, more language features, verification by formal proof based on axioms and inference rules, mechanical assistance for proof-checking and proof-generation, and correctness proofs about parsing and syntax analysis.

## 1.5.4 Verified Systems

Most of the previous work on the compiler correctness problem is ten to twenty years old. This work has generally followed the paradigm laid down by McCarthy and Painter of distancing the problem from the details of real hardware by using a target machine with idealized features.

Since this early work, proof-generation systems have developed considerably and have been used to construct some very large proofs. These developments, combined with recent successes in the formal verification of microprocessors such as VIPER and FM8501, have revived interest in the compiler correctness problem and given it a greater relevance than before.

### 1.5.4.1 The CLI Stack

The most remarkable achievement so far has been made by researchers [6,7] at Computational Logic, Inc. (CLI). The "short" version of the CLI stack consists of four layers:

| | |
|---|---|
| Micro-Gypsy | - a high level programming language [144] |
| Piton | - a high-level assembly language [109] |
| FM8502 | - 32-bit microprocessor based on the FM8501 prototype |
| Gates | - register-transfer level model of an implementation |

Each layer in this stack is intended to support realistic applications. A small operating system called Kit has also been implemented and proven correct. However, this operating system does not fit precisely onto the "short" stack due to minor architectural differences.

In addition to the four main layers of the CLI stack, there are several minor layers. For instance, the assembly of Piton programs involves an intermediate form called i-code which provides an intermediate layer between Piton and FM8502.

A distinctive aspect of the CLI approach is that each layer in the stack is cast into the same basic mold: each layer is described as a finite-state machine defined by an interpreter function in pure Lisp. This contrasts with the approach described in this dissertation of using special-purpose notations from well-established formalisms such as temporal logic and denotational description. Another distinctive feature of the CLI approach is its adherence to the closed-world principle where every operator and every data type is completely defined within each layer. This contrasts with our use of generic specification to filter out non-essential detail from each layer of a multi-layered stack.

#### 1.5.4.2 The SAFEMOS Stack

Research is jointly underway at Cambridge University, INMOS, Oxford University and SRI International (Cambridge Research Centre) on a project to develop a prototype design and verification environment for *real-time* mixed hardware/software systems.

The verified stack will include a real-time language with simple constructs based on OCCAM, a program verifier for that language, a verified processor and a verified translator for compiling the real-time language into the processor instruction set.

The project will combine the INMOS CAD system and HOL proof-generation system and will also involve other formal methods suitable for specifying real-time systems including Z and CSP.

## 1.6 Outline of this Dissertation

Chapter 2 provides an introduction to the HOL logic and the HOL proof-generation system. This chapter concludes with remarks on the use of formal proof and mechanical proof-generation to reason about the correctness of software and hardware.

Chapter 3 describes the operation and design of the TAMARACK-3 microprocessor in a conventional style of microprocessor description. This chapter introduces the idea of describing hardware generically and explains how the internal architecture of a microprocessor can be viewed as a hierarchy of interpretation levels.

Chapter 4 elaborates on the advantages of generic specification and shows how this technique is used to formally specify the TAMARACK-3 microprocessor. This chapter is central to the argument of this dissertation.

Chapter 5 illustrates fundamental proof strategies for verifying microprocessor systems. The main ideas presented in this chapter are: stating correctness results, structuring a proof into multiple levels, and using logic to symbolically execute a design.

Chapter 6 elaborates on the idea of embedding natural notations from special-purpose formalisms. This is illustrated by embedding a form of temporal logic in the HOL logic for the purpose of reasoning about the interaction of TAMARACK-3 with external memory using handshaking signals.

Chapter 7 concludes this dissertation with remarks on the scope and limitations of the TAMARACK-3 proof of correctness and its relationship to other levels of proof.

Chapter 2

# Formal Proof in the HOL System

The research described in this dissertation is based exclusively on 'the HOL logic'. This is a formulation of higher-order logic originally set out by Church [26] and adapted by Gordon [60,62] for representation in the HOL system.

In this chapter we provide brief descriptions of both the HOL logic and the HOL system. These descriptions have been tailored for the purposes of this dissertation; they are accurate in what is described but they do not represent a complete account of either the logic or the proof-generation system.

We also comment on the *security* and *extensibility* of the HOL system which we regard as its most important and distinctive features. Finally, we conclude this chapter with some remarks on why formal proof in a mechanized system such as HOL may be useful.

## 2.1 Introduction

Although higher-order logic was originally developed to study theoretical questions about the foundations of mathematics [70], a non-theoretician should be able to understand most aspects of this formalism without too much difficulty. Much of our notation will be familiar from the informal notation of mathematics, e.g., $\neg$, $\wedge$, $\vee$, $\Longrightarrow$, $\forall$, $\exists$. Some of the most important concepts will be familiar to readers with experience in strongly-typed programming languages and functional programming.

Higher-order logic extends first-order logic by allowing variables to range over functions and predicates. It also includes notation from the $\lambda$-calculus for describing functions. Functions which accept other functions as arguments or return functions as results are called 'higher-order' functions. The Lisp function `mapcar` is an example of a higher-order function in a programming language. In both logic and programming, higher-order functions result in specifications or definitions which are shorter, simpler and often easier to understand. Predicates in higher-order logic are just a particular kind of function, namely, a function which returns values corresponding to true and false. Thus, higher-order predicates are just higher-order functions which return values corresponding to true and false.

## 2.2 The HOL Logic

We begin with the syntax of the HOL logic which centers upon two main ideas: a set of well-formed *terms* composed from elementary terms and a set of *types* which are associated with terms in the logic.

### 2.2.1 Terms

More detailed descriptions of the HOL logic [60,62] reveal that there are just four different kinds of terms: *variables*, *constants*, *abstractions*, and *applications*. However, this conciseness is mostly hidden from a user of the HOL system and is not essential to our discussion.[1] Instead, we describe the syntax of the logic from a notational point of view where, for the purposes of this dissertation, the syntax consists of:

| | |
|---|---|
| variables | - e.g. `x`, `y`, `xyz` |
| constants | - e.g. `T`, `F`, `0`, `1`, `2` |
| negation | - ¬`t` where `t` is a term |
| conjunction | - `t1` ∧ `t2` where `t1` and `t2` are terms |
| disjunction | - `t1` ∨ `t2` where `t1` and `t2` are terms |
| implication | - `t1` ⟹ `t2` where `t1` and `t2` are terms |
| equality | - `t1` = `t2` where `t1` and `t2` are terms |
| universal quantification | - ∀`x.t` where `x` is a variable and `t` is a term |
| existential quantification | - ∃`x.t` where `x` is a variable and `t` is a term |
| function application | - `f t` where `f` and `t` are terms |
| pair | - `t1,t2` where `t1` and `t2` are terms |
| λ-expression | - λ`x.t` where `x` is a variable and `t` is a term |
| ε-expression | - ε`x.t` where `x` is a variable and `t` is a term |
| conditional expression | - `b` ⇒ `t1` \| `t2` where `b`, `t1` and `t2` are terms |
| let expression | - `let x = t1 in t2` where `x` is a variable and `t1` and `t2` are terms |

The meanings of ¬ ("not"), ∧ ("and"), ∨ ("or"), ⟹ ("implies"), = ("equals"), ∀ ("for all"), ∃ ("there exists") are common to many formal and informal notations of mathematics including the HOL logic.

A function application of the form `f t` denotes the application of the function `f` to the term `t`. If `f` is a predicate, then `f t` is an assertion which is either true or false. Certain function constants such as + ("addition") have a special syntactic status which allows them to be written as infix expressions, e.g., `1+2`.

In many other notations a function application must be written as `f(t)`. But in the HOL logic, `f(t)` is equivalent to `f t` unless matching parentheses are needed to enclose a compound term such as `1+2`, i.e., `f(1+2)`. Since, `f(t)` and `f t` are equivalent, there is no harm in using unnecessary parentheses (as we sometimes do) when this might help to make an expression easier to read.

Terms such as `(f t1 t2 t3)` are nested function applications. By convention, function application associates to the left which means that the term `(f t1 t2 t3)` is equivalent to `(((f t1) t2) t3)`. On the other hand, elements in an n-tuple associate to the right; for example, `(t1,t2,t3,t4)` is really just a compound term of nested pairs `(t1,(t2,(t3,t4)))`.

λ-expressions denote functions. For example, the λ-expression λ`x.x+1` denotes the successor function. This notation, borrowed from the λ-calculus, is just a succinct way

[1] The curious reader may find it interesting to know that ε, ⟹, and = are the only primitive constants in the HOL logic. More detailed descriptions of the logic [60,62] show how other constants such as `T`, ∀, and ∃ (which are often primitive constants in other formalisms) can be defined in terms of these three primitive constants.

to say "the function such that ..." without having to invent a name for the function. λ-expressions are often used in functional programming languages such as Lisp as arguments to higher-order functions like `mapcar`.

ε-expressions are a bit more exotic. The term `εx.P x` denotes a value satisfying the predicate `P` if such a value exists; otherwise, it denotes an arbitrary value of the appropriate type. For example, `εx.x < 10` denotes some natural number less than ten but `εx.x < 0` denotes an arbitrary natural number since the predicate "less than zero" cannot be satisfied in the natural numbers.

Conditional expressions of the form `b ⇒ t1 | t2` may be read as "*if* `b` *then* `t1` *else* `t2`". Let expressions of the form `let x = t1 in t2` denote the result of replacing all free occurrences of `x` in `t2` by `t1`; it is equivalent to `(λx.t2) t1`.

### 2.2.2 Types

Types are needed in the HOL logic to prevent certain inconsistencies such as Russell's paradox. When specifying hardware, types can be used to check the consistency (in an informal sense) of a design, e.g., if different types are used to represent 16-bit and 13-bit busses, then type-checking will reveal when a 16-bit bus has been mistaken for a 13-bit bus.

A type in the HOL logic is either a *type constant*, *compound type* or *type variable*. Built-in (or pre-defined) type constants include Boolean values, the natural numbers and string tokens.

```
:bool  - Boolean values T,F
:num   - natural numbers 0,1,2, ...
:tok   - string tokens, e.g., `a`, `abc`, `abc123`
```

Compound types are built up from existing types using type constructors. Built-in type constructors include Cartesian product,

$$ty1 \times ty2$$

which denotes the type of all pairs whose first element belongs to $ty1$ and second element belongs to $ty2$. There is also a type constructor for function types,

$$ty1 \rightarrow ty2$$

which denotes the type of all total functions with arguments of type $ty1$ and results of $ty2$. There are other type constructors built into the HOL logic but these are not needed for the purposes of this dissertation.

The third kind of type in the HOL logic is a type variable which stands for an arbitrary type. Any type which involves a type variable is called a *polymorphic type*. Type variables and polymorphism play an important role in this dissertation; in particular, they provide a way to write specifications that avoid details which are not relevant to a particular proof problem. Small greek letters, $\sigma$, $\beta$, $\gamma$, are often used for type variables, but in this dissertation we adhere to the machine-readable HOL notation in which type variables are prefixed by an asterisk. The HOL notation allows us to give meaningful names to type variables such as:

`:*wordn, :*word3, :*address, :*memory, :*word4`

However, these names are merely informal suggestions about the role of specific type variables in a particular proof.

### 2.2.3 Typed Terms

Every well-formed term in the HOL logic is associated with a type that is consistent with the types associated with its sub-terms. For example, a function from numbers to Booleans can only be applied to a number in a function application. This is similar to the idea of a strongly-typed programming language such as Ada where, for instance, the types of actual parameters in a function call must match the declared types of the corresponding formal parameters.

The type of a term can often be inferred from the types of its sub-terms, in particular, from fixed types associated with constants. For example, it can be inferred that the variable `b` is a Boolean value when it appears in the term $\neg$`b`. There is an algorithm, due to Milner [108], for such inferences. When the type of a term is ambiguous, its type can be indicated by a type annotation. For instance, the expression `b:bool` indicates that the variable `b` is a Boolean variable. We also use type annotations in our discussion to denote types themselves, e.g., `:bool`.

Once types have been unambiguously associated with every elementary sub-term, the type of a term can be derived hierarchically from the types of its sub-terms according to type deduction rules associated with each type constructor. This process is called *type-checking* or *type deduction.* For the HOL logic, type-checking is decidable which means that it can be performed automatically by the HOL system.[2]

### 2.2.4 Axioms and Inference Rules

In the HOL formulation of higher-order logic, a *formal proof* is a sequence of lines; each line has the form $\Gamma$ `|- t` where $\Gamma$ is a set of assumptions and `t` is a conclusion. Assumptions and conclusions are terms belonging to the type `:bool`. For each line $\Gamma$ `|- t`, the conclusion `t` is a true proposition if all of the assumptions in $\Gamma$ are true. If there are no assumptions, then the proposition is a theorem.

A sequence of such lines is a proof if every line is either an axiom or can be obtained from a previous line by a rule of inference. The HOL formulation of higher-order logic has five axioms and eight primitive inference rules. For example, when an implication `t1` $\Longrightarrow$ `t2` and its antecedent `t1` appear on previous lines of a proof, the consequent `t2` can follow on a later line inheriting assumptions from these two previous lines. This rule, called *Modus Ponens*, is illustrated below.

$$\frac{\Gamma 1 \text{ |- } t1 \Longrightarrow t2 \qquad \Gamma 2 \text{ |- } t1}{\Gamma 1 \cup \Gamma 2 \text{ |- } t2}$$

[2]For other logics with more powerful type systems, for instance, logics with dependent types, type-checking is not always decidable. However, Hanna et al. [68] reports that this loss of decidability is not necessarily a practical difficulty.

The main advantage of a formal proof is that every single step has a precise justification which can be mechanically checked. The main disadvantage is that the formal proof of a non-trivial theorem typically involves a large number of steps. However, the complexity of doing large formal proofs can be overcome with mechanical assistance from a computer both to check that each proof step is correct and to automatically generate large portions of the proof.

### 2.2.5 Theories and Definitions

A *theory* in the HOL logic is a set of constants, types and axioms. Theories can be organized into hierarchies where they inherit types, constants and axioms from their ancestors. In the HOL system, the term 'theory' has a slightly extended meaning which we explain in Section 2.3.3.

There is an initial or underlying theory in the HOL logic which provides the built-in types, constants and axioms of the logic. New term constants and new type constants can be introduced if they are not already present in a theory or in any of its ancestors. A *definitional axiom* for a new term constant or new type constant can be optionally introduced when the constant is introduced. The syntactic form of a definitional axiom is severely restricted to ensure that it is a *conservative extension* of the logic: this means that it does not introduce any new inconsistency that was not already present.

We refer to definitional axioms for new term constants as just plain *definitions* since they are the only kind of definitional axioms used in this dissertation. To a user of the HOL system (and in this dissertation) these definitions look similar to the definition of a function in a functional programming language. The syntactic constraints mentioned above require, in effect, a definition to be an equation of the form,

```
f x1 ...xn = t
```

where `f` is a term constant and all the free variables of `t` are included among `x1 ...xn`. As mentioned earlier, predicates are just functions which return Boolean values.

Non-definitional axioms can also be introduced in a theory but these are generally avoided since they have the potential of introducing inconsistency. In almost all cases, it is better for unproven assertions to appear as explicit assumptions. Non-definitional axioms are *not* used anywhere in the work described in this dissertation. This means that all of our theories are *definitional theories.*

## 2.3 The HOL Proof Generating System

The HOL system is an interactive programming environment interfaced to the HOL logic. It is used to construct formal proofs in the HOL logic. The user interacts with the HOL system through a strongly typed, functional programming language called ML. Terms, types, theorems and other features of the logic are data objects which are manipulated by ML functions.

### 2.3.1 Proving Theorems

At the start of a HOL session, the only theorems in the system are the axioms of the HOL logic and a standard set of previously derived theorems built into the HOL system. New theorems are generated from existing theorems by ML functions corresponding to the eight primitive inference rules. The type discipline of ML ensures that inference rules can only be applied to data objects representing previously generated theorems. This guarantees that the only way to obtain a theorem in the system is by the generation of a formal proof.

When introducing a definition, the HOL system enforces the logical requirement that it must be a conservation extension by ensuring that it satisfies syntactic constraints mentioned earlier. In this dissertation, we use a system function called `Define` to introduce definitions; this is a 'wrapper' function for the built-in HOL system functions that are normally used to create new definitions.

Primitive inference rules correspond to the smallest steps in a formal proof; to make large proofs feasible, ML functions can be programmed to apply sequences of primitive inference rules as single steps in a proof. A large number of these ML functions, called *derived inference rules*, are built into the HOL system. Some these of rules can potentially collapse hundreds (or even thousands) of primitive inferences into a single step. The user can also implement derived rules to either extend the built-in repertoire of general purpose rules or to perform a specific sequence of inferences in a particular proof (e.g. symbolic execution of a microinstruction in the microprocessor correctness proof).

### 2.3.2 Forward and Backward Proof

The development of a formal proof as a linear sequence of proof lines is a bottom-up process, called *forward* proof, which starts with axioms and uses inference rule to eventually prove the desired theorem. When developing a proof in this manner, it is often difficult to tell beforehand whether or not a sequence of proof steps will ultimately be successful. In practice, proofs are usually developed in a top-down manner, called *backwards* or *goal-oriented* proof, starting with the conclusion of the desired theorem and reducing the problem of proving this theorem to simpler sub-goals.

In general, top-down reasoning reflects how we intuitively solve problems: "I could prove X if I had a proof of Y and Z". In this case, the goal X is reduced to the sub-goals Y and Z. Sub-goals are repeatedly reduced to one or more sub-goals until they are all achieved by trivial inferences. Once all of the sub-goals have been achieved, the top-down development of the proof is used to guide the mechanical generation of a forward proof for the desired theorem.

The HOL system supports the top-down development of proofs by providing semi-automatic mechanisms to reduce a goal to sub-goals. Furthermore, the system constructs a forward proof of the desired theorem as a side-effect of generating sub-goals.

### 2.3.3 System Theories

The mathematical notion of a theory as a set of constants, types and axioms is extended in the HOL system to also serve as a database where derived theorems can be stored

for future use. Like the mathematical notion, system theories can be organized into a theory hierarchy. A theory created during a HOL session can be saved on disk in the computing environment and sometime later retrieved by loading the theory file into a subsequent HOL session or made accessible to a new theory as one of its ancestors. Large verification efforts such as the VIPER project [29,30,31] or the work described in this dissertation are made possible by this ability to build up a hierarchy of theories which serve both the mathematical purpose of imposing logical structure and the practical purpose of managing a very large and complex verification task.

In addition to the purely logical content of the HOL logic, the HOL system provides a number of built-in theories and a library of assorted theories which may be used as required. This includes, for instance, an axiomatization of Peano arithmetic and a set of theorems derived from this axiomatization. A sharper distinction between the purely logical content of the HOL system and additional theories built upon this foundation is given in [60,62].

### 2.3.4 Security

One of the most important features of the HOL system is that the generation of any theorem ultimately depends only on a finite set of known axioms and the eight primitive inference rules of the HOL logic. This has several important consequences including:

- All theorems have bona-fide proofs in higher-order logic.
- Any user can implement extensions to the built-in support for proof-generation (e.g., derived inference rules, decision procedures) without danger of compromising proof security.
- A record of primitive inferences generated by the system could be checked independently by another program which implements a very simple, non-interactive proof checker.

The security of proof-generation in the HOL system is inherited from its software parent, the LCF system [54] and is shared in common with several other systems derived from or influenced by the LCF approach, e.g., ISABELLE [116], LAMBDA [47], NUPRL [34], VERITAS$^+$ [68]. This feature distinguishes HOL and other systems which 'guarantee'[3] proof security by this or some other means from other verification systems which are either based on *ad hoc* rules or an insecure implementation of inference rules.

### 2.3.5 Extensibility

Some verification systems take full control during proof-generation and only ask for help from the user when standard heuristics fail to produce a complete proof. With the HOL system, the user has direct control over proof-generation down to the most detailed level of primitive inference steps. Most often, the user guides proof-generation at a much

[3]Even the extreme measures taken in the HOL system to 'guarantee' proof security are not infallible as demonstrated two years ago by the discovery of a very subtle form of insecurity (now repaired) concerning type variables.

higher level using derived rules and other kinds of transformations based on inference rules. When built-in proof support is not enough, the user has the full generality of a complete programming language to implement new, re-usable proof procedures. As Gordon remarks [62]:

> Trivial deductions sometimes require elaborate tactics, but on the other hand one never reaches an impasse. HOL experts can prove arbitrarily complicated theorems if they are willing to use sufficient ingenuity. Furthermore, the type discipline ensures that no matter how complicated and *ad hoc* are the tactics, it is impossible to prove an invalid theorem.

Although the HOL system currently lacks much needed support to automate specific kinds of proofs, e.g., simple facts of arithmetic, it is only a matter of time before this support is gradually acquired by the system. Like several other extensible systems such as the UNIX operating system and EMACS text editor (now highly refined and widely used systems), the HOL system is an open system programmed largely in the same language that users use to interact with the system. We think that the following comment by Stallman with regard to EMACS suggests how the HOL system might also grow to maturity:

> User customization helps ... by making the whole user community into a breeding and testing ground for new ideas. Users think of small changes, try them, and give them to other users. If an idea becomes popular, it can be incorporated into the core system. When we poll users on suggested changes, they can respond on the basis of actual experience rather than thought experiments.

### 2.3.6 More Detailed Descriptions

More detailed descriptions of the HOL system may be found in a paper by Gordon [60] and the system documentation [62]. The on-line version of the system documentation includes the HOL sources for several tutorial-style case studies including the TAMARACK-3 correctness proof described in this dissertation.

The HOL system is a descendent of the Edinburgh LCF system which is described in a book by Gordon et al. [54]. The general idea of representing a logic in the ML programming language, the meta-language of both HOL and LCF, is discussed by Gordon in [56]. Many of LCF system features inherited by the HOL system are described in a book by Paulson [115].

## 2.4 Reasons for Mechanized Formal Proof

About a decade ago, some probing questions were asked by De Millo, Lipton and Perlis [41] about the relevance of formal verification. Although these commentators had software verification in mind, their remarks also apply to hardware verification. We are sympathetic with some of their concerns, but obviously disagree with their main conclusion that formal verification does not have a significant role to play in computer science and computer engineering.

One area of concern in their paper which especially interests us is the role of *fully* automated proof (which is an extreme form of mechanized proof):

> It seems to us that the scenario envisioned by the proponents of verification goes something like this: The programmer inserts his 300-line input/output package into the verifier. Several hours later, he returns. There is his 20,000-line verification and the message "VERIFIED".

We think that the key point raised here by De Millo et al. is that fully automated proof (when this is even possible) does nothing to increase confidence in our understanding of what a generated theorem really says or why it is true. This worry is underlined by remarks made by Cohn on the notion of proof in hardware verification in which she stresses the difference between a formal description and a designer's intention: "that one may end up proving properties of a formal description bearing an imperfect relation to the intended design - and possibly never know it" [31]. A possible danger of misplaced confidence in formal verification is the so-called 'Titanic effect' [41] where conventional redundancies and safeguards to handle errors that 'cannot' happen are removed.

We agree wholeheartedly with the view that the answer "VERIFIED" generated automatically without thoughtful human participation is not enough. One of the most concrete reasons for this view is the fact that some purported correctness results turn out to be true theorems but they are true in a meaningless way: the best known class of meaningless results are instances of the 'false implies everything' problem [18].

On the other hand, the answer "VERIFIED FALSE" may be useful even if it is obtained without thoughtful human participation. First of all, we assume that "VERIFIED FALSE" means that a contradiction was found which is a different case than the answer "GIVING UP" or never producing an answer. A negative result almost certainly indicates a specification error, that is, an inconsistency in the formal specification of either the design or its intended behaviour or an inconsistency in the formally stated relationship between them. This may or may not indicate an error in the implementation of the design (e.g., a microchip) since neither the implementation nor the designer's intentions are necessarily consistent with the formal description (as Cohn has remarked from experience with the VIPER project [31]). In either case, it would certainly seem worthwhile investigating why the verification system produced the answer "VERIFIED FALSE" especially if the verification system also returns some hints about why the proof failed.[4] Admittedly, simulation is a much easier way to *possibly* find some errors but formal verification, unlike simulation, is certain to reveal specification errors if they exist.

Another purpose of mechanized formal proof is to support the thoughtful participation of a human verifier in a formal proof. Here, the user takes the initiative and the proof process itself is the main object of interest; the eventual outcome, "VERIFIED" or "VERIFIED FALSE", is not necessarily as significant. In this view, the human verifier, who might indeed also be the designer, has to construct an argument for why the design is correct. The use of mechanically-checked formal proof ensures that his or her reasoning is absolutely correct to the extent that the formal descriptions match physical

[4]The Boyer-Moore theorem-prover [13] is an example of a verification system which produces proof readable summaries. Cohn is currently working on a facility for the HOL system which automatically generates proof summaries that may play a similar role in assisting a human verifier to debug a proof session [32].

reality and intention. Clearly, some automation, in fact, a great deal of automation is needed to allow the user to guide proof-generation at a level which supports clear thinking without being side-tracked by tedious proofs of relatively simple facts.

There are several reasons to believe that the primary role of mechanized proof in formal verification is to support thoughtful human participation in the proof process. These reasons include:

- A significant amount of human participation is necessary for non-trivial verification problems; to be extensible, this should be a well-conceived part of the methodology rather than provisions for sporadic intervention.
- An experienced human verifier is likely to recognize the tell-tale signs that a purported correctness result is actually meaningless.
- Insights gained from formal verification may actually contribute usefully to the design process itself in a verification-driven approach to design.

Lastly, there is the view that the answer "VERIFIED", or more importantly, the proven theorem is a significant result after all. But we argue, as Cohn [31] has also argued, that a correctness result is only meaningful if its scope and limitations are clearly understood. We think that one of the best ways to gain an appreciation of what has and what has not been verified is by thoughtful participation in the verification process.

Indeed, many other serious proponents of formal verification have expressed similar views. Among others, Rushby [123] has emphasized the role of a verification system as an "implacably skeptical colleague". Also speaking from substantial experience, Shankar [129] observes:

> The utility of proof-checkers is in clarifying proofs rather than in validating assertions. The commonly held view of proof-checkers is that they do more of the latter than the former. In fact, very little of the time spent with a proof-checker is actually spent proving theorems. Much of it goes into finding counterexamples, correcting mistakes, and refining arguments, definitions, or statements of correctness. A useful automatic proof-checker plays the role of a devil's advocate for this purpose.

Chapter 3

# A Simple Microprocessor

This chapter describes the operation and design of TAMARACK-3 in a conventional style of microprocessor description. It is divided into three main sections: programming level model, memory interface and internal architecture.

The only significant difference (in style) from a conventional description is the use of uninterpreted data types and uninterpreted primitives in place of defined data types and defined symbols.

We also describe how the internal architecture can be viewed as a series of increasingly concrete interpretation levels. This hierarchy of interpretation levels will be important in subsequent chapters as the basis of a proof strategy for verifying the design of TAMARACK-3.

## 3.1 Programming Level Model

The programming level model, or external architecture, of TAMARACK-3 is a description of its operation as seen by a programmer. This model hides all aspects of the internal architecture which the programmer does not need to know about when writing programs for this microprocessor.

The programming level model can be viewed as an interpreter for manipulating a set of variables which corresponds to the externally visible state of the microprocessor. It consists of four main parts:

- Basic data types and primitive operations.
- Variables manipulated by the interpreter.
- Format of instructions.
- Instruction semantics.

Our presentation of the programming level model is organized around these four main parts. Although we describe hardware interrupts separately from the semantics of ordinary program instructions, hardware interrupts in TAMARACK-3 can be regarded as just another kind of instruction in the programming level model.

### 3.1.1 Basic Data Types and Primitive Operations

We begin with the basic data types and primitive operations used in the programming level model. The data type `:bool` is used to represent voltage values or logical conditions. The data type `:num` is used when some lower level form of data is interpreted

as the representation of a natural number. The remaining data types correspond to machine words, groups of bits within a machine word, and memory states. A complete list of basic data types used in the programming level model is shown below.

| | |
|---|---|
| `:bool` | - Boolean values {T,F} |
| `:num` | - natural numbers {0,1,2,...} |
| `:*wordn` | - full-size machine words |
| `:*word3` | - instruction opcodes |
| `:*address` | - memory addresses |
| `:*memory` | - memory states |

A conventional description would typically be very precise about details such as the number of bits in a machine word and the size of memory. However, we avoid specifying these details by regarding `:*wordn`, `:*word3`, `:*address` and `:*memory` as *uninterpreted types.* The actual representation of these basic data types may be thought of as implementation dependent details. We use the prefix * to distinguish these as uninterpreted types.[1]

Functional elements such as the ALU (*Arithmetic Logic Unit*) at the lowest level of architectural description perform various operations on data. It is also possible and desirable to avoid specifying any details about these operations. Instead, these operations are regarded as *uninterpreted primitives.* Primitive operations used to describe the design and operation of TAMARACK-3 are listed below along with an informal description of their types. Although we use the syntax of the HOL logic to describe these types, a slightly different set of types is used in the formal theory for reasons explained later in Chapter 4.

| | | |
|---|---|---|
| `iszero` | `:*wordn→bool` | test if zero |
| `inc` | `:*wordn→*wordn` | increment |
| `add` | `:(*wordn×*wordn)→*wordn` | addition |
| `sub` | `:(*wordn×*wordn)→*wordn` | subtraction |
| `wordn` | `:num→*wordn` | representation of a number |
| `valn` | `:*wordn→num` | value of a full-size word |
| `opcode` | `:*wordn→*word3` | extract opcode field |
| `val3` | `:*word3→num` | value of an opcode |
| `address` | `:*wordn→*address` | extract address field |
| `fetch` | `:(*memory×*address)→*wordn` | read memory |
| `store` | `:(*memory×*address×*wordn)→*memory` | write memory |

The above list also gives a suggested interpretation for each of the uninterpreted primitives. Although we avoid specific details about the operations denoted by these uninterpreted primitives, we sometimes relax our presentation style by referring to an uninterpreted primitive in terms of its suggested interpretation.

The use of uninterpreted data types and uninterpreted primitives is an important concept in this dissertation. As mentioned earlier in Chapter 1, they are key mechanisms for creating generic specifications with the aim of filtering out non-essential detail from

[1]Our use of the prefix * can be taken as a hint in this informal description that uninterpreted types will be represented by type variables in the HOL logic.

a formal description. Furthermore, by not specifying any details about basic data types or primitive operations on data, we have an informal description which is generalized over a whole range of possible word and memory sizes and possible interpretations for the primitive operations.

Parenthetically, we note that the term 'interpretation' has several different meanings in this dissertation. For example, we will eventually describe how programming level operations are 'interpreted' by sequences of microinstructions at the microprogramming level; this idea of an interpretation is standard from conventional concepts about structured computer organization. On the other hand, we also use the term 'interpretation' for the informal concept of assigning a set of values to a type variable or assigning a value to a term variable. Uninterpreted types and uninterpreted primitives can be thought of as variables that stand for "for any" type and "for any" operation respectively.

### 3.1.2 Externally Visible State

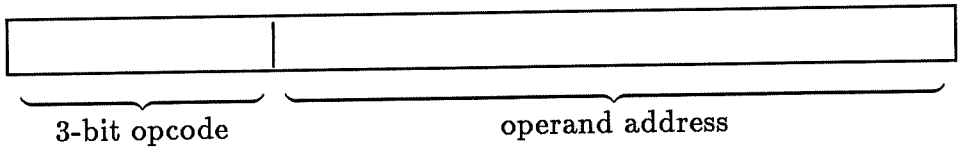
The set of variables manipulated by the programming level model corresponds to the externally visible state of the microprocessor. In TAMARACK-3, these variables are:

| | |
|---|---|
| `mem` | - memory |
| `pc` | - program counter |
| `acc` | - accumulator |
| `rtn` | - return address register |
| `iack` | - interrupt acknowledge flag |

The memory stores memory states, represented by the data type `:*memory`. Each of the registers stores full-size memory words, represented by the data type `:*wordn`. The interrupt acknowledge flag is stored internally by a flipflop whose value belongs to the data type `:bool`.

### 3.1.3 Instruction Word Format

Instructions are exactly one full-size machine word. Although specific details about word size and instruction word format are not given in this description, we can assume that the instruction word consists of a 3-bit opcode (since there are eight different instructions) with the remaining bits used as an operand address. The operand address is the absolute address of a memory word which may be used as the address of either data or an instruction.




Opcodes and operand addresses are represented by the uninterpreted types `:*word3` and `:*address`. They are extracted from an instruction word by the uninterpreted primitives `opcode` and `address`.

| Instruction | Opcode Value | Effect |
|---|---|---|
| JZR | 0 | jump if zero |
| JMP | 1 | jump |
| ADD | 2 | add accumulator |
| SUB | 3 | subtract accumulator |
| LDA | 4 | load accumulator |
| STA | 5 | store accumulator |
| RFI | 6 | return from interrupt |
| NOP | 7 | no operation |

Table 3.1: TAMARACK-3 Instruction Set

### 3.1.4 Instruction Set Semantics

The eight TAMARACK-3 programming level instructions are in Table 3.1. Their opcode values and a brief explanation of each instruction are also given in the table. The opcode is extracted from the current instruction word by `opcode` and its numerical value is then obtained by applying `val3` to the extracted opcode.

The four data processing instructions, ADD, SUB, LDA and STA, involve both the accumulator `acc` and memory `mem`. The other four instructions, JZR, JMP, RFI and NOP, are control instructions with no effect on either the accumulator or memory. The only conditional branch, JZR, tests whether the accumulator `acc` contains the machine representation of zero.

The address of the current instruction is always given by the program counter pc at the beginning of each instruction cycle. Operationally, the program counter pc is a full-size register but only the address field of this register is used to access the current instruction word from memory. When a jump is taken as a result of either a JMP or JZR instruction, the entire instruction word is loaded into the program counter pc but only the address field has any significance.

The eight programming level instructions are described below. Only changes to the current state of the external architecture are described. Unaffected components of the externally visible state are not mentioned. Some of these descriptions are simplified by using the abbreviations,

```
inst = fetch (mem,(address pc))
operand = fetch (mem,(address inst))
```

for the current instruction word and the operand addressed by this instruction. The informal notation,

<destination> ← <expression>

is used to denote when a value computed from the current machine state is loaded into a register, flipflop or memory to form a component of the next machine state.

### JZR - jump if zero

```
pc ← if iszero acc then inst else inc pc
```

If the result of applying `iszero` to the current contents of the accumulator `acc` is T, then the current instruction word is loaded into the program counter `pc`. Otherwise, the instruction is completed by incrementing the program counter `pc`.[2]

### JMP - jump

```
pc ← inst
```

The current instruction word is unconditionally loaded into the program counter `pc`.

### ADD - add accumulator

```
acc ← add (acc,operand)
pc ← inc pc
```

The `add` operation is applied to the current contents of the accumulator `acc` and the memory word addressed by the operand address field of the current instruction. The result is loaded into the accumulator `acc`. The instruction is completed by incrementing the program counter `pc`.

### SUB - subtract accumulator

```
acc ← sub (acc,operand)
pc ← inc pc
```

The `sub` operation is applied to the current contents of the accumulator `acc` and the memory word addressed by the operand address field of the current instruction. The result is loaded into the accumulator `acc`. The instruction is completed by incrementing the program counter `pc`.

### LDA - load accumulator

```
acc ← operand
pc ← inc pc
```

The memory word addressed by the operand address field of the current instruction is loaded into the accumulator `acc`. The instruction is completed by incrementing the program counter `pc`.

[2]This is an instance of when we have relaxed our presentation style by referring to an uninterpreted primitive, namely, `inc`, in terms of its suggested interpretation.

**STA - store accumulator**

```
mem ← store (mem,address inst,acc)
pc ← inc pc
```

The current contents of the accumulator acc are stored in external memory at the location specified by the operand address field of the current instruction. The instruction is completed by incrementing the program counter pc.

**RFI - return from interrupt**

```
pc ← rtn
iack ← F
```

The current contents of the return address register rtn are loaded into the program counter pc and the interrupt acknowledge flag iack is reset to F. This instruction does not check whether the interrupt acknowledge flag iack is currently set.

**NOP - no operation**

```
pc ← inc pc
```

Skip to the next instruction by incrementing the program counter pc.

### 3.1.5 Hardware Interrupts

In the normal flow of program execution, instructions are sequentially executed according to the semantics given in the previous section. The only kind of hardware exception is a single level, non-vectored, non-maskable hardware interrupt which is generated by setting the interrupt request pin irq to T.

Hardware interrupts can be regarded as just another kind of instruction in the TAMARACK-3 programming level model. This is because the current instruction cycle is completed before an interrupt request is allowed to interrupt the normal flow of program execution. Instructions are also indivisible (with respect to external interrupt requests) in many commercially-available microprocessors such as the MC68000 [27].

Normally, the interrupt will be detected within a few clock cycles but this may be delayed for an arbitrary number of clock cycles when the microprocessor is operating in either fully asynchronous mode or extended cycle mode. Because only a single level of interrupt is supported, the value of the interrupt request pin irq will be ignored if the interrupt acknowledge flag is already T indicating that a previous interrupt is still being serviced.

```
if iack = F then
    pc ← 0
    rtn ← pc
    iack ← T
```

The interrupt request is processed by saving the current value of the program counter pc in the return address register rtn, setting the interrupt acknowledge flag iack to T and loading the machine representation of zero into the program counter pc. The interrupt service routine is assumed to begin at location zero in memory.

At the end of the interrupt service routine, a return-from-interrupt instruction RFI is executed causing the saved return address stored in rtn to be loaded into the program counter pc and the interrupt acknowledge flag to be reset to F.

## 3.2 Memory Interface

The microprocessor can be interfaced to external memory to operate in one of three possible modes: fully synchronous, fully asynchronous, or extended cycle mode. The mode of operation is selected by the input pins dack and idle. The dack pin is used as a handshaking signal in fully asynchronous mode and extended cycle mode. In extended cycle mode, the idle pin is used in place of a handshaking signal to indicate when the external memory is idle and ready to begin another interaction.

Although a bi-directional bus would typically be used to transfer data between a microprocessor and external memory, the design of TAMARACK-3 uses two separate uni-directional busses, datain and dataout.[3] Data is sent to external memory on the dataout bus and received from external memory on the datain bus. Memory addresses are sent to external memory on the addr bus.

The operations performed by external memory are denoted by the uninterpreted primitives fetch and store. Although the synchronization details depend on the memory mode, the result of a read request is described by the equation,

```
datain = fetch (mem,addr)
```

and the result of a write request is described by the following update to the internal state of memory.

```
mem ← store (mem,addr,dataout)
```

Synchronization details for each of the three memory modes are described below.

### 3.2.1 Fully Synchronous Mode

In fully synchronous mode, every memory interaction is completed in a single cycle. The microprocessor is made to operate in this mode by wiring both of the pins idle and dack to T. A detailed timing analysis is needed to ensure that the external memory can always satisfy memory requests within a single clock cycle.

