Technical Report

UCAM-CL-TR-140
ISSN 1476-2986

Number 140

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Executing behavioural definitions in higher-order logic

Albert John Camilleri

July 1988

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

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

This technical report is based on a dissertation submitted February 1988 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Darwin College.

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

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

ISSN 1476-2986

Abstract

Over the past few years, computer scientists have been using formal verification techniques to show the correctness of digital systems. The verification process, however, is complicated and expensive. Even proofs of simple circuits can involve thousands of logical steps. Often it can be extremely difficult to find correct device specifications and it is desirable that one sets off to prove a correct specification from the start, rather than repeatedly backtrack from the verification process to modify the original definitions after discovering they were incorrect or inaccurate.

The main idea presented in the thesis is to amalgamate the techniques of simulation and verification, rather than have the latter replace the former. The result is that behavioural definitions can be simulated until one is reasonably sure that the specification is correct. Furthermore, proving the correctness with respect to these simulated specifications avoids the inadequacies of simulation, where it may not be computationally feasible to demonstrate correctness by exhaustive testing. Simulation here has a different purpose: to get specifications correct as early as possible in the verification process. Its purpose is not to demonstrate the correctness of the implementation—this is done in the verification stage when the very same specifications that were simulated are proven correct.

The thesis discusses the implementation of an executable subset of the HOL logic, the version of Higher Order Logic embedded in the HOL theorem prover. It is shown that hardware can be effectively described using both relations and functions; relations being suitable for abstract specification, and functions being suitable for execution. The differences between relational and functional specifications are discussed and illustrated by the verification of an n-bit adder. Techniques for executing functional specifications are presented and various optimisation stratagies are shown which make the execution of the logic efficient. It is further shown that the process of generating optimised functional definitions from relational definitions can be automated. Example simulations of three hardware devices (a factorial machine, a small computer and a communications chip) are presented.

Declaration

I hereby declare that this thesis is the result of my own work and, unless explicitly stated in the text, includes nothing which is the outcome of work done in collaboration. No part of this thesis has already been, or is currently being, submitted for any degree, diploma or other qualification at any other university.

Albert John Camilleri
February 14, 1988

Acknowledgements

I am deeply grateful to my supervisor Mike Gordon for his inspiration, guidance and encouragement. Special thanks are also due to Tom Melham for several technical discussions and for reading various drafts of this thesis, including those early drafts which I was too embarrassed to spread around. Mike Gordon also made many useful contributions to later drafts of the thesis.

Other people contributed in various ways: John Herbert advised on the details of the ECL chip; Jon Fairburn came up with all the answers to the 101 questions I posed on lazy-evaluation; and Simone Camilleri made many suggestions on the layout and helped with the intricate work involved in drawing LaTeX diagrams.

Thanks are due to Martyn Johnson and Graham Titmus for keeping the systems going, and to all in the Hardware Verification Group for all the interesting meetings, discussions, parties and pub-crawls, especially to the main protagonists: Avra Cohn, Francisco Corella, Inder Dhingra, Thomas Forster, Don Gaubatz, Mike Gordon, Roger Hale, John Herbert, Jeff Joyce, Miriam Leeser, Tom Melham and Ben Moszkowski. All members of the department helped by keeping a cheerful and lively atmosphere.

I would also like to thank Simone for her constant support, encouragement and patience, and my parents for giving me the opportunity to learn and teaching me never to give up.

My studies at Cambridge were supported by a scholarship from the Association of Commonwealth Universities, and by an award from the Lundgren Research Fund. The Dean of Darwin College and Graham Birtwistle of the University of Calgary helped with funding to attend conferences which contributed tremendously towards my research.

Dedication

There is one special person without whom this thesis could never have been—the person who inspired me, filled me with hope and confidence, and spurred me on when I most needed it. To this person I dedicate this thesis.

To Simone

Contents

1 Introduction 1

1.1 Hardware Verification and Simulation 1

1.2 Background and Related Work 3

1.2.1 Simulators and Hardware Description Languages 3

1.2.2 Verification 6

1.2.3 Verification and Simulation 6

1.2.4 Programming Languages and Simulation 9

1.3 Overview of Thesis 10

2 The HOL Theorem Prover 13

2.1 The HOL logic 13

2.1.1 Terms 14

2.1.2 Types 15

2.1.3 Hilbert's ε-operator 17

2.2 The HOL Meta-language 17

2.3 The HOL System 23

2.3.1 Theories: Definitions, Axioms and Theorems 24

2.3.2 Primitive Inference Rules 25

2.3.3 Tactics and Tacticals 26

3 Specifying and Verifying Hardware 29

3.1 Relational and Functional Specifications 29

3.1.1 Verification using Relations 32

3.1.2 Verification using Functions 33

3.2 Representation of n-bit Words 33

3.3 A 1-bit Full-Adder 35

3.4 An n-bit Adder 39

3.4.1 Formal Proof of Relational Specifications 42

3.4.2 Formal Proof of Functional Specifications 43

3.5 Proofs Relating Functions and Relations 46

4 Executing Specifications 49
4.1 ML as a Simulation Language 49
4.2 Combinational Circuits 50
4.2.1 Modelling Logic Gates 50
4.2.2 Modelling Behaviour 52
4.2.3 Modelling Structure 53
4.2.4 Dealing with Partial Specifications 55
4.3 Sequential Circuits 56
4.3.1 Time, Delay and Clocks 57
4.3.2 Feedback 59
4.4 Example of a Simple Counter 60
4.5 ML or ELLA 63

5 Faster Simulation Techniques 69
5.1 Inefficiencies in Basic Method 69
5.2 Memoisation 73
5.2.1 The Algorithm 73
5.2.2 Implementation 76
5.2.3 Memoisation of the Counter 77
5.3 Lazy Evaluation 79
5.3.1 General Simulation Principles 80
5.3.2 The Counter using Lazy Evaluation 84
5.4 Memo-functions or Infinite Lists 85

6 Executing the HOL logic 87
6.1 From Relations to Functions 88
6.2 Automatically Translating Relations 90
6.2.1 Translating Predicates 91
6.2.2 Simple Relational Definitions 92
6.2.3 Primitive Recursive Definitions 95
6.2.4 Relations Involving History Functions 96
6.3 Relations Not Automatically Translated 98
6.3.1 Relations with No Functional Interpretation 99
6.3.2 Relations Requiring Normalisation 100
6.3.3 Relations with Non-Executable Interpretations 101
6.4 Automatic Translation of HOL to ML 103

7 Simulating a Factorial Machine **107**
7.1 The Specification 108
7.2 The Implementation 109
7.3 Implementation Using Simpler Primitives 114
7.4 Example Simulations 121

8 Simulating a Computer **125**
8.1 Description of Gordon's Computer 126
8.2 Setting Up *wordn* Types in ML 128
8.3 The Target Machine 132
8.4 The Host Machine 135
8.5 Executing Programs on Gordon's Computer 144

9 Simulating the ECL Chip **149**
9.1 The Specification 149
9.2 The Implementation 151
9.3 Example Simulations 157

10 Conclusions and Future Research **161**
10.1 Summary of Thesis 161
10.2 Specifying Behaviour 163
10.2.1 Automatic Manipulation of Specifications 163
10.3 Executing Logic Specifications 163
10.3.1 From HOL Relations to HOL Functions 164
10.3.2 From HOL Functions to ML Functions 165
10.3.3 Optimising ML Functions 165
10.4 Automatically Translating Types 166
10.5 Simulation at a Lower Level 166

Bibliography **169**

A Grammar for Parsing Relations **177**

B Example Interactive Session **179**

List of Figures

1.1 The Ideal Verification Process 2

3.1 A Full-Adder 36
3.2 Implementations of CARRY and SUM 36
3.3 Specification of a Binary Adder 39
3.4 Implementation of a Binary Adder 40
3.5 Correctness Theorems 47

4.1 Specification Diagram of a Counter 61
4.2 Implementation of a Counter 61
4.3 Implementation of a Half-Adder 64

5.1 Example Device for Illustrating Inefficient Modelling 70
5.2 Tree Illustration of Inefficient Evaluation 72

6.1 Relational Expressions in the HOL logic 89

7.1 A Factorial Machine 108
7.2 Implementation of Factorial Device 110
7.3 Implementation of Down 117
7.4 Implementation of Mult 118
7.5 Implementation of Test 119

8.1 Front Panel of Gordon's Computer 126
8.2 Implementation of Gordon's Computer 136
8.3 Representation of Memory for Program to Add Two Integers . . . 145
8.4 Representation of Program Executed on Gordon's Computer . . . 146
8.5 Table Displaying Stages of Simulation 147

9.1 The ECL Chip 150
9.2 Top-level Implementation of ECL Chip 152
9.3 Implementation of the SHIFTREGS Device 153

9.4 Implementation of a Shift Register 154
9.5 Implementation of a Register Cell 154
9.6 Waveforms Showing Inputs to ECL Chip 157
9.7 Waveforms Showing Outputs from ECL Chip 158

Chapter 1

Introduction

1.1 Hardware Verification and Simulation

Over the past few years, computer scientists have been applying verification techniques to the correctness of digital systems. This was mainly brought on by the increasing inadequacy of conventional approaches (i.e. simulation, prototyping, etc.) which could only demonstrate the presence of a bug, never its absence.

Several mechanical theorem provers have been designed and used for the verification of hardware [27,24,31]. Various techniques and approaches have been adopted but the goal has always been the same: to verify that a given system actually behaves in the desired way.

The idea behind hardware verification is to use a mathematical and logical notation to represent the desired behaviour of a digital circuit (specification) and to prove it equivalent to the representation of a contemplated implementation (verification). The verification process is complicated and expensive. Even proofs of simple circuits can involve thousands of logical steps. Often it can be extremely difficult to find correct device specifications and it is therefore desirable that one sets off to prove a correct specification from the start, rather than repeatedly backtrack from the verification process to modify the original definitions after discovering they were incorrect or inaccurate.

The idea discussed in this thesis is that the techniques of simulation and verification should be amalgamated, rather than have the latter replace the former. The result is that behavioural definitions can be simulated until it is reasonably sure that the specification is correct. Furthermore, proving the correctness with respect to these simulated specifications avoids the inadequacies of simulation, where it may not be computationally feasible to demonstrate correctness by exhaustive testing. In other words, simulation here has a different purpose: to

discover obvious design bugs and to get specifications correct as early as possible in the verification process. Its purpose is not to demonstrate the correctness of the implementation—this is done in the verification stage.

The process of designing and manufacturing a digital circuit can be summarised as shown in Figure 1.1. Ideally, the number of times one backtracks along arc 2 is kept as small as possible by the opportunity of backtracking along arc 1.

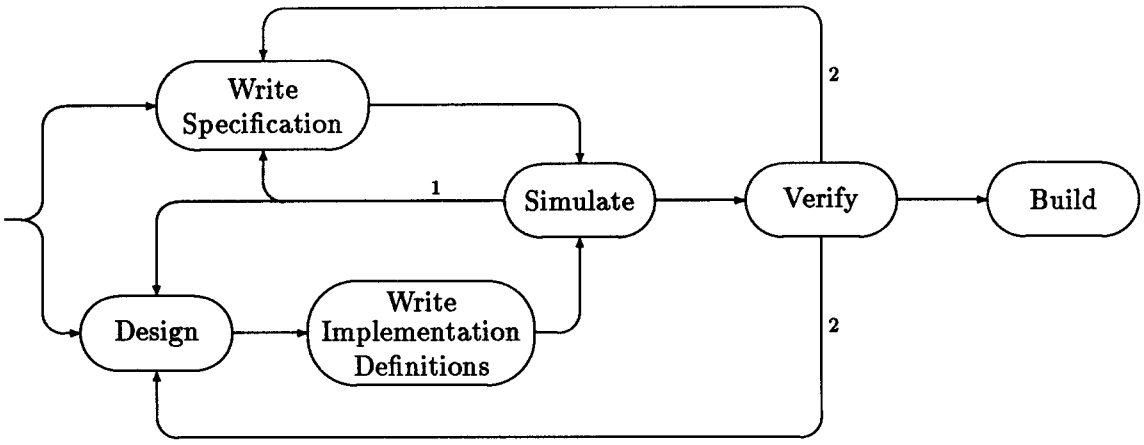


Figure 1.1: The Ideal Verification Process

The process starts at the specification and the design stage where specification definitions are written, a design to implement the specification is contemplated, and implementation definitions are written to represent the design. These definitions are simulated until they are fully understood before proceeding to verify them. Inaccurate models and specifications can be corrected by backtracking from the simulation stage to the design and specification stage until one is confident that the models reflect the desired behaviour. Without simulation, however, it would only be possible to backtrack from the verification stage, which usually involves a waste of time and effort. Some backtracking from the verification stage can still be required since design errors not trapped by simulation should be discovered during verification. Once the circuit is verified it can be fabricated. Ideally, there should be no backtracking from the fabrication stage for the purpose of correcting inaccurate designs since verification should yield design implementations which are 100% correct.

In this thesis we describe how the facility to simulate and verify digital systems can be supported in a theorem proving system called HOL, developed by Mike Gordon at the University of Cambridge [22] and based on the LCF system

developed by Robin Milner at the University of Edinburgh [16]. The system uses a version of higher order logic as a formalism for specifying and verifying hardware.

In the following chapters, we show how hardware specifications written in the HOL logic (see Chapter 2) can be cleanly transformed into executable programs which are notationally almost identical to the original non-executable specifications. This enables one to perform simulation in a formal verification environment, and avoids many dangers and inconsistencies introduced if different notations and systems were used for conducting simulation and verification separately.

The ideas presented revolve around the aim of performing simulation and formal verification using the same specifications. The ultimate goal would be to develop an infrastructure which supported several techniques used for designing correct hardware (e.g. silicon compilation, synthesis, timing analysis, simulation, verification), all of which used the same notation. In the rest of this thesis, however, we only discuss techniques that can be used to execute specifications within a theorem proving framework, and hence enable simulation. Before moving on to present the ideas behind the chosen approach, some related works in the areas of simulation and verification are discussed.

1.2 Background and Related Work

Some mechanical theorem provers combine the two notions of simulation and verification. The trend, however, has been either to develop a hardware simulator with no infrastructure for conducting formal proof, or to develop a theorem prover (or proof checker) which carries out formal proof by manipulating specifications but which does not do simulation.

Some mechanical systems which have been used for simulation, verification, or both, are described below. It is impossible to cover all the existing simulators and theorem provers; there are too many of them. In general, only those which are directly related to the modelling of digital circuits at the levels of description presented in this thesis are discussed. Furthermore, the discussions of these systems only provide a brief outline of their approach to demonstrating hardware correctness; further details can be found in the bibliography.

1.2.1 Simulators and Hardware Description Languages

Simulation has been the traditional approach for demonstrating the correctness of hardware devices. It is the process in which hardware descriptions are subjected

to various input stimuli to generate output values which can be examined and checked for errors.

Over the years, many hardware simulators have been developed to model circuits at various levels of description. For example, SPICE [51] is a simulator aimed at the detailed circuit level in which capacitors, resistors, transistors, etc., are represented in terms of their physical magnitudes such as voltage and current, and MOSSIM [6] is a switch level simulator in which transistors are modelled as bidirectional, voltage controlled switches.

In this thesis we will be mainly concerned with writing specifications at the register-transfer level. A simulator which has become widely used at this level of description is ELLA [48], developed by the Royal Signals and Radar Establishment and marketed by Praxis Systems plc.

ELLA

ELLA supports the simulation of both behavioural and structural specifications of digital systems [55]. In ELLA, circuits are described as networks of nodes connected by wires, where nodes are described by functions and have at least one input and one output. Nodes are allowed to operate in parallel to give a realistic model of hardware, and they can be decomposed into networks of subnodes to allow a hierarchical description of circuits. The notion of nodes, in fact, is similar to that of devices used in the logic specifications presented in this thesis. A case study comparing hardware descriptions in ELLA with those in the general purpose functional programming language ML is given in Chapter 4.

Like other special purpose simulators, ELLA does not adopt a formal approach to demonstrating correctness. Because of the absence of verification facilities in the ELLA system, attempts to introduce formal proof in design processes involving ELLA simulations have required that the ELLA definitions be translated into formal specifications used by mechanical theorem provers (see Chapter 6 and [13]). Such use of different notations is dangerous because inconsistencies can be introduced in the specifications in the process of translation, and also because the semantic differences between two languages can impose completely different representations.

VHDL

Hardware description languages (HDL) are often used as input languages for computer aided design (CAD) tools. They provide a textual description of structure and/or behaviour which can then be input to simulators, silicon compilers, etc.

Once again, many HDLs have been designed for describing hardware at different levels of abstraction. One example of a HDL which enables the description of both hardware behaviour and structure at a register transfer level is VHDL, a language originated by the United States Department of Defence [57]. The main theme behind VHDL is to support the design, documentation, and simulation of circuits irrespective of different technologies, and to offer the flexibility to cope with new or upgraded CAD technologies.

Circuit descriptions in VHDL make use of ADA constructs to define functions that map from inputs to outputs. The descriptions are divided into two parts: a definition of the interface between the design and the outside world (i.e. the ports), and a functional definition of the circuit.

As is typical of other HDLs, however, VHDL does not have a formal semantics. For this reason, the approach described in this thesis is to translate formal specifications written in the HOL logic into a language which has a similar semantics, rather than to interface HOL with a HDL where the lack of formal semantics can give rise to ambiguous descriptions.

DAISY

One approach to hardware specification which enables simulation and has a formal semantics basis is DAISY, a system designed by Steven Johnson at Indiana University for conducting synthesis of hardware designs [32]. Specifications in DAISY are typed recursion equations based on the notation and semantics of the Scott-Strachey calculus [59].