Figure 3.1 shows the interconnections between external memory and TAMARACK-3 when operating in fully synchronous mode. The memory request pin dreq is not needed in this mode because every clock cycle is assumed to be either a read or write request. The type of request is indicated by the wmem pin which is normally reset to F except when writing to memory.

[3] A single bi-directional bus would be more realistic than two uni-directional busses but for simplicity we have chosen to avoid modelling bi-directional behaviour in this part of the design. However, a bi-directional model of bus operation is used for the system bus inside the TAMARACK-3 datapath.

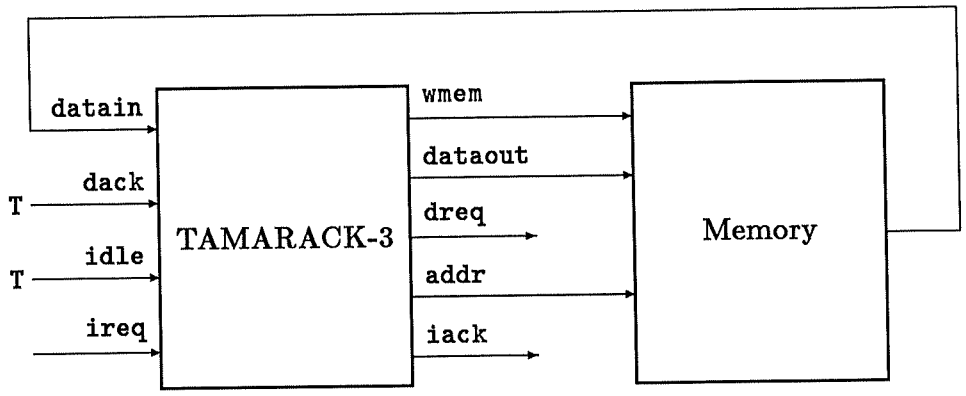



Figure 3.1: Fully Synchronous Operation.

### 3.2.2 Fully Asynchronous Mode

Fully asynchronous interaction with external memory is achieved when no assumptions are made about the speed of the external memory relative to the microprocessor clock speed. This allows the microprocessor to be interfaced to a mixture of fast and slow devices in the address space of external memory. The transfer of data (and memory addresses) between the microprocessor and external memory is synchronized by handshaking signals following the four-phase bundled data convention illustrated in Figure 3.2. It is only assumed that wire delays between the microprocessor and external memory are approximately uniform.

Figure 3.3 shows the interconnections between external memory and TAMARACK-3 when operating in fully asynchronous mode. In this mode, the `idle` pin is permanently wired to `F`. The acknowledgement signal `dack` is generated by the external memory (or by peripheral devices in the case of memory-mapped I/O).

A memory request is signaled by setting the memory request pin `dreq` to `T`. The type of request is indicated by the `wmem` pin which is normally reset to `F` except when writing to memory. After signaling a memory request, `dreq` must remain `T` and the `wmem` flag, address bus `addr` and dataout bus `dataout` must remain at stable values until `dack` becomes `T` signaling that the request has been satisfied. In the case of a read request, incoming data from the external memory will be stable from the instant when the acknowledgement signal `dack` becomes `T` until the request signal `dreq` returns to its original value of `F`. Finally, the microprocessor waits for `dack` to also return to `F` before starting another memory request.

The use of handshaking signals to synchronize data transfers between the microprocessor and external memory requires very little extra circuitry and no additional control states. Best case performance by the external memory will result in exactly the same number of clock cycles as fully synchronous mode.

### 3.2.3 Extended Cycle Mode

Extended cycle mode can be viewed as a compromise between the real-time constraints of fully synchronous mode and the delay-insensitive nature of fully asynchronous mode.

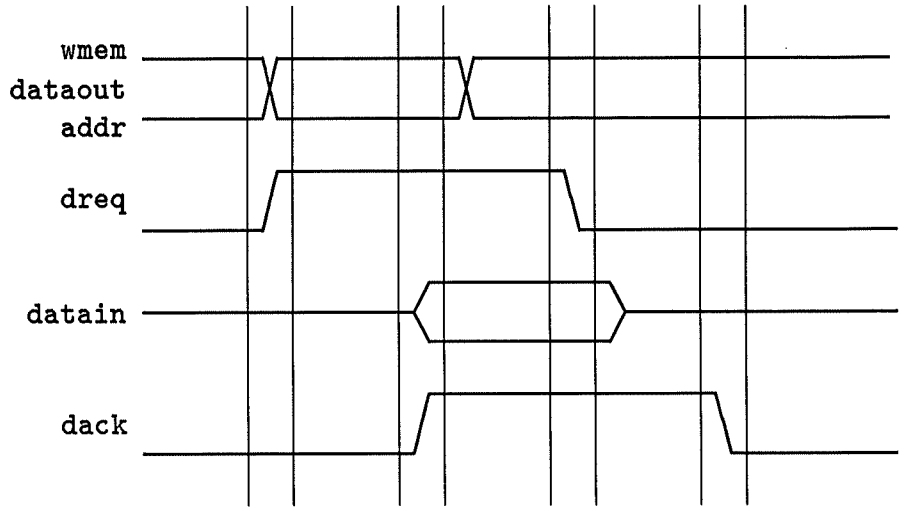



Figure 3.2: Synchronizing Data Transfer with Handshaking Signals.

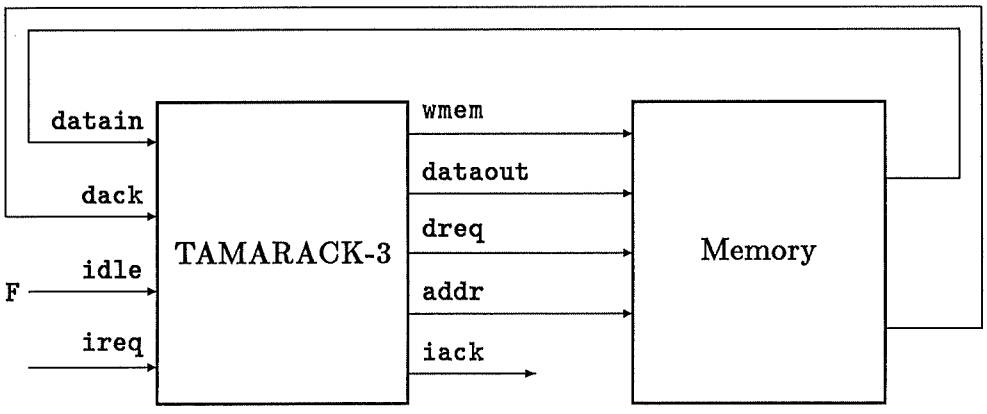



Figure 3.3: Fully Asynchronous Operation.

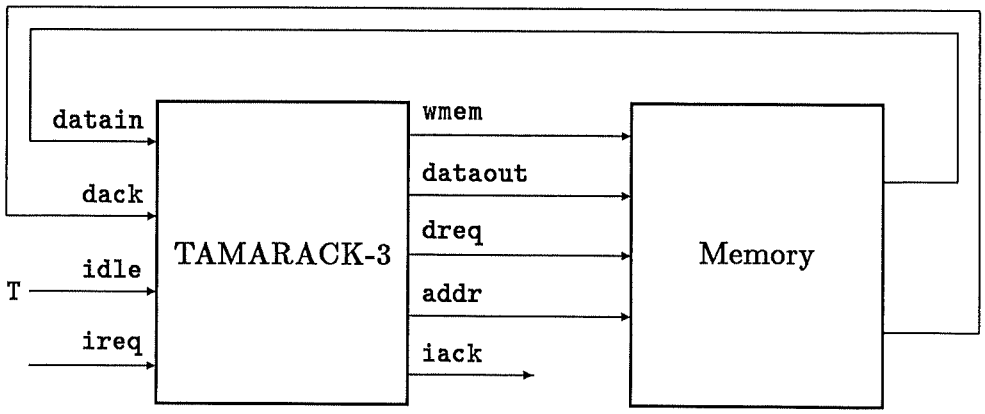



Figure 3.4: Extended Cycle Operation.

In TAMARACK-3, this mode is selected by permanently wiring the `idle` pin to `T` as shown in Figure 3.4. As before, the acknowledgement signal `dack` is generated by external memory.

The only difference concerns the completion of the memory cycle after the request signal `dreq` has been reset to `F`. Unlike fully asynchronous mode, the control logic does not force the microprocessor to wait until the acknowledgement signal `dack` also returns to `F` before starting another memory request. Instead, it is assumed that the external memory will always complete the current memory cycle in time for another request to begin as early as the next clock cycle. This is a significant difference from fully asynchronous mode because the speed of external memory is no longer independent of the microprocessor clock speed.

Many commercially-available microprocessors feature a memory mode which is similar to this mixture of handshaking signals and real-time constraints. For example, the VIPER microprocessor has a memory mode which allows the acknowledgement of a memory request to be delayed multiple clock cycles, but once the memory request terminates, it is assumed that another memory request can begin after 200 nanoseconds [120].

Extended cycle mode is provided in TAMARACK-3 simply because it was easy to implement with a trivial addition to the hardware. However, we ignore extended cycle mode in the rest of our informal description and in the formal proof of correctness since, in general, our research has not focussed on the use of formal methods at detailed level timing levels involving real-time constraints. Related work by Hanna and Dache [67], by Herbert [72,74] and by Leeser [93] suggest techniques for reasoning about real-time constraints at hardware levels.

## 3.3 Internal Architecture

This section begins with a structural view of the internal architecture and an overview of how programming level instructions are interpreted by the hardware. This is followed by a more detailed view of the internal architecture as a series of increasingly concrete

interpretation levels. Finally, we outline some bottom level assumptions which bridge the gap between our most detailed level of description and actual hardware.

### 3.3.1 Register-Transfer Level Structure

Figure 3.5 shows a structural view of the TAMARACK-3 register-transfer level architecture. It consists of two main parts: a microcoded control unit and a single-bus datapath.

The control unit is implemented by the microcode program counter `mpc`, a ROM (*Read Only Memory*) for storing microcode, a decoder which separates the ROM output into various microinstruction fields, and combinational logic for computing the address of the next microinstruction.

The datapath includes the program counter `pc`, accumulator `acc`, return address register `rtn`, and interrupt acknowledge flag `iack` which are components of the externally visible state at the programming level. In addition to these, several internal registers are needed to interpret programming level instructions. These additional, full-size word registers are:

| | |
|---|---|
| `mar` | - memory address register |
| `ir` | - instruction word register |
| `arg` | - argument register for ALU input |
| `buf` | - buffer for ALU output |

The datapath also includes several functional elements:

| | |
|---|---|
| `alu` | - four functions: `add`, `sub`, `inc` and outputting zero (a constant) |
| `interface` | - switching between system bus and memory data pins |
| `opc` | - implements `opcode` for extracting opcode field |
| `addr` | - implements `address` for extracting address field |
| `zeroflag` | - implements `iszero` to test for zero |
| `dreq` | - two-input OR-gate |

The storage devices and functional elements of the datapath are interconnected by a single system bus. The width of the system bus is exactly one full-size machine word. The datapath is controlled by signals from the control unit which, in turn, receives feedback from the datapath.

### 3.3.2 Overview of Instruction Interpretation

Each TAMARACK-3 instruction is executed by a sequence of steps which varies depending on the particular instruction, the machine state and external inputs. In general, the following actions are taken by the internal architecture to interpret each programming level instruction.

- Check for interrupt request, otherwise continue ...
- Fetch instruction addressed by program counter `pc`.
- Decode instruction.

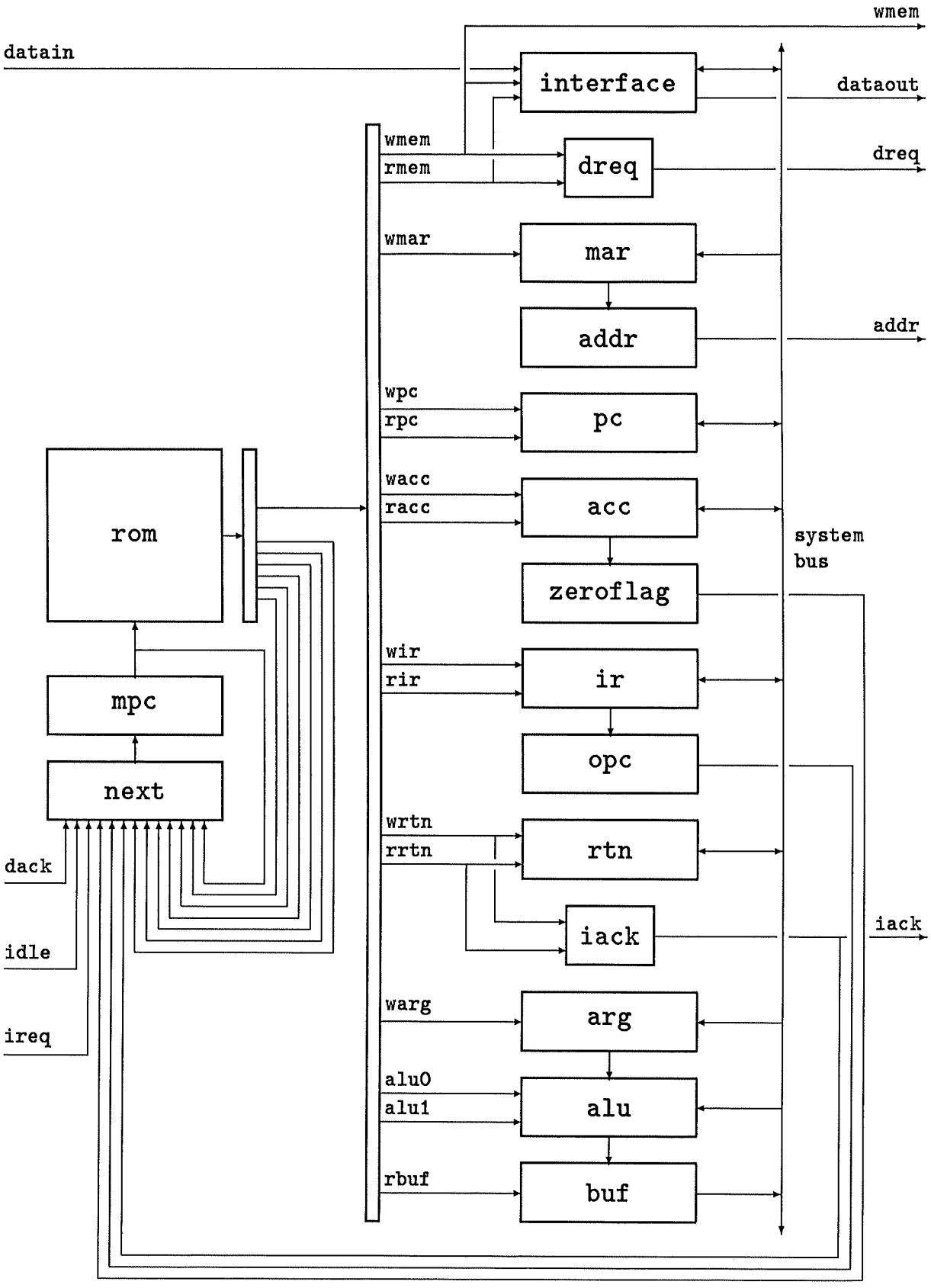



Figure 3.5: Register-Transfer Level Architecture.

- Fetch operand if needed.
- Execute instruction.
- Increment program counter pc if necessary.

Not all of these actions are performed for each instruction. In particular, the control instructions JZR, JMP, RFI and NOP do not require an operand to be fetched from memory. Similarly, the program counter does not need to be incremented for the JMP and RFI instructions or for the JZR instruction when the jump is taken. There are several opportunities for overlapping some of these steps (e.g. incrementing the program counter while executing the current instruction) but this has not been done for the current design of TAMARACK-3.

The interpretation of an ADD instruction illustrates with greater detail how the internal architecture of TAMARACK-3 is used to implement its instruction set. An ADD instruction is interpreted by a sequence of data transfers over the system bus, interactions with memory, and operations on data performed by the various functional elements. The informal notation,

<destination> ← <expression>

is used here to denote when a value computed from the current machine state is loaded into a register, flipflop or memory to form a component of the next machine state. But unlike its previous use to describe state changes in the programming level model between instruction cycles, this informal notation now describes state changes at the register-transfer level between clock cycles.

Exactly eight steps are required to interpret the ADD instruction. These eight steps are executed in the sequential order shown below; however, some of these steps might be repeated (i.e., they are repeat-loops) when the microprocessor is operating in fully asynchronous mode.[4]

```
mar ← pc                                              repeat if ¬(idle or ¬dack)
ir ← fetch (mem,(address mar))                        repeat if ¬dack
mar ← ir                                              repeat if ¬(idle or ¬dack)
arg ← acc                                             repeat if ¬(idle or ¬dack)
buf ← add (arg,fetch (mem,(address mar)))             repeat if ¬dack
acc ← buf
buf ← inc pc
pc ← buf
```

The equation,

```
clock cycles = 8+n1+n2+n3+n4+n5
```

[4]The `fetch` operation performed by external memory is not necessarily completed until the last iteration of the repeat-loop.

gives the total number of clock cycles needed to interpret an ADD instruction. Since every step is executed at least once, at least eight clock cycles are required. The variables n1, n2, n3, n4 and n5 denote the number of additional clock cycles spent waiting for the external memory at various steps in the instruction cycle (due to the repeat-loops shown above).

In fully synchronous mode, none of the eight steps are ever repeated because the pins idle and dack are both wired to T. In this case, the variables n1, n2, n3, n4 and n5 will all be equal to zero. Hence, the instruction cycle will be completed in exactly eight clock cycles.

In fully asynchronous mode, some of the above steps may be repeated but this will have no untoward effect except to increase the number of clock cycles. In this case when the microprocessor is using handshaking signals to interact with memory, the variables n1, n2, n3, n4 and n5 will depend on the latency of external memory. For best case performance, these variables will all be zero resulting in exactly the same number of clock cycles as fully synchronous mode. More generally, it is only known that each wait loop will eventually terminate. This fact depends on the correct implementation of the handshaking protocol by both the microprocessor and external memory; establishing this fact is a major step in the TAMARACK-3 proof of correctness.

In either mode of operation, the cumulative effect of the ADD instruction interpretation sequence is described by the following updates to the accumulator acc and the program counter pc.

```
acc ← add (acc,fetch (mem,(address fetch (mem,(address pc)))))
pc ← inc pc
```

Simplifying these expressions with the earlier mentioned abbreviations inst and operand, yields the programming level description of the ADD instruction semantics given earlier in Section 3.1.4.

```
acc ← add (acc,operand)
pc ← inc pc
```

The interpretation of the other seven programming level instructions and the processing of a hardware interrupt can be described in a similar way by a sequence of steps. Showing by formal proof for each programming level instruction that the cumulative effect of each sequence satisfies the semantics of that particular instruction is another major part of the TAMARACK-3 proof of correctness.

### 3.3.3 Multiple Interpretation Levels

The programming level model of a microprocessor sits at the top of a hierarchy of interpretation levels implemented by the internal architecture. The internal operation of a microprocessor can generally be described in terms of the following levels [2].

| | |
|---|---|
| Programming level | - sequential execution of user programs |
| Microprogramming level | - sequential execution of microcode |
| Phase level | - concurrent elementary hardware operations |
| Instant level | - asynchronous sequencing in a clock phase |
| Basic logic components | - circuit level behaviour |

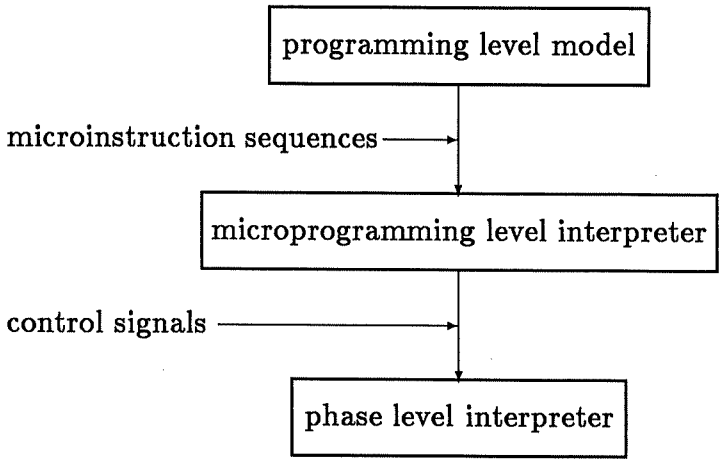



Figure 3.6: Hierarchy of Interpretation Levels.

This description of TAMARACK-3 focuses on the three highest levels of interpretation (i.e., only down to the phase level). The programming level has already been described in terms of the instruction set semantics and the hardware interrupt facility. Below the programming level, increasingly concrete views of the internal operation of TAMARACK-3 are described at the microprogramming level and at the phase level.

At the microprogramming level, a programming level instruction is interpreted by executing a sequence of microinstructions. This sequence of microinstructions is generated by a FSM (*Finite State Machine*) implemented by the control unit. Microinstructions are executed by an operational part corresponding to the datapath.

The phase level description decomposes the interpretation of a single microinstruction into the parallel execution of a set of elementary operations. This decomposition reveals the structural organization of the internal architecture in terms of register-transfer level components.

The concept of multiple interpretation levels is used by architects to achieve a "progressive translation of functions in several stages" [2]. We will later describe how this concept also provides a very effective strategy for controlling proof complexity in the formal verification of TAMARACK-3.

#### 3.3.3.1 Microprogramming Level

Every programming level instruction is interpreted by a different sequence of actions even though they share many individual steps in common. This sequence partially depends on the instruction opcode which is only known part way through the sequence after the instruction word has been fetched from memory. The sequence of steps taken may also depend on the machine state, in particular, on the contents of the accumulator acc in the case of a jump-if-zero JZR instruction.

The interpretation of each instruction, that is, the sequence of actions taken for each instruction, and similarly, the actions taken to process a hardware interrupt are determined by the control unit. The control unit FSM generates datapath commands (represented by microinstructions) each clock cycle. Each command causes an action

to be performed by the datapath during the current clock cycle.

The interpretation algorithm implemented by the control unit FSM is based on *conditional branches*, that is, a Moore machine approach in contrast to a Mealy machine approach based on *conditional instructions*. Inputs are used to select the next machine state but do not determine the current output of the state machine. Some of these inputs consist of feedback from the datapath, in particular, the opcode field of the instruction word register `ir` contents, a test-accumulator-for-zero flag `zeroflag`, and the current value of the interrupt acknowledge flag `iack`. The FSM also receives external inputs from the `idle`, `dack` and `ireq` pins which determine the behaviour of the FSM. The FSM for the TAMARACK-3 control unit is described by the flow graph in Figure 3.7.

The start of an instruction cycle occurs when the FSM is in state 0 and about to exit to either state 1 or state 2. If an interrupt is requested and the `iack` flag is not already set, then the FSM exits to state 1 to process the interrupt. Otherwise, the FSM exits to state 2 and causes the current instruction word to be fetched from memory. In state 3, the FSM dispatches on the opcode field to the start of the remaining interpretation sequence for the current instruction. For instance, the interpretation of an ADD instruction would cause a transition from state 3 to state 6. From this point onwards, the FSM follows a sequence of state transitions leading back to state 0. In the case of a JZR instruction, the FSM selects one of two possible exits from state 4 depending on the test-accumulator-for-zero feedback from the datapath. Assuming that the FSM never loops indefinitely in a particular state, then in all cases, including the processing of interrupts, the FSM always returns to state 0 to begin the next instruction cycle.

For example, the interpretation of an ADD instruction in fully synchronous mode results in the following sequence of states.

0, 2, 3, 6, 13, 15, 11, 12, and back to 0.

In fully asynchronous mode, additional clock cycles caused by delayed handshaking signals from the external memory could result in a sequence such as:

0, 2, 2, 2, 3, 3, 6, 13, 13, 13, 15, 11, 12, and back to 0.

Each FSM state causes a specific action to be performed by the datapath. The mapping from FSM states to actions is shown in Figure 3.8. This mapping, combined with the flow graph in Figure 3.7, gives a complete description of the internal architecture of TAMARACK-3 at one level of abstraction. This level of description contains no structural details aside from the conceptual distinction between the function of the control unit and the operation of the datapath. Although the components of the internal state are visible in this view, updates to the machine state are described functionally. This abstract view of the internal architecture is the basis of an intermediate step in the formal verification of TAMARACK-3.

#### 3.3.3.2 Phase Level

Each command generated by the FSM is a microinstruction which is interpreted by a set of register-transfer level operations. The phase level description decomposes a microprogramming level action such as,

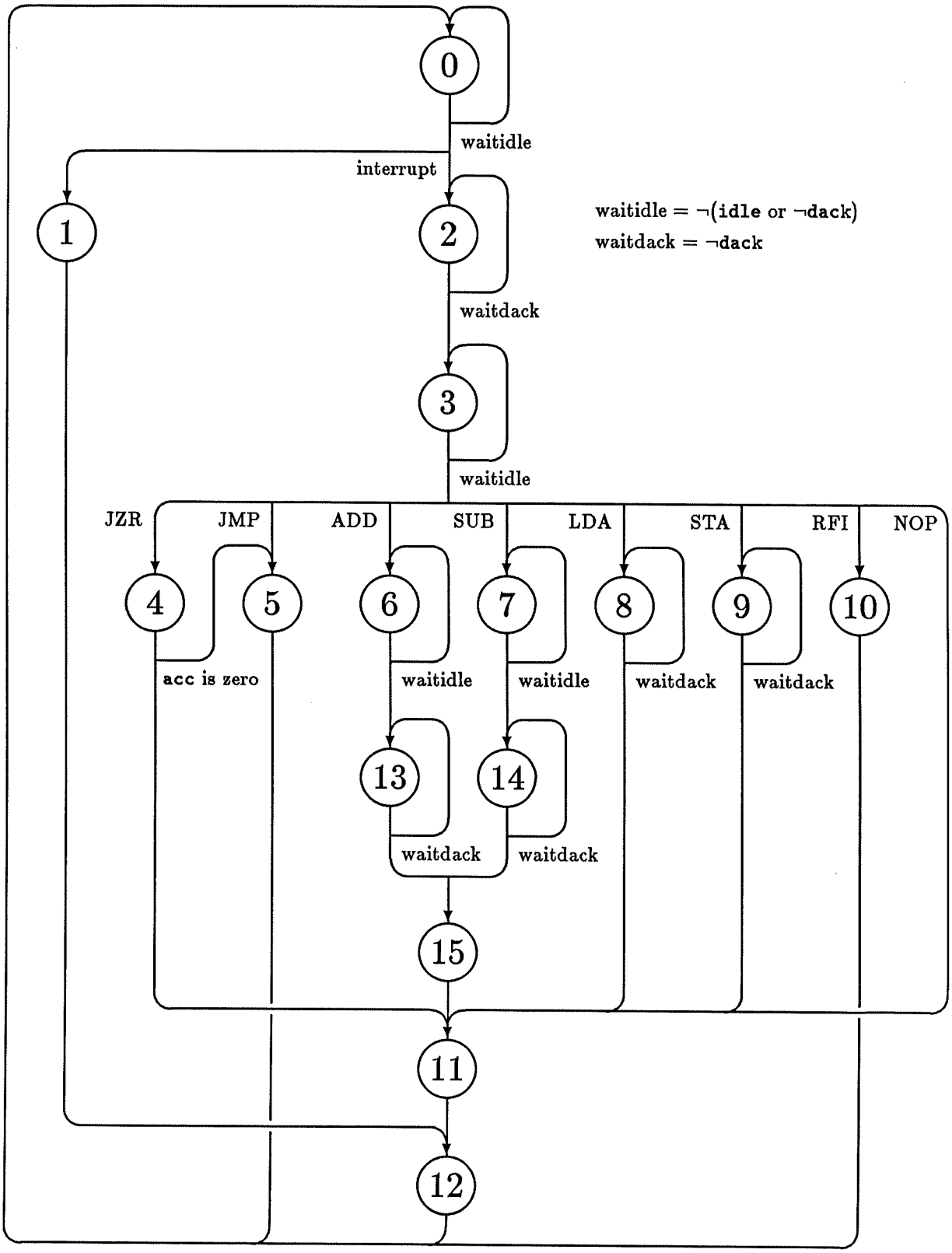
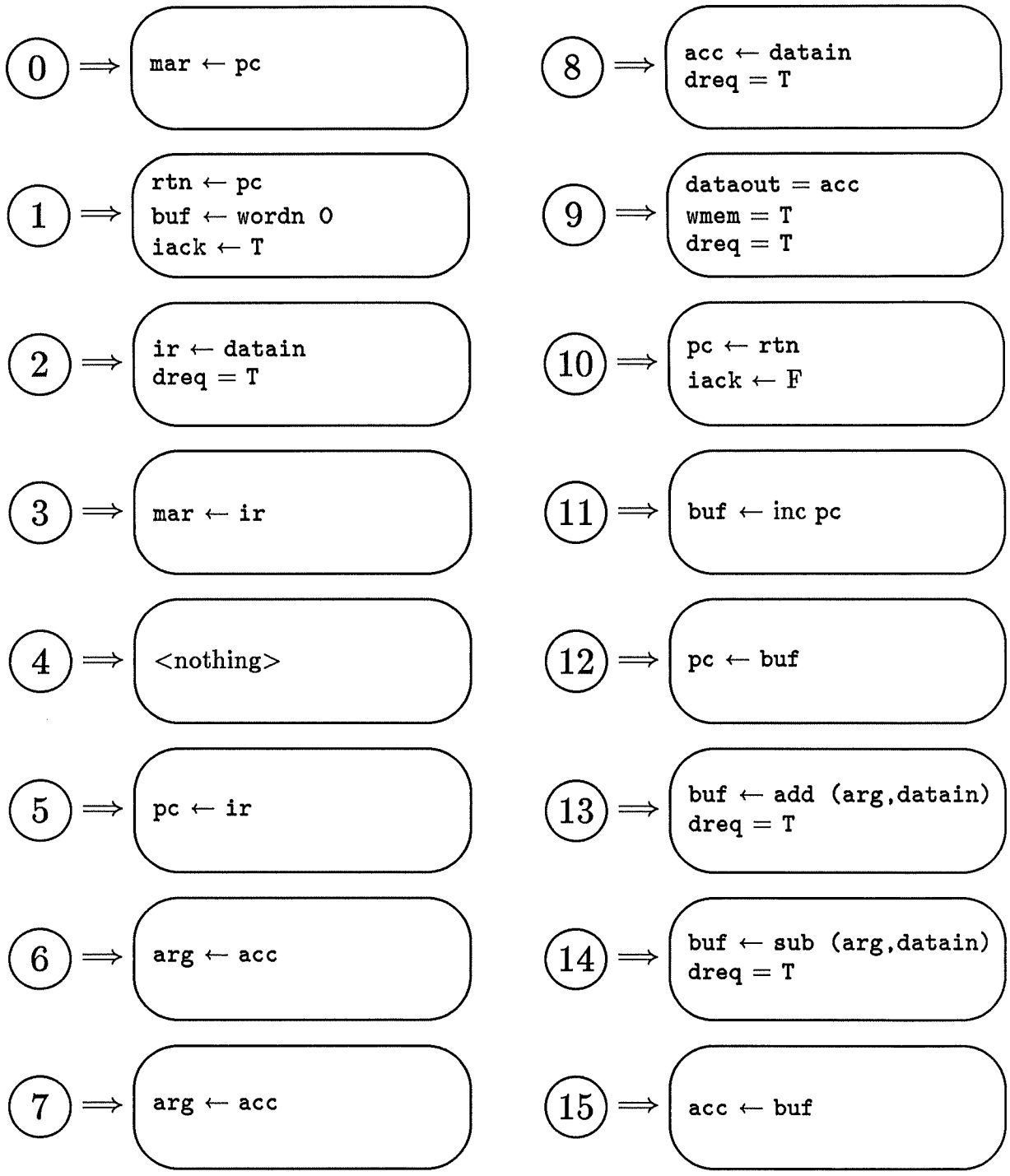



Figure 3.7: Control Unit Finite-State Machine Flow Diagram.




wmem = F
dreq = F
dataout = wordn 0
addr = address mar

} in all states except where indicated otherwise

Figure 3.8: Mapping from FSM States to Datapath Actions.

```
buf ← add (arg,fetch (mem,(address mar)))
```

into the parallel execution of a set of elementary operations (described by equations) followed by a single update to buf:

```
addr = address mar
datain = fetch (mem,addr)
bus = datain
alu = add (arg,bus)
buf ← alu
```

This decomposition reveals the structural organization of the microprocessor. Each of the elementary operations shown above corresponds to a functional element of the register-transfer architecture or, in the case of `fetch`, to a primitive operation performed by external memory.

The phase level description of TAMARACK-3 also refines certain abstract features of the microprogramming level description. In particular, it reveals that the FSM is implemented by:

- A microcode program counter `mpc` to hold the current state of the FSM.
- The microcode ROM which encodes the abstract mapping from FSM states to actions given in Figure 3.8.
- A decoder which separates the ROM output into various microinstruction fields.
- The next address logic which computes the next FSM state according to the flow graph given in Figure 3.7.

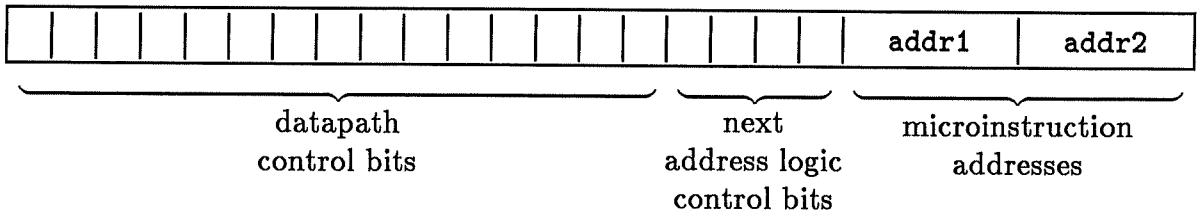
The microcode program counter is a register which stores values belonging to an uninterpreted data type:

```
:*word4   - FSM state
```

Several uninterpreted primitives for this data type are needed later on to describe the computation performed by the next address logic.

| | | |
|---|---|---|
| `word4` | `:num→*word4` | representation of a number |
| `val4` | `:*word4→num` | numerical value of an FSM state |

Every clock cycle, the current value of the microcode program counter `mpc` is used to fetch the current microinstruction from the microcode ROM. The microcode is *horizontal* which means, in this case, that every control signal is determined by a unique bit in the microinstruction word. The format of microinstruction words is shown below.



The fifteen datapath control bits are extracted from the output of the microcode ROM and sent to the datapath over a set of control lines. These control bits generate individual control signals including 'write' signals for storage elements, 'read' signals for devices which can assert values onto the system bus, and signals to control the operation of the functional elements such as the ALU.

In addition to control bits for the datapath, the microinstruction word contains several other fields which are used to encode a partial computation of the next FSM state. The four next address logic control bits determine how the next address logic computes the next FSM state. The two microinstruction address fields, `addr1` and `addr2` (both of the type `:*word4`), specify destination fields for various kinds of conditional branches in the microcode which may be selected by the next address logic as the next FSM state. The next address logic is simply a block of combinational logic whose function can be derived from the flow graph in Figure 3.7.

The interpretation of a microinstruction at the phase level during a single clock cycle results in a sequence of events which include:

- Fetch the current microinstruction.
- Compute the address of the next microinstruction.
- Read data onto the system bus.
- Evaluate operations performed by functional elements.
- Update storage elements such as memory, registers and flipflops.

Some of these events clearly precede other events. For instance, the current microinstruction has to be fetched from the microcode ROM before it can be interpreted. On the other hand, many of these events take place concurrently: for example, the address of the next microinstruction is computed by the next address logic while the datapath executes the actions specified by the current microinstruction. Indeed, much of the activity during a clock cycle is not necessarily synchronized by an explicit control mechanism: a change in the inputs to a functional element such as the ALU might be propagated to its outputs after a few nanoseconds depending on its implementation.

Even though some constraints on the order of events during a clock cycle could be described, this has not been done in this description of TAMARACK-3. Instead, functional elements such as the ALU are modelled without delay and the update of storage elements is described as an atomic action. A fundamental feature of this abstraction is the assumption that updates to storage elements do not propagate to their outputs until the end of the clock cycle, i.e., there is no possibility of a closed feedback path resulting in a race condition.

### 3.3.4 Some Bottom Level Assumptions

The relatively abstract view of how a microinstruction is interpreted during a single clock cycle is the lowest level in this description of TAMARACK-3. The phase level view is also the lowest level of specification in the formal proof of correctness. The semantic gap between this abstraction and actual hardware is bridged by several informally stated assumptions.

First of all, it is assumed that the design is implemented by digital circuitry in some kind of synchronous logic design style (e.g., two-phase, non-overlapping logic). Depending on the design style, certain rules must be followed to ensure, for instance, that updates to storage elements do not propagate to their outputs until the end of the clock cycle. Assuming that these rules have been followed and certain other constraints have been satisfied,[5] synchronous logic gives rise to the abstract view where functional elements are modelled without delay and the update of storage elements is described as an atomic action.

Synchronous logic only ensures that this abstraction is valid for the internal logic; some additional considerations are required to ensure that this abstraction is valid for the interface between internal logic and external devices.

Asynchronous input events, such as a transition from `F` to `T` on the hardware interrupt request pin `ireq`, may occur at any time. In particular, they may occur at any point during the clock cycle used to synchronize the internal logic. Asynchronous input events may or may not be detected during the clock cycle depending on when they occur with respect to the minimum set-up time of any internal storage device that samples (directly or indirectly) this input. Carefully designed interface circuitry would be required to minimize the *metastability* problem, that is, the possibility of unstable equilibria in cross-coupled circuits [127].

In addition to sampling asynchronous inputs, the internal logic generates asynchronous outputs such as the interrupt acknowledge flag `iack`. External outputs may be sampled at any point in continuous time and every change in their value is significant. During a clock cycle, synchronous logic may generate several transient values before settling on a set of final values. For signals used internally, these transient values have no effect, but additional interface circuitry may be required to ensure that transient values do not propagate to asynchronous outputs.

These informal assumptions underlie the abstract view of how a microinstruction is interpreted during a single clock cycle. In the formal theory, asynchronous input and output events are modelled with respect to the same discrete time scale used to model internal synchronous logic. The model represents the *observed* behaviour of the external environment as seen by the internal logic. An asynchronous output event is described by changing the value of an output signal between adjacent points of discrete time. It is assumed that the output signal is otherwise stable with respect to continuous time.

[5] Conventional CAD tools are commonly used to check that synchronous logic has been correctly implemented with respect to the rules of a particular design style and that other constraints are satisfied, e.g., timing constraints.