It is not possible, at present, to conduct verification using DAISY, although it would seem that a theorem prover could be added to the system. This would address the concept of combining verification and simulation using the same specifications as presented in this thesis, but using the opposite approach (i.e. adding proof infrastructure to an existing system that supports executable specifications, rather than adding tools to an existing theorem prover that enable the specifications to be executed). The DAISY system, however, was mainly intended for synthesis, a technique for deriving correct implementation designs from specifications by construction. Hence, an alternative approach to verification is adopted for demonstrating correctness of designs.

Recent work by John O'Donnell at Indiana University and the University of Glasgow, has concentrated on automatically constructing the geometrical layout of hardware designs [52], and is highly influenced by the methodology of DAISY.

1.2.2 Verification

Higher Order Logic

At the other end of the scale, VERITAS, developed by Keith Hanna at the University of Kent at Canterbury, is a mechanical theorem prover which has been applied to hardware verification [24]. The VERITAS logic is a species of higher order, polymorphically-typed logic [25]; its type structure is based on Martin-Löf's Intuitionistic Type Theory [37].

The use of higher order logic for hardware verification was first advocated by Keith Hanna, and has proved to be a promising formalism for specifying all aspects of hardware. Other theorem provers based on higher order logic include HOL (described in this thesis), and a system developed by Mike Fourman at Brunel University [15].

The higher order logic used in VERITAS is different from that presented in this thesis. Currently, specifications in VERITAS cannot be executed so no simulation is possible using it. It is only possible to conduct formal verification.

Hans Eveking also uses a version of higher order predicate logic for hardware verification [14]. Both the inadequecies of simulation, and the inability of HDLs to allow formal reasoning about specifications are identified in [14], and hence, Eveking's methods combine the use of HDLs with predicate calculus, replacing the HDL specifications by logical formulae in order to enable verification. Our approach is to avoid translating to and from HDLs, and to show that execution of a subset of the logic provides good enough simulation facilities, making the use of HDLs and special purpose simulators unnecessary to aid the verification process.

1.2.3 Verification and Simulation

In this subsection, we present descriptions of various systems in which it is possible to conduct both simulation and verification. Each system has been chosen to typify a different approach. Some of these systems can seem more natural in supporting simulation than the approach described for HOL in this thesis. For example, in BOYER-MOORE and CIRCAL (see below) specifications can be simulated without requiring any translation into executable languages.

The research explained in this thesis, however, is also intended to show that although higher order logic sentences do not in general have an executable interpretation, the subset used for modelling hardware can be executed. Predicate logic specifications are abstract definitions (unlike in the BOYER-MOORE system

where specifications are programs) and so they have to be transformed into an executable notation if they are to be executed. From the experiences gained with HOL, the addition of simulation facilities to the system greatly helps the process of understanding specifications and conducting verification. The transformation process from abstract specifications to programs is not overly tedious since the translation process has been automated. It is recognised that it is insufficient to claim that specifications can be executed; they must be executed with an acceptable efficiency. It is later shown, therefore, that the execution strategies adopted enable efficient simulation of large, and real circuit designs.

In the light of the above, the provision of simulation facilities in HOL as described hereafter makes a useful contribution to the application of logical inference theorem provers to formal hardware verification by enabling efficient execution of specifications to support verification.

BOYER-MOORE

The BOYER-MOORE theorem prover, developed by Robert Boyer and J Strother Moore, is one of the leading automatic theorem provers [5]. This theorem prover is a mechanisation of a quantifier-free first order logic. Automatic proof is conducted by applying a set of heuristics in turn to a goal; if the goal is found to be true it is returned as a theorem and saved in a data base where it can be used to prove other theorems. Often, theorems cannot be proven straight away and require simpler and more general theorems to be proved and inserted in the database before the proof of the intended theorems can be conducted successfully. Primarily, the BOYER-MOORE logic is not typed although a form of type restriction can be employed explicitly by the use of functions which determine the nature of data types.

The BOYER-MOORE prover has been recently used to formally specify and verify a microprocessor called FM8501 by Warren Hunt Jr. [31]. Since the specifications are actually LISP programs, the LISP evaluation function *eval* [65] can be used to execute the specifications. Thus, hardware specifications written in the BOYER-MOORE logic can be directly used for both simulation and verification.

CIRCAL

Another formalism which has been applied to both simulation and verification of hardware circuits is CIRCAL [43], an algebraic calculus developed by George Milne. CIRCAL (CIRcuit CALculus) is based on the Dot Calculus, also invented by George

Milne [42], and is related to the CCS [45] and SCCS [46] calculi developed by Robin Milner.

In CIRCAL one can model concurrency (simultaneous occurrence of events), synchrony (occurrence of events at clocked intervals), and asynchrony (occurrence of events not controlled by clocks) [44]. These three features are among the strong points of CIRCAL which make it attractive both as a hardware description language and as an analytical framework within which to describe, specify and analyse the behaviour of communicating computing agents.

The general approach of CIRCAL to the description of hardware circuits is to formulate the behaviour as communications between agents or processes. Behaviour is specified by a CIRCAL *expression* which has an associated set of labels (called a *sort*) to indicate where devices may interact. A set of CIRCAL laws is used to manipulate expressions to conduct verification, whereas simulation is performed by composing expressions describing the behaviour of a device and expressions describing a single pattern of event stimuli [43]. A detailed account of CIRCAL as a medium for conducting simulation and formal verification can be found in [61].

BSPL

The behavioural specification language BSPL, developed by Martin Richards, is the input to an automatic tool designed primarily for the verification of microcode programs, but which has also been used to specify synchronous circuits [56]. The only data types available in BSPL are words of a specified number of signals, where each signal can take one of four values: 0 (low), 1 (high), X (undefined) and Z (floating). Specifications in BSPL are formal representations of clocked, finite-state machines with inputs, outputs and internal states. One of the hardware examples that has been specified in BSPL is the microcomputer presented in Chapter 8 of this thesis.

ITL

Another approach to reasoning about concurrency and hardware is Temporal Logic [23,47]. The imperative programming language Tempura, developed by Ben Moszkowski [50], is based on Moszkowski's earlier work on Interval Temporal Logic (ITL) [49]. In Tempura, the programs are themselves temporal logic formulae, so statements in Tempura can be used to specify, simulate and verify hardware circuits. In temporal logic, one makes use of special built-in operators (such as *always*

and *sometimes*) which have an implicit notion of time to represent time-dependent concepts when modelling hardware.

μFP

The integrated circuit design language μFP, developed by Mary Sheeran, can describe both the behaviour and the layout of a circuit [58]. It is a formal design language with a semantics based on FP [3] but especially geared towards hardware design. The main extension over Backus's FP lies in the introduction of the combining form μ which is used to represent memory in sequential circuits.

Circuit descriptions in μFP are functions which take sequences of inputs and return sequences of outputs. These specifications can be manipulated by using algebraic laws to demonstrate correctness by construction. Interpreters for executing μFP specifications have been written in both functional and imperative languages, providing the facility of performing logic level simulation. Design tools also exist for interpreting the specifications to produce a floor-plan of the described circuit.

1.2.4 Programming Languages and Simulation

Over the years, several attempts have also been made to apply general purpose programming languages to hardware simulation. Essentially this approach is a methodology with which features characteristic of programming languages can be exploited to model and simulate hardware circuits. The use of programming languages, however, only enables simulation; there is no infrastructure to support verification. Our approach, therefore, has been to translate formal specifications used by a theorem prover to programs in a general purpose programming language, where the syntax and semantics of the subsets of the two notations required to model hardware at the register transfer level are almost identical.

The fact that general purpose programming languages are expressive enough to model many aspects of hardware designs has been widely demonstrated. Below are a few examples of such programming languages that have been used for hardware simulation.

MIRANDA

In [29] for example, Stephen Hill shows how the lazy functional programming language MIRANDA [62] can be used to simulate hardware. His techniques rely

heavily on the lazy nature of the language which enables the manipulation of infinite data structures. The ability to evaluate finite portions of infinite data structures allows signals to be represented as infinite lists of logic values. The use of lazy evaluation for hardware simulation is discussed further in Chapter 5.

PROLOG

William Clocksin has used the logic programming language PROLOG [10] in the role of simulation [9]. Among other features, the relational nature of logic programs is exploited to enable the modelling of bi-directional flow in low-level representations of circuits. This is one advantage over the functional approach since with relations there is no distinction between inputs and outputs. Comparisons between relations and functions form a major topic in this thesis.

OCCAM

Finally, the features for expressing parallelism in OCCAM [33] were used extensively by David May and David Shepherd to specify, simulate and validate a microprocessor called the IMS T800 [38]. Hardware circuits are specified in OCCAM as a collection of concurrent processes communicating via channels. Parallel computation in OCCAM is very efficient and makes simulation fast. The OCCAM language has a set of laws which can be used to demonstrate correctness by transformation.

1.3 Overview of Thesis

In this thesis we show how a subset of the higher order logic embedded within the HOL theorem proving system can be executed to enable simulation of hardware specifications.

The use of higher order logic as a formalism for specifying and verifying hardware circuits is discussed in [21,7], and some substantial and real circuits which have been verified using HOL are presented in [11,28,34]. The suitability of higher order logic as a medium for hardware verification, therefore, has been well established. In this thesis we give evidence that the HOL system can also be used for hardware simulation as a means for supporting verification.

Although logic specifications are in general not executable, we show how the subset of specifications used for hardware verification can be automatically transformed into executable programs. These programs are also automatically optimised to perform at an acceptable speed. In support of the practicality and

versatility of such automatic tools, the specifications of some major examples previously verified in HOL are automatically translated into executable programs and simulated for test data.

Below is a brief description of the organisation of the material presented in this thesis.

- Chapter 2 provides a brief description of the HOL system. We describe the species of higher order logic used, the meta-language ML in which the logic is formulated, and the theorem proving strategies used to conduct proofs in HOL. The main emphasis in this chapter is on those features of the HOL system which are used in later chapters of this thesis.

- Chapter 3 shows how hardware can be specified in HOL. The differences between relations and functions are explained and their respective advantages for modelling hardware are discussed. As an example, an n-bit adder is specified and verified using both relational and functional definitions. It is shown that the two different styles are not equivalent and a relationship between them is described.

- In Chapter 4 the HOL meta-language ML is shown to be a good hardware simulator at the register-transfer level. Furthermore, the style of writing programs in ML that model hardware is shown to be extremely similar to that used in writing HOL specifications in Chapter 3.

- The ML programs, however, are extremely inefficient to execute. In Chapter 5, the nature of the inefficiency is discussed and two optimisation strategies (memoisation and lazy evaluation) are proposed. Both are shown to provide good solutions, but only one technique (memoisation) is adopted and discussed further in the rest of the thesis.

- Chapter 6 covers the algorithms for automatically translating HOL relations to ML functions (or programs). The algorithms are shown to cater for most of the commonly used techniques in register-transfer level representations of hardware. An account is also given of which kind of relations can be automatically translated into executable functions and which do not have functional interpretations.

- Chapters 7–9 illustrate the techniques presented in Chapter 6 by describing the translations of the specifications of three hardware examples which were

previously verified in HOL. The examples are: a factorial machine, a simple microcomputer, and a communications chip. The problems and success encountered with each example are discussed, giving an idea of the practicality and versatility of the automation. The derived programs in each example are simulated over test data to show how this tool can be useful as an aid to verification.

- Finally, Chapter 10 evaluates the research described in this thesis. Some ideas for further research, possible solutions to problems encountered, and improvements to current strategies are also proposed.

Chapter 2

The HOL Theorem Prover

The HOL system, developed by Mike Gordon at the University of Cambridge, is a tool intended primarily for hardware specification and verification using higher order logic. It is implemented on top of Cambridge LCF [53] and supersedes the earlier system LCF-LSM [18].

HOL is the name given to the entire theorem proving system which supports higher order logic as a formalism for writing specifications and conducting proofs. In cases where it is necessary to distinguish between the computer system and the species of higher order logic embedded within it, the terminology HOL system and HOL logic is used respectively.

A detailed account of both the HOL system and the HOL logic can be found in [22]. In order to make this thesis self-contained, however, a brief introduction to HOL is given in the following sections. This should enable the reader with little or no experience with HOL to follow the rest of this thesis. Some familiarity with predicate logic is assumed. Readers familiar with HOL may wish to skip to Chapter 3.

2.1 The HOL Logic

The species of higher order logic used within the HOL system is a version of Church's Simple Type Theory [8]. The HOL logic uses standard predicate logic notation in which one makes use of the propositional logic connectives denoting negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), implication ($\supset$) and equivalence ($\equiv$) to connect propositions (such as properties and relations) to form more complicated sentences. Variables in such sentences are bound using universal ($\forall$) and existential ($\exists$) quantification.

Table 2.1 outlines the syntax and informal semantics of predicate logic. In the table, t, t_1 and t_2 stand for arbitrary *terms* while $t[x]$ stands for some term containing free occurrences of the variable x.

Notation	Meaning
$P(x)$	x has property P
$R(x,y)$	relation R holds between x and y
$\neg t_1$	not t_1
$t_1 \vee t_2$	t_1 or t_2
$t_1 \wedge t_2$	t_1 and t_2
$t_1 \supset t_2$	t_1 implies t_2
$t_1 \equiv t_2$	t_1 if and only if t_2
$\forall x.\, t[x]$	$t[x]$ is true for all x
$\exists x.\, t[x]$	$t[x]$ is true for some x
$(t \Rightarrow t_1 \mid t_2)$	if t is true then t_1 else t_2

Table 2.1: Predicate Logic Notation

Higher order logic generalises first order predicate calculus by allowing higher order variables—i.e. variables ranging over functions and predicates. For example, the induction axiom for natural numbers can be written as:

$$\forall P.\, [P(0) \wedge (\forall n.\, P(n) \supset P(n{+}1)) \supset \forall n.\, P(n)]$$

Here, the variable P is quantified and ranges over predicates; such variables are said to be higher order.

2.1.1 Terms

The HOL logic uses four kinds of terms: *variables*, *constants*, *function applications* and *lambda expressions*.

Variables and constants are denoted by sequences of letters or digits starting with a letter. A few other symbols are also allowed in variable and constant names but will not be mentioned here. For example, x, $y1$ and gnd can be names of variables or constants. The difference between variables and constants is not apparent at this stage but will be dealt with in a later section on the HOL system when the notion of a *theory* is introduced.

Function applications have the form $t_1(t_2)$, where the subterm t_1 is called the *operator* and t_2 is called the *operand* (or *argument*). Due to the higher order

nature of the logic, the results of function applications can themselves be functions, i.e. functions can take functions as arguments or return functions as results.

To minimise bracketing, function applications can be written as $f\ x$ instead of $f(x)$. Furthermore, application associates to the left and so, $t_1\, t_2 \ldots t_n$ abbreviates $((t_1\ t_2) \ldots t_n)$.

Lambda-expressions are the means for denoting functions within higher order logic. The term $\lambda x.\ t$ (where t is any expression) denotes the function f, say, defined by

$$f(x) = t$$

If we take t in the above lambda-expression to be the expression $x+y$ such that we have the term $\lambda x.\ x+y$ then x is said to be a *bound variable*, y is a *free* variable and $x+y$ is called the *body* of the λ-expression.

2.1.2 Types

The HOL logic is a strongly typed logic, i.e. all terms expressed in this version of higher order logic must have a *type*. Without types, the HOL logic would be unsound as the availability of higher order variables can give rise to a version of Russell's paradox. This can be shown using the following definition of a predicate P:

$$P\ x = \neg(x\ x)$$

from which one can derive the paradox:

$$P\ P = \neg(P\ P)$$

The above paradox is prevented by the use of types and the reasons for this are presented in [20].

The type system used in the HOL logic is derived from that of PPLAMBDA [16] which, in turn, descends from the type system formulated by Alonzo Church [8]. It allows types to be either:

- *atomic* (e.g. *bool* to denote the sets of booleans or *num* to denote natural numbers), or
- *compound* (i.e. those types built from atomic (or other compound) types by using *type operators*).

Examples of type operators in the HOL logic are *list*, $\rightarrow$ and #, where *list* is a unary type operator used to denote a list of values (e.g. *num list* denotes a list of natural numbers) while $\rightarrow$ and # are infixed binary type operators used to denote sets of functions and pairs respectively. For example, $(num\#num)\rightarrow bool$ denotes the type of a function with a domain of pairs of natural numbers and a range of boolean truth values.

Types in the HOL logic can contain variables. In order to demonstrate this, let us consider the function *compose* defined below.

$$compose = \lambda f.\ \lambda g.\ \lambda x.\ f(g\ x)$$

If *compose* is applied to two functions, f and g say, then the result would be a function which would apply g to its argument and f to that result.

For example, if *not* is the boolean negation function of type $bool\rightarrow bool$ and *even* is a function of type $num\rightarrow bool$ which returns 'true' if its arguments are even natural numbers and 'false' otherwise, then the result of applying *compose* to *not* and *even* in that order would be $\lambda x.\ not\,(even\ x)$ which is a function of type $num\rightarrow bool$.

On the other hand, if *rnd* is a function of type $real\rightarrow num$ which rounds off a positive real number to the nearest natural number, and *log* is the arithmetic logarithmic function of type $real\rightarrow real$, the result of applying *compose* to *rnd* and then *log* is the function $\lambda x.\ rnd\,(log\ x)$ of type $real\rightarrow num$.

The function *compose*, therefore, appears to have two different types:

$$(bool\rightarrow bool)\rightarrow(num\rightarrow bool)\rightarrow(num\rightarrow bool)$$

and

$$(real\rightarrow num)\rightarrow(real\rightarrow real)\rightarrow(real\rightarrow num)$$

Indeed it appears that it can have many different types, depending on the types of the functions f and g it is applied to. In the HOL logic, *type variables* are used to allow functions with more than one possible type to be expressed within the logic. Without type variables, a different function would have to be defined for every type because a single function is not allowed to denote several types.