Chapter 4

# Formal Specification

> When converting a natural-language description, or mental image, into a precise form, no algorithmic procedure is possible. One can develop only heuristic approaches and constantly check to see if the resulting formal description is consistent with the original informal idea. Frequently the process of writing a formal specification generates questions about what is desired in situations not originally considered. Thus, the process of constructing a flow table may clarify the problem even if no further use is made of the table.
>
> [Stephen H. Unger, Asynchronous Sequential Switching Circuits, 1969]

> Viper's top-level specification and its major-state level were both supplied in a logical language; but its block-level model was given partly formally and partly pictorially (as was natural). Combining these two parts required both ingenuity and some guesswork. The guesses were based on the coincidence of line names, on the names of bound variables in functional definitions, and on the annotations in the text of the definitions. None of these notational devices can be regarded as a formal specification.
>
> [Avra Cohn, The Notion of Proof in Hardware Verification, 1989]

Among the most interesting and creative aspects of using formal methods to verify microprocessor systems is the translation of an informal description, such as the one just given in the previous chapter, into a formal specification.

We begin this chapter by elaborating on the use of generic specification, in particular, some of the motivations for filtering out non-essential detail from the formal specification of TAMARACK-3. We also describe how generic specifications can be created in the HOL logic using only existing constructs.

The rest of this chapter shows how these techniques are used to formally specify the design and operation of TAMARACK-3. This includes formal specifications of the internal architecture, the programming level model and external memory.

## 4.1 Generic Specification

### 4.1.1 Motivation

The formal verification of TAMARACK-3 focuses very specifically on the register-transfer level operation of the internal architecture. Below and above this level there are aspects of a complete design for TAMARACK-3 which are not formally considered. For instance, the formal proof does not consider whether the register-transfer components have been

designed correctly. Similarly, the formal proof only shows that the design satisfies the semantics of the instruction set but it does not consider whether these instructions could be combined to write useful programs.

To clearly demarcate the boundaries between what has and what has not been formally considered in this proof, we have used generic descriptions in both the informal description and formal specification of TAMARACK-3 to eliminate as much detail as possible without simplifying the verification problem.[1]

This is quite different from conventional microprocessor descriptions which are usually very specific about details such as word size and various operations performed on data. Nevertheless, it is possible to explain and reason about a great deal of the internal architecture without specifying these details. The ability to make use of the abstract description without knowing these details is one way in which formal verification is distinguished from conventional simulation where these details would generally need to be known.

To illustrate the point that symbols are often just place-holders in a formal proof, we consider the uninterpreted primitive add which appears in both the bottom and top level descriptions of TAMARACK-3. At the bottom level, it is used to describe one of the functions performed by the ALU. At the top level, it is used to give the semantics of the TAMARACK-3 ADD instruction. The formal proof only shows that the ADD instruction is correctly implemented in the sense that the add operation is applied to the correct operands and the result stored in the correct destination by the end of the instruction cycle. Since add is only a place-holder in the formal proof, it is possible (and desirable) to regard this symbol as an uninterpreted primitive instead of a defined symbol.

A more pragmatic reason for using generic specifications is to avoid having to build up computational models 'from scratch' for particular cases when the verification problem has very little to do with computation. In the case of TAMARACK-3, building up a computational model for machine words, e.g., defining arithmetic operations on machine words, would be wasted effort for the purposes of verifying its register-transfer level operation. In this respect, the specification of TAMARACK-3 differs from earlier examples of microprocessor verification in the HOL system [29,30,58,78,79] by not using the built-in HOL types and postulated axioms (inherited from the LCF_LSM system) for bit strings and machine words. Not using this built-in infra-structure also means that this example could be more easily reproduced in other formalisms which lack infra-structure for reasoning about hardware.

Finally, there is the additional benefit that the generic description of TAMARACK-3 is re-usable for a wide range of possible word sizes and possible interpretations for the primitive operations. Later in Chapter 7, we show how correctness results for TAMARACK-3 can be specialized for the case of a 16-bit machine.

The cost of generic specification for TAMARACK-3 is surprisingly small: just two assumptions about the data types and primitive operations are needed to verify its register-transfer level operation. One of these assumptions, for instance, states that the numerical value of every 3-bit word is less than eight. These two assumptions appear explicitly in the correctness theorems.

[1] Our approach was prompted by discussions with researchers at SRI International, Menlo Park. The EHDM verification system developed at SRI also supports a form of generic specification using parameterized modules [142].

### 4.1.2 Formal Basis

Our use of the prefix * for uninterpreted data types in the informal description of TAMARACK-3, e.g., :*word, hinted at our intention to use type variables for uninterpreted types in the formal specification. We also need a formal basis for uninterpreted primitives, e.g., add, which denote operations on these basic data types.

One possibility would be to simply introduce the names of these operations as constants in the formal theory; the operations denoted by these constants would have fixed, unknown values. Although this approach is appealingly simple, the association of operation names with fixed, unknown values would be inconvenient if we ever wanted to extend the formal theory by giving an interpretation to the heretofore uninterpreted data types and primitive operations.[2]

Instead of a theory with constants denoting fixed, unknown values for primitive operations, we would like to 'parameterize', in an intuitive sense, the theory by the data types and primitive operations mentioned in the theory. The theory could be 'instantiated' by giving an interpretation to the data types and primitive operations (as we show in Chapter 7). This would be useful when combining correctness results for register-transfer architecture of TAMARACK-3 with another theory about the implementation of various register-transfer level components.

Although there is no existing mechanism to parameterize (in a literal sense) a HOL theory, the desired effect can be achieved in the HOL logic by parameterizing definitions in the theory by an extra variable, rep, which denotes the 'representation' of a theory.

The representation parameter can be thought of as a package containing values for each of the primitive operations. A particular operation can be specified by applying a selector function to the representation. To avoid introducing more names than necessary, we have just re-used names from the informal description for the corresponding selector functions. Whereas these names were used in the informal description to directly refer to primitive operations, they are used in the formal specification to denote selector functions which must be applied to the representation variable. Instead of saying,

> 'the operation denoted by add'

we refer to this operation as:

> 'the operation selected by add'

For instance, the terms inc, add and sub denote selector functions for the three operations performed by the ALU. Applying each of these terms to the representation variable rep yields the three ALU operations specified by rep.

Like any other term in the HOL logic, the representation variable rep is associated with a type. We have taken the straightforward approach of representing this variable in the HOL logic as an n-tuple where each element corresponds to one of the primitive operations on uninterpreted data types. The particular ordering of elements in this

[2]There is also a technical reason why this scheme might not work very well for the HOL system, in particular, that unbound type variables are not allowed in definitions.

n-tuple is completely arbitrary. The type of rep is denoted by the following type abbreviation.[3]

```
AbbreviateType (
  rep_ty,
  ":(*wordn→bool)×                              % iszero %
    (*wordn→*wordn)×                            % inc %
    (*wordn×*wordn→*wordn)×                     % add %
    (*wordn×*wordn→*wordn)×                     % sub %
    (num→*wordn) ×                              % wordn %
    (*wordn→num) ×                              % valn %
    (*wordn→*word3)×                            % opcode %
    (*word3→num)×                               % val3 %
    (*wordn→*address)×                          % address %
    (*memory×*address→*wordn)×                  % fetch %
    (*memory×*address×*wordn→*memory)×          % store %
    (num→*word4)×                               % word4 %
    (*word4→num)");;                            % val4 %
```

Selectors for primitive operations are defined in the formal theory by composing various sequences of the two primitive selectors FST and SND. For instance, the first three selectors, iszero, inc, and add, have the following definitions:[4]

```
Define ("iszero (rep:rep_ty) = FST rep");;

Define ("inc (rep:rep_ty) = FST(SND rep)");;

Define ("add (rep:rep_ty) = FST(SND(SND rep))");;
```

The rest of the selectors are defined in a similar manner such that the following theorem is true:

[3]The built-in HOL system utility for creating type abbreviations would not allow this particular abbreviation since it contains type variables. However, there is an alternative way to introduce names to stand for fully expanded type expressions (using ML variables and ML antiquotation). To mask these unimportant details, we have 'invented' the meta-language function AbbreviateType for the purpose of introducing type abbreviations without restrictions on the occurrence of type variables in abbreviations.

[4]We have also 'invented' the meta-language function Define for creating definitions; this function masks a few unimportant details about using the built-in HOL system utilities for creating definitions.

```
|- rep =
    ((iszero rep),
     (inc rep),
     (add rep),
     (sub rep),
     (wordn rep),
     (valn rep),
     (opcode rep),
     (val3 rep),
     (address rep),
     (fetch rep),
     (store rep),
     (word4 rep),
     (val4 rep))
```

As just mentioned, two assumptions about uninterpreted types and uninterpreted primitives are needed to verify TAMARACK-3. These assumptions are expressed by predicates `Val3_CASES_ASM` and `Val4Word4_ASM` which are both parameterized by the representation variable `rep`.

```
Define ("Val3_CASES_ASM (rep:rep_ty) = ∀w. ((val3 rep) w) < 8");;

Define (
  "Val4Word4_ASM (rep:rep_ty) =
    ∀n. n < 16 ⟹ (((val4 rep) ((word4 rep) n)) = n)");;
```

The predicate `Val3_CASES_ASM` expresses the assumption that the numerical value of every 3-bit word is less than eight. The predicate `Val4Word4_ASM` expresses the assumption that `:val4` and `:word4` are inverses for numbers less than sixteen.

## 4.2 TAMARACK-3 Specification

The formal specification of TAMARACK-3 is hand-translated from the informal description given earlier in Chapter 3. It consists of three main parts:

- The design of the internal architecture in terms of its structural organization and behavioural models for primitive components.
- The programming level model based on the semantics of the instruction set and the processing of hardware interrupts.
- A specification of the external environment, in particular, a behavioural model for external memory.

We begin by describing how structure and behaviour are represented in the formal specification. Following these preliminaries, we present the three main parts of the formal specification of TAMARACK-3.

### 4.2.1 Specifying Structure and Behaviour

The specification of TAMARACK-3 models devices at all levels of the specification hierarchy by predicates which express relations on time-dependent signals. These predicates are parameterized, in part, by variables representing physical input and output signals. They may also be parameterized by other signals representing the internal state or external conditions governing the behaviour of the device. Time-dependent signals are modelled as functions from discrete time to signal values. As shown in the following type abbreviation, discrete time is represented by the natural numbers.

```
AbbreviateType ('time',":num");;
```

The behaviour of a primitive device is specified directly in terms of defined operators such as logical connectives and arithmetic functions. The uninterpreted primitives just described in Section 4.1.2 may also be used to directly specify the behaviour of a device. Typically, the behaviour of a primitive component is expressed by an equation for output signals in terms of input signals, internal state, and external conditions. Universal quantification over explicit time variables is used to state that the behaviour holds at all points in discrete time.

The behaviour of a non-primitive device is specified indirectly by composing behaviours for simpler devices. The interconnection of components through similarly-named ports is expressed by logical conjunction. Existential quantification is used to hide internal signals. This can be viewed as a structural specification of the device.

### 4.2.2 Internal Architecture

The formal specification of the internal architecture of TAMARACK-3 begins with behaviour models for primitive components such as the ALU and the registers. Higher up the specification hierarchy, non-primitive devices such as the control unit and datapath are specified structurally as the composition of simpler devices.

As explained earlier in Chapter 3, the internal operation of TAMARACK-3 is described at a relatively abstract level where functional elements such as the ALU are modelled without delay and the update of a storage element is an atomic action. The formal specifications in this section are based on this abstract view of behaviour.

#### 4.2.2.1 Primitive Components of the Datapath

The ALU implements four different operations: the operation performed by the ALU is determined by the two ALU control signals `f0` and `f1`. The operation selected from the representation variable `rep` by the selector function `inc` takes a single full-size word as an argument and returns a full-size word as a result. The operations selected by `add` and `sub` take two full-size words as arguments and return a single full-size word as a result. The fourth operation implemented by the ALU takes no arguments: it yields a constant result, namely, the full-size word representation of the number zero.

```
Define (
  "ALU (rep:rep_ty) (f0,f1,inp1,inp2,out) =
    ∀t:time.
      out t =
        (((f0 t,f1 t) = (T,T)) ⇒ ((inc rep) (inp2 t)) |
         ((f0 t,f1 t) = (T,F)) ⇒ ((add rep) (inp1 t,inp2 t)) |
         ((f0 t,f1 t) = (F,T)) ⇒ ((sub rep) (inp1 t,inp2 t)) |
                                  ((wordn rep) 0))");;
```

Definitions for `OpcField` and `AddrField` specify devices which implement the operations selected by `opcode` and `address` respectively for extracting the opcode and address fields from a full-size word. Depending on how the opcode and address fields are represented, the implementation of these two devices could be nothing more than some wiring connections to appropriate bits.

```
Define (
  "OpcField (rep:rep_ty) (inp,out) =
    ∀t:time. out t = (opcode rep) (inp t)");;

Define (
  "AddrField (rep:rep_ty) (inp,out) =
    ∀t:time. out t = (address rep) (inp t)");;
```

The definition of `TestZero` specifies a device which implements the operation selected by `testzero`.

```
Define (
  "TestZero (rep:rep_ty) (inp,out) =
    ∀t:time. out t = (iszero rep) (inp t)");;
```

#### 4.2.2.2 Modelling System Bus Operation

The system bus of the TAMARACK-3 datapath is used to transfer data between various devices in the datapath. Modelling the operation of this bus presents some special problems because control over the bus signal is decentralized. A correct design will never allow more than one device at a time to read a value onto the system bus. Some very simple models of the datapath bus (e.g., modelling a bus as a many-input multiplexor) are based on the informal assumption that this aspect of the design is correct. However, we would like to establish this fact as part of our formal proof.

One possibility is to invent an extra value (sometimes called 'Z') to denote the floating (or high-impedance) state. The tri-state word types built into the HOL system (but not used here) are an example of this approach [57,58,79]. The value of bus is determined by a function which combines the outputs of all the bus drivers. If more than one driver has a non-floating output, then the result returned by the combining function is either

an 'error' value or unknown. This scheme is appealing partly because it is familiar from switch level simulation models such as MOSSIM [16]. It is also similar in concept to *resolution functions* in the VHDL hardware description language [3].

We have chosen to model the datapath in a different and (to our knowledge) novel way which takes advantage of our relationship style of formal specification. In this approach, the behavioural model of a bus device is regulated by a time-dependent condition on its environment: the device may only assert a value onto the system bus if no other device is also attempting to assert a value onto the bus. This condition may be thought of as an additional control signal to the device although it does not correspond to any physical signal.

At the register-transfer level, each bus device has a 'read' signal which controls when the device attempts to assert a value onto the system bus. In TAMARACK-3, these 'read' signals are:

`rmem`, `rpc`, `racc`, `rir`, `rrtn`, and `rbuf`

The time-dependent condition regulating the behaviour of bus drivers in the TAMARACK-3 datapath can be defined in terms of these signals: a value can be successfully read onto the system bus if and only if at most one of these signals is equal to `T`. This condition is expressed by the following definition of `BusOkay`. (The right-hand side of this equation looks complicated but it is just a Boolean expression for 6-way exclusive or-ing.)

```
Define (
  "BusOkay (rmem,rpc,racc,rir,rrtn,rbuf,busokay) =
    ∀t:time.
      busokay t =
        ((rmem t) ⇒ (¬(rpc t ∨ racc t ∨ rir t ∨ rrtn t ∨ rbuf t)) |
        ((rpc t) ⇒ (¬(racc t ∨ rir t ∨ rrtn t ∨ rbuf t)) |
        ((racc t) ⇒ (¬(rir t ∨ rrtn t ∨ rbuf t)) |
        ((rir t) ⇒ (¬(rrtn t ∨ rbuf t)) |
        ((rrtn t) ⇒ (¬(rbuf t)) |
                    T)))))");;
```

`BusOkay` may be thought of as a virtual device in the datapath. Although it does not correspond to a physical component of the datapath, its specification can be derived directly from a structural description of the datapath by determining which devices share the system bus as a common output port.

Admittedly, some aspects of this particular approach to modelling bus operation may appear to be rather artificial, e.g., 'virtual' devices. However, other approaches to modelling bus operation may look artificial as well, e.g., modelling a bus as a device with inputs and outputs. In general, there does not appear to be any one "best" way to model (in pure logic) the de-centralized nature of control over the operation of a bus.

#### 4.2.2.3 More Primitive Components of the Datapath

The definition of `Interface` specifies a device which provides a two-way interface between the system bus and the memory data pins. The device attempts to read data

received from memory onto the system bus when the read signal r is equal to T. In the output direction, the current value of the system bus is connected to the output pins when the write signal w is equal to T; otherwise, the machine representation of zero is assigned to the output pins as a default value.

```
Define (
  "Interface (rep:rep_ty) (busokay,w,r,bus,datain,dataout) =
    ∀t:time.
      ((busokay t) ⟹ (r t) ⟹ (bus t = datain t)) ∧
      (dataout t = ((w t) ⇒ (bus t) | ((wordn rep) 0)))");;
```

Following our scheme for modelling the operation of the system bus, the predicate **Interface** is parameterized by the virtual signal **busokay** which is the time-dependent condition specified in the definition of **BusOkay**. When data is being read from memory and no other device is attempting to assert a value onto the system bus, then the memory data will be successfully read onto the bus.

The basic storage element in the datapath of TAMARACK-3 is a selectively loadable register for storing full-size words. If the 'write' signal is equal to T, then the current input will be loaded into the register; otherwise, the contents of the register are unchanged. The out signal serves as both an output signal and a signal representing the internal state of the register. The register can also be interfaced to the system bus: the register attempts to read its contents onto the system bus when its 'read' signal is equal to T. Like the definition of **Interface**, the definition of **Register** is parameterized by the virtual signal **busokay** which determines when the register can successfully assert its contents onto the system bus.

```
Define (
  "Register (busokay,w,r,inp,bus,out:time→*wordn) =
    ∀t:time.
      ((busokay t) ⟹ (r t) ⟹ (bus t = out t)) ∧
      (out (t+1) = ((w t) ⇒ (inp t) | (out t)))");;
```

A set of 1-bit control signals runs from the control unit to the datapath. It is convenient to view these control signals collectively as a single input to the datapath. Once inside the datapath, this bundle of control signals is separated into the fifteen individual control signals.

In the formal specification, this bundle can be represented as a signal whose value at any discrete point in time is an n-tuple with fifteen elements. The following definition of **DecodeCntls** specifies a block of wiring which separates this bundle of control signals into fifteen individual signals by 'assigning' elements of n-tuple representation to corresponding control signals. This definition is expressed in terms of an equation where the left and right hand sides of the equation are n-tuples. Two n-tuples are equal if and only if they have exactly the same number of elements and matching elements of each n-tuple are equal both in type and in value. In effect, we are using properties of n-tuples to model bit manipulation operations, in particular, the extraction of individual bits from a group of bits.

```
AbbreviateType (
  'cntls_ty',
  ":bool×bool×bool×bool×bool×bool×bool×
    bool×bool×bool×bool×bool×bool×bool×bool");;

Define (
  "DecodeCntls (
    cntls:time→cntls_ty,
    wmem,rmem,wmar,wpc,rpc,wacc,racc,
    wir,rir,wrtn,rrtn,warg,alu0,alu1,rbuf) =
    ∀t:time.
      (wmem t,rmem t,wmar t,wpc t,rpc t,wacc t,racc t,
       wir t,rir t,wrtn t,rrtn t,warg t,alu0 t,alu1 t,rbuf t) =
      (cntls t)");;
```

The remaining three components needed to implement the datapath are a JK flip-flop, a two-input OR gate and voltages sources, i.e, 'power' and 'ground'.

```
Define (
  "JKFF (j,k,out) =
    ∀t:time. out (t+1) = (((j t) ∧ ¬(out t))  ∨ (¬(k t) ∧ (out t)))");;

Define ("OR (inp1,inp2,out) = ∀t:time. out t = ((inp1 t) ∨ (inp2 t))");;

Define ("PWR out = ∀t:time. out t = T");;

Define ("GND out = ∀t:time. out t = F");;
```

#### 4.2.2.4 Datapath Implementation

The register-transfer level implementation of the datapath is formally specified by the definition of Datapath in terms of the above-mentioned primitive components.

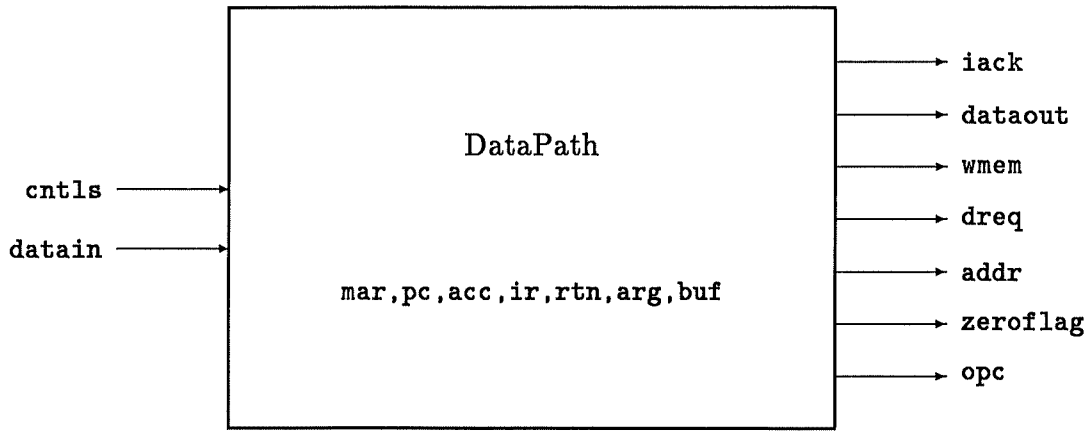



Figure 4.1: External View of the Datapath Specification

```
Define (
  "DataPath (rep:rep_ty)
    (cntls,datain,mar,pc,acc,ir,rtn,iack,
     arg,buf,dataout,wmem,dreq,addr,zeroflag,opc) =
    ∃bus busokay alu pwr gnd rmem wmar wpc rpc
     wacc racc wir rir wrtn rrtn warg alu0 alu1 rbuf.
      DecodeCntls (
        cntls,
        wmem,rmem,wmar,wpc,rpc,wacc,racc,
        wir,rir,wrtn,rrtn,warg,alu0,alu1,rbuf) ∧
      BusOkay (rmem,rpc,racc,rir,rrtn,rbuf,busokay) ∧
      Interface rep (busokay,wmem,rmem,bus,datain,dataout) ∧
      OR (wmem,rmem,dreq) ∧
      Register (busokay,wmar,gnd,bus,bus,mar) ∧
      AddrField rep (mar,addr) ∧
      Register (busokay,wpc,rpc,bus,bus,pc) ∧
      Register (busokay,wacc,racc,bus,bus,acc) ∧
      TestZero rep (acc,zeroflag) ∧
      Register (busokay,wir,rir,bus,bus,ir) ∧
      OpcField rep (ir,opc) ∧
      Register (busokay,wrtn,rrtn,bus,bus,rtn) ∧
      JKFF (wrtn,rrtn,iack) ∧
      Register (busokay,warg,gnd,bus,bus,arg) ∧
      ALU rep (alu0,alu1,arg,bus,alu) ∧
      Register (busokay,pwr,rbuf,alu,bus,buf) ∧
      PWR pwr ∧
      GND gnd");;
```

The predicate Datapath is parameterized by a number of signals. Some of these parameters correspond to signals which are externally available as physical inputs and outputs of the datapath as shown in the 'black box' view of Figure 4.1. Other signal

parameters represent internal state variables; these internal state variables are also shown in Figure 4.1.

#### 4.2.2.5 Microcode Source and Micro-Assembler

We use a straightforward representation for each microinstruction word as an n-tuple where each element corresponds to one of the sub-fields described earlier in Chapter 3. For example, the n-tuple,

```
((F,F,T,F,T,F,F,F,F,F,F,F,F,F,F),(T,F),(T,F),(addr1,addr2))
```

is the representation of a typical microinstruction word. The first three elements of this representation are also n-tuples; they represent sub-fields which consist of individual bits. The fourth element is a pair of microcode addresses belonging to the type `:*word4`.

Instead of specifying the microcode directly in terms of bit patterns, a simple micro-assembler allows the microcode to be specified in a more readable form. The micro-assembler consists of the following definitions.[5]

```
Define (
  "Cntls (tok1,tok2,tok3) =
    ((tok2 = 'wmem'),
     (tok1 = 'rmem'),
     (tok2 = 'wmar'),
     (tok2 = 'wpc'),
     (tok1 = 'rpc'),
     (tok2 = 'wacc'),
     (tok1 = 'racc'),
     (tok2 = 'wir'),
     (tok1 = 'rir'),
     (tok2 = 'wrtn'),
     (tok1 = 'rrtn'),
     (tok2 = 'warg'),
     ((tok3 = 'inc') ∨ (tok3 = 'add')),
     ((tok3 = 'inc') ∨ (tok3 = 'sub')),
     (tok1 = 'rbuf'))");;
```

[5]It is not essential to use string tokens in the microcode specification; they are merely convenient to use as a set of distinct literals with meaningful names e.g., `'wmem'` for "write memory".

```
Define ("waitdack = (T,T)");;
Define ("waitidle = (T,F)");;
Define ("continue = (F,F)");;

Define ("jump = (T,T)");;
Define ("jireq = (T,F)");;
Define ("jzero = (F,T)");;
Define ("jopcode = (F,F)");;

AbbreviateType (
  rom_ty,
  ":cntls_ty×(bool×bool)×(bool×bool)×(*word4×*word4)");;

Define (
  "CompileMicroCode (rep:rep_ty) (cntl,wcode,jcode,(n,m)) =
    ((Cntls cntl,wcode,jcode,((word4 rep) n,(word4 rep) m)):rom_ty)");;
```

The microcode source is specified by the definition of MicroCode. This function is a mapping from natural numbers to specifications of individual microinstructions.

```
Define (
  "MicroCode n =
    ((n = 0) ⇒ (('rpc','wmar','none'), waitidle, jireq, (1,2)) |
     (n = 1) ⇒ (('rpc','wrtn','zero'), continue, jump, (12,0)) |
     (n = 2) ⇒ (('rmem','wir','none'), waitdack, jump, (3,0)) |
     (n = 3) ⇒ (('rir','wmar','none'), waitidle, jopcode, (4,0)) |
     (n = 4) ⇒ (('none','none','none'), continue, jzero, (5,11)) |
     (n = 5) ⇒ (('rir','wpc','none'), continue, jump, (0,0)) |
     (n = 6) ⇒ (('racc','warg','none'), waitidle, jump, (13,0)) |
     (n = 7) ⇒ (('racc','warg','none'), waitidle, jump, (14,0)) |
     (n = 8) ⇒ (('rmem','wacc','none'), waitdack, jump, (11,0)) |
     (n = 9) ⇒ (('racc','wmem','none'), waitdack, jump, (11,0)) |
     (n = 10) ⇒ (('rrtn','wpc','none'), continue, jump, (0,0)) |
     (n = 11) ⇒ (('rpc','none','inc'), continue, jump, (12,0)) |
     (n = 12) ⇒ (('rbuf','wpc','none'), continue, jump, (0,0)) |
     (n = 13) ⇒ (('rmem','none','add'), waitdack, jump, (15,0)) |
     (n = 14) ⇒ (('rmem','none','sub'), waitdack, jump, (15,0)) |
                 (('rbuf','wacc','none'), continue, jump, (11,0)))");;
```

The formal specification of the microcode source is based on the informal description of the FSM given earlier in Chapter 3, in particular, the flow graph in Figure 3.7 and the mapping from FSM states to actions in Figure 3.8. Each line of the microcode specification consists of four parts corresponding to the four sub-fields of the microinstruction word. For example, the term,

```
(('rpc','wmar','none'),waitidle,jireq,(1,2))
```

specifies the microinstruction at location 0 in the microcode.

The first part specifies the action to be performed by the datapath in terms of a data transfer from a source to a destination and an ALU operation. In this case, the contents of the program counter pc are read onto the bus and written into the memory address register mar. The string 'none' indicates that no specific ALU operation is required.

The second part of each microinstruction specifies one of three wait conditions,

```
waitidle   - repeat if ¬(idle or ¬dack)
waitdack   - repeat if ¬dack
continue   - do not repeat
```

which may cause this particular microinstruction to be repeated.

The third part of the microinstruction specifies how to compute the address of the next microinstruction (when not waiting in a repeat-loop).

```
jump       - use addr1
jireq      - use addr1 if ireq is T, else use addr2
jzero      - use addr1 if acc is zero, else use addr2
jopcode    - add opcode to offset in addr1
```

The last part of the microinstruction specifies two microinstruction addresses or, in the case of jopcode, the offset to be added to the opcode field in computing the address of the next microinstruction.

Microinstruction specifications are individually assembled by the micro-assembler function CompileMicroCode. For example, the result of applying CompileMicroCode to the microinstruction at location 0 is described by the following theorem (after unfolding with definitions for the micro-assembler functions).

```
|- CompileMicroCode rep(MicroCode 0) =
   ((F,F,T,F,T,F,F,F,F,F,F,F,F,F,F),(T,F),(T,F),word4 rep 1,word4 rep 2)
```

Specifying the microcode source in pure logic and using a proof-generation system to unfold this specification is a very secure way (at least as secure as proof-generation) to assemble a microcode code source into bit patterns. The result of assembling the TAMARACK-3 microcode source is shown below (where a let-expression has been introduced for pretty-printing purposes).

```
|- CompileMicroCode rep(MicroCode n) =
   let addrs (p,q) = (word4 rep p,word4 rep q) in
   ((n = 0) ⇒ ((F,F,T,F,T,F,F,F,F,F,F,F,F,F,F),(T,F),(T,F),addrs(1,2)) |
    (n = 1) ⇒ ((F,F,F,F,T,F,F,F,F,T,F,F,F,F,F),(F,F),(T,T),addrs(12,0)) |
    (n = 2) ⇒ ((F,T,F,F,F,F,F,T,F,F,F,F,F,F,F),(T,T),(T,T),addrs(3,0)) |
    (n = 3) ⇒ ((F,F,T,F,F,F,F,F,T,F,F,F,F,F,F),(T,F),(F,F),addrs(4,0)) |
    (n = 4) ⇒ ((F,F,F,F,F,F,F,F,F,F,F,F,F,F,F),(F,F),(F,T),addrs(5,11)) |
    (n = 5) ⇒ ((F,F,F,T,F,F,F,F,T,F,F,F,F,F,F),(F,F),(T,T),addrs(0,0)) |
    (n = 6) ⇒ ((F,F,F,F,F,F,T,F,F,F,F,T,F,F,F),(T,F),(T,T),addrs(13,0)) |
    (n = 7) ⇒ ((F,F,F,F,F,F,T,F,F,F,F,T,F,F,F),(T,F),(T,T),addrs(14,0)) |
    (n = 8) ⇒ ((F,T,F,F,F,T,F,F,F,F,F,F,F,F,F),(T,T),(T,T),addrs(11,0)) |
    (n = 9) ⇒ ((T,F,F,F,F,F,T,F,F,F,F,F,F,F,F),(T,T),(T,T),addrs(11,0)) |
    (n = 10) ⇒ ((F,F,F,T,F,F,F,F,F,F,T,F,F,F,F),(F,F),(T,T),addrs(0,0)) |
    (n = 11) ⇒ ((F,F,F,F,T,F,F,F,F,F,F,F,T,T,F),(F,F),(T,T),addrs(12,0)) |
    (n = 12) ⇒ ((F,F,F,T,F,F,F,F,F,F,F,F,F,F,T),(F,F),(T,T),addrs(0,0)) |
    (n = 13) ⇒ ((F,T,F,F,F,F,F,F,F,F,F,F,T,F,F),(T,T),(T,T),addrs(15,0)) |
    (n = 14) ⇒ ((F,T,F,F,F,F,F,F,F,F,F,F,F,T,F),(T,T),(T,T),addrs(15,0)) |
                ((F,F,F,F,F,T,F,F,F,F,F,F,F,F,T),(F,F),(T,T),addrs(11,0)))
```

#### 4.2.2.6 Primitive Components of the Control Unit

Chapter 3 outlined the implementation of the control unit FSM by a microcode program counter, microcode ROM, ROM output decoder and next address logic.

The microcode program counter `mpc` is modelled as a register which is unconditionally updated each clock cycle with its current input.

```
Define ("MPC (inp,out:time→*word4) = ∀t:time. out (t+1) = inp t");;
```

The microcode ROM is specified as a combinational device which takes a FSM state as input and returns a microinstruction word as a result. The actual contents of the ROM have already been specified by the earlier definition of `MicroCode`.

```
Define (
  "ROM (rep:rep_ty) (inp,out) =
    ∀t:time.
      out t = (CompileMicroCode rep) (MicroCode ((val4 rep) (inp t)))");;
```

The output of the ROM is an n-tuple with elements corresponding to microinstruction sub-fields. The definition of `DecodeROM` specifies a device which separates the output of the ROM into various microinstruction sub-fields. The specification of this device is similar to the earlier definition of `DecodeCntls`.

```
Define (
  "DecodeROM (rom:time→rom_ty,f0,f1,f2,f3,addr1,addr2,cntls) =
    ∀t:time. (cntls t,(f0 t,f1 t),(f2 t,f3 t),(addr1 t,addr2 t)) = rom t");;
```

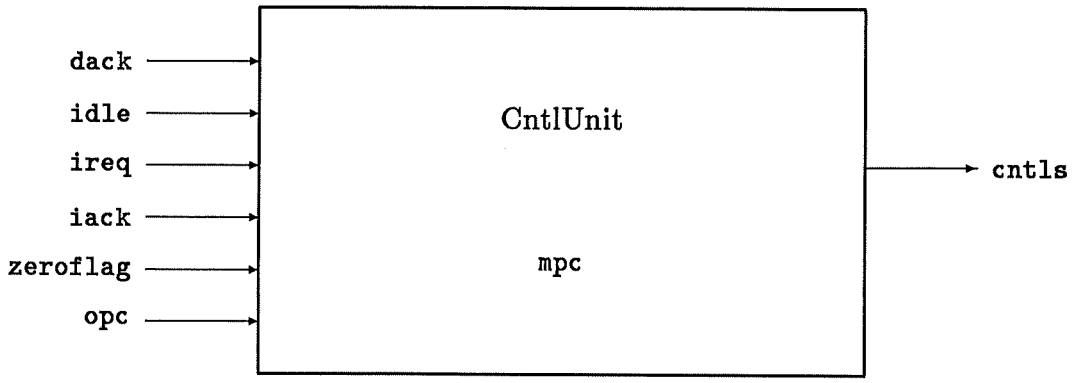



Figure 4.2: External View of the Control Unit Specification

The definition of NextMPC specifies the computation of FSM states according to the flow graph in Figure 3.7. Although the definition of NextMPC looks rather complex for a primitive component, it is just a block of combinational logic which could be implemented by a set of logic gates, some multiplexors and an adder. The formal verification of TAMARACK-3 could be extended by proving that a particular implementation of the next address logic satisfies the behavioural specification shown below.

```
Define (
  "NextMPC (rep:rep_ty)
    (f0,f1,f2,f3,dack,idle,ireq,iack,
     zeroflag,opc,addr1,addr2,mpc,next) =
    ∀t:time.
      let waitcond =
        ((((f0 t,f1 t) = waitdack) ∧ ¬(dack t)) ∨
         (((f0 t,f1 t) = waitidle) ∧ ¬((idle t) ∨ ¬(dack t)))) in
      (next t =
        (waitcond ⇒ (mpc t) |
         ((f2 t,f3 t) = jump) ⇒ (addr1 t) |
         ((f2 t,f3 t) = jireq) ⇒
           (((ireq t) ∧ ¬(iack t)) ⇒ (addr1 t) | (addr2 t)) |
         ((f2 t,f3 t) = jzero) ⇒
           ((zeroflag t) ⇒ (addr1 t) | (addr2 t)) |
           ((word4 rep)
             (((val3 rep) (opc t)) + ((val4 rep) (addr1 t))))))");;
```

#### 4.2.2.7 Control Unit Implementation

The register-transfer level implementation of the control unit FSM is formally specified by the definition of CntlUnit.

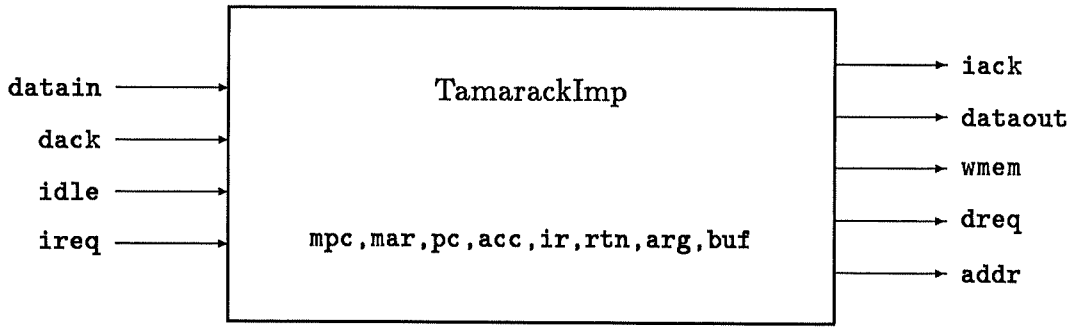



Figure 4.3: External View of the Top Level Specification

```
Define (
  "CntlUnit (rep:rep_ty) (dack,idle,ireq,iack,zeroflag,opc,mpc,cntls) =
    ∃f0 f1 f2 f3 addr1 addr2 next rom.
      NextMPC rep (
        f0,f1,f2,f3,dack,idle,ireq,iack,
        zeroflag,opc,addr1,addr2,mpc,next) ∧
      MPC (next,mpc) ∧
      ROM rep (mpc,rom) ∧
      DecodeROM (rom,f0,f1,f2,f3,addr1,addr2,cntls)");;
```

As shown in Figure 4.2, all the signal names in the parameter list of `CntlUnit` correspond to physical inputs and outputs of the control unit FSM with the exception of the microcode program counter `mpc` which is an internal state variable at this level.

#### 4.2.2.8 Top Level Structure

The control unit and datapath are combined to implement the internal architecture of TAMARACK-3. The control signals, `cntl`, and feedback signals, `zeroflag` and `opc`, are internal connections between the control unit and datapath.

```
Define (
  "TamarackImp (rep:rep_ty)
    (datain,dack,idle,ireq,mpc,mar,pc,
     acc,ir,rtn,arg,buf,iack,dataout,wmem,dreq,addr) =
    ∃zeroflag opc cntls.
      CntlUnit rep (dack,idle,ireq,iack,zeroflag,opc,mpc,cntls) ∧
      DataPath rep (
        cntls,datain,mar,pc,acc,ir,rtn,iack,
        arg,buf,dataout,wmem,dreq,addr,zeroflag,opc)");;
```

Figure 4.3 shows which signals in the parameter list correspond to physical input and output pins of the microprocessor and which signals correspond to internal state variables.

### 4.2.3 Programming Level Model

The formal specification of the programming level model is based on the informal descriptions given in Chapter 3 of the instruction set semantics and the hardware interrupt facility.