In HOL, however, it is only necessary to define a single function *compose*. If α, β and γ are type variables then *compose* is given the type:

$$(\beta\rightarrow\gamma)\rightarrow(\alpha\rightarrow\beta)\rightarrow(\alpha\rightarrow\gamma)$$

These type variables can be instantiated to different types according to the particular use of the function *compose*. Types containing type variables are called *polymorphic*.

2.1.3 Hilbert's ε-operator

Hilbert's choice operator, ε, plays a very important part in the HOL logic. It is most commonly used to denote values one knows to exist but have no name.

More precisely, if $t[x]$ is a boolean term containing a free variable x of type α, then the term $\varepsilon\, x.\, t[x]$ denotes some value of type α, a say, such that $t[a]$ is true. For example, $\varepsilon\, x.\, (7 < x) \wedge (x < 9)$ denotes 8 while the term $\varepsilon\, x.\, x \geq 0$ denotes some unspecified positive number.

In the case that there is no value a such that $t[a]$ is true, then $\varepsilon\, x.\, t[x]$ denotes a fixed but unspecified value of type α. For example, $\varepsilon\, n{:}num.\, \neg(n = n)$ denotes an unspecified number. The notation *term:type* is used within the HOL logic to explicitly specify the type of a term.

No further detail regarding ε is given here. For a thorough discussion of Hilbert's ε-operator see [36]. A detailed description of how Hilbert terms are included in HOL to build in the Axiom of Choice [26] is given in [20].

The features of the HOL logic necessary to enable an understanding of the rest of this thesis have now been covered and we can go on to show (very briefly) how the logic is implemented in the HOL system. First, however, a brief introduction to ML, the meta-language embedded in the HOL system and with which most of the system is coded, is given.

2.2 The HOL Meta-language

The aim of this section is to give an introduction to the ML language; mainly covering those features which are discussed later on in the thesis. The version of ML described here as part of the HOL system is not Standard ML [64] but the LCF meta-language version described in the ML Handbook [12], where a complete description of the ML syntax and semantics is presented.

ML is a typed interactive functional programming language. At the 'top-level' one can:

- evaluate expressions
- perform declarations

The #-symbol is the prompt issued by ML to indicate it is ready for input. An input sequence is terminated by two consecutive semi-colons, ';;', and ML does not respond until these terminating characters are input. To avoid cluttering the text in the following examples, the prompt and terminating characters are only used

when it is necessary to distinguish between the system's response and the user's input.

Comments in ML are enclosed within two %-characters and are completely ignored.

```
# %This is a comment%
```

Anything else, however, is evaluated. If one types in 5;; to the system, ML returns the value 5, specifying its type, *int*. If one tries 9+7;;, the two numbers are added and the value 16 is returned. The special identifier 'it' in ML, recalls the last value evaluated.

```
# 5;;
5 : int

# 9 + 7;;
16 : int

# it;;
16 : int
```

Several expressions can be evaluated in sequence by separating the expressions with a single semi-colon. Thus, when an expression $e_1; \ldots; e_n$ is evaluated, each sub-expression e_i is evaluated in turn and the value of the entire expression is that of e_n. For example,

```
# 5+3;
  true;;
true : bool
```

Declarations are performed using let statements. These are of the form let$x = e$, where the identifier x is bound to the value of expression e. Several declarations can be done in parallel by using 'and' between individual declarations. Local declarations make use of the phrase 'in' before the body of the expression within which the declaration applies.

```
# let x = 5;;
x = 5 : int

# let y = 2+3 and z = 0;;
y = 5 : int
z = 0 : int
```

```
# let x = 10 in x×x;;
100 : int

# x;;
5 : int
```

Destructive assignment is possible in ML provided the variables to be assigned values are first declared for this purpose. This is done by using the keyword letref which declares the identifiers and sets them to an initial value. The infix assignment operator ':=' can then be used to assign values to the declared variables. For example:

```
# letref y = 0;;
y = 0 : int

# y := 5+y;;
y = 5 : int
```

Another use of the let declarations is to define functions in ML. For example, the statement:

```
let suc n = n+1
```

defines the successor function, *suc*, with parameter n and body $n+1$. To apply *suc* to a particular parameter one merely evaluates the application. For example:

```
# suc 5;;
6 : int
```

Once a function or other value is defined in ML and bound to an identifier, that identifier denotes the same function throughout an ML session unless redefined.

In the case of functions with several parameters, the parameters can be either *curried* or *tupled*. For example, the function *add* can be defined with curried parameters:

```
let add x y = x+y
```

or with a single parameter of the cartesian product type (*int#int*):

```
let add(x,y) = x+y
```

The advantage of curried functions is that they can be partially applied.

As well as taking tuples for parameters, functions can also return tupled results. For example, one may wish to define a function *order* to take a pair of integers x and y, and return a pair with the two integers in ascending order. One possible

definition of *order* is given below which makes use of the ML conditional statement.

$$\textsf{let } order(x, y) = \textsf{if } x{>}y \textsf{ then } (y, x) \textsf{ else } (x, y)$$

The conditional statement: **if** a **then** b **else** c is often abbreviated to the equivalent notation: $a \Rightarrow b \mid c$.

Functions with two parameters can be, and often are, declared as infix functions or operators. In our examples so far, we have already made use of several pre-declared infix operators such as +, =, >, etc.

Recursive functions can be defined in ML in almost the same way as ordinary functions. The only difference is that **letrec** is used instead of **let**. For example, the factorial function, *fact*, can be defined recursively as:

$$\textsf{letrec } fact\ n = \textsf{if } n{=}0 \textsf{ then } 1 \textsf{ else } n{\times}(fact(n{+}1))$$

Of course, *fact* can be defined iteratively but since we will not be using iterative loops anywhere in this thesis, the reader is referred to [12] for further details.

One can also represent functions in ML as lambda-expressions. The following two definitions of a function f are equivalent.

$$(\textsf{let } f\ x = e) \equiv (\textsf{let } f = \lambda x.\ e)$$

Lists in ML are represented by a sequence of objects separated by semi-colons and enclosed within square brackets. All objects within a list must be of the same type. The expressions [ ], [1; 2; 3] and [*true*; *false*] are all examples of lists. A list such as [1; *true*] will not type check as the objects in the list, 1 and *true*, are of different types.

The standard functions on lists are:

- *hd*—returns the head of a list (e.g. $hd\,[1;2] = 1$),
- *tl*—returns the tail of a list (e.g. $tl\,[1;2] = [2]$),
- *null*—boolean function which checks if a list is empty,
- .—infix *cons* operator (e.g. $1.[2] = [1;2]$),
- @—infix *append* operator (e.g. $[1]@[2] = [1;2]$).

Another data type represented in ML is *string*. Strings are any sequence of characters enclosed within single quotes (e.g. 'This is a string'). There are also standard functions on strings such as *concat*, *explode* and *implode*.

Variables and functions in ML can be polymorphic. The notion of polymorphism has already been explained in Section 2.1.2 and so it will suffice here to give an example of how a polymorphic function is defined in ML. Consider the function *map* defined as follows:

$$\textsf{letrec}\ map\ f\ l = (null\ l) \Rightarrow [\,] \mid f(hd\ l).(map\ f\ (tl\ l))$$

Since the types of the two arguments f and l are not specified in the definition of *map*, ML assumes the types of f and l to be polymorphic and defines *map* to be a polymorphic function of type:

$$(*\rightarrow **) \rightarrow *\ list \rightarrow **\ list$$

Sequences of asterisks are used to denote type variables in ML. The difference between ML and logic types are briefly explained in the next section.

There are three ways in which one can define new types in ML. The first is by using the command **typeabbrev** to define new names to abbreviate previously defined types. This does not really define a new type but simply binds a name to a previously defined type. It is useful for shortening long type names or for renaming types more appropriately. For example:

$$\textsf{typeabbrev}\ intpair = int\#int$$

defines a type *intpair* to denote pairs of integers.

The two other ways of defining types are used to define altogether new types rather than just abbreviations. New types can be defined to be *concrete* or *abstract.*

Concrete types are used when the objects in the type can be enumerated into subgroups. For example, the declaration:

$$\textsf{type}\ signal = HI \mid LO \mid FLOAT$$

defines a new type *signal* which has three possible values: HI, LO or $FLOAT$.

The other way of defining types is by using abstraction. The ML command **abstype** allows one to make an abstract type declaration. While defining a type ty, say, there are two primitive functions, abs_ty and rep_ty, which are usable within the context of the type definition. The function abs_ty maps the representation of ty to ty while the function rep_ty maps ty to its representation. One can also define, within the type declaration itself, further primitive functions to manipulate the new type.