The semantics of each instruction is given individually by the definition of a function which returns the next (externally visible) state of the microprocessor, i.e., the next values of the memory state `mem`, program counter `pc`, accumulator `acc`, return address register `rtn` and interrupt acknowledge flag `iack`. The following definitions specify how these values are computed from the current state of the microprocessor.

```
Define (
  "JZR_SEM (rep:rep_ty)
    (mem:*memory,pc:*wordn,acc:*wordn,rtn:*wordn,iack:bool) =
    let inst = (fetch rep) (mem,(address rep) pc) in
    let nextpc = ((iszero rep) acc) => inst | ((inc rep) pc) in
      (mem,nextpc,acc,rtn,iack)");;

Define (
  "JMP_SEM (rep:rep_ty)
    (mem:*memory,pc:*wordn,acc:*wordn,rtn:*wordn,iack:bool) =
    let inst = (fetch rep) (mem,(address rep) pc) in
      (mem,inst,acc,rtn,iack)");;

Define (
  "ADD_SEM (rep:rep_ty)
    (mem:*memory,pc:*wordn,acc:*wordn,rtn:*wordn,iack:bool) =
    let inst = (fetch rep) (mem,(address rep) pc) in
    let operand = (fetch rep) (mem,(address rep) inst) in
      (mem,(inc rep) pc,(add rep) (acc,operand),rtn,iack)");;

Define (
  "SUB_SEM (rep:rep_ty)
    (mem:*memory,pc:*wordn,acc:*wordn,rtn:*wordn,iack:bool) =
    let inst = (fetch rep) (mem,(address rep) pc) in
    let operand = (fetch rep) (mem,(address rep) inst) in
      (mem,(inc rep) pc,(sub rep) (acc,operand),rtn,iack)");;

Define (
  "LDA_SEM (rep:rep_ty)
    (mem:*memory,pc:*wordn,acc:*wordn,rtn:*wordn,iack:bool) =
    let inst = (fetch rep) (mem,(address rep) pc) in
    let operand = (fetch rep) (mem,(address rep) inst) in
      (mem,(inc rep) pc,operand,rtn,iack)");;
```

```
Define (
  "STA_SEM (rep:rep_ty)
    (mem:*memory,pc:*wordn,acc:*wordn,rtn:*wordn,iack:bool) =
    let inst = (fetch rep) (mem,(address rep) pc) in
    let newmem = (store rep) (mem,(address rep) inst,acc) in
      (newmem,(inc rep) pc,acc,rtn,iack)");;

Define (
  "RFI_SEM (rep:rep_ty)
    (mem:*memory,pc:*wordn,acc:*wordn,rtn:*wordn,iack:bool) =
    (mem,rtn,acc,rtn,F)");;

Define (
  "NOP_SEM (rep:rep_ty)
    (mem:*memory,pc:*wordn,acc:*wordn,rtn:*wordn,iack:bool) =
    (mem,(inc rep) pc,acc,rtn,iack)");;
```

The processing of a hardware interrupt is described in a similar way by the definition of a function which computes the next state of the microprocessor from its current state.

```
Define (
  "IRQ_SEM (rep:rep_ty)
    (mem:*memory,pc:*wordn,acc:*wordn,rtn:*wordn,iack:bool) =
    (mem,((wordn rep) 0),acc,pc,T)");;
```

The opcode of the current instruction word determines which instruction is executed during a particular instruction cycle. The following set of definitions specify the opcode value for each instruction.

```
Define ("JZR_OPC = 0");;
Define ("JMP_OPC = 1");;
Define ("ADD_OPC = 2");;
Define ("SUB_OPC = 3");;
Define ("LDA_OPC = 4");;
Define ("STA_OPC = 5");;
Define ("RFI_OPC = 6");;
Define ("NOP_OPC = 7");;
```

The opcode value of the current instruction is obtained by fetching the memory word addressed by the program counter, extracting the value of its opcode field and interpreting the opcode as a number between 0 and 7. This procedure is specified in the definition of `OpcVal`

```
Define (
    "OpcVal (rep:rep_ty) (mem,pc) =
      (val3 rep) ((opcode rep) ((fetch rep) (mem,(address rep) pc)))");;
```

Every instruction cycle results in the execution of a programming level instruction unless a hardware interrupt is detected at the beginning of this cycle. The following definition of NextState specifies the overall control mechanism for determining what happens during a particular instruction cycle.

```
Define (
  "NextState (rep:rep_ty) (ireq,mem,pc,acc,rtn,iack) =
    let opcval = OpcVal rep (mem,pc) in
    ((ireq ∧ ¬iack)     ⇒ IRQ_SEM rep (mem,pc,acc,rtn,iack) |
     (opcval = JZR_OPC) ⇒ JZR_SEM rep (mem,pc,acc,rtn,iack) |
     (opcval = JMP_OPC) ⇒ JMP_SEM rep (mem,pc,acc,rtn,iack) |
     (opcval = ADD_OPC) ⇒ ADD_SEM rep (mem,pc,acc,rtn,iack) |
     (opcval = SUB_OPC) ⇒ SUB_SEM rep (mem,pc,acc,rtn,iack) |
     (opcval = LDA_OPC) ⇒ LDA_SEM rep (mem,pc,acc,rtn,iack) |
     (opcval = STA_OPC) ⇒ STA_SEM rep (mem,pc,acc,rtn,iack) |
     (opcval = RFI_OPC) ⇒ RFI_SEM rep (mem,pc,acc,rtn,iack) |
                          NOP_SEM rep (mem,pc,acc,rtn,iack))");;
```

Finally, we use the function NextState to define the predicate TamarackBeh which specifies the intended behaviour of the microprocessor as a relation on the time-dependent signals mem, pc, acc, rtn and iack.

```
Define (
  "TamarackBeh (rep:rep_ty) (ireq,mem,pc,acc,rtn,iack) =
    ∀u:time.
      (mem (u+1),pc (u+1),acc (u+1),rtn (u+1),iack (u+1)) =
      NextState rep (ireq u,mem u,pc u,acc u,rtn u,iack u)");;
```

The programming level model not only hides structural details of the internal architecture but also timing details about the number of microinstructions executed for each instruction. To be more precise, the programming level model describes the operation of the microprocessor in terms of an abstract time scale where each instruction is uniformly executed in a single unit of time. This abstract time scale is different than the time scale used to specify the behaviour of register-transfer level components where a single unit of time corresponds to a single clock cycle. To emphasize this difference, we have used the explicit time variable u instead of t in the above definition of TamarackBeh (but there is no logical distinction between these two variable names). As we will see in Chapter 5, part of the verification task is to establish a formal relationship between these two granularities of discrete time.

### 4.2.4 External Memory Specification

A behavioural model for fully synchronous memory expressed by the predicate SynMemory. The definition of this predicate is shown below.

```
Define (
  "SynMemory (rep:rep_ty) (w,addr,dataout,mem,datain) =
    ∀t:time.
      (¬(w t) ⟹ (datain t = (fetch rep) (mem t,addr t))) ∧
      (mem (t+1) =
        ((w t) ⇒ ((store rep) (mem t,addr t,dataout t)) | (mem t)))");;
```

External memory implements the uninterpreted operations selected by `fetch` and `store`. The write signal `w` controls which operations are performed by the memory each clock cycle. Data is sent to external memory on the `dataout` bus and received from external memory on the `datain` bus. Memory addresses are sent to external memory on the `addr` bus. The internal state of memory is represented by the virtual signal `mem`.

A fully asynchronous version of external memory also implements the uninterpreted operations selected by `fetch` and `store` but uses handshaking signals to synchronize its interaction with the microprocessor. A formal specification for a fully asynchronous memory device is deferred until Chapter 6.

Chapter 5

# Formal Verification

The purpose of this chapter is to illustrate fundamental proof strategies for verifying microprocessor systems by describing a very straight-forward version of the TAMARACK-3 correctness proof.

The opening section on formulating a verification plan for TAMARACK-3 presents the main ideas of this chapter: stating correctness results, structuring a proof into multiple levels, and using logic to symbolically execute a design.

The next section summarizes the TAMARACK-3 correctness proof in terms of 'journal-level' proof steps based on the machine-generated correctness proof. The main point of interest in this section (with regard to the argument of this dissertation) is the role of uninterpreted primitives as mere place-holders in the formal proof.

Finally, the last section of the chapter describes an enhancement to the TAMARACK-3 correctness proof based on a general technique for synchronizing multiple levels of timing.

## 5.1 Verification Plan

This section describes the logical form used to state correctness results for TAMARACK-3, a plan to achieve these results and some of the basic proof techniques which are used to carry out this plan in the HOL system.

### 5.1.1 Stating Correctness Results

Although the terms 'correctness' and 'verification' may be understood in an informal context to mean different things to different people, these terms have a precise, technical meaning when formal logic is used to verify a hardware design. The formal verification (or proof of correctness) for TAMARACK-3 refers to the derivation of a theorem by formal proof in the HOL formulation of higher-order logic. This theorem relates the specification of the internal architecture, given by the predicate `TamarackImp`, to the specification of its intended behaviour, given by the predicate `TamarackBeh`.

We actually give three different versions of the top level correctness theorem but they are have the same basic form: that the constraints imposed by the implementation predicate `TamarackImp` satisfy the requirements expressed by the behaviour predicate `TamarackBeh`. The general form of this theorem is a logical implication:

```
|- Implementation Specification ⟹ Behaviour Specification
```

The precise form of this theorem (in each of the three cases) depends on a formally established relationship between the two different granularities of time used to specify

register-transfer level behaviour and the programming level model. The correctness theorems also include a behavioural model of external memory and a description of how memory is interfaced to the microprocessor.

For readers curious at this point about the appearance of these correctness results, the following theorem is (a slightly expanded form of) the result obtained by the procedure outlined later in this chapter.

```
|- ∀datain pwr dataout wmem dreq addr.
     Val3_CASES_ASM (rep:rep_ty) ∧
     Val4Word4_ASM rep ∧
     TamarackImp rep (
       datain,pwr,pwr,ireq,mpc,mar,pc,
       acc,ir,rtn,arg,buf,iack,dataout,wmem,dreq,addr) ∧
     SynMemory rep (wmem,addr,dataout,mem,datain) ∧
     PWR pwr ∧
     (((val4 rep) o mpc) 0 = 0)
     ⟹
     let f = TimeOfCycle rep (ireq,mem,pc,acc,rtn,iack) in
     TamarackBeh rep (ireq o f,mem o f,pc o f,acc o f,rtn o f,iack o f)
```

### 5.1.2 Multi-Level Verification

Although most of the tedious low level work of proof-generation can be done mechanically by the HOL system, the formal verification of non-trivial hardware (and other kinds of proofs) requires a considerable amount of guidance from the user.

Much of the creative work in using the HOL system centers upon the problem of discovering high-level proof strategies when routine HOL interactions like STRIP_TAC, BOOL_CASES_TAC and ASM_REWRITE_TAC are not enough to complete a proof. Proof-generation experience or mathematical insight is sometimes the source of a successful proof strategy. However, an equally important source for hardware verification, and more specifically, the verification of microprocessor systems is conventional design methodology where many useful strategies for controlling the complexity of reasoning informally about computer architecture are already well-known.

In particular, we have found that the concept of multiple interpretation levels, used by architects to achieve a "progressive translation of functions in several stages" [2] is a very effective strategy for controlling the complexity of verifying the correctness of TAMARACK-3. As explained earlier in Chapter 3, TAMARACK-3 has two levels of interpretation beneath the programming level: the microprogramming level and phase level. We use this structured view of the internal operation of the microprocessor to organize the bulk of the formal proof into two main steps:

1. Prove that each microinstruction is correctly interpreted at the phase level.

2. Prove that each programming level operation is correctly interpreted at the microprogramming level.

The proof is completed in a third (and relatively short) step by establishing a formally defined relationship between the two different granularities of discrete time used to specify the internal architecture and programming level model:

3. Relate clock cycles view of behaviour to instruction cycle view of behaviour.

Verifying the phase level operation of TAMARACK-3 in Step (1) does not depend on a behavioural model for external memory. However, different versions of Steps (2) and (3) are needed for fully synchronous mode and fully asynchronous mode.

### 5.1.3 Symbolic Execution

Many different kinds of computation systems are implemented by a hierarchy of interpretation levels. Showing that one level correctly implements the next higher level is equivalent to showing that the higher level is an abstract view of the lower level. Each operation at a given level in the hierarchy is typically interpreted (directly or indirectly) by one or more operations at the next lower level. To demonstrate that this level is correctly implemented by the next lower level, each operation is shown to be correctly implemented by the corresponding sequence of lower level operations. This is established by deriving the cumulative effect of each sequence and comparing it to the intended effect of the corresponding higher level operation.

It is relatively straightforward to reason about a fixed sequence (or a finite set of fixed sequences) of lower level operations. To derive the cumulative effect of a sequence of lower level operations, we use inference rules of higher-order logic to *symbolically execute* this sequence. The term 'symbolic execution' is used here in a purely descriptive sense for a proof technique which is actually nothing more than repeatedly unfolding various parts of the specification (or consequences of this specification derived at lower levels).[1]

It is natural to use forward proof to symbolically execute a sequence of operations starting with assumptions about the initial state and applying inference rules to derive subsequent states in the computation. Similar techniques were also used to verify the VIPER microprocessor. As Cohn remarks [30], forward proofs of this kind are a rather unsophisticated use of the HOL system which may produce unforeseen results. However, the use of forward proof in this case is indicated by the nature of the problem.

Even though forward proof lies at the heart of symbolic execution, it is very convenient (in the HOL system) to carry out these forward proofs in the overall context of a backward (or goal-oriented) proof using the built-in sub-goals package. After an initial bit of backward proof, forward inference steps are achieved by repeated use of resolution tactics with some direct manipulation of the intermediate results.

The following section illustrates the general concept of symbolic execution and the mechanics of using this proof technique in the HOL system.

[1] Reasoning about fixed sequences is a very simple form of symbolic execution. The more difficult case of reasoning about variable length sequences (due to microcode repeat-loops) is described in Chapter 6. Formal verification of the IMP compiler involves a hierarchical form of symbolic execution for reasoning about intermediate code generated by the compiler.

## 5.2 TAMARACK-3 Verification

In this chapter, we describe the formal verification of TAMARACK-3 when operating in fully synchronous mode. Verifying TAMARACK-3 in this mode is relatively simple because each programming level operation is implemented by a fixed sequence (or finite set of fixed sequences) of microinstructions. The considerably more difficult task of verifying TAMARACK-3 for fully asynchronous mode is outlined later in Chapter 6.

### 5.2.1 Phase Level

As explained earlier in Chapter 3, the phase level operation of TAMARACK-3 is described in terms of elementary operations performed by register-transfer level components. The interpretation of a microinstruction at the phase level during a single clock cycle results in a sequence of events which includes:

- Fetch the current microinstruction.
- Compute the address of the next microinstruction.
- Transfer data onto the system bus.
- Evaluate operations performed by functional elements.
- Update storage elements such as memory, registers and flipflops.

Verifying the operation of the microprocessor at this level is a matter of deriving the cumulative effect of these elementary operations for each of the sixteen microinstructions. We show that the cumulative effect of these operations satisfies the intended effect of each microinstruction.

To illustrate this process, we outline the steps taken to establish that microinstruction 0, i.e., the microinstruction at location 0 in the microcode, is correctly interpreted at the phase level. The intended effect of microinstruction 0,

```
MicroCode 0 = (('rpc','wmar','none'), waitidle, jireq, (1,2))
```

is to transfer the contents of the program counter pc to the memory address register mar and update the microcode program counter mpc according to conditions given in the flow graph of Figure 3.7. Furthermore, the execution of this microinstruction must leave the internal state of the memory mem and the contents of the program counter pc, accumulator acc, return address register rtn and interrupt acknowledge flag iack unchanged. The effect of this microinstruction on the remaining components of the internal state, namely, the arg and buf registers, is unimportant and can be ignored. Finally, the memory data output bus dataout must be set to its default value and the memory control flags, wmem and dreq, must be set to F.

These requirements are formally expressed in the goal term of the following call to the ML function set_goal. This is the first step in an interactive proof to formally derive this goal[2] as a theorem using the HOL sub-goal package.

[2]The term 'goal' has technical meaning in the HOL system (as an ML type abbreviation) but we use it more generally to mean a yet-to-be-proven theorem.

```
set_goal (
  [],
  "Val3_CASES_ASM (rep:rep_ty) ∧
   Val4Word4_ASM rep ∧
   TamarackImp rep (
     datain,dack,idle,ireq,mpc,mar,pc,
     acc,ir,rtn,arg,buf,iack,dataout,wmem,dreq,addr)
   ⟹
   ∀t.
    (((val4 rep) o mpc) t = 0)
    ⟹
    (((val4 rep) o mpc) (t+1) =
       ((¬((idle t) ∨ ¬(dack t))) ⇒ 0 |
        ((ireq t) ∧ ¬(iack t)) ⇒ 1 | 2)) ∧
    (mar (t+1) = pc t) ∧
    (pc (t+1) = pc t) ∧
    (acc (t+1) = acc t) ∧
    (rtn (t+1) = rtn t) ∧
    (iack (t+1) = iack t) ∧
    (dataout t = ((wordn rep) 0)) ∧
    (wmem t = F) ∧
    (dreq t = F)");;
```

A sequence of interactions with the sub-goal package eventually results in the generation of this goal as a theorem. To illustrate the strategy which underlies this sequence of interactions and more generally, how formal proof can be used to symbolically execute hardware, we describe a condensed view of an interactive HOL session for solving this goal.

We begin by expanding the above goal with the definitions for `TamarackImp`, `CntlUnit` and `DataPath`. We then apply standard goal reduction techniques to strip antecedents from the goal and break them up into a number of assumptions which are put into the assumption list. The result is shown below where each assumption appears as a term between a matching pair of brackets `[` and `]`. Most of these assumptions correspond to primitive components of the internal architecture with internal connections (hidden in the specification by existential quantification) now exposed as free variables in the assumption list.

```
OK..
"(val4 rep(mpc(t + 1)) =
  ((¬(idle t ∨ ¬dack t)) ⇒ 0 | ((ireq t ∧ ¬iack t) ⇒ 1 | 2))) ∧
 (mar(t + 1) = pc t) ∧
 (pc(t + 1) = pc t) ∧
 (acc(t + 1) = acc t) ∧
 (rtn(t + 1) = rtn t) ∧
 (iack(t + 1) = iack t) ∧
 (dataout t = wordn rep 0) ∧
 (wmem t = F) ∧
 (dreq t = F)"
    [ "Val3_CASES_ASM rep" ]
    [ "Val4Word4_ASM rep" ]
    [ "NextMPC
       rep
       (f0,f1,f2,f3,dack,idle,ireq,iack,zeroflag,opc,addr1,addr2,mpc,
        next)" ]
    [ "MPC(next,mpc)" ]
    [ "ROM rep(mpc,rom)" ]
    [ "DecodeROM(rom,f0,f1,f2,f3,addr1,addr2,cntls)" ]
    [ "DecodeCntls
       (cntls,wmem,rmem,wmar,wpc,rpc,wacc,racc,wir,rir,wrtn,rrtn,warg,
        alu0,alu1,rbuf)" ]
    [ "BusOkay(rmem,rpc,racc,rir,rrtn,rbuf,busokay)" ]
    [ "Interface rep(busokay,wmem,rmem,bus,datain,dataout)" ]
    [ "OR(wmem,rmem,dreq)" ]
    [ "Register(busokay,wmar,gnd,bus,bus,mar)" ]
    [ "AddrField rep(mar,addr)" ]
    [ "Register(busokay,wpc,rpc,bus,bus,pc)" ]
    [ "Register(busokay,wacc,racc,bus,bus,acc)" ]
    [ "TestZero rep(acc,zeroflag)" ]
    [ "Register(busokay,wir,rir,bus,bus,ir)" ]
    [ "OpcField rep(ir,opc)" ]
    [ "Register(busokay,wrtn,rrtn,bus,bus,rtn)" ]
    [ "JKFF(wrtn,rrtn,iack)" ]
    [ "Register(busokay,warg,gnd,bus,bus,arg)" ]
    [ "ALU rep(alu0,alu1,arg,bus,alu)" ]
    [ "Register(busokay,pwr,rbuf,alu,bus,buf)" ]
    [ "PWR pwr" ]
    [ "GND gnd" ]
    [ "val4 rep(mpc t) = 0" ]

() : void
```

After this initial bit of backward proof, forward inference steps are achieved by repeated use of resolution tactics. Theorems generated by resolution in this manner are logical consequences of the assumptions already in the assumption list. These theorems contribute various facts which incrementely advance the state of the symbolic execution.

As theorems are generated, they are added to the current list of assumptions and may be used by subsequent resolution steps to generate more theorems.

Since this part of the proof is only concerned with generating new theorems for the assumption list, we just show what is added onto the end of the assumption list after each main step. We begin by using the definition of ROM to generate the following equation for the current output of the ROM.

```
rom t = CompileMicroCode rep(MicroCode 0)
```

Instead of adding this equation directly to the assumption list, we use the microcode specification MicroCode and definitions for the micro-assembler functions given in Chapter 4 to transform it into an equation which gives the assembled form of microinstruction 0.

```
OK..
    ...
    [ "rom t =
       (F,F,T,F,T,F,F,F,F,F,F,F,F,F,F),(T,F),(T,F),word4 rep 1,
       word4 rep 2" ]

() : void
```

The definition of DecodeROM is then used to separate the ROM output into datapath control bits, cntls, next address logic control bits, f0, f1, f2 and f3, and microcode addresses, addr1 and addr2.

```
OK..
    ...
    [ "cntls t = F,F,T,F,T,F,F,F,F,F,F,F,F,F,F" ]
    [ "f0 t = T" ]
    [ "f1 t = F" ]
    [ "f2 t = T" ]
    [ "f3 t = F" ]
    [ "addr1 t = word4 rep 1" ]
    [ "addr2 t = word4 rep 2" ]

() : void
```

The next address logic control bits and microcode addresses extracted from the current output of the ROM are used to derive an expression for the output of the next address logic. This expression is obtained from the definition of NextState. The value of this expression depends on external inputs, idle, dack, and ireq, and the current value of the interrupt acknowledge flag iack.

```
OK..
    ...
    [ "next t =
       ((¬(idle t ∨ ¬dack t)) ⇒
        mpc t |
        ((ireq t ∧ ¬iack t) ⇒ word4 rep 1 | word4 rep 2))" ]

() : void
```

At the end of the clock cycle, the value computed by the next address logic is loaded into the microcode program counter mpc. The definition of MPC is used to establish this fact.

```
OK..
    ...
    [ "mpc(t + 1) =
       ((¬(idle t ∨ ¬dack t)) ⇒
        mpc t |
        ((ireq t ∧ ¬iack t) ⇒ word4 rep 1 | word4 rep 2))" ]

() : void
```

Meanwhile, the datapath is performing actions specified by the datapath control bits. To derive the result of these actions, we start with the definition of DecodeCntls to separate the bundle of control bits into individual control signals.

```
OK..
   ...
    [ "wmem t = F" ]
    [ "rmem t = F" ]
    [ "wmar t = T" ]
    [ "wpc t = F" ]
    [ "rpc t = T" ]
    [ "wacc t = F" ]
    [ "racc t = F" ]
    [ "wir t = F" ]
    [ "rir t = F" ]
    [ "wrtn t = F" ]
    [ "rrtn t = F" ]
    [ "warg t = F" ]
    [ "alu0 t = F" ]
    [ "alu1 t = F" ]
    [ "rbuf t = F" ]

() : void
```

The definitions of PWR and GND yield equations for the voltage sources pwr and gnd. These two signals are used to permanently enable or disable various functions in the datapath.

```
OK..
   ...
    [ "pwr t = T" ]
    [ "gnd t = F" ]

() : void
```

All of the 'read' signals are equal to F except for rpc satisfying the condition that only one bus device can attempt to transfer a value onto the system bus. The definition of BusOkay is used to establish that the virtual signal busokay is equal to T indicating that this condition is satisfied.

```
OK..
   ...
    [ "busokay t = T" ]

() : void
```

Using these values for rpc and busokay, the definition of Register yields an equation for the value of the system bus.

```
OK..
   ...
    [ "bus t = pc t" ]

() : void
```

Hence, the contents of the program counter pc are successfully transferred onto the bus. The data transfer is completed by writing the value of the bus into the memory address register mar. The contents of other registers in the datapath (ignoring buf) and the value of the interrupt acknowledge flag iack remain unchanged. These facts can be established from the above equations for the control signals and the definitions of Register and JKFF.

```
OK..
   ...
    [ "mar(t + 1) = pc t" ]
    [ "pc(t + 1) = pc t" ]
    [ "acc(t + 1) = acc t" ]
    [ "ir(t + 1) = ir t" ]
    [ "rtn(t + 1) = rtn t" ]
    [ "arg(t + 1) = arg t" ]
    [ "buf(t + 1) = alu t" ]
    [ "iack(t + 1) = iack t" ]

() : void
```

Finally, the control signals cause certain values to be generated as external outputs. The definitions of `OR` and `Interface` yield the following equations for `dreq` and `dataout` (an equation for `wmem` has already appeared in a previous step).

```
OK..
   ...
    [ "dreq t = F" ]
    [ "dataout t = wordn rep 0" ]

() : void
```

This completes the symbolic execution part of the proof: the results of symbolic execution are used to solve all the remaining goals of the backward proof. At the beginning of the symbolic execution part of the proof, the top level goal was:

```
"(val4 rep(mpc(t + 1)) =
  ((¬(idle t ∨ ¬dack t)) ⇒ 0 | ((ireq t ∧ ¬iack t) ⇒ 1 | 2))) ∧
 (mar(t + 1) = pc t) ∧
 (pc(t + 1) = pc t) ∧
 (acc(t + 1) = acc t) ∧
 (rtn(t + 1) = rtn t) ∧
 (iack(t + 1) = iack t) ∧
 (dataout t = wordn rep 0) ∧
 (wmem t = F) ∧
 (dreq t = F)"
```

But by the end of symbolic execution, most of the conjuncts in this goal have already been solved as a side-effect of various manipulations of the goal stack (e.g., uses of `ASM_REWRITE_TAC`). The original goal has been reduced to:

```
"val4
 rep
 ((¬(idle t ∨ ¬dack t)) ⇒
  mpc t |
  ((ireq t ∧ ¬iack t) ⇒ word4 rep 1 | word4 rep 2)) =
 ((¬(idle t ∨ ¬dack t)) ⇒ 0 | ((ireq t ∧ ¬iack t) ⇒ 1 | 2))"
```

This reduced goal is easily solved by case analysis on Boolean terms and using the assumption expressed by `Val4Word4_ASM`.

```
OK..
goal proved

 ... additional output deleted here ...

|- Val3_CASES_ASM rep ∧
   Val4Word4_ASM rep ∧
   TamarackImp
   rep
   (datain,dack,idle,ireq,mpc,mar,pc,acc,ir,rtn,arg,buf,iack,dataout,
    wmem,dreq,addr) ⟹
   (∀t.
     (((val4 rep) o mpc)t = 0) ⟹
     (((val4 rep) o mpc)(t + 1) =
      ((¬(idle t ∨ ¬dack t)) ⇒ 0 | ((ireq t ∧ ¬iack t) ⇒ 1 | 2))) ∧
     (mar(t + 1) = pc t) ∧
     (pc(t + 1) = pc t) ∧
     (acc(t + 1) = acc t) ∧
     (rtn(t + 1) = rtn t) ∧
     (iack(t + 1) = iack t) ∧
     (dataout t = wordn rep 0) ∧
     (wmem t = F) ∧
     (dreq t = F))

Previous subproof:
goal proved
() : void
```

A similar pattern of reasoning is used to derive correctness results for the remaining fifteen microinstructions. Each of these theorems shows that the microinstruction satisfies a set of equations for the next state of the microprocessor at time `t+1` in terms of its state at time `t`. Each theorem expresses the minimal (or close to minimal) set of results needed at higher proof levels; irrelevant details (such as the effect of microinstruction 0 on `arg` and `buf`) are ignored.

These sixteen correctness results collectively capture the informal description conveyed by the flow graph in Figure 3.7 and corresponding datapath actions described in Figure 3.8.

### 5.2.2 Microprogramming Level

The second main step of the verification procedure is to examine the interpretation of programming level operations by sequences of microinstructions. Here we consider the operation of the microprocessor in fully synchronous mode where each programming level operation is implemented by a fixed sequence of microinstructions, or in the case of a JZR instruction, by one of two possible sequences. Symbolic execution of these microinstruction sequences is used to show that their cumulative effect satisfies the semantics of the instruction set formally defined in Chapter 4. Symbolic execution of microinstructions is also used to show that hardware interrupts are correctly implemented.

A behavioural model for external memory was not needed to verify the phase level operation of the microinstruction since correctness results at that level are simply concerned with sampling and generating inputs and outputs on pins of the microchip. However, a behavioural model for external memory is needed to verify the microprogramming level. Whereas the terms `fetch` and `store` did not appear in the correctness results for the phase level, they do appear in correctness results for the microprogramming level.

The predicate `SynSystem` is defined below to specify a system consisting of the TAMARACK-3 microchip interfaced to a fully synchronous memory device. The specification of the memory device was given earlier in Chapter 4 by the definition of `SynMemory`. The internal architecture is made to operate in fully synchronous mode by wiring the two pins `dack` and `idle` to `T`, i.e., the voltage source `pwr`.

```
Define (
  "SynSystem (rep:rep_ty)
    (ireq,mpc,mar,pc,acc,ir,rtn,arg,buf,iack,mem) =
    ∃datain pwr dataout wmem dreq addr.
      TamarackImp rep (
        datain,pwr,pwr,ireq,mpc,mar,pc,
        acc,ir,rtn,arg,buf,iack,dataout,wmem,dreq,addr) ∧
      SynMemory rep (wmem,addr,dataout,mem,datain) ∧
      PWR pwr");;
```

Each programming level operation is interpreted by a non-empty, fixed sequence of microinstructions (or pair of possible sequences in the case of the JZR instruction). It is convenient to define the function `MicroCycles` which specifies the length of sequence about to be executed given the current state of the microprocessor (assuming that the microprocessor is operating in fully synchronous mode).

```
Define (
  "MicroCycles (rep:rep_ty) (ireq,mem,pc,acc,rtn:*wordn,iack) =
    (ireq ∧ ¬iack) ⇒ 3 |
    let opcval = OpcVal rep (mem,pc) in
    ((opcval = 0) ⇒ (((iszero rep) acc) ⇒ 5 | 6) |
     (opcval = 1) ⇒ 4 |
     (opcval = 2) ⇒ 8 |
     (opcval = 3) ⇒ 8 |
     (opcval = 4) ⇒ 6 |
     (opcval = 5) ⇒ 6 |
     (opcval = 6) ⇒ 4 |
                    5)");;
```

MicroCycles provides the basis for relating behaviour models of TAMARACK-3 expressed in terms of different granularities of time.[3] Later in Section 5.3, we describe a more general way of relating different granularities of time that does not involve MicroCycles.

To reason about the implementation of a particular programming level operation at the microprogramming level, we assume that the value of the microcode program counter mpc is equal to zero at time t, that is, at the beginning of the instruction cycle. The results of symbolic execution are used to determine the state of the microprocessor and memory at the end of the instruction cycle denoted as time t+m where m is the value returned by MicroCycles. Correctness results are obtained by showing that the state of the system at the end of the interpretation sequence is consistent with the definition of NextState. To eventually get rid of the assumption about the value of the microcode program counter mpc at the beginning of the cycle, we must also show that its value returns to zero at the end of the cycle.

These requirements could be translated directly into an individual statement of correctness for each programming level operation. But all of these correctness statements would be nearly identical except for minor (textual) distinctions such as the opcode value of the current instruction in each case. We can avoid a great deal of redundant text by defining a predicate SynCorrectness1 which is a generalized form of correctness statement. It is parameterized by the variables,

| | |
|---|---|
| ircond | - interrupt condition |
| opcval | - opcode value |
| iszerocond | - result of testzero operation |

which distinguish one programming level operation from another.

[3]The function MicroCycles is similar in purpose to the function NUMBER_OF_STEPS in the VIPER proof and the oracle function in the verification of FM8501 (except that the FM8501 oracle also takes account of handshaking delays).

```
Define (
  "SynCorrectness1 (rep:rep_ty) (ircond,opcval,iszerocond) =
    ∀ireq mpc mar pc acc ir rtn arg buf iack mem.
      Val3_CASES_ASM (rep:rep_ty) ∧
      Val4Word4_ASM rep ∧
      SynSystem rep (ireq,mpc,mar,pc,acc,ir,rtn,arg,buf,iack,mem)
      ⟹
      ∀t:time.
        (((val4 rep) o mpc) t = 0) ∧
        ((ireq t ∧ ¬(iack t)) = ircond) ∧
        (OpcVal rep (mem t,pc t) = opcval) ∧
        ((iszero rep) (acc t) = iszerocond)
        ⟹
        let m = MicroCycles rep (ireq t,mem t,pc t,acc t,rtn t,iack t) in
        ((((val4 rep) o mpc) (t+m) = 0) ∧
         ((mem (t+m),pc (t+m),acc (t+m),rtn (t+m),iack (t+m)) =
          NextState rep (ireq t,mem t,pc t,acc t,rtn t,iack t)))");;
```

To illustrate details of this verification step, we consider the sequence of microinstructions for an ADD instruction. As shown earlier in Chapter 3, the ADD instruction is interpreted in fully synchronous mode by the sequence of microinstructions stored in the microcode ROM at the following locations.

0, 2, 3, 6, 13, 15, 11 and 12

The ADD instruction case is specified by the assumption that the opcode of the current instruction is the ADD instruction opcode and that a hardware interrupt is not about to be processed in this cycle. Since the result of the `testzero` operation is not relevant in this case, the value for `iszerocond` is given by a variable `b`. The desired correctness result is expressed by the goal term in the following call to `set_goal`.

```
set_goal ([],"∀b. SynCorrectness1 (rep:rep_ty) (F,ADD_OPC,b)");;
```

We begin with a little backward proof to expand the correctness condition expressed by `SynCorrectness1` and move the expanded result into the assumption list. Definitions for `MicroCycles` and `NextState` along with some arithmetic facts are then used to simplify the goal for the ADD instruction case.

```
OK..
"let m = 8
 in
  ((((val4 rep) o mpc)(t + m) = 0) ∧
   (mem(t + m),pc(t + m),acc(t + m),rtn(t + m),iack(t + m) =
    ADD_SEM rep(mem t,pc t,acc t,rtn t,iack t)))"
    [ "Val3_CASES_ASM rep" ]
    [ "Val4Word4_ASM rep" ]
    [ "TamarackImp
       rep
       (datain,pwr,pwr,ireq,mpc,mar,pc,acc,ir,rtn,arg,buf,iack,dataout,
        wmem,dreq,addr)" ]
    [ "SynMemory rep(wmem,addr,dataout,mem,datain)" ]
    [ "∀t. pwr t = T" ]
    [ "((val4 rep) o mpc)t = 0" ]
    [ "ireq t ∧ ¬iack t = F" ]
    [ "val3 rep(opcode rep(fetch rep(mem t,address rep(pc t)))) = 2" ]
    [ "iszero rep(acc t) = b" ]

() : void
```

After this initial bit of backward proof, we begin the symbolic execution of the microcode for the ADD instruction. The symbolic execution begins with the value of the microcode program counter mpc equal to zero. We use correctness results for microinstruction 0 (obtained earlier from the verification of the phase level) to derive the state of the machine at time t+1.

```
OK..
    ...
    [ "iack(t + 1) = iack t" ]
    [ "rtn(t + 1) = rtn t" ]
    [ "acc(t + 1) = acc t" ]
    [ "pc(t + 1) = pc t" ]
    [ "mar(t + 1) = pc t" ]
    [ "mem(t + 1) = mem t" ]
    [ "((val4 rep) o mpc)(t + 1) = 2" ]

() : void
```

As we have assumed that the microprocessor is operating in fully synchronous mode and that an interrupt request is not about to be processed in this cycle, we can show that the value of the microcode program counter mpc becomes equal to two at time t+1. To advance the state of the symbolic execution to time t+2, we use the correctness results for microinstruction 2 and the behavioural model of external memory to obtain:

```
OK..
    ...
    [ "iack(t + 2) = iack t" ]
    [ "rtn(t + 2) = rtn t" ]
    [ "ir(t + 2) = fetch rep(mem t,address rep(pc t))" ]
    [ "acc(t + 2) = acc t" ]
    [ "pc(t + 2) = pc t" ]
    [ "mar(t + 2) = pc t" ]
    [ "mem(t + 2) = mem t" ]
    [ "((val4 rep) o mpc)(t + 2) = 3" ]

() : void
```

By time t+2, the instruction addressed by the program counter pc has been read from memory and loaded into the instruction register ir. The opcode field of the instruction register now contains the opcode for the current instruction which, in the case of ADD, has the value two. Correctness results for microinstruction 2 show that the address of the next microinstruction is obtained by adding four to the value of the opcode; hence, the address of the next microinstruction is six.

```
OK..
    ...
    [ "iack(t + 3) = iack t" ]
    [ "rtn(t + 3) = rtn t" ]
    [ "ir(t + 3) = fetch rep(mem t,address rep(pc t))" ]
    [ "acc(t + 3) = acc t" ]
    [ "pc(t + 3) = pc t" ]
    [ "mar(t + 3) = fetch rep(mem t,address rep(pc t))" ]
    [ "mem(t + 3) = mem t" ]
    [ "((val4 rep) o mpc)(t + 3) = 6" ]

() : void
```

This sequence continues with the symbolic execution of microinstruction at locations 6, 13, 15, 11 and 12 (in this order). New assumptions added to the assumption list for each step are shown below.

```
OK..
    ...
    [ "iack(t + 4) = iack t" ]
    [ "arg(t + 4) = acc t" ]
    [ "rtn(t + 4) = rtn t" ]
    [ "acc(t + 4) = acc t" ]
    [ "pc(t + 4) = pc t" ]
    [ "mar(t + 4) = fetch rep(mem t,address rep(pc t))" ]
    [ "mem(t + 4) = mem t" ]
    [ "((val4 rep) o mpc)(t + 4) = 13" ]

() : void
```

```
OK..
    ...
    [ "iack(t + 5) = iack t" ]
    [ "buf(t + 5) =
       add
       rep
       (acc t,
        fetch rep(mem t,address rep(fetch rep(mem t,address rep(pc t)))))" ]
    [ "arg(t + 5) = acc t" ]
    [ "rtn(t + 5) = rtn t" ]
    [ "acc(t + 5) = acc t" ]
    [ "pc(t + 5) = pc t" ]
    [ "mar(t + 5) = fetch rep(mem t,address rep(pc t))" ]
    [ "mem(t + 5) = mem t" ]
    [ "((val4 rep) o mpc)(t + 5) = 15" ]

() : void
```

```
OK..
    ...
    [ "iack(t + 6) = iack t" ]
    [ "rtn(t + 6) = rtn t" ]
    [ "acc(t + 6) =
       add
       rep
       (acc t,
        fetch rep(mem t,address rep(fetch rep(mem t,address rep(pc t)))))" ]
    [ "pc(t + 6) = pc t" ]
    [ "mem(t + 6) = mem t" ]
    [ "((val4 rep) o mpc)(t + 6) = 11" ]

() : void
```

```
OK..
    ...
    [ "iack(t + 7) = iack t" ]
    [ "buf(t + 7) = inc rep(pc t)" ]
    [ "rtn(t + 7) = rtn t" ]
    [ "acc(t + 7) =
       add
       rep
       (acc t,
        fetch rep(mem t,address rep(fetch rep(mem t,address rep(pc t)))))" ]
    [ "mem(t + 7) = mem t" ]
    [ "((val4 rep) o mpc)(t + 7) = 12" ]

() : void
```

Eventually we obtain a set of equations for the state of the system at the end of this particular execution sequence, that is, at time t+8. These equations describe the net effect of executing the microinstruction sequence which implements an ADD instruction.

```
OK..
    ...
    [ "iack(t + 8) = iack t" ]
    [ "rtn(t + 8) = rtn t" ]
    [ "acc(t + 8) =
       add
       rep
       (acc t,
        fetch rep(mem t,address rep(fetch rep(mem t,address rep(pc t)))))" ]
    [ "pc(t + 8) = inc rep(pc t)" ]
    [ "mem(t + 8) = mem t" ]
    [ "((val4 rep) o mpc)(t + 8) = 0" ]

() : void
```

At this point, the current goal is unchanged from the start of the symbolic execution part of this proof. We recall that this goal was:

```
"let m = 8
 in
  ((((val4 rep) o mpc)(t + m) = 0) ∧
   (mem(t + m),pc(t + m),acc(t + m),rtn(t + m),iack(t + m) =
    ADD_SEM rep(mem t,pc t,acc t,rtn t,iack t)))"
```

The backward proof is easily completed by expanding this goal with the definition of ADD_SEM and using the assumption list to satisfy the resulting subgoals.

```
OK..
goal proved

 ... additional output deleted here ...

|- ∀b. SynCorrectness1 rep(F,ADD_OPC,b)

Previous subproof:
goal proved
() : void
```

There are nine further cases to consider: one case for the JZR instruction when the value of the accumulator is zero; another case for JZR instruction when the accumulator is non-zero; one case for each of the six remaining instructions; and finally, a case for the processing of a hardware interrupt. Correctness results for these nine further cases are expressed by the following theorems:

```
|- ∀b. SynCorrectness1 rep(F,JZR_OPC,T)

|- ∀b. SynCorrectness1 rep(F,JZR_OPC,F)

|- ∀b. SynCorrectness1 rep(F,JMP_OPC,b)

|- ∀b. SynCorrectness1 rep(F,SUB_OPC,b)

|- ∀b. SynCorrectness1 rep(F,LDA_OPC,b)

|- ∀b. SynCorrectness1 rep(F,STA_OPC,b)

|- ∀b. SynCorrectness1 rep(F,RFI_OPC,b)

|- ∀b. SynCorrectness1 rep(F,NOP_OPC,b)

|- ∀n b. SynCorrectness1 rep(T,n,b)
```

The penultimate step in verifying the microprogramming level is to combine the above correctness results for individual programming level operations into a single theorem. The theorem shown below is obtained by case analysis on the three parameters of `SynCorrectness1`. Since `ircond` and `iszerocond` are Boolean variables, case analysis on these variables yields a finite number of cases to consider in the analysis. The variable `opcval`, a natural number, also yields a finite number of cases: either it is equal to one of the eight opcode values, i.e., $\texttt{opcval} < 8$, or it is not a valid opcode value, i.e., $8 \leq \texttt{opcval}$, in which case the theorem is vacuously true because of the condition expressed by `Val3_CASES_ASM`.

```
|- ∀b1 n b2. SynCorrectness1 rep(b1,n,b2)
```

The predicate SynCorrectness1 is useful as a parameterized correctness condition when the correctness of programming level operations are considered individually. But when these individual correctness results are combined in the above theorem, the parameterized variables of SynCorrectness1 no longer have a useful role. The final step in verifying the microprogramming level is to eliminate this overhead to obtain the following theorem.

```
|- Val3_CASES_ASM rep ∧
   Val4Word4_ASM rep ∧
   SynSystem rep (ireq,mpc,mar,pc,acc,ir,rtn,arg,buf,iack,mem)
   ⟹
   ∀t.
     (((val4 rep) o mpc) t = 0)
     ⟹
     let m = MicroCycles rep (ireq t,mem t,pc t,acc t,rtn t,iack t) in
      ((((val4 rep) o mpc) (t+m) = 0) ∧
       ((mem (t+m),pc (t+m),acc (t+m),rtn (t+m),iack (t+m)) =
        NextState rep (ireq t,mem t,pc t,acc t,rtn t,iack t)))
```

Although we extend our correctness proof with one more step to obtain the top level statement of correctness, the above theorem can stand alone as a substantial result about the correctness of the TAMARACK-3 design. For all possible instruction cycles, we have shown that the net effect of the instruction cycle corresponds to the state change specified by NextState and that MicroCycles correctly specifies the length of each instruction cycle.