Consider, for example, the type declaration below which introduces a new type *trigger*, represented by the type *bool*.

```
abstype  trigger   = bool
with     ON        = abs_trigger true
and      OFF       = abs_trigger false
and      bool_of   = rep_trigger
and      inv c     = abs_trigger (not (rep_trigger c))
and      clock n   = λt. (n = 0) ⇒ (abs_triggerfalse) |
                         ((t÷n)×n = t) ⇒ (abs_trigger true) |
                         (abs_trigger false)
```

The type declaration makes use of the two locally available functions *abs_trigger* and *rep_trigger* to define the following primitive functions:

- *ON* and *OFF*—two constants of type *trigger* represented by *true* and *false* respectively.
- *bool_of*—a function of type *trigger→bool* which maps a value of type *trigger* to its representative of type *bool*. For example, *bool_of ON = true*.
- *inv*—a function of type *trigger→trigger* which inverts values of type *trigger*, i.e. (*inv ON = OFF*) and (*inv OFF = ON*).
- *clock*—a function of type *int→int→trigger*. The application (*clock n*) returns a function which models a clock with a pulse interval of n. Thus when n is 0, *clock*(0) returns a function which models a clock which is always *OFF* (pulse interval is 0, i.e. no pulse). The application *clock*(1), on the other hand, returns a clock which is always *ON* (pulse interval is 1, i.e. constant pulse). The application *clock*(2) models a clock which toggles *ON* and *OFF* alternately, and so on. Hence, in the application (*clock n t*), n determines the frequency of the clock and t represents the time at which the clock value may be evaluated.

Finally, it is worth mentioning the *fail* mechanism built into ML. This can be triggered by the *fail* command and is usually followed by an error message explaining the reason for failure. The *fail* command is especially useful as an escape from a program.

The error messages that follow a *fail* command usually consist of the name of the function in which the failure occurred, thus indicating the code that caused the failure. Messages following a failure, however, can be set to anything desired

simply by using the command *failwith s* (where s is a value of type *string*) instead of *fail*, thus generating failure with an error message s.

```
# hd [];;
evaluation failed    hd

# failwith 'main program';
evaluation failed    main program
```

Failure can also be trapped. The value of the expression $e_1?e_2$ is that of e_1 if e_1 does not fail; otherwise it is the value of e_2. This feature is useful as a 'switch' where the sub-program e_1 is executed by default but, in the case that e_1 fails, e_2 will be evaluated rather than failing the entire program.

```
# (1÷0)?100;;
100 : int
```

2.3 The HOL System

Terms of the HOL logic are represented in ML by enclosing them in double quotes. The syntax of HOL terms has been described in section 2.1.1 although in practice, various combinations of ascii characters are used to represent those logic symbols not supported by the ordinary ascii terminal.

For example,

$$\forall a\ b.\ a \supset b \equiv \neg a \vee b$$

is represented by

```
!a b. a ==> b  ==  ~a \/ b
```

In the rest of this thesis we shall use the notation of Section 2.1.1 (i.e. we will be using $\forall$ instead of ! and $\supset$ instead of ==>). For a detailed account of the representation of the logic in the HOL system see [22].

When a HOL term is entered in ML, it is type-checked (according to the type rules of the logic—not ML) and, if successful, it is given the ML type *term*. Care should be taken here not to confuse the terminology HOL terms and ML expressions and, moreover, HOL types and ML types. The rule is as follows:

- A HOL term is a special kind of ML expression and is distinguished by a pair of double quotes enclosing the logical term. HOL terms have an ML type called *term*.

- A HOL type is the type of a HOL term and forms an ML type called *type*. HOL types are expressions of the form " : ...".

For example:

- $(1, 2)$ is an ML expression with type $(int\#int)$.
- "$(1, 2)$" is an ML expression with type *term* (since anything enclosed within double quotes represents a HOL term). The HOL type of this term is $(num\#num)$.
- Likewise, ("1", "2") is an ML expression with type $(term\#term)$ where each term has HOL type *num*.
- " : *num*" is an ML expression with ML type *type*. It represents the HOL type *num*.

2.3.1 Theories: Definitions, Axioms and Theorems

In [20], a *theorem* is defined as a *sequent* that is either an *axiom* or follows from other theorems by *rules of inference*, where

- a *sequent* is a pair (Γ, t) consisting of a finite set of boolean terms Γ (called *assumptions*) and a boolean term t (called a *conclusion*),
- an *axiom* is a sequent postulated to be a theorem, and
- *rules of inference* are procedures for deducing new theorems from existing ones (see Section 2.3.2).

When a sequent (Γ, t) is a theorem it is written as $\Gamma \vdash t$ or, if Γ is empty, as $\vdash t$.

Certain types of axioms are classed as *definitions*. Definitions are those axioms of the form $\vdash c = t$ where c is a constant not previously defined and t is a term containing no free variables. Of course, this kind of axiom is always safe as it merely defines an abbreviation. Ideally, all axioms should be of a definitional form since the freedom to postulate arbitrary axioms allows the introduction of inconsistencies.

To make a definition, prove a theorem, or declare a new HOL type, one must first enter a *theory*. A theory is a collection of types, type operators, constants, definitions, axioms and theorems.

New constants and types can be declared within theories. The distinction between variables and constants is, therefore, that variables are those terms (excluding function applications and λ-expressions) which are not declared as constants within a theory.

Theories can have other theories as *parents*. If one is working within a theory, th say, and an object from theory th' is required in th, then th' must be declared a parent of th. If th' is a parent of th then all the types, constants, definitions, axioms and theorems available in th' are available in th. Thus, th is said to be a *descendant* of th'.

2.3.2 Primitive Inference Rules

Theorems in HOL are represented by values of type *thm* and must be distinguished from values of type $(term\ list)\#term$. For one to obtain a new value of type *thm*, one must apply a sequence of events (constituting a *proof*) to either axioms or previously proved theorems.

Such procedures for deriving new theorems are called *rules of inference*. The following is an example of a rule of inference called *modus ponens*. The example uses standard natural deduction notation where t_1 and t_2 denote arbitrary terms, the theorems above the horizontal line are called the *hypotheses* of the rule and the theorem below the line is called the *consequent*.

$$\frac{\Gamma_1 \vdash t_1 \supset t_2 \qquad \Gamma_2 \vdash t_1}{\Gamma_1 \cup \Gamma_2 \vdash t_2}$$

Hence, if we have a theorem of the form $\Gamma_1 \vdash t_1 \supset t_2$, say $y > 1 \vdash y \geq y \supset y^2 > y$, and we also have the theorem which says that $\vdash y \geq y$ (i.e. the antecedent of the implication in the first theorem is true, $\Gamma_2 \vdash t_1$), then by the rule of modus ponens the theorem $y > 1 \vdash y^2 > y$ is derived. In the example above, the assumption Γ_2 is empty.

Inference rules are represented in the HOL system as functions in ML. The core of the HOL system is made up of a small set of inference rules called *primitive inference rules* and a small number of definitions and axioms from which all the standard rules of logic can be derived. Indeed, one can derive further inference rules (called *derived inference rules*) which can be justified solely on the basis of these primitive inference rules and axioms.

The choice of primitive inference rules and primitive axioms in HOL is, to a certain extent, arbitrary, although it is desirable to keep them as small in number as possible so that the implementation of the logic can be kept simple and clean.

For the purpose of this thesis it is not important to list all the inference rules and axioms. The reader is referred to [20] for further details including a complete list of axioms and inference rules in the current version of HOL.

2.3.3 Tactics and Tacticals

In the previous section we described rules of inference and how they can be used to carry out a proof. One starts with a set of definitions and theorems and manipulates them using the inference rules until the desired theorem to be proved is achieved. In other words, truth is preserved from truth. This form of proof is sometimes called *forward* proof.

The HOL system supports another way of carrying out a proof called *goal directed proof* or *backward* proof. The problem with forward proof is that it can often be difficult to foresee which definitions and theorems are required to prove the end result, especially if the proof is long and complicated. The idea of goal directed proof is to do the proof backwards, i.e. start from the desired result (called the *goal*) and manipulate it until it is reduced to a subgoal which is obviously true.

A *tactic* is an ML function which reduces goals to subgoals. The concept of tactics was invented by Robin Milner [16]. They are used for goal directed proving as described above. Tactics are written in a similar notation to inference rules, but with a double horizontal line. For example, mathematical induction can be coded as a tactic of the form:

$$\frac{\forall n.\ P[n]}{P[0] \quad \forall n.\ P[n] \supset P[n+1]}$$

If the induction tactic is applied to a term of the form $\forall n.\ P[n]$, then the two subgoals $P[0]$ and $\forall n.\ P[n] \supset P[n+1]$ are generated.

A goal consists of a pair of values and has ML type $(term\ list)\#term$. The first element of the pair denotes the assumption list and the second element is the term to be proved. A theorem is proved by applying tactics to every subgoal generated until all subgoals are shown to be true, without the addition of invalid assumptions (see [22]).

Tactics can be combined together by using certain ML functions called *tacticals*. An example of a tactical is THEN where, if T_1 and T_2 are tactics then T_1 THEN T_2 evaluates to a tactic which first applies T_1 to a goal and then applies T_2 to the resulting subgoal or subgoals.