### 5.2.3 Completing the Proof

As mentioned earlier in Chapter 4, the programming level operation and internal architecture are formally specified at different granularities of time. A formal relationship needs to be established between these two granularities of discrete time in order to complete the verification of TAMARACK-3 (operating in fully synchronous mode).

The chief source of difficulty in the formal definition of this timing relationship is that a unit of discrete time at the *abstract* programming level time scale does not correspond to a constant number of clock cycles on the *concrete* microprogramming level time scale. If every programming level operation was implemented by the same number of microinstructions, then the relationship between the two time scales could be expressed by a very simple arithmetic equation. However, this is not the case for our implementation of the microprocessor. For instance, the ADD instruction is implemented by a sequence of eight microinstructions as shown in Figure 5.1 whereas the JMP instruction is implemented by a sequence of four microinstructions as shown in Figure 5.2.

Instead, we define a primitive recursive function TimeOfCycle which computes the concrete time of every abstract time point using MicroCycles to determine the number of microinstructions executed between adjacent points on the abstract time scale.[4] To compute the concrete time of u+1, we recursively compute the concrete time of u and

[4]The definition of TimeOfCycle is actually just a 'wrapper' for the definition of CURRIED_TimeOfCycle;

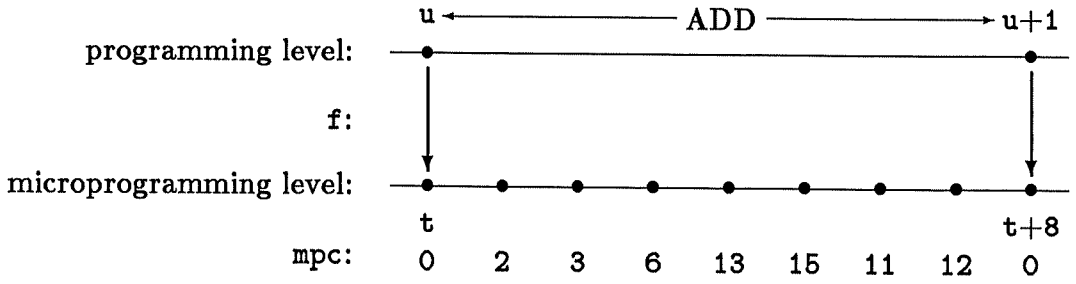



Figure 5.1: ADD Instruction Cycle Timing.

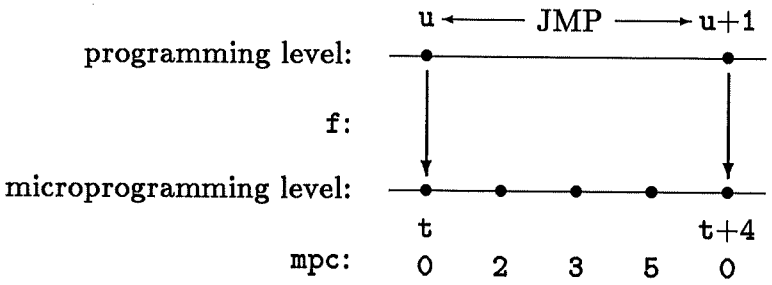



Figure 5.2: JMP Instruction Cycle Timing.

then add the number given by `MicroCycles` for the length of the next microinstruction sequence to be executed. As with the definition of `TamarackBeh` in Chapter 4, we emphasize the distinction between abstract and concrete time by using the variable `u` for abstract time and the variable `t` for concrete time.

```
Define (
  "(CURRIED_TimeOfCycle (rep:rep_ty) ireq mem pc acc rtn iack 0 = 0) ∧
   (CURRIED_TimeOfCycle rep ireq mem pc acc rtn iack (SUC u) =
    let t = CURRIED_TimeOfCycle rep ireq mem pc acc rtn iack u in
    (t + (MicroCycles rep (ireq t,mem t,pc t,acc t,rtn t,iack t))))");;

Define (
  "TimeOfCycle (rep:rep_ty) (ireq,mem,pc,acc,rtn,iack) u =
    CURRIED_TimeOfCycle rep ireq mem pc acc rtn iack u");;
```

The function denoted by,

```
TimeOfCycle (rep:rep_ty) (ireq,mem,pc,acc,rtn,iack)
```

this is a consequence of pragmatic restrictions imposed by the HOL system on the format of parameter lists in primitive recursive definitions. The definition of `CURRIED_TimeOfCycle` uses the successor function `SUC` since this is the format expected by the HOL system, i.e., `(SUC u)` instead of `(u+1)`.

is the desired mapping from abstract time to concrete time represented by the downward arrows in Figures 5.1 and 5.2. For instance, in the ADD microinstruction sequence where t is the concrete time of u, the next point on the abstract time scale u+1, is mapped to t+8 on the concrete time scale. In the JMP microinstruction sequence, u+1 is mapped to t+4.

Using this specification of the relationship between abstract and concrete time scales, we can derive correctness results expressed in terms of the abstract time scale from results already obtained which are expressed in terms of the concrete time scale. In particular, we want to show that from one abstract time point to the next, the microprocessor executes a single programming level instruction.

We first need to show that every point on the abstract time scale corresponds to the start of a microinstruction sequence. It is sufficient to show that every abstract time point maps to a concrete time point when the microcode program counter is zero since every microinstruction sequence begins at this location. It is necessary to assume that the microcode program counter initially has the value zero; that is, time begins when the microcode program counter is reset. The following theorem can be proved by mathematical induction on u.

```
|- Val3_CASES_ASM (rep:rep_ty) ∧
   Val4Word4_ASM rep ∧
   SynSystem rep (ireq,mpc,mar,pc,acc,ir,rtn,arg,buf,iack,mem) ∧
   (((val4 rep) o mpc) 0 = 0)
   ⟹
   let f = TimeOfCycle rep (ireq,mem,pc,acc,rtn,iack) in
   ∀u. ((val4 rep) o mpc) (f u) = 0");;
```

Once we have shown that every point on the abstract time scale corresponds to the beginning of an instruction cycle, it is a relatively simple matter to complete the last step in this version of the correctness proof for TAMARACK-3. The top-level correctness theorem is expressed by the goal term in the following call to set_goal.

```
set_goal (
  [],
  "Val3_CASES_ASM (rep:rep_ty) ∧
   Val4Word4_ASM rep ∧
   SynSystem rep (ireq,mpc,mar,pc,acc,ir,rtn,arg,buf,iack,mem) ∧
   (((val4 rep) o mpc) 0 = 0)
   ⟹
   let f = TimeOfCycle rep (ireq,mem,pc,acc,rtn,iack) in
   TamarackBeh rep (ireq o f,mem o f,pc o f,acc o f,rtn o f,iack o f)");;
```

We begin the backward proof for this theorem by expanding the goal with the definition of TamarackBeh. We can see from the result of this step that the proof is simply a matter of combining correctness results for the microprogramming level with facts derived from the definition of TimeOfCycle. In a nutshell, we must show that the externally visible state of the microprocessor at time u+1 is determined by the function NextState from its state at time u.

```
OK..
"Val3_CASES_ASM rep ∧
 Val4Word4_ASM rep ∧
 SynSystem rep(ireq,mpc,mar,pc,acc,ir,rtn,arg,buf,iack,mem) ∧
 (((val4 rep) o mpc)0 = 0) ⟹
 let f = TimeOfCycle rep(ireq,mem,pc,acc,rtn,iack)
 in
  (∀u.
    (mem o f)(u + 1),(pc o f)(u + 1),(acc o f)(u + 1),(rtn o f)(u + 1),
    (iack o f)(u + 1) =
    NextState
    rep
    ((ireq o f)u,(mem o f)u,(pc o f)u,(acc o f)u,(rtn o f)u,(iack o f)u))"

() : void
```

Next, we use standard HOL techniques to further expand the goal and move antecedents into the assumption list. However, the use of standard goal reduction techniques alone would result in a goal with some cumbersome sub-terms. These particular sub-terms are unnecessary and it is helpful (as a technique for managing proof complexity) to replace them with simple variables, namely, `t` and `m`. These variables are introduced by means of the following two theorems (which are trivial facts of logic).

```
|- ∃t. TimeOfCycle rep(ireq,mem,pc,acc,rtn,iack)u = t

|- ∃m. MicroCycles rep(ireq t,mem t,pc t,acc t,rtn t,iack t) = m
```

After these simplifications, the new goal is:

```
OK..
"mem(t + m),pc(t + m),acc(t + m),rtn(t + m),iack(t + m) =
 NextState rep(ireq t,mem t,pc t,acc t,rtn t,iack t)"
    [ "Val3_CASES_ASM rep" ]
    [ "Val4Word4_ASM rep" ]
    [ "SynSystem rep(ireq,mpc,mar,pc,acc,ir,rtn,arg,buf,iack,mem)" ]
    [ "((val4 rep) o mpc)0 = 0" ]
    [ "TimeOfCycle rep(ireq,mem,pc,acc,rtn,iack)u = t" ]
    [ "MicroCycles rep(ireq t,mem t,pc t,acc t,rtn t,iack t) = m" ]

() : void
```

We have already shown that every point on the abstract time scale corresponds to the beginning of an instruction cycle. In particular, we know that the value of the microcode program counter `mpc` at time `t` is 0.

```
OK..
...
    [ "((val4 rep) o mpc)t = 0" ]

() : void
```

We have also shown (in Section 5.2.2) that the externally visible state of the microprocessor at the end of the instruction cycle is related to its initial state at the beginning of the instruction cycle by the function NextState. This fact is used to solve the current goal and complete the final step in this version of the correctness proof for TAMARACK-3.

```
OK..

 ... additional output deleted here ...

goal proved
|- Val3_CASES_ASM rep ∧
   Val4Word4_ASM rep ∧
   SynSystem rep(ireq,mpc,mar,pc,acc,ir,rtn,arg,buf,iack,mem) ∧
   (((val4 rep) o mpc)0 = 0) ⟹
   let f = TimeOfCycle rep(ireq,mem,pc,acc,rtn,iack)
   in
    TamarackBeh rep(ireq o f,mem o f,pc o f,acc o f,rtn o f,iack o f)

Previous subproof:
goal proved
() : void
```

We recall that the predicate SynSystem specifies the implementation of a synchronous TAMARACK-3 system in terms of TamarackImp, SynMemory and PWR. The above theorem relates the formal specification of this system to its behavioural specification given by the predicate TamarackBeh. In particular, we show that the constraints imposed by TamarackImp on the register-transfer level signals of the internal architecture, together with behavioural models for the memory and voltage source, satisfy the constraints imposed by TamarackBeh on the corresponding programming level signals.

## 5.3 Synchronizing Multiple Levels of Timing

The simple approach just described in Section 5.2.3 for relating different granularities of time is not very general because it depends on the implementation of programming level operations by fixed microinstruction sequences whose predetermined lengths are specified by the definition of MicroCycles. In general, programming level operations may not necessarily be implemented at lower levels by fixed sequences.

An example of when it is not possible to define a simple function like MicroCycles to compute the exact length of each instruction cycle is the operation of TAMARACK-3

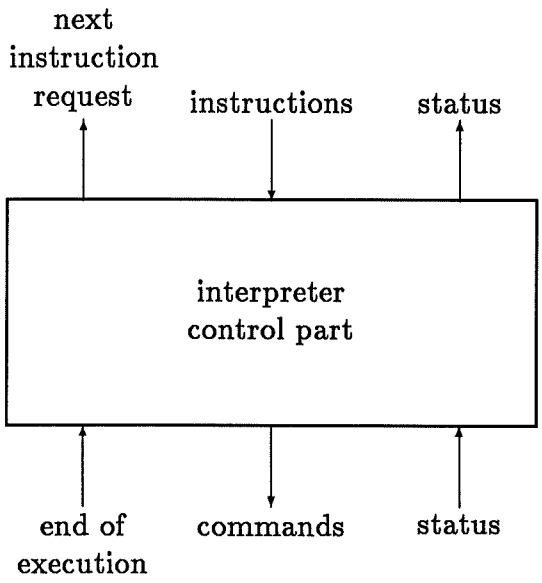



Figure 5.3: Control Part of an Interpreter.

in fully asynchronous mode when delayed handshaking signals may cause microinstructions to be repeated. Although there are ways to partially circumvent this problem using an oracle function to 'guess' the length of each instruction cycle, the result may not be completely general[5] or correspond very well to an intuitive view of this timing relationship.

The term *temporal abstraction* (or *temporal projection*) has been used in verification circles [43,45,65,72,100,112,133] to denote the idea of relating different rates of computation in a formal specification. This idea naturally arises from the more general concept of viewing computer hardware as a hierarchy of interpretation levels where a timing relationship between two levels of interpretation is sometimes called a *synchronization scheme* (or *synchronization mechanism*). The control part of each level in the interpretation hierarchy can be viewed as a virtual machine with the generalized form shown in Figure 5.3. To paraphrase the description by Anceau in [2]:

> A given level is only allowed to generate a new command once it has received an end-of-execution signal for the present command from the levels below. In turn, it sends a similar signal to the control part immediately above requesting the next instruction. The end-of-instruction signal received by the lowest level of synchronous logic is simply the clock signal of the machine. The next-instruction signal issued by the microprogramming level implies the fetching of the next instruction from memory.

[5] If all functions must be total, then use of an oracle function to guess the length of each instruction cycle implies that every instruction cycle is finite, in particular, that handshaking sequences always run to completion. A completely general approach would not imply that every handshaking sequence terminates but, instead, would require (as we do) that this fact be established by formal proof.

In this section, we describe a formalization which corresponds directly to the imagery of Figure 5.3 where a next-instruction signal at the microprogramming level implies the fetching of the next programming level instruction from memory. At this level, the next-instruction signal is not a physical signal, but rather, a time-dependent condition on the internal state of the machine.

In fully synchronous mode, the end of the instruction cycle occurs when the value of the microprocessor program counter returns to zero. Hence, the next-instruction signal generated by the microprogramming level is the time-dependent condition expressed by the term,

```
((val4 rep) o mpc) Eq 0
```

where Eq is an infix temporal operator for relating a signal to a particular value in terms of equality.

```
Define ("(P:time→*) Eq c = λt. P t = c");;
```

If an instruction cycle begins at time t, then it terminates at time t+m if and only if t+m is the first time that the microcode program counter mpc becomes equal to zero since time t. This condition is expressed by the term,

```
Next (((val4 rep) o mpc) Eq 0) (t,t+m)
```

where the predicate Next is defined as:

```
Define (
  "Next g (t1,t2) = t1 < t2 ∧ (∀t. t1 < t ∧ t < t2 ⟹ ¬(g t)) ∧ (g t2)");;
```

It is possible to show that this condition holds for each of the fixed microinstruction sequences that implement programming level operations in fully synchronous mode. In place of the predicate SynCorrectness1, a slightly different form of parameterized correctness condition is used to verify the implementation of each programming level operation.

```
Define (
  "SynCorrectness2 (rep:rep_ty) (ircond,opcval,iszerocond) =
    ∀ireq mpc mar pc acc ir rtn arg buf iack mem.
      Val3_CASES_ASM (rep:rep_ty) ∧
      Val4Word4_ASM rep ∧
      SynSystem rep (ireq,mpc,mar,pc,acc,ir,rtn,arg,buf,iack,mem)
      ⟹
      ∀t:time.
        (((val4 rep) o mpc) t = 0) ∧
        ((ireq t ∧ ¬(iack t)) = ircond) ∧
        (OpcVal rep (mem t,pc t) = opcval) ∧
        ((iszero rep) (acc t) = iszerocond)
        ⟹
        ∃m.
          Next (((val4 rep) o mpc) Eq 0) (t,t+m) ∧
          ((mem (t+m),pc (t+m),acc (t+m),rtn (t+m),iack (t+m)) =
           NextState rep (ireq t,mem t,pc t,acc t,rtn t,iack t))");;
```

In this version of the correctness condition, the length of the instruction cycle is an existentially quantified variable instead of a value computed by `MicroCycles`. The proofs of correctness for each programming level operation are almost the same as before except that we must keep track of the microcode program counter `mpc` throughout the instruction cycle and show that it is never equal to zero before the end of the instruction cycle. Combining individual correctness results for each programming level operation and eliminating, as before, the overhead of a parameterized correctness condition yields the following theorem.

```
|- Val3_CASES_ASM (rep:rep_ty) ∧
   Val4Word4_ASM rep ∧
   SynSystem rep (ireq,mpc,mar,pc,acc,ir,rtn,arg,buf,iack,mem)
   ⟹
   ∀t.
     (((val4 rep) o mpc) t = 0)
     ⟹
     ∃m.
       Next (((val4 rep) o mpc) Eq 0) (t,t+m) ∧
       ((mem (t+m),pc (t+m),acc (t+m),rtn (t+m),iack (t+m)) =
        NextState rep (ireq t,mem t,pc t,acc t,rtn t,iack t))
```

The next step is to define a general purpose function `TimeOf` which uses a next-instruction signal `g` to construct a mapping `f` from an abstract time scale to a concrete time scale. The relationship between abstract and concrete time is shown in Figure 5.4. The following definition of `TimeOf` is similar to definitions given in previous work by Dhingra [43], Herbert [72,73] and Melham [100] (the name '`TimeOf`' is used here for

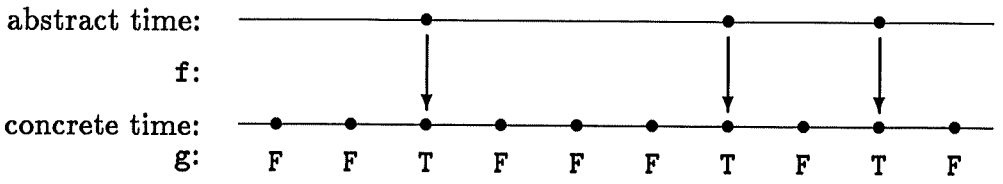



Figure 5.4: Abstract and Concrete Time Scales.

continuity with previous work). It is defined in terms of the predicates `First` and `Next` using primitive recursion and Hilbert's $\varepsilon$-operator.[6]

```
Define ("First g t = (∀p. p < t ⟹ ¬(g p)) ∧ (g t)");;

Define (
  "(TimeOf g 0 = εt. First g t) ∧
   (TimeOf g (SUC u) = εt. Next g (TimeOf g u,t))");;
```

The first point on the abstract time scale corresponds to the first time that the next-instruction signal g is true with respect to the concrete time scale. Subsequent points on the abstract time scale are defined recursively. The next point after u on the abstract time scale, i.e., u+1, corresponds to the next time that the next-instruction signal g becomes true with respect to the concrete time scale.

Applying `TimeOf` to the next-instruction signal generated at the microprogramming level of TAMARACK-3,

```
TimeOf (((val4 rep) o mpc) Eq 0)
```

yields a mapping from points on the abstract time scale to points on the concrete time scale. This mapping is the synchronization scheme which relates behaviour at the abstract programming level time scale to behaviour at the concrete microprogramming level time scale.

As with most definitions that use Hilbert's $\varepsilon$-operator, some additional proof infrastructure is needed to reason about specifications involving `TimeOf`. For instance, it is necessary to establish the fact that the next-instruction signal is true infinitely often [78,100]. Fortunately, the main result of this proof infrastructure can be summed up in a compact theorem:

```
|- ∀g r.
     (∃t. g t) ∧ (∀t. g t ⟹ ∃m. Next g (t,t+m) ∧ r (t,t+m))
     ⟹
     ∀u. r (TimeOf g u,TimeOf g (u+1))
```

[6] Our definition of `TimeOf` is simplified by avoiding the existential quantification found in corresponding definitions given by [43,72,73,100]. When the constraint that g is 'true infinitely often' holds, this function yields exactly the same mapping from abstract time to concrete time. When this constraint does not hold, both versions (with and without existential quantification) yield an incomplete mapping.

For any next-instruction signal g and next state relation r, this theorem can be used to reduce the problem of establishing,

```
∀u. r (TimeOf g u,TimeOf g (u+1))
```

to a pair of simpler problems:

```
∃t. g t
∀t. g t ⟹ ∃m. Next g (t,t+m) ∧ r (t,t+m)
```

The last step in verifying TAMARACK-3 is essentially a question about the validity of the synchronization scheme relating behaviour at the programming level time scale to behaviour at the microprogramming level time scale. The above theorem provides a way to translate this question into a pair of simpler problems. In particular, it is used mid-way through a backward proof of the top-level correctness statement to reduce the goal,

```
OK..
"∀u.
  mem(TimeOf g(u + 1)),pc(TimeOf g(u + 1)),acc(TimeOf g(u + 1)),
  rtn(TimeOf g(u + 1)),iack(TimeOf g(u + 1)) =
  NextState
  rep
  (ireq(TimeOf g u),mem(TimeOf g u),pc(TimeOf g u),acc(TimeOf g u),
   rtn(TimeOf g u),iack(TimeOf g u))"
    [ "((val4 rep) o mpc) Eq 0 = g" ]
    [ "Val3_CASES_ASM rep" ]
    [ "Val4Word4_ASM rep" ]
    [ "SynSystem rep(ireq,mpc,mar,pc,acc,ir,rtn,arg,buf,iack,mem)" ]
    [ "∃t. ((val4 rep) o mpc)t = 0" ]

() : void
```

to the following pair of sub-goals.

```
"∃t. g t"
    [ "((val4 rep) o mpc) Eq 0 = g" ]
    [ "Val3_CASES_ASM rep" ]
    [ "Val4Word4_ASM rep" ]
    [ "SynSystem rep(ireq,mpc,mar,pc,acc,ir,rtn,arg,buf,iack,mem)" ]
    [ "∃t. ((val4 rep) o mpc)t = 0" ]

() : void
```

```
"∀t.
  g t ⟹
  (∃m.
    Next g(t,t+m) ∧
    (mem(t + m),pc(t + m),acc(t + m),rtn(t + m),iack(t + m) =
     NextState rep(ireq t,mem t,pc t,acc t,rtn t,iack t)))"
    [ "((val4 rep) o mpc) Eq 0 = g" ]
    [ "Val3_CASES_ASM rep" ]
    [ "Val4Word4_ASM rep" ]
    [ "SynSystem rep(ireq,mpc,mar,pc,acc,ir,rtn,arg,buf,iack,mem)" ]
    [ "∃t. ((val4 rep) o mpc)t = 0" ]

() : void
```

The first sub-goal is satisfied by members of the assumption list combined with the definition of Eq. The second sub-goal is satisfied by previously mentioned correctness results for the microprogramming level. The satisfaction of these two sub-goals yields the following correctness theorem for TAMARACK-3.

```
|- Val3_CASES_ASM (rep:rep_ty) ∧
   Val4Word4_ASM rep ∧
   SynSystem rep (ireq,mpc,mar,pc,acc,ir,rtn,arg,buf,iack,mem) ∧
   (∃t. ((val4 rep) o mpc) t = 0)
   ⟹
   let f = TimeOf (((val4 rep) o mpc) Eq 0) in
   TamarackBeh rep (ireq o f,mem o f,pc o f,acc o f,rtn o f,iack o f)
```

This is nearly identical to the previous version of the top level correctness statement in Section 5.2.3 except for the sub-term which specifies the relationship between the two time scales.

Chapter 6

# Embedding Other Notations

The ability to embed notation from formalisms is a major advantage of using higher-order logic to reason about microprocessor-based systems. We illustrate this point by embedding a form of temporal logic in the HOL logic for the purpose of verifying the operation of TAMARACK-3 in fully asynchronous mode, i.e., the use of handshaking signals to synchronize the transfer of data between the microprocessor and external memory.

The main specification problem in this chapter is to define a behavioural model for fully asynchronous memory which is both succinct and independent from the microprocessor specification. The main verification problem is to derive an abstract view of the instruction cycle where handshaking interactions are collapsed into fixed sequences of steps.

Temporal logic is especially useful for specifying and reasoning about relative order in a set of events. We take advantage of this natural notation by regarding a set of temporal logic operators as abbreviations for higher-order functions. This approach results in succinct specifications and simplifies the verification task by hiding some low-level aspects of proof in derived inference rules for temporal logic operators.

## 6.1 Specification Using Temporal Logic Operators

We use the temporal logic operators,

- $\Box$ - "henceforth"
- $\Diamond$ - "eventually"
- $\bigcirc$ - "next"
- $\cup$ - "until"

and a number of connectives ($\sim$, $\longrightarrow$, and) corresponding to logical connectives in propositional logic. Later on, we give precise definitions for these operators and connectives as abbreviations for higher-order functions. For now, the intuitive meanings suggested above are enough to illustrate the use of these operators in the formal specification of a handshaking protocol.

Figure 6.1 shows a stylized timing diagram for a four-phase handshaking sequence. A request at time `t1` by the sender is acknowledged at time `t2` by the receiver. A request to end the interaction is signaled at time `t3` and eventually acknowledged at time `t4`.

The handshaking protocol is formally specified by a number of constraints imposed on both the sender and receiver. For now, these can be understood as constraints imposed on the sender and receiver viewed together as a single system; later on, we say which constraints are imposed on which process.

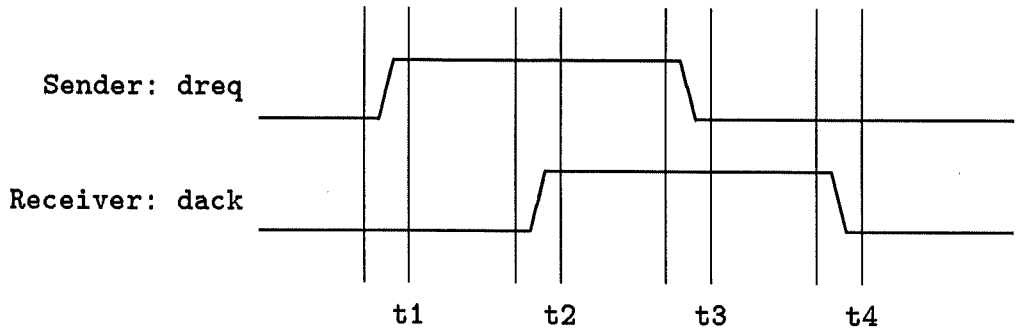



Figure 6.1: Four-Phase Handshaking Timing Diagram.

A handshaking interaction is initiated whenever dreq becomes true. dreq must continue to be true until this is acknowledged. This constraint is expressed by:

$$(\text{dreq} \longrightarrow (\text{dreq} \cup \text{dack}))$$

A request must eventually be acknowledged. Furthermore, once dack becomes true, it must remain true until dreq becomes false.

$$(\text{dreq} \longrightarrow (\Diamond\ \text{dack}))$$
$$(\text{dack} \longrightarrow (\text{dack} \cup (\sim\text{dreq})))$$

Once a request has been acknowledged, dreq must eventually become false again to signal the end of the interaction. The next request cannot be signaled until dack also returns to false.

$$(\text{dack} \longrightarrow (\Diamond\ (\sim\text{dreq})))$$
$$((\sim\text{dreq}) \longrightarrow ((\sim\text{dreq}) \cup (\sim\text{dack})))$$

When dreq becomes false, dack must also become false and stay false until the next request.

$$((\sim\text{dreq}) \longrightarrow (\Diamond\ (\sim\text{dack})))$$
$$((\sim\text{dack}) \longrightarrow ((\sim\text{dack}) \cup \text{dreq}))$$

Upon completion of the handshaking interaction, the protocol requires another request to be initiated sometime in the future. This final constraint is usually included in the handshaking protocol specification to obtain the property that system activity never ceases.

$$((\sim\text{dack}) \longrightarrow (\Diamond\ \text{dreq}))$$

The use of temporal logic operators to specify the relative order of events in a handshaking protocol has been described by others including Bochman [11], Dill and Clarke [44], and Fujita et al. [49].

## 6.2 Temporal Logic in Higher-Order Logic

Temporal logics are often treated as primitive systems defined axiomatically or based on semantic definitions for temporal logic operators. However, our higher-order logic framework allows us to regard temporal logic operators as simply abbreviations for higher-order functions. The following definitions yield temporal logic operators for a form of temporal logic known as *Linear Temporal Logic*.

```
Define ("□ P = λt. ∀n. P (t+n)");;

Define ("◇ P = λt. ∃n. P (t+n)");;

Define ("○ P = λt. ((P (t+1)):bool)");;

Define ("P ∪ Q = λt. ∀n. (∀m. m < n ⟹ ¬(Q (t+m))) ⟹ P (t+n)");;

Define ("~P = λt. ¬(P t)");;

Define ("P ⟶ Q = λt. P t ⟹ Q t");;

Define ("P and Q = λt. P t ∧ Q t");;
```

Each operator is defined in terms of a function which maps discrete points in time, modelled by the natural numbers, to Boolean values. These operators can be combined with variables such as P and Q to form assertions in temporal logic. In both first-order logic and higher-order logic, every assertion, for instance,

$$\forall b.\ b \vee \neg b$$

is a Boolean expression which is either true or false. However, an assertion in temporal logic such as,

$$(P \longrightarrow \Diamond\ (Q \cup R))$$

is only true or false relative to a particular instant of time. When stated as an assertion, this is informally understood to mean that the assertion is true at every instant of time. However, to formally represent temporal logic assertions as assertions in our higher-order logic framework, we introduce a notion of validity[1] where a temporal logic assertion is valid if and only if it is true at all times.

```
Define ("VALID P = ∀t. P t");;
```

[1]Although this definition of validity allows us to express temporal logic assertions in higher-order logic (and is essentially the same as the approach used by Hale [65]), it does not capture the 'whole meaning' of validity in a model-theoretic sense [66].

Powerful inference rules for direct manipulation of temporal logic assertions can be derived in higher-order logic from the above definitions. Many such rules effectively 'package up' what would otherwise be tedious and repetitive patterns of inference. For instance, the following theorem provides a particularly useful rule; this rule achieves, in a single step, an inference which would otherwise involve a proof by mathematical induction.

```
|- ∀P Q. VALID((P and (~Q)) ⟶ ( ○ P)) ⟹ VALID(P ⟶ (P ∪ Q))
```

A recent survey article [22] describes 'traditional logic' (which includes higher-order logic) and temporal logic as alternative kinds of pure formalism for reasoning about hardware. But when temporal logic operators are simply abbreviations for higher-order functions, anything which can be done with the temporal logic operators can also be done without them using explicit time variables. In fact, we have taken a mixed-mode approach of using both temporal operators and explicit time variables. The right mixture of temporal operators and explicit time variables yields relatively succinct specifications and much easier proofs.

Previous work by Hale [65] demonstrated that the idea of embedding a temporal logic in higher-order logic was of practical use. Leeser [93] has also embedded temporal logic in a proof-generation environment to reason about hardware. Both Hale and Leeser used another form of temporal logic called *Interval Temporal Logic* developed by Moszkowski [112,113] to reason about digital systems in general. However, the form of temporal logic captured in our definitions for $\Box$, $\Diamond$,$\bigcirc$, etc., is adequate for the very specific purpose of reasoning about handshaking interactions between TAMARACK-3 and external memory. Moreover, this form of temporal logic is easier to embed in higher-order logic.

## 6.3 Sender and Receiver Specifications

Earlier we gave a formal specification of the handshaking protocol in terms of constraints expressed by a set of temporal logic assertions. Seitz [127] distinguishes between *functional constraints* on outputs which must be satisfied by a process and *domain constraints* which are allowable assumptions about inputs. In the handshaking protocol, constraints imposed on `dreq` are functional constraints for the sender process and domain constraints for the receiver process. Constraints imposed on `dack` are functional constraints for the receiver process and domain constraints for the sender process.

It turns out that only some of the domain constraints are actually needed in each case. Furthermore, these domain constraints are only needed to establish specific functional constraints. For example, the domain constraint,

```
(dreq ⟶ (dreq ∪ dack))
```

that `dreq`, once true, must remain true can be assumed in showing that the receiver satisfies the functional constraint,

```
(dreq ⟶ (◇ dack))
```

that it will detect and eventually acknowledge the request.

The following definitions give the functional constraints for the sender and receiver respectively and, where required, the domain constraints which can be assumed in showing that a process satisfies a particular functional constraint. An equivalent formulation of these constraints could be given using □ and **and** instead of VALID and ∧ but this would only be a matter of personal taste since the result (enclosed by VALID) would be equivalent to the definitions shown below.

```
Define (
  "Sender (dreq,dack) =
    VALID (dreq ⟶ (dreq ∪ dack)) ∧
    VALID ((~dreq) ⟶ ((~dreq) ∪ (~dack))) ∧
    (VALID (dack ⟶ (dack ∪ (~dreq))) ⟹
     VALID (dack ⟶ (◇ (~dreq)))) ∧
    (VALID ((~dack) ⟶ ((~dack) ∪ dreq)) ⟹
     VALID ((~dack) ⟶ (◇ dreq)))");;

Define (
  "Receiver (dreq,dack) =
    VALID (dack ⟶ (dack ∪ (~dreq))) ∧
    VALID ((~dack)  ⟶ ((~dack) ∪ dreq)) ∧
    (VALID (dreq ⟶ (dreq ∪ dack)) ⟹
     VALID (dreq ⟶ (◇ dack))) ∧
    (VALID ((~dreq) ⟶ ((~dreq) ∪ (~dack))) ⟹
     VALID ((~dreq) ⟶ (◇ (~dack))))");;
```

When the sender and receiver parts of the handshaking protocol specification are both satisfied, this results in the set of constraints mentioned at the beginning of this section for the system as a whole. This is shown by the following theorem.

```
|- Sender(dreq,dack) ∧
   Receiver(dreq,dack)
   ⟹
   VALID (dreq ⟶ (dreq ∪ dack)) ∧
   VALID (dreq ⟶ (◇ dack)) ∧
   VALID (dack ⟶ (dack ∪ (~dreq))) ∧
   VALID (dack ⟶ (◇ (~dreq))) ∧
   VALID ((~dreq) ⟶ ((~dreq) ∪ (~dack))) ∧
   VALID ((~dreq) ⟶ (◇ (~dack))) ∧
   VALID ((~dack) ⟶ ((~dack) ∪ dreq)) ∧
   VALID ((~dack) ⟶ (◇ dreq))
```

## 6.4 Memory Specification

The predicate `Receiver` only specifies constraints on the acknowledgement signal `dack`. The rest of the memory specification is given by the definitions of `ReceiverData`,

ReadFunc, WriteFunc and AsynMemory. These predicates specify constraints on the address and data signals in relation to the handshaking signals.

First of all, it is convenient to introduce another temporal operator to express assertions of the form "P will be true if and when Q becomes true". This operator is defined directly in terms of $\longrightarrow$ and $\cup$.

```
Define ("P when Q = ((Q ⟶ P) ∪ Q)");;
```

The behavioural specification of asynchronous memory is actually a very localized case of generic specification: we specify a set of generic constraints on data transfer for a generic receiver process and later instantiate this specification for the particular requirements of the TAMARACK-3 memory interface. One advantage of this approach is that the generic specification of the receiver processor could be re-used to model other devices besides the asynchronous memory device, e.g., a sensor or actuator in a real-time control system. However, the main reason for using a generic specification in this case is to produce a more readable specification by filtering out details about the precise nature of the data being transferred between TAMARACK-3 and external memory.[2]

In particular, we are able to treat the three separate signals wmem, dataout and addr of the TAMARACK-3 interface as one signal: in this view, the item of data sent to external memory actually consists of three sub-items, namely, the read/write flag, memory data (only needed for a write operation), and a memory address. Grouping these three signals into one signal avoids repetitiously specifying constraints for each sub-item.

In a generic view of data transfer, the sender transfers a single item of data to the receiver and the receiver eventually replies with another item of data. We use the uninterpreted data types ** and *** for data produced by the sender and receiver respectively. Additionally, we use the uninterpreted data type * for the internal state of the receiver. These uninterpreted types are just type variables in the HOL logic.

The uninterpreted primitives f1 and f2 denote operations performed by the receiver on the uninterpreted data types *, ** and ***. f1 denotes the operation performed by the receiver to compute its reply to the sender. f2 denotes the operation performed by the receiver to update its internal state as a result of an interaction with the sender. The polymorphic types of f1 and f2 are:

f1:(*×**)→***

f2:(*×**)→*

A set of constraints for data transfer is captured in the following definition of ReceiverData. The uninterpreted primitives f1 and f2 appear explicitly as parameters in this definition. To be consistent with earlier uses of the names datain and dataout, data is sent to the generic receiver from the sender on the dataout bus while the reply to the sender is sent on the datain bus.

[2]Cardell-Oliver [23] has described a similar idea in using the HOL logic to specify a generalized sliding window protocol based on properties common to a family of protocol implementations.

```
Define (
  "ReceiverData (f1:(*×**)→***,f2) (req,ack,mem,dataout,datain) =
    (∀x y.
      VALID (
        ((~req) and (◯ req)) ⟶
        (◯ (mem Eq x)) ⟶
        (◯ ((dataout Eq y) ⟶ ((dataout Eq y) ∪ ack))) ⟶
        (◯ ((((datain Eq (f1 (x,y))) ∪ (~req)) when ack) and
            (((mem Eq (f2 (x,y))) ∪ (~req)) when ack))))) ∧
    (∀x. VALID (((~req) and (mem Eq x)) ⟶ (◯ (mem Eq x))))");;
```

The definition of ReceiverData captures the essential features of how data is transferred between a sender and receiver by means of handshaking signals. Paraphrasing this specification, suppose that the sender initiates an interaction with the receiver:

```
((~req) and (◯ req))
```

Also suppose that the internal state (called mem) of the receiver process is equal to x and that the value of the incoming data is stable and equal to y:

```
(◯ (mem Eq x))
(◯ ((dataout Eq y) ⟶ ((dataout Eq y) ∪ ack)))
```

If these conditions hold, then datain will be equal to f1(x,y) and the new value of the internal state will be f2(x,y) when ack becomes true. Furthermore, these values will remain stable at least until req becomes false:

```
(◯ ((((datain Eq (f1 (x,y))) ∪ (~req)) when ack) and
    (((mem Eq (f2 (x,y))) ∪ (~req)) when ack)))
```

Finally, the internal state of the receiver will be stable as long as req remains false:

```
∀x. VALID (((~req) and (mem Eq x)) ⟶ (◯ (mem Eq x)))
```

This generic specification is specialized for the TAMARACK-3 memory interfacing by using the functions ReadFunc and WriteFunc as values for the uninterpreted primitives f1 and f2. We note that WriteFunc and ReadFunc are also generic specifications; the definition of AsynMemory illustrates how one generic specification can be a refinement (or refine aspects of) another generic specification.

```
Define (
  "ReadFunc (rep:rep_ty) (memval,(wmemval,addrval,datainval)) =
    (wmemval ⇒ datainval | ((fetch rep) (memval,addrval)))");;

Define (
  "WriteFunc (rep:rep_ty) (memval,(wmemval,addrval,datainval)) =
    (wmemval ⇒ ((store rep) (memval,addrval,datainval)) | memval)");;
```

```
Define (
  "AsynMemory (rep:rep_ty) (
    req,ack,mem,wmem,addr,datain,dataout) =
    Receiver (req,ack) ∧
    ReceiverData (ReadFunc rep,WriteFunc rep)
      (req,ack,mem,(λt:time.(wmem t,addr t,datain t)),dataout)");;
```

The memory specification is a formal rendering of the description given earlier in Chapter 3. In particular, it is precise statement about details such as the constraint that:

> After signaling a memory request, dreq must remain equal to T and the wmem flag, address bus addr and dataout bus dataout must remain at stable values until dack becomes equal to T signaling that the request has been satisfied.

As with the specification of fully synchronous memory, SynMemory, the operations performed by memory are selected from the representation variable by `fetch` and `store`. For example, when dack becomes equal to T at time t+n in response to a read request initiated at time t, the value of the datain bus is described by the following equation:

```
datain (t+n) = (fetch rep) (mem t,(address rep) (mar t))
```

One of the main differences from the FM8501 specification [76] is that our relational specification style allows the memory model to have an entirely independent specification. Thus, it is possible to reason independently about the internal operation of TAMARACK-3 at the phase level without need of a memory model. It is only necessary to compose specifications for TAMARACK-3 and external memory in order to reason about their interaction at the microprogramming level. This contrasts with the functional specification style used for FM8501 where "the characterization of external devices is wrapped up in the same function which specifies the microprocessor" [76].

## 6.5 Verification

Verifying the operation of TAMARACK-3 in fully asynchronous mode is much more difficult than fully synchronous mode because of the unknown length of microcode repeat-loops during an instruction cycle. The overall strategy is to first show that every handshaking sequence runs to completion, then collapse repeat-loops to single steps in an abstract view of behaviour. Symbolic execution is then used to reason about the fixed sequences of operations in this abstract view of behaviour. Finally, we use the function `TimeOf` described earlier in Chapter 5 to specify a synchronization scheme for asynchronous mode.

### 6.5.1 Phase Level

Earlier correctness results described in Chapter 5 for the phase level operation of TAMARACK-3 are re-used in this version of the correctness proof without change or addition. We recall that phase level correctness results do not depend on any particular model of external memory. The interaction of the microprocessor with external memory only concerns the operation of its internal architecture at the microprogramming level.

### 6.5.2 Implementation of the Sender Specification

One of the first steps in verifying the operation of TAMARACK-3 in fully asynchronous mode is to show that the microprogramming level correctly implements the sender part of the handshaking specification. The derivation of the following theorem is tedious but very routine. This proof step does not involve a behavioural model for external memory.

```
|- Val3_CASES_ASM rep ∧
   Val4Word4_ASM rep ∧
   TamarackImp rep (
     datain,dack,gnd,idreq,mpc,mar,pc,
     acc,ir,rtn,arg,buf,idack,dataout,wmem,dreq,addr) ∧
   GND gnd ∧
   (((val4 rep) o mpc) 0 = 0)
    ⟹
   Sender (dreq,dack)
```

Once it has been established that the microprogramming level satisfies the constraints expressed by `Sender`, this result can be combined with the specification of external memory, in particular, the constraints expressed by `Receiver` to obtain some very useful facts for reasoning about handshaking interactions. Among other things, one of the useful facts implied by the combined constraints of `Sender` and `Receiver` is that every handshaking sequence runs to completion, that is, every repeat-loop in the microcode eventually terminates.

### 6.5.3 Collapsing Repeat-Loops to Single Steps

The microprocessor specification and behavioural model of external memory are combined together in the definition of `AsynSystem` to reason about interactions of the TAMARACK-3 microchip with fully asynchronous memory. As shown in the definition of `AsynMemory`, the memory specification includes the constraints expressed by `Receiver`, i.e., the memory device must satisfy the receiver part of the handshaking protocol.

```
Define (
  "AsynSystem (rep:rep_ty)
    (ireq,mpc,mar,pc,acc,ir,rtn,arg,buf,iack,dack,mem) =
    ∃datain gnd dataout wmem dreq addr.
      TamarackImp rep (
        datain,dack,gnd,ireq,mpc,mar,pc,
        acc,ir,rtn,arg,buf,iack,dataout,wmem,dreq,addr) ∧
      AsynMemory rep (dreq,dack,mem,wmem,addr,dataout,datain) ∧
      GND gnd");;
```

The fact that every repeat-loop in the microcode terminates makes it possible to derive theorems which, in effect, collapse repeat-loops to single steps in an abstract view of behaviour at the microprogramming level.

For instance, the following theorem provides a collapsed view of the repeat-loop in state 6 of the FSM flow graph. This theorem states that the repeat-loop will terminate after zero or more clock cycles with no changes to the internal state of the microprocessor.

```
|- Val3_CASES_ASM rep ∧
   Val4Word4_ASM rep ∧
   AsynSystem rep (idreq,mpc,mar,pc,acc,ir,rtn,arg,buf,idack,dack,mem) ∧
   (((val4 rep) o mpc) 0 = 0)
   ⟹
   ∀t.
     (((val4 rep) o mpc) t = 6)
     ⟹
     ∃n.
       (dack (t+n) = F) ∧
       (∀m. m < n ⟹ (dack (t+m) = T)) ∧
       (((val4 rep) o mpc) (t+n) = 6) ∧
       (mem (t+n) = mem t) ∧
       (mar (t+n) = mar t) ∧
       (pc (t+n) = pc t) ∧
       (acc (t+n) = acc t) ∧
       (rtn (t+n) = rtn t) ∧
       (idack (t+n) = idack t)
```

Another example is the theorem shown below which provides a collapsed view of the repeat-loop in state 13. It is convenient to state this result as a transition which begins in state 6 and terminates in state 15. During this transition, a value is fetched from memory mem, added to the current contents of the accumulator acc, and the result temporarily placed in the ALU buffer buf.

```
|- Val3_CASES_ASM rep ∧
   Val4Word4_ASM rep ∧
   AsynSystem rep (idreq,mpc,mar,pc,acc,ir,rtn,arg,buf,idack,dack,mem) ∧
   (((val4 rep) o mpc) 0 = 0)
   ⟹
   ∀t.
     (((val4 rep) o mpc) t = 6) ∧
     (dack t = F)
     ⟹
     ∃n.
       (∀m. m < n ⟹ ¬(((val4 rep) o mpc) (t+m) = 0)) ∧
       (((val4 rep) o mpc) (t+n) = 15) ∧
       (mem (t+n) = mem t) ∧
       (pc (t+n) = pc t) ∧
       (buf (t+n) =
         add rep (acc t,fetch rep (mem t,(address rep (mar t))))) ∧
       (rtn (t+n) = rtn t) ∧
       (idack (t+n) = idack t)
```

The above pair of theorems describe two successive steps in the symbolic execution of the microprogramming level. In the course of reasoning about the microinstruction sequence for an ADD instruction, the first theorem advances the state of the symbolic execution to the end of the repeat-loop in state 6 and the second theorem advances the symbolic execution from this point to state 15.

### 6.5.4 Symbolic Execution

A complete set of theorems for collapsing repeat-loops into single steps represents an abstract view of behaviour where programming level operations are implemented by fixed sequences of actions. Hence, the symbolic execution proof technique described earlier in Chapter 5 for fully synchronous mode can also be used here to show that programming level operations are correctly implemented by microinstruction sequences which involve handshaking interactions with external memory.

Correctness results for each programming level operation are combined by case analysis to obtain the following theorem.

```
|- Val3_CASES_ASM rep ∧
   Val4Word4_ASM rep ∧
   AsynSystem rep (idreq,mpc,mar,pc,acc,ir,rtn,arg,buf,idack,dack,mem) ∧
   (((val4 rep) o mpc) 0 = 0)
   ⟹
   ∀t.
     (((val4 rep) o mpc) t = 0) ∧
     (dack t = F)
     ⟹
     ∃n.
       Next ((((val4 rep) o mpc) Eq 0) and (~dack)) (t,t+n) ∧
       ((mem (t+n),pc (t+n),acc (t+n),rtn (t+n),idack (t+n)) =
        NextState rep (idreq t,mem t,pc t,acc t,rtn t,idack t))
```

The above theorem is almost identical to the top-level correctness theorem for fully synchronous mode given in Chapter 5 except for the term used to denote the next-instruction signal. Instead of,

```
(((val4 rep) o mpc) Eq 0)
```

the next-instruction signal is denoted by the term,

```
(((val4 rep) o mpc) Eq 0) and (~dack)
```

which states that the next instruction cycle does not begin until the control unit FSM is about to leave state 0.

### 6.5.5 Top Level Correctness Statement

This version of the proof is completed by repeating the proof procedure outlined earlier in Chapter 5 for using the function `TimeOf` to construct a mapping from abstract time to concrete time. The top-level correctness statement for this version of the proof is shown below.

```
|- Val3_CASES_ASM rep ∧
   Val4Word4_ASM rep ∧
   AsynSystem rep (idreq,mpc,mar,pc,acc,ir,rtn,arg,buf,idack,dack,mem) ∧
   (((val4 rep) o mpc) 0 = 0)
   ⟹
   let f = TimeOf ((((val4 rep) o mpc) Eq 0) and (~dack)) in
   TamarackBeh rep (idreq o f,mem o f,pc o f,acc o f,rtn o f,idack o f)
```

Chapter 7

# Summary

## 7.1 What Has Been Proved ?

We have described three different correctness theorems for the design of TAMARACK-3. The first two theorems state correctness results for the operation of TAMARACK-3 in fully synchronous mode. These two theorems differ only in how they define a timing relationship between different levels of specification. The third theorem states correctness results for the operation of TAMARACK-3 in fully asynchronous mode. As mentioned earlier, we have not formally considered the operation of TAMARACK-3 in extended cycle mode.

Each correctness theorem includes explicit assumptions about the external environment, in particular, a behavourial model of external memory and how it is interfaced to the microprocessor. In each case, the correctness theorem states that the predicate `TamarackBeh` is an abstract model of the internal architecture as specified by the predicate `TamarackImp`. Therefore, true statements about the behaviour of the abstract model can be related to true statements about the specification of the internal architecture.

In the course of establishing these general results, we have obtained some more specific results about particular aspects of the design including:

- That the datapath bus is driven by (at most) one bus device whenever it is being used to transfer data[1]
- That the microprocessor design correctly implements the sender part of the handshaking protocol.
- A precise description of the next-instruction signal marking the end of an instruction cycle (which might not be obvious in the case of fully asynchronous mode).

Most of the proof effort concerns the fetch-decode-execute sequence for each programming level operation during an instruction cycle. But the proof goes further than this by considering the overall operation of the microprocessor, that is, how one instruction cycle is related to the next instruction cycle. For instance, it shows that an interrupt request does not interrupt the completion of a memory interaction (i.e. the repeat-loop in state 0). It also shows that an interrupt request takes precedence, as one would expect, over the normal flow of program control (when a previous interrupt is not still being serviced).

[1]Strictly speaking, the proof has not directly addressed the question of whether bus conflicts can ever occur.

It is also important to describe the limitations of this proof of correctness. As we remarked earlier, the formal verification of TAMARACK-3 focuses very specifically on the register-transfer level operation of the internal architecture. Below and above this level there are aspects of a complete design for TAMARACK-3 which are not formally considered.

Chapter 3 described some aspects of the semantic gap between the abstract view of sequential behaviour in our lowest level of formal specification and timing relationships in actual hardware. This gap is bridged by a number of informal assumptions which are *not* explicitly mentioned in the correctness results. Although our formal theory could be extended with several more, increasingly detailed, levels of timing,[2] there will always be a gap between formal proof (a mathematical concept) and actual hardware (a physical device). As Cohn [30] remarks,

> ...a material device can only be observed and measured, not verified. It can be described in an abstracted way, and the simplified description verified, but again, there is no way to assure the accuracy of the description.

It is also important to understand that our formal theory says nothing about the basic data types used to represent data in the microprocessor and primitive operations involving these data types with the exception of two assumptions which appear explicitly in the correctness theorems. It may surprise some readers that these details do not need to be known, but their insignificance underlines the fact that very few aspects of the computation performed by the hardware are actually taken account of in the formal proof. The only significant forms of computation considered in the proof are:

- Computation of microinstruction addresses by the next address logic.
- Resolving de-centralized control over access to the system bus in the datapath.
- Extraction of individual bits and other sub-fields from the current microinstruction word by decoding logic.

This absence of detail about basic types and primitive operations should not be seen as a shortcoming or unfinished step in our formal proof. We have deliberately avoided these details to clearly demarcate the boundaries between what has and what has not been considered in the formal verification of TAMARACK-3. Building more details than necessary (for this particular proof) into the computation model would have risked the false impression that these details have been formally considered in the proof. Instead, other levels of concern should be considered in separate theories. The next section suggests how a hierarchy of parameterized theories (in the non-technical, general sense) could be stacked upon one another by linking them together through representation variables.

[2]Herbert [72,74] describes proofs techniques in the HOL logic for relating more detailed levels of timing to synchronous level models. Dhingra [43] has used the HOL logic to give a formal foundation for the design rules of a dynamic CMOS integrated circuit design style.

## 7.2 Relating This Proof to Other Levels

The representation variable rep which appears as an extra parameter in definitions throughout the formal specification of TAMARACK-3 is effectively a parameterization of the formal theory. It provides a means of relating this theory to both lower and higher level models of computation. We illustrate this point with several examples of how our formal theory for TAMARACK-3 might be linked into a verified stack.

### 7.2.1 Lower Levels

By assigning an appropriate value to the representation variable rep, the formal theory about TAMARACK-3 can be made to stack upon a lower level theory about the implementation of register-transfer level devices. This lowel level theory might, in turn, be a generic specification parameterized by its own representation variable and stacked upon an even lower level of representation at the transistor level.

To illustrate this idea with a simple example, the constant REP16 is defined as a value for rep based on the built-in HOL data types described in [30,57,79]. In this case, we have created data types for a 16-bit version of TAMARACK-3.

```
Define ("ISZERO16 w = ((VAL16 w) = 0)");;

Define ("INC16 w = WORD16 ((VAL16 w) + 1)");;

Define ("ADD16 (w1,w2) = WORD16 ((VAL16 w1) + (VAL16 w2))");;

Define ("SUB16 (w1,w2) = WORD16 ((VAL16 w1) - (VAL16 w2))");;

Define ("OPCODE w = WORD3 (V (SEG (0,2) (BITS16 w)))");;

Define ("ADDRESS w = WORD13 (V (SEG (3,15) (BITS16 w)))");;

Define (
  "REP16 =
    ISZERO16,                          % iszero %
    INC16,                             % inc %
    ADD16,                             % add %
    SUB16,                             % sub %
    WORD16,                            % wordn %
    VAL16,                             % valn %
    OPCODE,                            % opcode %
    VAL3,                              % val3 %
    ADDRESS,                           % address %
    (λ(x,y). FETCH13 x y),             % fetch %
    (λ(x,y,z). STORE13 y z x),         % store %
    WORD4,                             % word4 %
    VAL4");;                           % val4 %
```

The current version of these built-in HOL data types (as given by the `eval` library in the HOL88 system) is not fully axiomatized or secure,[3] but with a complete axiomatization it would be possible to derive the assumptions expressed by the predicates `Val3_CASES_ASM` and `Val4Word4_ASM` as proven theorems.

```
|- Val3_CASES_ASM REP16

|- Val4Word4_ASM REP16
```

If these assumptions were established as theorems for this particular value of the representation variable, we could then obtain the following correctness result for a 16-bit version of TAMARACK-3 (along with similar results for fully synchronous mode).

```
|- AsynSystem REP16 (idreq,mpc,mar,pc,acc,ir,rtn,arg,buf,idack,dack,mem) ∧
   ((VAL4 o mpc) 0 = 0)
   ⟹
   let f = TimeOf (((VAL4 o mpc) Eq 0) and (not dack)) in
   TamarackBeh REP16 (idreq o f,mem o f,pc o f,acc o f,rtn o f,idack o f)
```

Hence, one of the purposes of relating our formal theory to lower levels is to replace assumptions about uninterpreted data types by theorems derived from lower level representations of data.

Another purpose is to derive correctness results for implementations of register-transfer level primitives, for example, to verify that an implementation of the ALU correctly implements a set of arithmetic/logical operations.

Because the current version of the built-in HOL data types for machine words is not fully axiomatized or secure, we have used an alternate representation where bit patterns are modelled by functions which map bit positions to bit values. In addition to completeness and security, this representation offers the advantage of being able to parameterize specifications by the number of bits in a machine word instead of the fixed word widths provided by the built-in HOL data types. Functions such as `Valn` and `Wordn` are defined to establish relationships between this function-based representation and higher-algebraic levels, i.e., the natural numbers. Correctness results for implementations of register-transfer level primitives are expressed in terms of `Valn` and `Wordn`. We have used this approach to verify a complete implementation of an earlier version of TAMARACK-3 down to the transistor level [80,81,84,87,88].

### 7.2.2 Higher Levels

While lower levels of representation may yield theorems to replace assumptions about data types, stacking higher levels upon our correctness results for TAMARACK-3 may

[3] An incomplete set of postulated axioms exists in the form of evaluation rules which use `mk_thm` to create unproven theorems. The recursive data types package developed for HOL by Melham [103] provides a foundation for building a secure set of data types for reasoning about machine words and bit patterns.

conversely require the introduction of more assumptions about the data types. As explained earlier, the use of uninterpreted data types and uninterpreted primitives for operations on these data types makes computational aspects of our programming level model somewhat transparent. To reason about the execution of programs requires the introduction of more primitives and more assumptions to provide the programming level model with more computation details.

At the very least, it would likely be necessary to provide some details of the operation selected by inc, used to increment the program counter pc, in order to reason about the sequential execution of TAMARACK-3 instructions. For example, the predicate INC_ASM,

```
Define (
  "INC_ASM n (rep:rep_ty) =
    ∀w. ((valn rep) ((inc rep) w)) = ((((valn rep) w) + 1) MOD (2 EXP n))");;
```

could be defined to provide an arithmetic interpretation for the operation selected from the representation variable by inc.[4] In this case, machine words (of various sizes) are modelled at this higher level by natural numbers where the finite precision of operations performed by hardware is modelled by modular arithmetic. The parameter n is the number of bits in a full-size machine word; for instance, n would be equal to sixteen for a 16-bit version of TAMARACK-3.

Similar assumptions expressed by ADD_ASM and SUB_ASM provide arithmetic interpretations for the operations selected by add and sub.[5] The predicate FETCH_STORE_ASM provides an interpretation for memory operations.

```
Define (
  "ISZERO_ASM (rep:rep_ty) =
    ∀w. ((iszero rep) w) = (((valn rep) w) = 0)");;

Define (
  "ADD_ASM n (rep:rep_ty) =
    ∀w1 w2.
      ((valn rep) ((add rep) (w1,w2))) =
        ((((valn rep) w1) + ((valn rep) w2)) MOD (2 EXP n))");;

Define (
  "SUB_ASM n (rep:rep_ty) =
    ∀w1 w2.
      ((valn rep) ((sub rep) (w1,w2))) =
        ((((valn rep) w1) - ((valn rep) w2)) MOD (2 EXP n))");;
```

[4]The assumption expressed by INC_ASM is stronger than necessary if it is only necessary to know that the address field of the full-size machine word is incremented. This assumption would be too restrictive for an implementation where the address bits are not necessarily the low-order bits of the full-size machine word stored in the program counter pc.

[5]Subtraction in the natural numbers as axiomatized in HOL does not correspond exactly to subtraction in two's complement arithmetic. The definition of two's complement subtraction given in [84] provides a more accurate model of how this ALU operation would likely be implemented in hardware.

```
Define (
  "FETCH_STORE_ASM (rep:rep_ty) =
    ∀m a1 a2 w.
      ((a1 = a2) ⇒
        ((fetch rep) (((store rep) (m,a1,w)),a2) = w) |
        ((fetch rep) (((store rep) (m,a1,w)),a2) = (fetch rep) (m,a2)))");;
```

Like the assumptions expressed by `Val3_CASES_ASM` and `Val4Word4_ASM`, these additional assumptions could be proven as theorems from a lower level representation of data. For instance, a complete axiomatization of built-in HOL data types for machine words would yield the theorems,

```
|- INC_ASM 16 REP16

|- ISZERO_ASM REP16

|- ADD_ASM 16 REP16

|- SUB_ASM 16 REP16

|- FETCH_STORE_ASM REP16
```

as properties of the constant `REP16`. In the case of `INC_ASM`, `ADD_ASM` and `SUB_ASM`, the axiomatization would need to include the following property. A similar theorem has been derived for the function-based representation described in [81,84].

```
|- ∀m. VAL16 (WORD16 m) = (m MOD (2 EXP 16))
```

Together with some additional assumptions about operations for extracting the opcode and operand address bits from a full-size machine word, the above assumptions would introduce enough substance to the programming level model of TAMARACK-3 to reason about the execution of programs. In particular, it would provide enough computational power to support the compiler verification study reported in [85,86].

## 7.3 Putting Formal Specifications to Work

At the beginning of this dissertation, we emphasized that formal descriptions must easily translate into established notations to be understood in a wider context. In this section, we briefly mention two experiments in verification-driven design based on earlier versions of TAMARACK-3.

### 7.3.1 A Fabricated TAMARACK-1 Microchip

An 8-bit version of TAMARACK-1 was implemented as a 5,000 transistor CMOS microchip in December 1985 while the author was a visiting student at Xerox PARC. The purpose of this exercise was to study the role of formal specification in the implementation of a design. The design was small enough to be completed in two months but was sufficiently varied to be representative of many aspects of VLSI design. The fabricated chip was tested in April 1986 at the University of Calgary and found to be fully functional.

A significant difference between the HOL specification and the design hierarchy used to build the chip was the functional-slice[6] organization of the datapath in the HOL specification in contrast to the bit-slice organization of the datapath in the design hierarchy.

A more interesting result was the discovery that the formal verification of the design missed a design error: there was no reset button to initialize the microinstruction program counter. After the design was submitted for fabrication and long after its formal verification, we decided to simulate the design using a switch level simulator involving a more accurate model of a signal value. The design failed to simulate properly because the initial state of the microcode program counter was undefined and there was no way to force this signal to a defined state. Fortunately, when the actual chips were returned and tested we found that the fabricated chip would eventually reset to a defined state due to electrical factors not modelled at the switch level and the chip worked correctly.

The discovery of this error lead us to extend the original correctness proof down to the switch level. In addition to showing that the revised design of TAMARACK-1 could be reset to a well-defined state, it was necessary to prove that, once initialized, the hardware model would remain in a well-defined state. Establishing this fact was more work than expected partly because it involved functional aspects of the design in addition to switch level considerations. It depended, for instance, on showing that at most one bus device would ever attempt to assert a value onto the datapath bus.

### 7.3.2 Silicon Compiler Interface

In collaboration with researchers at SRI International [89], we undertook a simple experiment to investigate how formal methods could be interfaced with practical CAD tools by hand-translating the HOL specification for TAMARACK-2 into a design hierarchy for a commercial silicon compiler called GENESIL.[7]

> The GENESIL Silicon Design System is a silicon compiler, an automated tool that contains the IC design expertise to transform a functional specification into a database from which an IC can be produced. The system designer using GENESIL need not know the lowest details of IC design.
>
> [GENESIL System Users Manual, page 1-1]

We originally hoped that formally verified design would be very close to the level of description required by the silicon compiler. These two levels of description indeed turned out to be very closely related from a structural point of view. However, we found

[6] Anceau [2] elaborates on the difference between functional-slice and bit-slice views.

[7] GENESIL Silicon Design System, Silicon Compiler Systems Corp., San Jose, California, 1988

a wider semantic gap than expected between the view of sequential behaviour in the formally verified model and the view of sequential behaviour supported by the silicon compiler.

This semantic gap contributed to a design error even though the design was correct with respect to the built-in rules for the two-phase, non-overlapping design style. In particular, there was a mismatch between the single-phase view of sequential behaviour in the formal specification and the two-phase view supported by the silicon compiler. Although this experiment in verification-driven design revealed a semantic gap for this particular level of bottom level specification, this does *not* represent any kind of inherent limitation of formal methods.

## 7.4 Conclusion

This dissertation has described methods based on formal proof and mechanical proof-generation which can be used to increase confidence in the design of a microprocessor-based system. In particular, these methods can be used to demonstrate that a design is free from errors to the extent that formal descriptions of the design and requirements are related by a formal proof.

The principal contribution of this research is to recommend the use of generic specification and the use of natural notations from other formalisms in a single unifying framework. We have also shown how generic specification can be implemented in a pure formalism, namely higher-order logic, without adding new primitive constructs.

Previous work, using more constrained approaches, has made significant progress towards the formal specification and verification of microprocessor-based systems. However, genericity and the use of natural notations are both likely to be key mechanisms when verification problems are scaled upwards to the complexity of realistic applications.

Finally, we have argued that the primary purpose of using mechanical proof generation techniques to reason about software and hardware is to support the intelligent participation of a human verifier in the rigorous analysis of a design at a level which supports clear thinking.

# References


[1] Alfred V. Aho and Jeffrey D. Ullman, Principles of Compiler Design, Addison-Wesley, 1977.

[2] F. Anceau, The Architecture of Microprocessors, Addison-Wesley Publishing Company, Wokingham, 1986.

[3] James R. Armstrong, Chip-Level Modelling with VHDL, Prentice-Hall, 1989.

[4] J. G. P. Barnes, Programming in Ada, Addison-Wesley, 1984 (second edition).

[5] H. Barrow, VERIFY: A Program for Proving Correctness of Digital Hardware Designs, Artificial Intelligence, Vol. 24, No. 1-3, December 1984, pp. 437-491.

[6] William R. Bevier, Warren A. Hunt, Jr., and William D. Young, in: Towards Verified Execution Environments, in: Proceedings of the 1987 IEEE Symposium on Security and Privacy, 27-29 April 1987, Oakland, California Computer Society Press, Washington, D.C., 1987 pp. 106-115. Also Report No. 5, Computational Logic, Inc., Austin, Texas, February 1987.

[7] W. Bevier, W. Hunt, J Moore, and W. Young, An Approach to Systems Verification, Journal of Automated Reasoning, Vol. 5, No. 4, November 1989. Also Report No. 41, Computational Logic, Inc., Austin, Texas, April 1989.

[8] Mark Bickford and Mandayam Srivas, Microprocessor Verification using Clio, M. Leeser and G. Brown, eds., Specification, Verification and Synthesis: Mathematical Aspects, Proceedings of a Workshop, 5-7 July 1989, Ithaca, N.Y., Springer-Verlag, 1989.

[9] G. Birtwistle, J. Joyce, B. Liblong, T. Melham and R. Schediwy, Specification and VLSI, in: G. Milne and P. Subrahmanyam, eds., Formal Aspects of VLSI Design, Proceedings of the 1985 Edinburgh Conference on VLSI, North-Holland, 1986, pp. 83-97.

[10] Graham Birtwistle and Brian Graham, Verifying an SECD chip in HOL, Luc Claesen, ed., Applied Formal Methods For Correct VLSI Design, 13-16 November 1989, Leuven, Belgium, (to be published by Elsevier Science Publishers).

[11] G. Bochman, Hardware Specification with Temporal Logic, IEEE Transactions on Computers, Vol. C-31, No. 3, March 1982, pp. 223-231.

[12] Johnathan Bowen, The Formal Specification of a Microprocessor Instruction Set, Report PRG-60, Computing Laboratory, Oxford University, January 1987.

[13] R. S. Boyer and J S. Moore, A Computational Logic, Academic Press, 1979.

[14] Bishop C. Brock and Warren A. Hunt, Jr., The Formalization of a Simple Hardware Description Language, in Luc Claesen, ed., Applied Formal Methods For Correct VLSI Design, 13-16 November 1989, Leuven, Belgium, (to be published by Elsevier Science Publishers).

[15] Geoffrey M. Brown and Miriam E. Leeser, From Programs to Transistors: Verifying Hardware Synthesis Tools, in: M. Leeser and G. Brown, eds., Specification, Verification and Synthesis: Mathematical Aspects, Proceedings of a Workshop, 5-7 July 1989, Ithaca, N.Y., Springer-Verlag, 1989.

[16] Randy Everitt Bryant, A Switch-Level Simulation Model for Integrated Logic Circuits, Ph.D. Thesis, Dept. of Electrical Engineering and Computer Science, Report No. MIT/LCS/TR-259, Laboratory for Computer Science, Massachusetts Institute of Technology, March 1981.

[17] R. M. Burstall and P.J. Landin, Programs and their Proofs: an Algebraic Approach, in: B. Meltzer and D. Mitchie, eds., Machine Intelligence, Vol. 4, Edinburgh University Press, 1969. pp. 17-43.

[18] A. Camilleri, M. Gordon and T. Melham, Hardware Verification using Higher-Order Logic, in: D. Borrione, ed., From HDL Descriptions to Guaranteed Correct Circuit Designs, Proceedings of the IFIP WG 10.2 International Working Conference, Grenoble, France, 9-11 September 1986, North-Holland, Amsterdam, 1987. pp. 43-67. Also Report No. 91, Computer Laboratory, Cambridge University, June 1986.

[19] Albert John Camilleri, Executing Behavioural Definitions in Higher Order Logic, Ph.D. Thesis, Report No. 140, Computer Laboratory, Cambridge University, February 1988.

[20] Albert John Camilleri, Simulation as an Aid to Verification Using the HOL Theorem Prover, in: D. Edwards, ed., Proceedings of the IFIP TC10 Working Conference on Design Methodology in VLSI and Computer Architecture, Pisa, Italy, 19-21 September 1988, North-Holland, Amsterdam, 1989, pp. 147-168. Also Report No. 150, Computer Laboratory, Cambridge University, October 1988.

[21] Albert John Camilleri, Mechanizing CSP Trace Theory in Higher Order Logic, (in preparation), Hewlett-Packard Laboratories, Bristol, England, 1989.

[22] Paolo Camurati and Paolo Prinetto, Formal Verification of Hardware Correctness, IEEE Computer, Vol. 21, No. 7, July 1988, pp. 8-19.

[23] Rachel Cardell-Oliver, The Specification and Verification of Sliding Window Protocols in Higher Order Logic, Report No. 183, Computer Laboratory, Cambridge University, October 1989.

[24] L.M. Chirica, Contributions to Compiler Correctness, Ph.D. Thesis, Report UCLA-ENG-7697, Computer Science Department, University of California, Los Angeles, October 1976.

[25] Laurian M. Chirica and David F. Martin, Toward Compiler Implementation Correctness Proofs, ACM Transactions on Programming Languages and Systems, Vol. 8, No. 2, April 1986, pp. 185-214.

[26] A. Church, A Formulation of the Simple Theory of Types, Journal of Symbolic Logic, Vol. 5, 1940.

[27] Alan Clement, Microprocessor Systems Design: 68000 Hardware, Software and Interfacing, PWS Publishers, 1987.

[28] Avra Jean Cohn, Machine Assisted Proofs of Recursion Implementation, Ph.D. Thesis, Technical Report CST-6-79, Department of Computer Science, University of Edinburgh, April 1980.

[29] Avra Cohn, A Proof of Correctness of the Viper Microprocessor: The First Level, in: G. Birtwistle and P. Subrahmanyam, eds., VLSI Specification, Verification and Synthesis, Kluwer Academic Publishers, Boston, 1988, pp. 27-71. Also Report No. 104, Computer Laboratory, Cambridge University, January 1987.

[30] Avra Cohn, Correctness Properties of the Viper Block Model: The Second Level, in: G. Birtwistle and P. Subrahmanyam, eds., Current Trends in Hardware Verification and Automated Theorem Proving, Springer-Verlag, 1989, pp. 1-91. Also Report No. 134, Computer Laboratory, Cambridge University, May 1988.

[31] Avra Cohn, The Notion of Proof in Hardware Verification, Journal of Automated Reasoning, Vol. 5, May 1989, pp. 127-139.

[32] Avra Cohn, private communication, 1989.

[33] P. A. Collier, Simple Compiler Correctness - A Tutorial on the Algebraic Approach, The Australian Computer Journal, Vol. 18, No. 3, August 1986, pp. 128-135.

[34] R. L. Constable et al., Implementing Mathematics with the Nuplr Proof Development System, Prentice-Hall, 1986.

[35] Stephen D. Crocker, Eve Cohen, Sue Landauer and Hilarie Orman, Reverification of a Microprocessor, Proceedings of the 1988 IEEE Symposium on Security and Privacy, 18-21 April 1988, Oakland, California Computer Society Press, Washington, D.C., 1988, pp. 166-176.

[36] W.J. Cullyer, Implementing Safety-Critical Systems: The VIPER Microprocessor, in: G. Birtwistle and P. Subrahmanyam, eds., VLSI Specification, Verification and Synthesis, Kluwer Academic Publishers, 1988, pp. 1-25.

[37] W.J. Cullyer, High Integrity Computing, in: M. Joseph, ed., Formal Techniques in Real-Time and Fault-Tolerant Systems, Lecture Notes in Computer Science, No. 331, Springer-Verlag, Berlin, 1988. pp. 1-35.

[38] W. J. Cullyer, Using VIPER in Railroad Signalling, SafetyNet, Issue 4, Viper Technologies Ltd., Worcester, England, March 1989, pp. 7-12.

[39] Paul Curzon, A Structured Approach to the Verification of Low Level Microcode, Ph.D. Thesis, Computer Laboratory, Cambridge University, (to appear).

[40] Bruce S. Davie, A Formal, Hierarchical Design and Validation Methodology for VLSI, Ph.D. Thesis, Report CST-55-88, Dept. of Computer Science, University of Edinburgh, October 1988.

[41] Richard A. De Millo, Richard J. Lipton and Alan J. Perlis, Social Processes and Proofs of Theorems and Programs, Communications of the ACM, Vol. 22, No. 5, May 1979. pp. 271-280.

[42] Joëlle Despeyroux, Proof of Translation in Natural Semantics, Proceedings of the 1986 Symposium on Logic in Computer Science, 16-18 June 1986, Cambridge, MA., Computer Society Press, Washington, D.C., 1986, pp. 193-205. Also Rapports de Recherche No. 514, INRIA, Sophia Antipolis, France, April 1986.

[43] Inderpreet S. Dhingra, Formalising an Integrated Circuit Design Style in Higher-Order Logic, Ph.D. Thesis, Report No. 151, Computer Laboratory, Cambridge University, 1989.

[44] D. Dill and E. Clarke, Automatic Verification of Asynchronous Circuits using Temporal Logic, IEE Proceedings, Vol. 133, Pt. E, No. 5, September 1986, pp. 276-282.

[45] H. Eveking, Formal Verification of Synchronous Systems, in: G. Milne and P. Subrahmanyam, eds., Formal Aspects of VLSI Design, Proceedings of the 1985 Edinburgh Conference on VLSI, North-Holland, Amsterdam, 1986, pp. 137-152.