In fact, what really happens when tactics are sequentially applied to goals is that a proof tree is built at the same time as new subgoals are generated. Consider a tactic T and a goal g. If $(T\ g)$ is evaluated, a pair $([g_1;\ldots;g_n],p)$ is obtained, where p is a justification of the reduction of goal g to subgoals $g_1,\ldots,g_n$ (i.e. p can be seen as a forward proof that can 'reverse' the tactic if the subgoal reduction is correct). Hence, if one proceeds to apply more tactics to the subgoals $g_1,\ldots,g_n$ until all further subgoals are reduced to empty lists of subgoals, and if all the steps reducing $[g_1;\ldots;g_n]$ to $[\,]$ are justified by a justification p', then $p'[\,]$ evaluates to a list of theorems $[\vdash th_1;\ldots;\vdash th_n]$ such that $p[th_1;\ldots;th_n]$ evaluates to a theorem $\vdash th$ achieving g.

In the next chapter we move on to show how HOL can be applied to specifying and verifying hardware. This chapter, although not intended as a HOL manual, has served as an introduction to the main features of HOL which will enable a thorough understanding of the rest of this thesis. Further information on all aspects of the HOL theorem proving system can be found in the various references suggested throughout the text (e.g. [20,22]).

Chapter 3

Specifying and Verifying Hardware

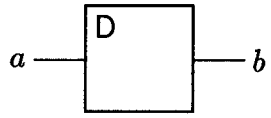
The first stage in the process of hardware verification is to write down the mathematical and logical definitions that describe the behaviour of the particular hardware to be verified. The next stage is to design a circuit which operates in the manner specified in the behavioural definitions, and to write down definitions that describe the implementation.

In this chapter we show various ways in which the HOL logic can be used to write hardware specifications and implementation definitions. We also demonstrate how these definitions are used in proof to verify hardware, and illustrate the general ideas presented on the specification and verification of hardware by the example of an n-bit adder.

3.1 Relational and Functional Specifications

In HOL, behaviour can be modelled in two different ways: by relations or functions. The relational style is the usual way of writing specifications but the functional style is necessary to facilitate simulation, as described later in the thesis.

Consider the device D shown below, with input line a and output line b.



The behaviour of this device can be modelled by defining a relation as follows:

$$\mathsf{D}_{rel}(a, b) \equiv t\,[a, b]$$

where D_{rel} is a predicate symbol of two arguments which abbreviates some boolean term $t\,[a, b]$ involving a and b. $\mathsf{D}_{rel}(a, b)$ holds if and only if a and b are allowable values on the corresponding lines of D.

The behaviour of D can also be modelled using a function:

$$\mathsf{D}_{fun}(a) = t\,[a]$$

where D_{fun} is a function symbol of one argument and $t\,[a]$ is some term involving a. The application $\mathsf{D}_{fun}(a)$ computes the output value on line b of device D given an input value a.

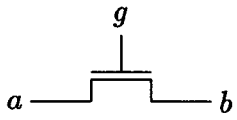
The two models are different. In the relational model all ports of the device are parameters to the specification and no distinction is made between inputs and outputs. In the functional style, however, one is only interested in evaluating the values on the output ports as functions of the inputs. Only the input lines are needed as parameters.

For simple cases, the correspondence between the two styles of definitions is:

$$\mathsf{D}_{rel}(a, b) \equiv (b = \mathsf{D}_{fun}(a))$$

In this chapter, however, we show that this is not generally the case, and an alternative correspondence is explained.

Not only are the two styles different, but in certain cases a relational model is possible when a functional model is not. For example, a delayless CMOS n-transistor shown below:



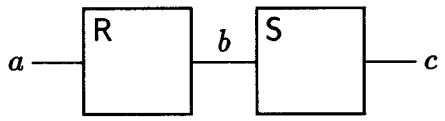
can be modelled using a relation as follows:

$$\mathsf{Ntran}(g, a, b) \equiv (g \supset (a = b))$$

where Ntran is a predicate symbol abbreviating the definition of the behaviour of an n-transistor [7]. The above definition states that $\mathsf{Ntran}(g, a, b)$ holds if and only if a and b are equal whenever g is true. The predicate Ntran specifies a bidirectional device because there is no distinction between inputs and outputs for the ports a and b. The definition can be used to model port a as input and port b as output, or vice-versa.

With functions, however, a model like the above is not possible. One would have to define functions which model the transistor as a unidirectional device, either with a as an input port and b as an output port, or vice-versa. Reasons supporting the use of both relations and functions are given on page 31.

To see how structure can be specified in logic, consider a simple case when internal lines are involved. The diagram below shows the connection of two devices R and S via an internal line b to form a larger device with external ports a and c.



The relational definition for the overall device is:

$$\mathsf{Dev}_{rel}(a, c) \equiv \exists b.\ \mathsf{R}_{rel}(a, b) \wedge \mathsf{S}_{rel}(b, c)$$

while the functional definition is:

$$\mathsf{Dev}_{fun}(a) = (\mathsf{let}\ b = \mathsf{R}_{fun}(a)\ \mathsf{in}\ (\mathsf{S}_{fun}\ b))$$

The let statement used in the definition above is part of the HOL syntax. In fact it is merely syntactic sugaring for the lambda abstraction used in the alternative definition of Dev_{fun} below.

$$\mathsf{Dev}_{fun}(a) = (\lambda b.\ (\mathsf{S}_{fun}\ b))(\mathsf{R}_{fun}(a))$$

In order to be consistent, let statements will be used in functional definitions throughout this thesis. Reasons for using let expressions instead of λ-abstractions are given in Chapter 4.

Two aspects of the above definitions come to attention here.

- The first is that of *composition*. In the relational model, this is represented by conjunction ($\wedge$). In the functional definition this is represented by evaluating the output of the first 'block' in a structure and passing it on as input to the next 'block'.

- *Hiding* is the other aspect. In the relational model, the use of existential quantifiers ($\exists$) enables one not to mention the internal line variables as parameters to the external relation. The same hiding effect is achieved in the functional model by the use of let statements to declare local variables.

Traditionally, in HOL, it has been more common to model behaviour in a relational way because it is relatively easy and natural to express the behaviour derived from structure relationally. With the use of predicates, one merely states boolean conditions which define the intended behaviour of a device and so one has the advantage of:

- only stating the conditions describing the features of a device which are of interest, thus forming a *partial specification*,

- dealing with bidirectional devices by merely defining relations between ports without distinguishing inputs from outputs (as seen in the transistor example above).

Functional definitions, however, also have advantages.

- They are necessarily total. This will be demonstrated in the following sections discussing the verification of an adder.
- Given a suitable interpreter, they can be executed.

Hence, both relations and functions have advantages. It is therefore desirable to have a theorem prover which supports both styles, and perhaps to translate from one style to another.

Comparisons like "how both relations and functions cope with common techniques such as hiding or structure" are important in developing some form of automatic mapping between relational and functional definitions, and are discussed in a later chapter.

The relations and functions shown above are indeed very similar. This is not, however, always the case, and the verification of an n-bit adder described in later sections illustrates this. We first describe general statements that could be proved to demonstrate the correctness of hardware designs using both relational and functional models.

3.1.1 Verification using Relations

In this section, we present the general form of correctness statements that can be proved using relational specifications. For simple devices it is possible to prove that the implementation definition and the behavioural specification are equivalent. The correctness statement is of the form:

$$\forall i_1 \ldots i_m\ o_1 \ldots o_n.$$
$$\mathsf{Imp}_{rel}(i_1, \ldots, i_m, o_1, \ldots, o_n) \equiv \mathsf{Spec}_{rel}(i_1, \ldots, i_m, o_1, \ldots, o_n)$$

where Imp_{rel} and Spec_{rel} are predicate symbols representing the implementation and specification respectively, $i_1 \ldots i_m$ represent the input ports, and $o_1 \ldots o_n$ represent the output ports.

When modelling complex devices, however, the correctness statement is generally formulated as an implication of the form:

$$\forall i_1 \ldots i_m\ o_1 \ldots o_n.$$
$$\mathsf{Imp}_{rel}(i_1, \ldots, i_m, o_1, \ldots, o_n) \supset \mathsf{Spec}_{rel}(Abs\,(i_1, \ldots, i_m, o_1, \ldots, o_n))$$

instead of as a logical equivalence.

In the above definition, *Abs* is some data abstraction function. The implementation and specification definitions can model a device using different data types to represent external ports and internal lines and so a data abstraction function can be required to convert the data representations of the implementation to that of the specification [39].

The above implication is the typical form of a correctness statement using relational definitions. An explanation of why it is usually wrong to require that the implementation be logically equivalent to the specification when modelling complex devices is given in [7] along with an account of the *false implies everything problem* which can result from correctness statements like the above.