[46] H. Eveking, How to Design Correct Hardware and Know It. G. Milne, ed., The Fusion of Hardware Design and Verification, Proceedings of the IFIP WG 10.2 International Working Conference, Glasgow, Scotland, 3-6 July 1988, North-Holland, 1988, pp. 250-262.

[47] Simon Finn, Michael P. Fourman, Michael Francis and Robert Harris, Formal System Design - An Interactive Synthesis based on Computer-Assisted Formal Reasoning, in: Luc Claesen, ed., Applied Formal Methods For Correct VLSI Design, 13-16 November 1989, Leuven, Belgium, (to be published by Elsevier Science Publishers).

[48] Michael P. Fourman and Eleanor M. Mayger, Formally Based System Design - Interactive Hardware Scheduling, in: G. Musgrave and U. Lauther, eds., VLSI 89, Proceedings of the IFIP TC10/WG 10.5 International Conference on Very Large Scale Integration, Munich, Germany, 16-18 August 1989, (to be published by Elsevier Science Publishers).

[49] M. Fujita, H. Tanaka and T. Moto-oka., Temporal Logic Based Hardware Description and Its Verification with Prolog, New Generation Computing, No. 1, 1983, pp. 195-203.

[50] J. A. Goguen, J.W. Thatcher, E.G. Wagner, and J.B. Wright, Initial Algebra Semantics and Continuous Algebra, Journal of the ACM, Vol. 24, No. 1, January 1977, pp. 68-95.

[51] Joseph A. Goguen, Parameterized Programming, IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984, pp. 528-543.

[52] Joseph A. Goguen, OBJ as a Theorem Prover with Applications to Hardware Verification, in: G. Birtwistle and P. Subrahmanyam, eds., Current Trends in Hardware Verification and Automated Theorem Proving, Springer-Verlag, 1989, pp. 219-267. Also Report No. SRI-CSL-4R2, Computer Science Laboratory, SRI International, Menlo Park, August 1988.

[53] Michael J. C. Gordon, The Denotational Description of Programming Languages, Spring-Verlag, 1979.

[54] M. J. C. Gordon, A. J. R. G. Milner and C. P. Wadsworth, Edinburgh LCF: A Mechanized Logic of Computation, Lecture Notes in Computer Science, No. 78, Springer-Verlag, 1979.

[55] M. Gordon, A Model of Register Transfer Systems with Applications to Microcode and VLSI Correctness, Internal Report CSR-82-81, Department of Computer Science, University of Edinburgh, 1981.

[56] M. J. C. Gordon, Representing a Logic in the LCF Metalanguage, in: D. Néel, ed., Tools and Notions for Program Construction, Cambridge University Press, 1982.

[57] M. Gordon, LCF_LSM, Report No. 41, Computer Laboratory, Cambridge University, 1983.

[58] M. Gordon, Proving a Computer Correct using the LCF_LSM Hardware Verification System, Report No. 42, Computer Laboratory, Cambridge University, 1983.

[59] M. J. C. Gordon, Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware, in: G. Milne and P. Subrahmanyam, eds., Formal Aspects of VLSI Design, Proceedings of the 1985 Edinburgh Conference on VLSI, North-Holland, 1986, pp. 153-177.

[60] Michael J. C. Gordon, A Proof Generating System for Higher-Order Logic, in: G. Birtwistle and P. Subrahmanyam, eds., VLSI Specification, Verification and Synthesis, Kluwer Academic Publishers, 1988, pp. 73-128. Also Report No. 103, Computer Laboratory, Cambridge University, January 1987.

[61] Michael J. C. Gordon, Mechanizing Programming Logics in Higher Order Logic, in: G. Birtwistle and P. Subrahmanyam, eds., Current Trends in Hardware Verification and Automated Theorem Proving, Springer-Verlag, 1989, pp. 387-439. Also Report No. 145, Computer Laboratory, Cambridge University, September 1988.

[62] Michael J. C. Gordon et al., The HOL System Description, Cambridge Research Centre, SRI International, Suite 23, Miller's Yard, Cambridge CB2 1RQ, England.

[63] Michael Gordon, private communication, 1989.

[64] Brian Graham and Graham Birtwistle, Formalising the Design of an SECD Chip, in: M. Leeser and G. Brown, eds., Specification, Verification and Synthesis: Mathematical Aspects, Proceedings of a Workshop, 5-7 July 1989, Ithaca, N.Y., Springer-Verlag, 1989.

[65] Roger W. S. Hale, Programming in Temporal Logic, Ph.D. Thesis, Report No. 173, Computer Laboratory, Cambridge University, July 1989.

[66] Roger Hale, private communication, 1989.

[67] F. K. Hanna and N. Daeche, Specification and Verification of Digital Systems using Higher-Order Predicate Logic, IEE Proceedings, Vol. 133, Part E, No. 5, September 1986, pp. 242-254.

[68] F. K. Hanna, N. Daeche and M. Longley, VERITAS$^+$: A Specification Language Based on Type Theory, in: M. Leeser and G. Brown, eds., Specification, Verification and Synthesis: Mathematical Aspects, Proceedings of a Workshop, 5-7 July 1989, Ithaca, N.Y., Springer-Verlag, 1989.

[69] F. K. Hanna, M. Longley and N. Daeche, Formal Synthesis of Digital Systems, in: Luc Claesen, ed., Applied Formal Methods For Correct VLSI Design, 13-16 November 1989, Leuven, Belgium, (to be published by Elsevier Science Publishers).

[70] William S. Hatcher, The Logical Foundations of Mathematics, Pergamon Press, 1982.

[71] John P. Hayes, Computer Architecture and Organization, McGraw-Hill, 1978.

[72] John M. J. Herbert, Application of Formal Methods to Digital System Design, Ph.D. Thesis, Computer Laboratory, Cambridge University, 1986.

[73] John Herbert, Temporal Abstraction of Digital Designs, in: G. Milne, ed., The Fusion of Hardware Design and Verification, Proceedings of the IFIP WG 10.2 International Working Conference, Glasgow, Scotland, 3-6 July 1988, North-Holland, 1988, pp. 1-25.

[74] John Herbert, Formal Reasoning about the Timing and Function of Basic Memory Devices, in: Luc Claesen, ed., Applied Formal Methods For Correct VLSI Design, 13-16 November 1989, Leuven, Belgium, (to be published by Elsevier Science Publishers).

[75] C. A. R. Hoare, An Axiomatic Basis for Computer Programming, Communications of the ACM, Vol. 12, No. 10, October 1969, pp. 576-583.

[76] Warren A. Hunt, FM8501, A Verified Microprocessor, Ph.D. Thesis, Report No. 47, Institute for Computing Science, University of Texas, Austin, December 1985.

[77] Steven D. Johnson, Robert M. Wehrmeister and Bhaskar Bose, On the Interplay of Synthesis and Verification, in: Luc Claesen, ed., Applied Formal Methods For Correct VLSI Design, 13-16 November 1989, Leuven, Belgium, (to be published by Elsevier Science Publishers).

[78] J. Joyce, G. Birtwistle and M. Gordon, Proving a Computer Correct in Higher Order Logic, Report No. 100, Computer Laboratory, Cambridge University, 1986.

[79] Jeffrey J. Joyce, Formal Verification and Implementation of a Microprocessor, in: G. Birtwistle and P. Subrahmanyam, eds., VLSI Specification, Verification and Synthesis, Proceedings of a Workshop, 12-16 January 1987, Kluwer Academic Publishers, 1988, pp. 129-157.

[80] Jeffrey J. Joyce, Multi-Level Verification of a Simple Microprocessor, First Year Ph.D. Progress Report and Research Proposal, Computer Laboratory, Cambridge University, December 1987.

[81] Jeffrey J. Joyce, Generic Structures in the Formal Specification and Verification of Digital Circuits, in: G. Milne, ed., The Fusion of Hardware Design and Verification, Proceedings of the IFIP WG 10.2 International Working Conference, Glasgow, Scotland, 3-6 July 1988, North-Holland, 1988, pp. 50-74.

[82] Jeffery J. Joyce, Formal Specification and Verification of Asynchronous Processes in Higher-Order Logic, in: C. Rattray, ed., BCS-FACS Workshop on Specification and Verification of Concurrent Processes, Stirling, Scotland, 6-8 July 1988, (to be published by Springer Verlag). Also Report No. 136, Computer Laboratory, Cambridge University, June 1988.

[83] Jeffrey J. Joyce, Formal Specification and Verification of Microprocessor Systems, in: S. Winter and H. Schumny, eds., Euromicro 88, Proceedings of the 14th Symposium on Microprocessing and Microprogramming, Zurich, Switzerland, 29 August - 1 September, 1988, North-Holland, 1988, pp. 371-378. Also Report No. 147, Computer Laboratory, Cambridge University, September 1988.

[84] Jeffery J. Joyce, Using Higher-Order Logic to Specify Computer Hardware and Architecture, in: D. Edwards, ed., Design Methodologies for VLSI and Computer Architecture, Proceedings of the IFIP TC10 Working Conference on Design Methodology in VLSI and Computer Architecture, Pisa, Italy, 19-21 September 1988, North-Holland, 1989, pp. 129-146.

[85] Jeffrey J. Joyce, A Verified Compiler for a Verified Microprocessor, Report No. 167, Computer Laboratory, Cambridge University, March 1989.

[86] Jeffrey J. Joyce, Totally Verified Systems: Linking Verified Software to Verified Hardware, in: M. Leeser and G. Brown, eds., Specification, Verification and Synthesis: Mathematical Aspects, Proceedings of a Workshop, 5-7 July 1989, Ithaca, N.Y., Springer-Verlag, 1989. Also Report No. 178, Computer Laboratory, Cambridge University, September 1989.

[87] Jeffrey J. Joyce, Formal Specification and Verification of Synthesized MOS Structures, in: G. Musgrave and U. Lauther, eds., VLSI 89, Proceedings of the IFIP TC10/WG 10.5 International Conference on Very Large Scale Integration, Munich, Germany, 16-18 August 1989, (to be published by Elsevier Science Publishers).

[88] Jeffrey J. Joyce, Formal Specification and Verification of Microprocessor Systems, Integration, the VLSI journal, Vol. 7, September 1989, pp. 247-266.

[89] J. Joyce, E. Liu, J. Rushby, N. Shankar, R. Suaya, F. von Henke: From Hardware Verification to Silicon Compilation, (in preparation) SRI International, Menlo Park, 1990.

[90] Donald M. Kaplan, Correctness of a Compiler for Algol-like Programs, Stanford Artificial Intelligence Memo No. 48, Stanford University, July 1967.

[91] M. Katevenis, Reduced Instruction Set Computer Architectures for VLSI, Ph.D. Thesis, Department of Electrical Engineering, University of California at Berkeley, ACM Doctoral Dissertation Award, MIT Press, 1984.

[92] John C. Knight and Nancy G. Leveson, An Experimental Evaluation of the Assumption of Independence in Multiversion Programming, IEEE Transactions on Software Engineering, Vol SE-12, No. 1, January 1986, pp. 96-109.

[93] Miriam E. Leeser. Reasoning about the Function and Timing of Integrated Circuits with Prolog and Temporal Logic, Ph.D. Thesis, Computer Laboratory, Report No. 132, Computer Laboratory, Cambridge University, April 1988.

[94] T. E. Leonard, private communication, 1989.

[95] A. M. Lister, Fundamentals of Operating Systems, Macmillan, 1979 (second edition).

[96] Paul Loewenstein, Reasoning about State Machines in Higher-Order Logic, in: M. Leeser and G. Brown, eds., Specification, Verification and Synthesis: Mathematical Aspects, Proceedings of a Workshop, 5-7 July 1989, Ithaca, N.Y., Springer-Verlag, 1989.

[97] Alain J. Martin, Steve M. Burns, T. K. Lee, Drazen Borkovic and Pieter J. Hazewindus, The Design of an Asynchronous Microprocessor, Report CS-TR-89-2, Computer Science Department, California Institute of Technology, (to appear in Proceedings of the Decennial CalTech Conference on VLSI, 20-22 March 1989).

[98] Alain J. Martin, Steve M. Burns, T. K. Lee, Drazen Borkovic and Pieter J. Hazewindus, The First Asynchronous Microprocessor: The Test Results, Report CS-TR-89-6, Computer Science Department, California Institute of Technology, April 1989.

[99] J. McCarthy and J. Painter, Correctness of a Compiler for Arithmetic Expressions, in: J. Schwartz, ed., Proceedings of a Symposia on Applied Mathematics, American Mathematical Society, 1967, pp. 33-41.

[100] Thomas F. Melham, Abstraction Mechanisms for Hardware Verification, in: G. Birtwistle and P. Subrahmanyam, eds., VLSI Specification, Verification and Synthesis, Kluwer Academic Publishers, 1988, pp. 267-291. Also Report No. 106, Computer Laboratory, Cambridge University, January 1987.

[101] Thomas F. Melham, Automating Recursive Type Definitions in Higher Order Logic, to be published in: G. Birtwistle, ed., Proceedings of the Banff Hardware Verification Workshop, Banff, Canada, June 12-18, 1988. Also Report No. 146, Computer Laboratory, Cambridge University, September 1988.

[102] Thomas F. Melham, Using Recursive Types to Reason about Hardware in Higher Order Logic, in: G. Milne, ed., The Fusion of Hardware Design and Verification, Proceedings of the IFIP WG 10.2 International Working Conference, Glasgow, Scotland, 3-6 July 1988, North-Holland, 1988, pp. 26-49. Also Report No. 135, Computer Laboratory, Cambridge University, January 1987.

[103] Thomas F. Melham, Formalizing Abstraction Mechanisms for Hardware Verification in Higher Order Logic, Ph.D. Thesis, Computer Laboratory, Cambridge University, (to appear).

[104] G. J. Milne, Behavioural Description and VLSI Verification, IEE Proceedings, Vol. 133, Part E, No. 3., May 1986, pp. 127-137.

[105] George J. Milne, Design for Verifiability, in: M. Leeser and G. Brown, eds., Specification, Verification and Synthesis: Mathematical Aspects, Proceedings of a Workshop, 5-7 July 1989, Ithaca, N.Y., Springer-Verlag, 1989.

[106] Robert Milne and Christopher Strachey, A Theory of Programming Language Semantics, Chapman and Hall, 1976.

[107] R. Milner and R. Weyhrauch, Proving Compiler Correctness in a Mechanized Logic, in: B. Meltzer and D. Mitchie, eds., Machine Intelligence, Vol. 7, Edinburgh University Press, Edinburgh, Scotland, 1972, pp. 51-70.

[108] A. R. J. G. Milner, A Theory of Type Polymorphism in Programming, Journal of Computer and System Science, Vol. 17, 1978.

[109] J Strother Moore, A Mechanically Verified Language Implementation, Report No. 30, Computational Logic Inc., Austin, Texas, September 1988.

[110] Francis Lockwood Morris, Correctness of Translations of Programming Languages, Ph.D. Thesis, Report STAN-CS-72-303, Computer Science Department, Stanford University, August 1972.

[111] F. Lockwood Morris, Advice on Structuring Compilers and Proving Them Correct, in: Proceedings of the ACM Symposium on Principles of Programming Languages, Boston, Mass., October 1973, pp. 144-152.

[112] Benjamin C. Moszkowski, Reasoning about Digital Circuits, Ph.D. Thesis, Report CS-83-970, Dept. of Computer Science, Stanford University, 1983.

[113] Benjamin Moszkowski, A Temporal Logic for Multilevel Reasoning about Hardware, IEEE Computer Vol. 18, No. 2, February 1985, pp. 10-19.

[114] Paliath Narendran and Johnathan Stillman, Formal Verification of the Sobel Image Processing Chip, in: G. Birtwistle and P. Subrahmanyam, eds., Current Trends in Hardware Verification and Automated Theorem Proving, Springer-Verlag, 1989, pp. 92-127.

[115] L. C. Paulson, Logic and Computation: Interactive Proof with Cambridge LCF, Cambridge University Press, 1987.

[116] L. C. Paulson, The Foundation of a Generic Theorem Theorem, Report No. 130, Computer Laboratory, Cambridge University, March 1987.

[117] Gordon D. Plotkin, A Structured Approach to Operational Semantics, Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, September 1981.

[118] Wolfgang Heinz Polak, Theory of Compiler Specification and Verification, Ph.D. Thesis, Report No. STAN-CS-80-802, Department of Computer Science, Stanford University, May 1980.

[119] Wolfgang Heinz Polak, Compiler Specification and Verification, Lecture Notes in Computer Science, No. 124, Springer-Verlag, 1981.

[120] C. Pygott, Electrical, Environmental and Timing Specification of the Viper Microprocessor, Memorandum No. 3753, RSRE, British Ministry of Defence, December 1984.

[121] Martin Richards, BSPL: A Language for Describing the Behaviour of Synchronous Hardware, Report No. 84, Computer Laboratory, Cambridge University, July 1986.

[122] A. W. Roscoe and C. A. R. Hoare, The Laws of Occam Programming, Technical Monograph PRG-53, Computing Laboratory, Oxford University, February 1986.

[123] John Rushby, Quality Measures and Assurance for AI Software, Report No. SRI-CSL-88-7R, Computer Science Laboratory, SRI International, Menlo Park, September 1988.

[124] John Rushby and Friedrich von Henke, Formal Verification of the Interactive Convergence Clock Synchronization Algorithm using EHDM, Report No. SRI-CSL-89-3, Computer Science Laboratory, SRI International, Menlo Park, February 1989.

[125] John Rushby, private communication, 1989.

[126] Bruce D. Russell, Implementation Correctness involving a Language with goto Statements, SIAM Journal of Computing, Vol. 6, No. 3, September 1977, pp. 403-415.

[127] C. Seitz, Chapter 7: System Timing, in: C. Mead and L. Conway, Introduction to VLSI Systems, Addison-Wesley, 1980, pp. 218-262.

[128] R. C. Sekar and M. K. Srivas, Formal Verification of a Microprocessor Using Equational Techniques, in: G. Birtwistle and P. Subrahmanyam, eds., Current Trends in Hardware Verification and Automated Theorem Proving, Springer-Verlag, 1989, pp. 171-218.

[129] N. Shankar, A Mechanical Proof of the Church-Rosser Theorem, Journal of the ACM, Vol. 35, No. 3, July 1988, pp. 475-522.

[130] D. Shepherd, The Role of Occam in the Design of the IMS T800, in: Communicating Process Architecture, 1988, pp. 93-103.

[131] Richard M. Stallman, EMACS: The Extensible, Customizable, Self-Documenting Display Editor, in: David R. Barstow, Howard E. Strobe and Erik Sandewall, eds., Interactive Programming Environments, McGraw-Hill, 1984, pp. 300-325.

[132] Joseph E. Stoy, The Scott-Strachey Approach to Programming Language Theory, The MIT Press, 1977.

[133] P. A. Subrahmanyam, Toward a Framework for Dealing with System Timing in Very High Level Silicon Compilers, in: G. Birtwistle and P. Subrahmanyam, eds., VLSI Specification, Verification and Synthesis, Kluwer Academic Publishers, 1988, pp. 159-215.

[134] P. A. Subrahmanyam, What's in a Timing Discipline ?: Considerations in the Specification and Synthesis of Systems with Interacting Asynchronous and Synchronous Components, in: M. Leeser and G. Brown, eds., Specification, Verification and Synthesis: Mathematical Aspects, Proceedings of a Workshop, 5-7 July 1989, Ithaca, N.Y., Springer-Verlag, 1989.

[135] Andrew S. Tanenbaum, Computer Networks, Prentice-Hall, 1981.

[136] Andrew S. Tanenbaum, Structured Computer Organization, Prentice-Hall, 1984 (second edition).

[137] J.W. Thatcher, E.G. Wagner, and J.B. Wright, More on Advice on Structuring Compilers and Proving Them Correct, Theoretical Computer Science, Vol. 15, September 1981, pp. 223-245.

[138] D. Turner, R. Burns and H. Hecht, Designing Micro-Based Systems for Fail-Safe Travel, IEEE Spectrum, Vol. 24, No. 2, February 1987, pp. 58-63.

[139] Stephen H. Unger, Asynchronous Sequential Switching Circuits, Wiley & Sons, 1969.

[140] John P. Van Tassel, The Semantics of VHDL with VAL and HOL: Towards Practical Verification Tools, M.Sc. Thesis, Dept. of Computer Science and Engineering, Wright State University, 1989.

[141] John Van Tassel and David Hemmendinger, Toward Formal Verification of VHDL Specifications, in: Luc Claesen, ed., Applied Formal Methods For Correct VLSI Design, 13-16 November 1989, Leuven, Belgium, (to be published by Elsevier Science Publishers).

[142] F. W. von Henke, J. S. Crow, R. Lee, J. M. Rushby and R. A. Whitehurst, The EHDM Verification Environment: An Overview, Proceedings of the 11th National Computer Security Conference, Baltimore, October 1988, pp. 147-155.

[143] Daniel Weise, Formal Multilevel Hierarchical Verification of Synchronous MOS VLSI Circuits, Ph.D Thesis, Report No. 978, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1987.

[144] William D. Young, A Verified Code Generator for a Subset of Gypsy, Report No. 33, Computational Logic Inc., Austin, Texas, October 1988.

[145] H. Zimmermann, OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection, IEEE Transactions on Communications, Vol. COM-28, No. 4, April 1980, pp. 425-432.

# Appendix: Compiler Verification

This appendix consists of a paper presented
at a workshop in Ithaca, N.Y., 5-7 July 1989.

# Totally Verified Systems: Linking Verified Software to Verified Hardware

Jeffrey J. Joyce
University of Cambridge

**Abstract.** We describe exploratory efforts to design and verify a compiler for a formally verified microprocessor as one aspect of the eventual goal of building *totally verified systems.* Together with a formal proof of correctness for the microprocessor, this yields a precise and rigorously established link between the semantics of the source language and the execution of compiled code by the fabricated microchip. We describe, in particular: (1) how the limitations of real hardware influenced this proof; and (2) how the general framework provided by higher-order logic was used to formalize the compiler correctness problem for a hierarchically structured language.

**Keywords.** compiler correctness, hardware verification, machine-assisted theorem proving, higher-order logic, safety-critical systems.

## 1. Introduction

Many safety-critical systems are implemented by a combination of hardware and software. The reliability of these systems depends not only on correct hardware and correct software, but also on the correctness of the compiler which provides the link between hardware and software levels. This paper describes exploratory efforts to design and verify a compiler for a formally verified microprocessor called 'Tamarack'. The source language is a very simple, hierarchically structured language with only a few basic constructs, e.g., expressions, assignment statements, while-loops, but this is enough to demonstrate how our approach could be applied to more realistic languages. We have used higher-order logic to formally specify this compiler and prove that it generates Tamarack machine code which executes correctly with respect to a denotational semantics for the source language.

The verification of this compiler builds upon an earlier proof of correctness showing that a transistor level model of the target machine satisfies a behavioural spec-

[1]Author's current address: Computer Laboratory, University of Cambridge, Pembroke Street, Cambridge CB2 3QG, England. After January 1, 1990: Department of Computer Science, University of British Columbia, 6356 Agricultural Road, Vancouver B.C., Canada V6T 1W5.

ification based on the semantics of the target machine instruction set [16]. The verification of both the compiler and the target machine in higher-order logic have been mechanically checked by the HOL system [13]. The HOL system has also been used to automatically generate substantial portions of these proofs.

The compiler correctness problem has a very long history beginning in the mid-1960's, but almost all of the previous work on this problem has been restricted to abstract, idealized target machines. These idealizations can include infinite word size and memory size, read-only code and symbolically addressed memory. By contrast, our target machine is not idealized hardware; indeed, an 8-bit version of the Tamarack microprocessor has been implemented as a CMOS microchip. Hence, our use of non-idealized hardware contributes to the more novel aspects of the work reported here.

Previous work has also generally relied upon specialized frameworks such as domain theory and algebraic concepts which are well-suited to the compiler correctness problem. But in the context of verifying both a compiler and the hardware of the target machine, a very general framework is needed to handle this many-sided problem. Such a framework is provided by the HOL system, a mechanization of higher-order logic, which has been used to reason about all kinds of computational systems.

Like most other examples of compiler verification, we ignore the problems of parsing and syntax analysis and use the abstract syntax of the source language as our starting point. The compiler is defined as a function which is applied to the parse tree of a program to generate code for the target machine. Semantic functions are applied in a similar way to the parse tree to generate the denotation of a program.

The work described here explores one aspect of the eventual goal of building *totally verified systems*. Assuming that our transistor level specification is an accurate model of the hardware, the compiler correctness proof combined with our earlier proof of correctness for the target machine results in a precise and rigorously established connection between the source language semantics and the execution of compiled code on the fabricated microchip. Hence, the semantics of the source language can be used to directly reason about the effect of running compiled programs on real hardware.

A detailed description of the Tamarack compiler and its formal verification is given in a separate report [17]. In this paper, we briefly outline the structure of this proof describing, in particular: (1) how the limitations of real hardware influenced this proof; and (2) how the general framework provided by higher-order logic was used to formalize the compiler correctness problem for a hierarchically structured language.

## 2. The Compiler Correctness Problem

The compiler correctness problem is easier to formulate than the general problem of program correctness. Unlike the general case, the compiler correctness problem has a built-in starting point for stating correctness, namely, the semantics of the source language. Intuitively, this problem is a question of whether the execution of compiled code agrees with the semantics of the source language. Compiler correctness is often expressed by the commutativity of a diagram similar to the one shown in Figure 1 where the two paths in the diagram from the source language programs to target language meanings (around opposite corners) are functionally identical.

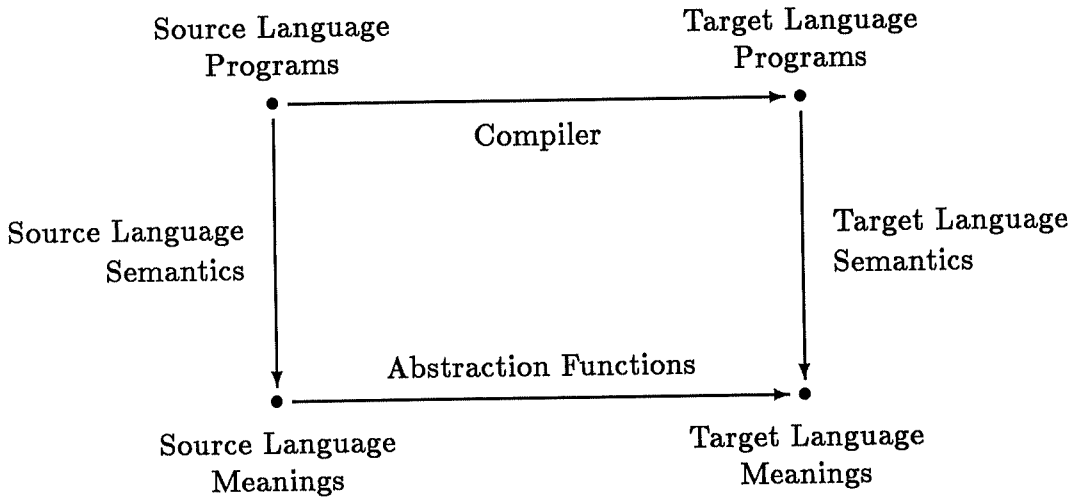



Figure 1: Compiler Correctness expressed by Commutativity.

The earliest example of compiler correctness (that we are aware of) was described more than twenty years ago by J. McCarthy and J. Painter [20]. They verified an algorithm for compiling arithmetic expressions into code for an abstract machine. This early work established a paradigm for subsequent work on compiler correctness (as summarized by A. Cohn [7]): (1) abstract syntax; (2) idealized hardware; (3) abstract specification of the compiler; (4) denotational source language semantics; (5) operational target machine semantics; (6) correctness stated as a relationship between the denotation of a program and the execution of its compiled form; and finally, (7) proofs by induction on the structure of source language expressions.

Subsequent developments include those described by: D. Kaplan [18]; R. Burstall and P. Landin [4]; R. Milner and R. Weyhrauch [23]; F. Morris [25,26]; L. Chirica [5]; R. Milne and C. Strachey [22]; J. Goguen et al. [11]; B. Russell [31]; A. Cohn [7]; W. Polak [29,30]; J. Thatcher et al. [33]; L. Chirica and D. Martin [6]; and P. Collier [9]. These developments include the use of algebraic methods and domain theory, more language features, verification by formal proof based on axioms and inference rules, mechanical assistance for proof-checking and proof-generation, and correctness proofs about parsing and syntax analysis.

However, all of the work mentioned above involves the use of a target machine with idealized features. Typically, the target machine has no finite limitations on word size or memory size. Another idealization is the use of read-only code, which avoids the problem of showing that a compiled program is not over-written during its execution. The target machine is occasionally provided with abstract mechanisms such as an infinite stack or display mechanism (admittedly, finite approximations of these mechanisms are available in real hardware). In some compiler verification examples, the memory of the target machine is addressed symbolically by program variables, dodging the problem of symbol table generation. Similarly, the target language may be block structured to avoid the complication of generating unique labels for instructions.

These idealizations, while simplifying the problem, can also be justified as reasonable strategies for structuring both the compiler and a proof of its correctness into several layers. Non-idealized aspects of hardware, in the context of programming language semantics and implementation, were recognized long ago; for instance, see papers by C. Hoare [15] and M. Newey [27]. But to our knowledge, these details and the attendant proof complexity have not been confronted until recently, in the work described here, and in J Moore's formal verification of the Piton assembler for the FM8502 microprocessor [24]. As part of the verified stack described by W. Bevier et al. [2], Piton provides considerable support as an intermediate language with stack-based instructions, typed data and recursive procedures [1]. Moore's proof takes account of the finite limitations of hardware; he also deals with issues such as allocating memory for program variables and loading compiled code and data into a single memory image. The semantics of Piton are given operationally by a formally defined interpreter expressed as a recursive function in the Lisp-like syntax of Boyer-Moore logic [3].

Our exploratory efforts with a simple 'toy' language are quite modest when compared to Moore's work on Piton. However, we have tackled a somewhat different problem by considering a hierarchically structured source language. We expect that methods similar to those described in this paper could be used to verify a compiler for a structured assembly language such as Vista [19] which is being used to write applications software for the (partially) verified Viper microprocessor [8,10,19]. Another important difference is the operational-style semantics of Piton in contrast to our denotational approach. We believe that the abstract and concise nature of a denotational semantics will be an advantage when compiler correctness results are used to relate conventional forms of reasoning about software (e.g., a verification condition generator based on Hoare logic) to the execution of compiled software on verified hardware.

[1] As another level in the verified stack described by Bevier et al. [2], W. Young has verified a code generator for a hierarchically structured source language with Piton as the target language [34].

## 3. The Source Language

Our source language is a very simple imperative language. It is not intended to be a useful programming language; it only provides a few basic constructs in order to demonstrate how our approach could be applied to more realistic languages. For instance, the only kind of compound arithmetic expression is a plus-expression. Conditional statements are not included because while-loops cover all the proof difficulties (and more) presented by conditional statements. We also simplify code generation by imposing an unusual restriction on plus-expressions and equal-expressions: the left-hand sides of these expressions must be atomic. An informal description of the abstract syntax for this language is shown below.

```
Aexp ::= {0,1,2,...} | Vars | Vars + Aexp
Bexp ::= Vars = Aexp | not Bexp
Cexp ::= skip | Vars := Aexp | Cexp ; Cexp | while Bexp do Cexp
```

There are three syntactic categories: arithmetic expressions, Boolean expressions and command expressions (or simply, commands). `Vars` is a set of string tokens which are used as variable names in programs, e.g., `'i'` and `'sum'` in the program shown in Figure 2. This program, called "SUM_0_to_9", computes the sum of the numbers 0 to 9 inclusive.

```
i := 0;
sum := 0;
while not (i = 10) do
  sum := sum + i;
  i := i + 1
```

Figure 2: The SUM_0_to_9 Program.

A denotational semantics for this simple language involves the definition of *semantic functions* for each syntactic category, namely, `SemAexp`, `SemBexp` and `SemCexp`. These functions map *syntactic objects* to their denotations as suggested by the type declarations,

$$\begin{aligned} \texttt{SemAexp}&: Aexp \rightarrow Asem \\ \texttt{SemBexp}&: Bexp \rightarrow Bsem \\ \texttt{SemCexp}&: Cexp \rightarrow Csem \end{aligned}$$

where $Aexp$, $Bexp$ and $Cexp$ are syntactic domains and $Asem$, $Bsem$ and $Csem$ are the corresponding semantic domains.

These semantic functions can be described informally by a set of *semantic clauses* using the *emphatic brackets* $[\![$ and $]\!]$ to surround syntactic objects when applying

semantic functions to them [12]. *Semantic operators* on the right-hand sides of these clauses are used to construct denotations from variables, constants and denotations of sub-expressions.

```
SemAexp ⟦v⟧ = SemVar v
SemAexp ⟦c⟧ = SemConst c
SemAexp ⟦v + aexp⟧ = SemPlus (v,SemAexp ⟦aexp⟧)

SemBexp ⟦v = aexp⟧ = SemEq (v,SemAexp ⟦aexp⟧)
SemBexp ⟦not bexp⟧ = SemNot (SemBexp ⟦bexp⟧)

SemCexp ⟦skip⟧ = SemSkip
SemCexp ⟦v := aexp⟧ = SemAssign (v,SemAexp ⟦aexp⟧)
SemCexp ⟦cexp1 ; cexp2⟧ = SemThen (SemCexp ⟦cexp1⟧,SemCexp ⟦cexp2⟧)
SemCexp ⟦while bexp do cexp⟧ = SemWhile (SemBexp ⟦bexp⟧,SemCexp ⟦cexp⟧)
```

To formally define the functions SemAexp, SemBexp and SemCexp, we need a suitable representation for syntactic objects. This representation must allow SemAexp, SemBexp and SemCexp to be defined as functions which satisfy the above (sometimes recursive) semantic clauses. The next section of this paper describes how syntactic objects can be represented in logic as parse trees.

## 4. Representing Hierarchical Structure

Many of the specialized frameworks used in earlier work on compiler verification directly support the representation of syntactic objects. While a general framework does not necessarily provide this support, it is still possible to represent syntactic objects using only rudimentary data types. We have demonstrated how this can be done in higher-order logic using a relatively concrete model for the representation of syntactic objects as parse trees such as the one shown in Figure 3.

In a conventional programming language, a parse tree can be implemented by an indexed list of records. The structure of the tree would be represented by pointers (record indices) in each record to zero, one or two sub-expression(s). Such data structures can be modelled in higher-order logic [2] using: (1) $n$-tuples to represent records; and (2) functions from indices to $n$-tuples to represent indexed lists of records. Since the representing type does not restrict how records are structurally composed into parse trees, it is necessary to have *validity predicates*, ValidAexp, ValidBexp and ValidCexp, to check whether a parse tree conforms to the abstract syntax of the source language.

[2]The HOL formulation of higher-order logic associates a *type* with every term. Every type is a primitive type (e.g., Booleans, natural numbers, string tokens) or built up from existing types using type constructors. Cartesian product is expressed by $ty1 \times ty2$ while $ty1 \rightarrow ty2$ denotes the type of all total functions with arguments of type $ty1$ and results of $ty2$.

Based on this representation for parse trees, we can define higher-order *mapping functions*, MapAexp, MapBexp and MapCexp, which allow a set of operations to be applied to the nodes of a parse tree in the same way that the Lisp function 'mapcar' allows an operation to be applied to the elements of a list. We use these mapping functions to define operations on parse trees by specifying operations for each kind of expression. These operations are applied recursively to the entire parse tree.

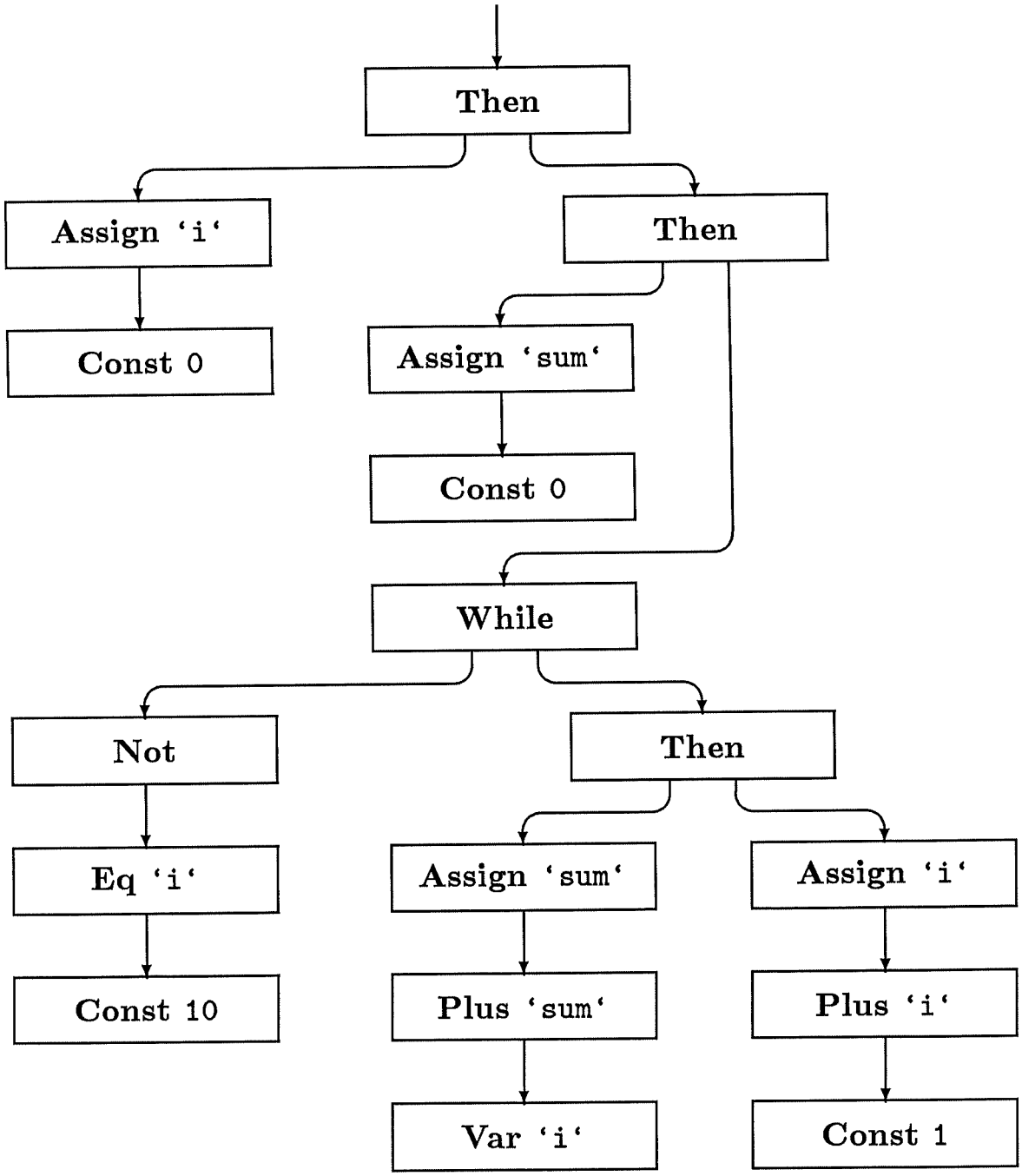



Figure 3: The Parse Tree for the SUM_0_to_9 Program.

For example, the definition of SemCexp (given in the next section) uses the mapping function MapCexp to recursively apply semantic operators to the parse tree of a command. This use of MapCexp is illustrated by the following term which denotes the result of applying SemCexp to the parse tree in Figure 3.

```
SemThen (
  SemAssign ('i',SemConst 0),
  SemThen (
    SemAssign ('sum',SemConst 0),
    SemWhile (
      SemNot (SemEq ('i',SemConst 10)),
      SemThen (
        SemAssign ('sum',SemPlus ('sum',SemVar 'i')),
        SemAssign ('i',SemPlus ('i',SemConst 1))))))
```

In addition to defining operations on expressions, we will also want to prove theorems about the result of applying such operations to expressions. To prove that a property holds for all expressions in a particular category, it is sufficient to show that the property holds for each kind of expression in the category assuming that it holds for all sub-expressions. This form of logical argument is called structural induction. Based on our representation for parse trees, we can prove structural induction theorems for each of the syntactic categories of the source language.

For instance, structural induction for arithmetic expressions is expressed by the following theorem. The predicates IsVar, IsConst and IsPlus are selectors for the three different kinds of arithmetic expressions and the function RightOf is used to obtain the sub-expression of a plus-expression.

$\vdash_{thm}$ $\forall$P.
(∀exp. IsVar exp ⟹ P exp) ∧
(∀exp. IsConst exp ⟹ P exp) ∧
(∀exp.
  IsPlus exp ∧ ValidAexp (RightOf exp) ⟹
  (P (RightOf exp) ⟹ P exp))
⟹
∀exp. ValidAexp exp ⟹ P exp

Structural induction only holds for valid parse trees; however, we may assume, as part of the inductive hypothesis, that the parse tree for the sub-expression is valid (in the case of a plus-expression). Structural induction theorems for Boolean and command expression have similar constraints.

The use of validity predicates to check whether a parse tree conforms to the abstract syntax of the source language is slightly cumbersome. Validity predicates provide a simple way to represent structure in a generalized framework using only rudi-

mentary data types. A more elegant approach avoids the use of validity predicates by formally introducing new types (as sub-types of the representing type) which contain (by definition) only valid syntactic objects.

We have used a relatively concrete representation for syntactic objects as collections of records organized into parse trees. The details of this representation are unimportant and are hidden at early point in our proof by the derivation of *abstract specifications* for the mapping functions `MapAexp`, `MapBexp` and `MapCexp` and the derivation of the above-mentioned structural induction theorems. In a more abstract approach, the unimportant details of a concrete representation can be entirely avoided by directly introducing a recursive type whose elements are (by definition) valid syntactic objects. This approach was taken by Cohn [7] working in LCF which is also a typed logic. This more abstract approach could also be followed in our higher-order logic framework - a task made easier by T. Melham's recent implementation of a recursive data types package for the HOL system [21].

To summarize this section, syntactic objects can be represented as parse trees which, in turn, can be represented by rudimentary data types in a generalized framework such as higher-order logic. Operations on parse trees can be defined in terms of a set of mapping functions; reasoning about parse trees is supported by a corresponding set of structural induction theorems. Full details on this representation, the mapping functions and the structural induction theorems are given in [17].

## 5. Semantics

A denotational semantics for the source language can be defined in higher-order logic using higher-order functions and relations as the denotations of expressions and commands respectively. This is a somewhat different framework than usual, i.e., Scott's logic for computable functions, but it is denotational in the sense that program constructs are mapped to abstract mathematical entities [12]. M. Gordon has also used higher-order logic to represent a denotational semantics in a similar manner [14].

The execution of a program is modelled by a sequence of states where each state is a mapping from variable names to their values. In this simple language only natural numbers can be assigned to variables. Hence, a state is represented by a function from string tokens to the natural numbers as shown by the following type abbreviation.

$$state = tok \rightarrow num$$

The execution of a source language program results in a sequence of updates to the current state. We use a standard model from denotational semantics for the effect

of an update. The function Update creates a new state identical to the current state except for the updated variable which is assigned a new value. The following definition introduces some of our notation: Update is defined in terms of a function-denoting $\lambda$-expression and a conditional expression of the form "b $\Rightarrow$ t1 | t2".

$\vdash_{def}$ Update (s:*state*,x,y) = $\lambda$z. (x = z) $\Rightarrow$ y | (s z)

The denotations for arithmetic and Boolean expressions are functions which specify the value of the expression in terms of the current state. The denotation of a command is a relation on pairs of initial and final states. The following type abbreviations summarize the types of denotations used for each of the three syntactic categories. These denotations are each parameterized by a number, namely, the word size of the target machine.

$$\begin{aligned} Asem &= num \rightarrow state \rightarrow num \\ Bsem &= num \rightarrow state \rightarrow bool \\ Csem &= num \rightarrow (state \times state) \rightarrow bool \end{aligned}$$

We can now begin to define semantic operators for expressions and commands in the source language. The definition of SemVar states that the denotation of a variable is its value in the current state. This operator is a *curried* function which takes its arguments 'one at a time'. When SemVar is applied to the first of its arguments, i.e., SemVar v, the result is a term with the type given by the type abbreviation *Asem* (where ws is the word size of the target machine).

$\vdash_{def}$ SemVar (v:*tok*) = $\lambda$ws. $\lambda$q. q v

The denotation of a constant is the value of the constant modulo the word size of the target machine. This use of the MOD function is due to our eventual goal of relating the semantics of the source language to the execution of compiled programs. Modular arithmetic is a convenient way of taking into account the finite word size of non-idealized hardware; an early example of this use of modular arithmetic appears in Hoare's seminal paper on axiomatic semantics [15].

$\vdash_{def}$ SemConst (c:*num*) = $\lambda$ws. $\lambda$q. c MOD $2^{ws}$

A plus-expression is an example of a compound expression; its denotation is obtained from its immediate constituents, in this case, from the sub-expression on the right-hand side of the '+'. Modular arithmetic is also used here to model the finite word size of the target machine.

$\vdash_{def}$ SemPlus (v:*tok*,s:*Asem*) = $\lambda$ws. $\lambda$q. ((q v) + (s ws q)) MOD $2^{ws}$

The semantic operator for equal-expressions is parameterized by the string token appearing on the left-hand side of the '=' and by the denotation of its arithmetic

sub-expression. The semantic operator for not-expressions is parameterized by the denotation of its Boolean sub-expression.

$\vdash_{def}$ SemEq (v:*tok*,s:*Asem*) = λws. λq. (q v) = (s ws q)

$\vdash_{def}$ SemNot (s:*Bsem*) = λws. λq. ¬(s ws q)

The semantic operators for commands yield relations on pairs of states. The simplest case is the Skip command which has no effect on the state. Therefore, the initial and final states of a Skip command are related if they are identical [3].

$\vdash_{def}$ SemSkip = λws. λ(q1,q2). q1 = q2

In the case of an assignment statement, the final state is obtained from the initial state by the Update function.

$\vdash_{def}$ SemAssign (v:*tok*,s:*Asem*) =
λws. λ(q1,q2). q2 = Update (q1,v,s ws q1)

In defining the semantics of a then-command (two commands in sequence), the two sub-commands must share a common intermediate state. Higher-order existential quantification is used to hide this intermediate state in the definition of SemThen. In a more standard framework, the denotation for a sequence of commands would be obtained by the functional composition of two partial functions. Partial functions allow for the possibility of non-terminating commands; however, all functions in higher-order logic are total. For this reason, we are using relations instead of partial functions. Our use of existential quantification for the denotation of a then-command is the analogue of functional composition for relations.

$\vdash_{def}$ SemThen (s1:*Csem*,s2:*Csem*) =
λws. λ(q1,q2). ∃q3. s1 ws (q1,q3) ∧ s2 ws (q3,q2)

The function Step is defined (by primitive recursion) to describe the condition where $n$ iterations of a while-loop result in a final state, that is, a state in which the Boolean condition is false. Zero iterations of the while-loop is equivalent to the execution of a Skip command; otherwise, $n$ iterations of the while-loop has the same effect as executing the body of the while-loop once followed by $n$-1 iterations of the while-loop. The semantic operators SemSkip and SemThen are used to define the zero and non-zero cases respectively. Since the actual number of iterations is not relevant to the semantics of a while-loop, this number is hidden by existential quantification in the definition of SemWhile.

[3] Predicates (including relations) in the HOL formulation of higher-order logic are simply functions which return Boolean values. Hence, the lambda expression, λ(q1,q2). q1 = q2 denotes the equality relation for pairs of states.

```
⊢def Step n (s1:Bsem,s2:Csem) ws (q1,q2) =
       (n = 0) ⇒ (((s1 ws q1) = F) ∧ SemSkip ws (q1,q2)) |
         (((s1 ws q1) = T) ∧
          SemThen (s2,Step (n-1) (s1,s2)) ws (q1,q2))

⊢def SemWhile (s1:Bsem,s2:Csem) =
       λws. λ(q1,q2). ∃n. Step n (s1,s2) ws (q1,q2)
```

Although our use of higher-order logic is an unusual framework for denotational semantics, some familiar properties can be derived for the semantic operators from the definitions given in this section. For instance, assuming for a moment that our source language also includes conditional statements, the while-loop ,

$$\ulcorner \text{while bexp do cexp} \urcorner$$

should have the same meaning as,

$$\ulcorner \text{if bexp then (cexp ; while bexp do cexp) else skip} \urcorner$$

This property is expressed formally by the theorem,

```
⊢thm ∀ s1 s2.
       SemWhile (s1,s2) =
         SemCond (s1,SemThen (s2,SemWhile (s1,s2)),SemSkip)
```

where SemCond is a semantic operator for conditional statements defined as:

```
⊢def SemCond (s1:Bsem,s2:Csem,s3:Csem) =
       λws. λ(q1,q2).
         ((s1 ws q1) = T) ⇒ (s2 ws (q1,q2)) | (s3 ws (q1,q2))
```

The operators, SemVar, SemConst, SemPlus, SemEq, SemNot, SemSkip, SemAssign, SemThen and SemWhile, describe how the denotation of an expression is obtained from its top-level form and the denotations of its sub-expressions. The denotation of a complete expression (including commands, and hence, complete programs) is obtained by using the mapping functions mentioned in Section 4 to recursively apply these operators to every node in a parse tree. From the *abstract specifications* for MapAexp, MapBexp and MapCexp given in [17], it is quite easy to show that the following definitions satisfy the semantic clauses given earlier in Section 3.

```
⊢def SemAexp = MapAexp (SemVar,SemConst,SemPlus)

⊢def SemBexp = MapBexp (SemAexp,SemEq,SemNot)

⊢def SemCexp =
       MapCexp (SemAexp,SemBexp,SemSkip,SemAssign,SemThen,SemWhile)
```

Later in this paper we will show how the mapping functions are used in a similar way to compile a complete program by recursively applying *compilation operators* to every node in a parse tree.

## 6. Compiler Overview

The Tamarack compiler is implemented by two phases. The original motivation for splitting the compilation process into two phases was to control the complexity of the formal proof of correctness. However, the use of an intermediate form is common practice in compiler design for more conventional reasons. For instance, it may be possible to compile more than one source language into the intermediate form and/or compile the intermediate form into the machine code of more than one target machine. This also suggests certain opportunities for re-using correctness results.

The first phase compiles the hierarchically structured program into a flat intermediate form called SM code. In general, this is a process of compiling an expression by first compiling its sub-expressions (if any) and then using the result to generate code for the expression itself. The second phase of the compiler assembles SM code into executable Tamarack machine code called TM code. To generate TM code from the intermediate form, a symbol table is constructed to map symbols in the source program to memory addresses. Each SM instruction is mapped to a fragment of TM code where each TM instruction is a 3-bit opcode and an address field packed together into a single memory word. This second phase of the compilation process performs (very simple versions of) the tasks associated with the assembler and linking loader in a conventional programming environment. The two phases of the compilation process are shown in Figure 4 where the example SUM_0_to_9 program is first compiled into SM code and then assembled into TM code.

As an intermediate form, SM code shares some common features with the source language. In both cases, storage is addressed symbolically by variable names and 'program space' is separate from data and cannot be over-written. However, SM code also shares some common features with the target language, in particular, they are both linear sequences of accumulator-based instructions.

The semantics of SM code are described operationally by the specification of an abstract machine (called an SM machine) which directly executes this intermediate form. The SM machine consists of a fixed program, an infinite, symbolically addressed store, a program counter and an accumulator. For simplicity, we have designed the SM machine to operate exclusively on natural numbers where multiple data types might otherwise have been used. The Boolean values `T` and `F` are represented by the natural numbers `0` and `1` respectively. Modular arithmetic is used to model the finite word size of the target machine.

```
i := 0;
sum := 0;
while not (i = 10) do
  sum := sum + i;
  i := i + 1
```

⇓

```
base:      ('CONST',ARB,0)
           ('ST','i',ARB)
           ('CONST',ARB,0)
           ('ST','sum',ARB)
base+4:    ('CONST',ARB,10)
           ('EQ','i',ARB)
           ('NOT',ARB,ARB)
           ('NOT',ARB,ARB)
           ('JZR',ARB,base+16)
           ('LD','i',ARB)
           ('ADD','sum',ARB)
           ('ST','sum',ARB)
           ('CONST',ARB,1)
           ('ADD','i',ARB)
           ('ST','i',ARB)
           ('JMP',ARB,base+4)
base+16:
```

⇒

```
base:      (LD n (base+2))
           (JMP n (base+3))
           (0 MOD 2^(n+3))
base+3:    (ST n (symtab 'i'))
base+4:    (LD n (base+6))
           (JMP n (base+7))
           (0 MOD 2^(n+3))
base+7:    (ST n (symtab 'sum'))
base+8:    (LD n (base+10))
           (JMP n (base+11))
           (10 MOD 2^(n+3))
base+11:   (SUB n (symtab 'i'))
           (JZR n (base+16))
           (LD n (base+15))
           (JMP n (base+16))
           1
base+16:   (JZR n (base+20))
           (LD n (base+19))
           (JMP n (base+23))
           0
           (LD n (base+22))
           (JMP n (base+23))
           1
base+23:   (JZR n (base+27))
           (LD n (base+26))
           (JMP n (base+30))
           0
           (LD n (base+29))
           (JMP n (base+30))
           1
base+30:   (JZR n (base+40))
base+31:   (LD n (symtab 'i'))
base+32:   (ADD n (symtab 'sum'))
base+33:   (ST n (symtab 'sum'))
base+34:   (LD n (base+36))
           (JMP n (base+37))
           (1 MOD 2^(n+3))
base+37:   (ADD n (symtab 'i'))
base+38:   (ST n (symtab 'i'))
base+39:   (JMP n (base+8))
base+40:
```

Figure 4.

## 7. Compiling Expressions and Commands

We begin to specify the compiler by defining a function for each kind of expression which compiles that expression into SM code. Each of these functions operates only on the top-level form of the expression; sub-expressions (if any) are compiled separately and the results supplied as arguments to the function. There is a close parallel between the role of these functions in compiling a hierarchically structured program and the semantic operators mentioned earlier in Section 5. For this reason, we call these functions *compilation operators.*

The intuitive sense in which the compilation operators for arithmetic and Boolean expressions are correct is fairly obvious. For instance, the compilation operator for plus-expressions is correct if and only if execution of the compiled code loads the sum of the sub-expression and the value of the program variable into the accumulator. In general, a compilation operator is correct if and only if the effect of executing the code generated for an expression or command agrees with its denotation generated by the corresponding semantic operator. In the case of an arithmetic expression, the value of the accumulator after executing the compiled code must be equal to the value given by its denotation in the current state. For a Boolean expression, the accumulator must contain either 0 or 1 depending on whether the denotation of the expression evaluates to true or false respectively.

Because commands do not necessarily terminate, the sense in which compilation operators for commands are correct is less obvious. By 'termination', we mean that the denotation of a command relates the initial state `q1` to a final state `q2`, i.e., that there exists a final state `q2`.

$\vdash_{def}$ `Terminates p ws q1 = ∃q2. SemCexp p ws (q1,q2)`

Termination, in this sense, is a property of the abstract mathematical entities denoted by source language programs; the question of whether the SM machine halts when the compiled form of the program is executed is *prima facie* a different matter. For an SM machine 'to halt', means that it eventually reaches the end of the SM code.

Using these distinct notions of termination and halting, the correctness of a compilation operator for a command is expressed by separate conditions for the terminating and non-terminating cases. In the terminating case, the SM machine must halt and the final state of its store must agree with the final state given by the corresponding denotation. In the non-terminating case, the SM machine must not halt.

After formalizing these intuitive notions of correctness, we prove that the compilation operator for each kind of expression is correct with respect to the corresponding semantic operator. These correctness results are obtained by a sequence of infer-

ences patterned on the *symbolic execution* of the compiled code for an expression. This use of the term 'symbolic execution' is purely descriptive; our proof technique is based entirely on the inference rules of higher-order logic.

This proof technique is straightforward for atomic expressions. Each step in the symbolic execution of the compiled code corresponds to the symbolic execution of a single SM instruction. A formal model of the SM machine is specified in terms of a *next state function* which is used to step through the code generated by the compilation operator for the atomic expression. After the appropriate number of steps, we show that the resulting state of the SM machine satisfies the correctness condition for this expression.

For compound expressions (including compound commands) symbolic execution involves steps corresponding to the execution of sub-expressions in addition to the execution of single SM instructions. We assume that the appropriate correctness conditions hold for the sub-expressions and use these assumptions to reason about the execution of each sub-expression as single steps in the symbolic execution of the compound expression. The remaining steps (steps corresponding to single SM instructions) are symbolically executed by an application of the next state function.

For example, the following theorem states that the top-level form of a plus-expression is compiled correctly by the compilation operator `CmpPlus` with respect to the denotation produced by the semantic operator `SemPlus`. The correctness condition for arithmetic expressions is expressed by the predicate `AexpCorrect`. The variables `c` and `s` are the compiled code and denotation respectively of the sub-expression on the right-hand side of the '+'.

$\vdash_{thm}$ $\forall$ `c s v.`
```
    AexpCorrect (c,s) ⟹
    AexpCorrect (CmpPlus (v,c),SemPlus (v,s))
```

Similar results are obtained for every other kind of expression in the source language. For most expressions, symbolic execution corresponds to a fixed sequence of steps. However, correctness results for while-loops are more difficult and involve proofs by mathematical induction. The terminating case for while-loops is proved by mathematical induction on the number of iterations. The non-terminating case is even more difficult because there is more than one way that a while-loop can fail to terminate: at any point, the body of the while-loop may fail to terminate, or else the while-loop itself may continue to loop forever.

There are two essential ideas being used here to reason about compound expressions. One is the idea of using assumptions about the correctness of sub-expressions to prove correctness results for compound expressions. The other is the idea of 'mixed-mode' symbolic execution where single steps correspond to either single SM instructions or to sub-expressions.

## 8. Compiling Complete Programs

In section 4 we showed how semantic functions for each syntactic category can be defined by applying the mapping functions MapAexp, MapBexp and MapCexp to the semantic operators. In a similar manner, *compilation functions* for each syntactic category can be obtained by applying the mapping functions to the compilation operators.

$\vdash_{def}$ `CmpAexp = MapAexp (CmpVar,CmpConst,CmpPlus)`

$\vdash_{def}$ `CmpBexp = MapBexp (CmpAexp,CmpEq,CmpNot)`

$\vdash_{def}$ `CmpCexp =`
`MapCexp (CmpAexp,CmpBexp,CmpSkip,CmpAssign,CmpThen,CmpWhile)`

The correctness of these compilation functions is easily established from correctness results for each compilation operator using the structural induction theorems mentioned in Section 4.

These correctness results lead directly to the following theorem where the variable p denotes any source language program. The predicate SMHalts is defined directly in terms of the formally specified model of the SM machine. For a given SM program, SMHalts describes a relation on pairs of states (q1,q2) where the SM machine begins execution in state q1 and eventually halts in state q2. Hence, SMHalts is a semantic function for SM code.

$\vdash_{thm}$ $\forall$`p. ValidCexp p` $\Longrightarrow$ `(SemCexp p = SMHalts (CmpCexp p))`

This theorem is the main result from the first part of our compiler correctness proof: it relates the denotational semantics of our source language to an operational semantics given by SMHalts applied to the compiled code generated by CmpCexp. We are using the term 'operational semantics' in a somewhat old-fashioned sense [4] where the semantics is given by an abstract machine and a translation from the source language into code for the abstract machine [32].

This result can also be expressed by the commutative diagram in Figure 5 which is similar to diagrams found in other discussions of the compiler correctness problem. In this case, there is no need for an abstraction function from source language meanings to target language meanings since they are identical. Consequently, the diagram has only three sides.

[4] A more recent form of operational semantics known as *Plotkin-style* or *natural* semantics has both structure and some denotational-style features [28].

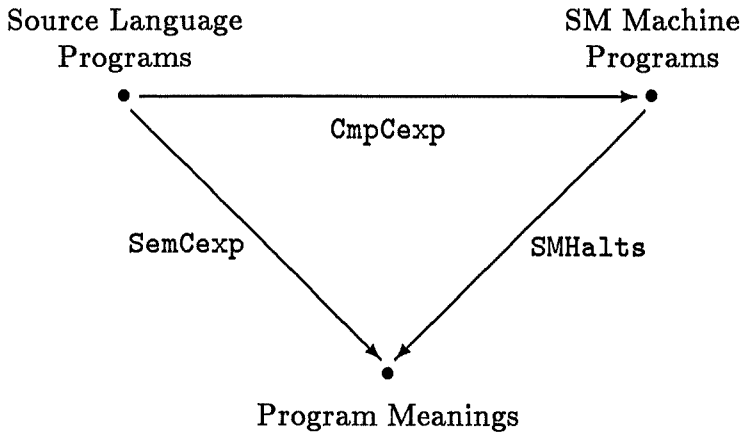



Figure 5: Compiler Correctness expressed by Commutativity.

The second part of our correctness proof considers the assembly of SM programs into TM code establishing a correspondence between the direct execution of an SM program and the execution of an assembled SM program by the target machine. Later, this result is combined with the above theorem to obtain a direct correspondence between the denotation of a source language program and the execution of its compiled form by the target machine.

## 9. Assembling Intermediate Code

The external architecture of the Tamarack microprocessor consists of three state components: the memory, program counter and accumulator. A single instruction word format is used by all Tamarack instructions: a 3-bit opcode followed by $n$ address bits. The actual size of the address field is given by a parameter throughout the formal proof of correctness. The transistor level model of the Tamarack implementation is also parameterized by the size of the address field. The correctness of this implementation has been established for all possible sizes.

The assembly of SM code into TM code requires the generation of a *symbol table*, `symtab`, which maps string tokens appearing in SM instructions to memory addresses. Symbols (i.e., string tokens) are only added to the table when they appear on the left-hand side of an assignment statement in the source language program. Since each assignment statement corresponds to an `ST` instruction in the resulting SM code, symbols are only added to the table when they appear in the SM code as an operand in an `ST` instruction. The symbol table is generated by a single pass over the SM program. When a new symbol is added to the table, it is assigned the address of the next available location in the data area of memory.

The assembly of SM code into TM code also requires an *address table*, `addrof`, which maps locations in the SM code to corresponding locations in the TM code.

The location of the TM code generated for a particular SM instruction can be determined by adding up the sizes of the code fragments generated for all preceding SM instructions. This table can also be generated by a single pass over the SM program.

The symbol table, `symtab`, and address table, `addrof`, are used in a third pass over the SM program to generate the TM code. Most SM instructions are assembled into a single TM instruction. However, a `CONST` instruction, used to load a constant into the accumulator, is assembled into a three word fragment of TM code: an `LD` instruction; a `JMP` instruction; and the constant itself which is stored in a separate memory word (i.e., as an immediate constant). The `JMP` instruction prevents the constant from being executed as an instruction. The SM instructions `EQ` and `NOT` are also assembled into multiple words of TM code. This is because `0` and `1`, representing true and false respectively, are stored as immediate constants [5].

For conceptual clarity, we have separated the assembly of SM code into three successive passes. However, there are well-known techniques, e.g., 'back-patching', which can be used to reduce the number of passes in a compiler [1].

Each of the three passes used to assemble SM code into TM code can be formally defined as an operation applied iteratively to a sequence of SM instructions; in concrete terms these functions can be defined by primitive recursion on the size of the SM code. As one might expect, correctness results for each of these passes over the SM code will involve proofs by induction on the size of the SM code.

Correctness results for symbol table generation show that the iteratively generated symbol table has several properties needed to prove that SM code is correctly assembled into TM code. For instance, we show that different symbols are mapped to different addresses. Several other less obvious properties are described in [17].

The rest of the proof is concerned with showing that SM code is correctly assembled into TM code. Intuitively, it is fairly obvious what conditions need to be satisfied: execution of the TM code must correspond to the execution of the SM program. There are several provisos, most of which arise from limitations of the finite word size and finite memory size of the target machine.

Earlier steps in the correctness proof have already been influenced by the finite limitations of the target machine: the finite word size of the target machine is a feature of both the denotational semantics of the source language and the operational semantics of SM code. However, correctness results for the first compiler phase place no bounds on the size of the SM code or the size of the store. Therefore, finite limitations of the target machine are more important in the second part of the

[5] The use of immediate constants was slightly easier (in the initial effort of developing this proof) than the more economical approach of storing a single instance of these constants in memory.

correctness proof when showing that SM code is correctly assembled into TM code. The size of addressable memory is limited by the number of bits in the address field of a target machine instruction. The memory area reserved for code must be large enough to accommodate the code generated by the assembler. Similarly, the area reserved for data must provide a separate memory word for each symbol in the symbol table. These two areas of memory must not overlap and cannot exceed the boundaries of addressable memory. We assume explicitly that these conditions are satisfied in proving that SM code is correctly assembled into TM code.

The sense in which the execution of TM code 'corresponds' to the execution of an SM program is, roughly speaking, the condition that updates to the memory state, program counter and accumulator of the target machine correspond to updates to the store, program counter and accumulator of the SM machine. There are three distinct steps in proving that execution of the assembled form of an SM program corresponds to its direct execution by the SM machine. These three steps are very briefly summarized in the next few paragraphs.

The first step establishes that the execution of the compiled form of individual SM instructions corresponds to their direct execution by the SM machine. This step in the proof is concerned with the fragments of TM code generated for each SM instruction. For each SM instruction, we prove that the symbolic execution of the TM code fragment by repeated applications of the next state function for the target machine corresponds to a single application of the next state function for the SM machine. This step also proves that execution of the code fragment does not over-write any part of the TM code.

The second step establishes that the fragments of TM code generated for each SM instruction are correctly composed into a single fragment of TM code for the entire SM program. This step is proved by mathematical induction on the size of the SM program.

The third step establishes that the execution of an assembled SM program corresponds to its direct execution by the SM machine for any number of execution steps (within the limitations of the target machine). This step is proved by mathematical induction on the number of execution steps.

The correctness result obtained from these three steps states precise details about the relationship between the execution of an assembled SM program and its direct execution by the SM machine. In very simple terms, there exists an SM machine which provides an abstract model of the target machine while executing the compiled SM program. Therefore, true statements about the direct execution of the SM program are also true statements about the execution of its compiled form by the target machine. This theorem is used in combination with earlier results to obtain a correctness result for the complete compilation process.

## 10. Combining Two Levels of Correctness Results

The final step in the verification of the Tamarack compiler combines correctness results for the two phases of the compilation process.

Earlier correctness results for the first compiler phase established that direct execution of the SM code generated from a terminating source language program will result in a final state which agrees with its denotation. In the case of a non-terminating program, the SM machine will not halt. For the second compiler phase, we have just seen that 'true statements about the direct execution of the SM program are also true statements about the execution of its assembled form by the target machine'.

The combination of these two results implies that a terminating source language program will be compiled into target machine code which will execute to completion and yield a final state which agrees with its denotation. This depends, of course, on whether the compiled program can be loaded into addressable memory. A precise statement of this result uses the symbol table generated by the compiler for this program to relate memory states of the target machine to the denotation of the source language program. In the case of a non-terminating program, the target machine will never complete execution of the compiled code. The correctness theorem for the terminating case is shown below.

$\vdash_{thm}$
```
∀ p n mem.
  ValidCexp p ∧
  CompiledAndLoaded n p (mem 0) ∧
  Terminates p (n+3) ((mem 0)∘(SymTab p))
  ⟹
  ∀ pc acc.
    TM n (mem,pc,acc) ∧
    (pc 0 = 0)
    ⟹
    ∃t.
      FirstReaches (pc,t,EndOfProg p) ∧
      SemCexp p (n+3) ((mem 0)∘(SymTab p),(mem t)∘(SymTab p))
```

To paraphrase this theorem: if the compiled code for a syntactically valid, terminating program is loaded into memory at location 0 and executed by the target machine (whose behaviour is given by the predicate TM) beginning at time 0, then the target machine will eventually reach the end of the code at some time t. When execution of this code is completed, an abstract view of the initial memory state will be related to an abstract view of the final memory state by the denotation of the program. An 'abstract view' of the memory state is obtained by using the symbol table to access the contents of the target machine memory; in the above theorem, this is expressed by use of the operator '∘' which denotes functional composition.

## 11. Summary

Our main correctness theorem provides a direct link between the semantics of the source language and the behavioural specification of the Tamarack microprocessor. When coupled with an earlier proof of correctness relating this behavioural specification to a transistor level model of the hardware, we obtain a precise and rigorously established connection between the denotation of a source language program and the effect of executing its compiled form on actual hardware.

A link between software and hardware levels provides a sound basis for using the semantics of the source language to reason about programs. In related work, Gordon [14] shows how Hoare logic can be embedded in higher-order logic by regarding the syntax of Hoare formulae as abbreviations for higher-order logic formulae. The axioms and inference rules of Hoare logic are then derived from semantic operators similar to the semantic operators defined in Section 5 of this paper. This means that theorems proved in Hoare logic using these axioms and rules are logical consequences of the underlying denotational semantics.

To relate this work to our correctness results for the Tamarack compiler, we would need to slightly re-formulate the axioms and rules of Hoare logic to take account of the finite size of memory words as we have done for the semantic operators in Section 5. It would then follow that theorems proved in Hoare logic about a particular program are true statements about the result of executing the compiled program on the fabricated microchip. This depends, of course, on both explicit conditions, e.g., whether the compiled code fits into addressable memory, and implicit assumptions, e.g., that the transistor level specification is an accurate model of the hardware.

In this exploratory effort, we have not ventured beyond a traditional view of formal semantics that the meaning of a program is either a partial function from initial states to final states or, as in our approach, a relation between initial and final states. However, we are interested in embedded systems where a 'batch processing' view of program behaviour is not entirely appropriate. These systems continuously interact with an environment; they are typically implemented by a fixed program compiled and loaded into the memory of one or more microprocessors. Unlike a batch job, execution of the compiled code is meant to execute forever, or at least, until the microprocessor is reset or switched off. Instead of a final outcome, we are interested in the on-going behaviour of the microprocessor while executing the compiled code. We are concerned, for instance, that the system responds correctly to external stimuli or that certain invariants are maintained. Therefore, an important direction of future work will be to investigate the relationship between suitable kinds of semantics for proving the correctness of a compiler and formalisms which can be used to reason about continuously-operating systems.

## Acknowledgements

I am grateful to Juanito Camilleri, Avra Cohn, Mike Gordon, John Herbert, Miriam Leeser and Glynn Winskel who have helped in various ways with this work. This research has been funded by the Cambridge Commonwealth Trust, the Canada Centennial Scholarship Fund, the Government of Alberta Heritage Fund, the Natural Sciences and Engineering Research Council of Canada, Pembroke College, and the UK Overseas Research Student Awards Scheme.

# References

[1] Alfred V. Aho and Jeffrey D. Ullman, Principles of Compiler Design, Addison-Wesley, Reading, MA., 1977.

[2] William R. Bevier, Warren A. Hunt, Jr., and William D. Young, in: Towards Verified Execution Environments, in: Procs. of the 1987 IEEE Symposium on Security and Privacy, 27-29 April 1987, Oakland, California Computer Society Press, Washington, D.C., 1987 pp. 106-115. Also Technical Report No. 5, Computational Logic, Inc., Austin, Texas, February 1987.

[3] R.S. Boyer and J S. Moore, A Computational Logic, Academic Press, New York, 1979.

[4] R.M. Burstall and P.J. Landin, Programs and their Proofs: an Algebraic Approach, in: B. Meltzer and D. Mitchie, eds., Machine Intelligence, Vol. 4, Edinburgh Univ. Press, Edinburgh, Scotland, 1969. pp. 17-43.

[5] L.M. Chirica, Contributions to Compiler Correctness, Ph.D. Thesis, Report UCLA-ENG-7697, Computer Science Dept., Univ. of California, Los Angeles, October 1976.

[6] Laurian M. Chirica and David F. Martin, Toward Compiler Implementation Correctness Proofs, ACM Transactions on Programming Languages and Systems, Vol. 8, No. 2, April 1986, pp. 185-214.

[7] Avra Jean Cohn, Machine Assisted Proofs of Recursion Implementation, Ph.D. Thesis, Technical Report CST-6-79, Dept. of Computer Science, Univ. of Edinburgh, April 1980.

[8] Avra Cohn, Correctness Properties of the Viper Block Model: The Second Level, in: G. Birtwistle and P. Subrahmanyam, eds., Current Trends in Hardware Verification and Automated Theorem Proving, Springer-Verlag, New York, 1989, pp. 1-91. Also Report No. 134, Computer Lab., Cambridge Univ., May 1988.

[9] P.A. Collier, Simple Compiler Correctness - A Tutorial on the Algebraic Approach, Australian Computer Journ., Vol. 18, No. 3, August 1986, pp. 128-135.

[10] W.J. Cullyer, High Integrity Computing, in: M. Joseph, ed., Formal Techniques in Real-Time and Fault-Tolerant Systems, Lecture Notes in Computer Science, No. 331, Springer-Verlag, Berlin, 1988. pp. 1-35.

[11] J.A. Goguen, J.W. Thatcher, E.G. Wagner, and J.B. Wright, Initial Algebra Semantics and Continuous Algebra, Journ. of the ACM, Vol. 24, No. 1, January 1977, pp. 68-95.

[12] Michael J. C. Gordon, The Denotational Description of Programming Languages, Spring-Verlag, Berlin, 1979.

[13] Mike Gordon, A Proof Generating System for Higher-Order Logic, in: G. Birtwistle and P. Subrahmanyam, eds., VLSI Specification, Verification and Synthesis, Kluwer Academic Publishers, Boston, 1988, pp. 73-128. Also Report No. 103, Computer Lab., Cambridge Univ., January 1987.

[14] Michael J. C. Gordon, Mechanizing Programming Logics in Higher Order Logic, in: G. Birtwistle and P. Subrahmanyam, eds., Current Trends in Hardware Verification and Automated Theorem Proving, Springer-Verlag, New York, 1989, pp. 387-439. Also Report No. 145, Computer Lab., Cambridge Univ., September 1988.

[15] C.A.R. Hoare, An Axiomatic Basis for Computer Programming, Communications of the ACM, Vol. 12, No. 10, October 1969, pp. 576-583.

[16] Jeffrey J. Joyce, Formal Specification and Verification of Microprocessor Systems, in: S. Winter and H. Schumny, eds., Euromicro 88, Procs. of the 14th Symposium on Microprocessing and Microprogramming, Zurich, Switzerland, 29 August - 1 September, 1988, North-Holland, Amsterdam, 1988, pp. 371-378. Also Report No. 147, Computer Lab., Cambridge Univ., September 1988.

[17] Jeffrey J. Joyce, A Verified Compiler for a Verified Microprocessor, Report No. 167, Computer Lab., Cambridge Univ., March 1989.

[18] Donald M. Kaplan, Correctness of a Compiler for Algol-like Programs, Stanford Artificial Intelligence Memo No. 48, Stanford Univ., July 1967.

[19] J. Kershaw, The VIPER Microprocessor, Report No. 87014, RSRE, Malvern, UK Ministry of Defence, November 1987.

[20] J. McCarthy and J. Painter, Correctness of a Compiler for Arithmetic Expressions, in: J. Schwartz, ed., Procs. of a Symposia on Applied Mathematics, American Mathematical Society, 1967, pp. 33-41.

[21] Thomas F. Melham, Automating Recursive Type Definitions in Higher Order Logic, in: G. Birtwistle and P. Subrahmanyam, eds., Current Trends in Hardware Verification and Automated Theorem Proving, Springer-Verlag, New York, 1989, pp. 341-386. Also Report No. 146, Computer Lab., Cambridge Univ., September 1988.

[22] Robert Milne and Christopher Strachey, A Theory of Programming Language Semantics, Chapman and Hall, London, 1976.

[23] R. Milner and R. Weyhrauch, Proving Compiler Correctness in a Mechanized Logic, in: B. Meltzer and D. Mitchie, eds., Machine Intelligence, Vol. 7, Edinburgh Univ. Press, Edinburgh, Scotland, 1972, pp. 51-70.

[24] J Strother Moore, A Mechanically Verified Language Implementation, Report No. 30, Computational Logic Inc., Austin, Texas, September 1988.

[25] F. Lockwood Morris, Correctness of Translations of Programming Languages, Ph.D. Thesis, Report STAN-CS-72-303, Computer Science Dept., Stanford Univ., August 1972.

[26] F. Lockwood Morris, Advice on Structuring Compilers and Proving Them Correct, in: Procs. of the ACM Symposium on Principles of Programming Languages, Boston, Mass., October 1973, pp. 144-152.

[27] Malcolm C. Newey, Proving Properties of Assembly Language Programs, in: B. Gilchrist, ed., Information Processing 77, North Holland, 1977, pp. 795-799.

[28] Gordon D. Plotkin, A Structured Approach to Operational Semantics, Technical Report DAIMI FN-19, Computer Science Dept., Aarhus Univ., September 1981.

[29] Wolfgang Heinz Polak, Theory of Compiler Specification and Verification, Ph.D. Thesis, Report No. STAN-CS-80-802, Dept. of Computer Science, Stanford Univ., May 1980.

[30] Wolfgang Heinz Polak, Compiler Specification and Verification, Lecture Notes in Computer Science, No. 124, Springer-Verlag, Berlin, 1981.

[31] Bruce D. Russell, Implementation Correctness involving a Language with goto Statements, SIAM Journ. of Computing, Vol. 6, No. 3, September 1977, pp. 403-415.

[32] Joseph E. Stoy, The Scott-Strachey Approach to Programming Language Theory, The MIT Press, Cambridge MA., 1977.

[33] J.W. Thatcher, E.G. Wagner, and J.B. Wright, More on Advice on Structuring Compilers and Proving Them Correct, Theoretical Computer Science, Vol. 15, September 1981, pp. 223-245.

[34] William D. Young, A Verified Code Generator for a Subset of Gypsy, Report No. 33, Computational Logic Inc., Austin, Texas, October 1988.