3.1.2 Verification using Functions

The kind of theorem which has to be proved when using functional definitions looks like:

$$\forall i_1 \ldots i_m.\ Abs'(\mathsf{Imp}_{fun}(i_1, \ldots, i_m)) = Abs''(\mathsf{Spec}_{fun}(Abs\,(i_1, \ldots, i_m)))$$

where *Abs*, *Abs'* and *Abs''* are some data abstraction functions. Again, a data abstraction function *Abs* can be required to convert data representations of the implementation to that of the specification. Furthermore, the functional definitions need not be equal for all valid data values and so data abstraction functions *Abs'* and *Abs''* are also required to restrict and select the data for which the implementation and specification definitions can be shown to be equal. The use of data abstraction to select and restrict the domain of a function is analogous to defining partial specifications when using relations. An example of the use of data abstraction functions for this purpose is given in Section 3.4.2. In the case that the specification and the implementation definitions are equivalent for all values of data then *Abs'* and *Abs''* are the identity functions.

Examples of the correctness statements explained above are given in following sections describing the formal specification and verification of an n-bit adder. In the next section we first define some data abstraction functions for handling bit-strings which will be required to carry out the adder proof.

3.2 Representation of n-bit Words

Bit-strings, or n-bit words, are sequences of values called bits, used to represent binary digits. In the adder example that follows, we represent n-bit inputs and

outputs with functions from integers (representing bit positions) to booleans (representing bits). For example, the 4-bit word TFTT is represented by a function f say, such that $f(0) = \mathsf{T}$, $f(1) = \mathsf{T}$, $f(3) = \mathsf{F}$ and $f(4) = \mathsf{T}$. The bit positions start at 0, the position of the least significant bit (the rightmost bit in the above example). The n^{th} bit, therefore, is at position $n-1$.

To perform arithmetic computations on the input values of the adder, we define a function *bitval* which maps the boolean truth-values, T and F, to the integer values, 1 and 0, respectively:

$$bitval(x) = (x \Rightarrow 1 \mid 0)$$

In order to relate bit-strings to natural numbers we also define a function *val* by primitive recursion, as follows:

$$val(0, f) = bitval(f(0)) \wedge$$
$$val(n{+}1, f) = (2^{n+1} \times bitval(f(n{+}1))) + val(n, f)$$

The higher order function *val* computes the natural number corresponding to the n-bit word represented by some function. This is done by using *bitval* to convert the n^{th} bit to 1 or 0, multiplying it by 2^n to obtain the integer value denoted by that bit at position n, and adding this result to the value represented by the remaining $n-1$ bits computed recursively. The recursion halts when the word is only one bit long (i.e. when $n{=}0$) and hence, returns an integer value of either 1 or 0.

We also need to define a function *boolval* which maps natural numbers to binary words, i.e. the inverse of the function *val*. The higher order function *boolval* is defined recursively on n, the number of bits which the word representation of a number v should have. The application $(boolval\, n\, v)$ returns a function f say, such that $f(n)\, f(n{-}1) \ldots f(0)$ is the word representation of:

- v, if $0 \leq v < 2^{n+1}$;
- $2^{n+1} - 1$, the largest number that can be represented by n bits, otherwise.

The formal definition of *boolval* is given below.

$$boolval\ 0\ v = (\lambda m.\ (m = 0) \Rightarrow (v \geq 1) \mid \mathsf{ARB\_VAL}) \wedge$$
$$boolval\ n{+}1\ v =$$
$$(\lambda m.\ (m = (n{+}1)) \Rightarrow (2^{n+1} \leq v) \mid$$
$$(2^{n+1} \leq v) \Rightarrow (boolval\ n\ (v{-}2^{n+1})\ m) \mid$$
$$(boolval\ n\ v\ m))$$

The base case of the definition above defines the representation of a one-bit word. The values of v that can be represented (as one bit) are either 1 or 0, which map to T or F respectively. Hence, the application $(boolval\ 0\ v)$ returns a function which, when applied to 0, returns the bit that represents the value of v. For example, if $v = 1$ then $(boolval\ 0\ v)$ returns a function f say, such that $f(0) = true$. If the function is evaluated for any other bit position, the result is undefined and so it is desirable to return some arbitrary value.

This can be done by defining a constant ARB_VAL which represents an arbitrary value of type *bool* by using Hilbert's choice operator ε. The constant ARB_VAL is defined as follows:

$$\mathsf{ARB\_VAL} = \varepsilon\ x{:}bool.\ \mathsf{T}$$

The definition states that ARB_VAL is that boolean value such that T holds, i.e. any value. Because of the way ε is axiomatised in higher order logic, nothing can be proved about ARB_VAL that does not also hold for every value of type *bool*; it therefore represents an arbitrary value.

In the recursive case of the definition of *boolval*, a function is returned which recursively computes the bit at the required position. For example, the application $(boolval\ n{+}1\ v)$ returns a function g say, which when evaluated for bit position $n{+}1$, returns T if $v \geq 2^{n+1}$; F otherwise. When g is evaluated for bit positions less than $n{+}1$, the bit is computed recursively from the integer v if the value of v is less than 2^{n+1}; from the result of $v{-}2^{n+1}$ otherwise.

The abstraction functions described above for relating natural numbers to binary digits, are required to specify and verify the n-bit adder presented in the following sections.

3.3 A 1-bit Full-Adder

This section illustrates the proof of a 1-bit full-adder in preparation for the verification of the n-bit adder discussed in the sections following.

A full-adder cell consists of two components, SUM and CARRY, which generate a sum and carry-out from two inputs and a carry-in. Together, the components SUM and CARRY (shown in Figure 3.1) compute the binary addition of three bits. For example, if i_1, i_2 and *cin* have values 1, 1, and 0 respectively, then SUM computes the binary sum 0 and CARRY computes the carry-over value of 1.

The desired behaviours of the components SUM and CARRY are specified by the predicates SUM_{rel} and CARRY_{rel}, or by the functions SUM_{fun} and CARRY_{fun},

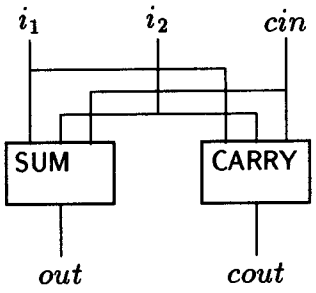


Figure 3.1: A Full-Adder

as follows:

$$\mathsf{SUM}_{rel}(i_1, i_2, cin, out) \equiv (out = (\neg i_1 \wedge \neg i_2 \wedge cin) \vee (\neg i_1 \wedge i_2 \wedge \neg cin) \vee (i_1 \wedge \neg i_2 \wedge \neg cin) \vee (i_1 \wedge i_2 \wedge cin))$$

$$\mathsf{CARRY}_{rel}(i_1, i_2, cin, cout) \equiv (cout = (i_1 \wedge i_2) \vee (i_1 \wedge cin) \vee (i_2 \wedge cin))$$

$$\mathsf{SUM}_{fun}(i_1, i_2, cin) = (\neg i_1 \wedge \neg i_2 \wedge cin) \vee (\neg i_1 \wedge i_2 \wedge \neg cin) \vee (i_1 \wedge \neg i_2 \wedge \neg cin) \vee (i_1 \wedge i_2 \wedge cin)$$

$$\mathsf{CARRY}_{fun}(i_1, i_2, cin) = (i_1 \wedge i_2) \vee (i_1 \wedge cin) \vee (i_2 \wedge cin)$$

The above definitions of SUM and CARRY are derived from the truth tables describing their behaviour. The implementations of the SUM and CARRY devices, as shown in Figure 3.2, consist of simpler components, namely or-gates, and-gates and xor-gates.

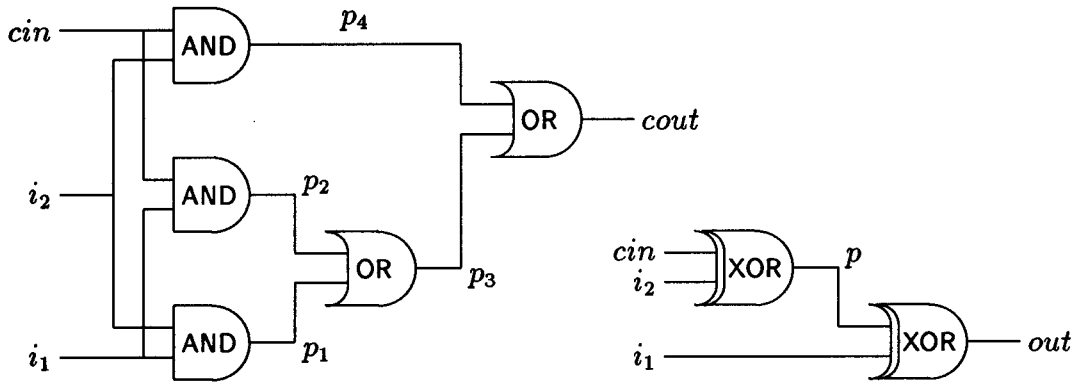


Figure 3.2: Implementations of CARRY (left) and SUM (right)

These logic gates are defined as primitives using the relations:

$$\mathsf{OR}_{rel}(i_1, i_2, o) \equiv (o = i_1 \vee i_2)$$

$$\mathsf{AND}_{rel}(i_1, i_2, o) \equiv (o = i_1 \wedge i_2)$$

$$\mathsf{XOR}_{rel}(i_1, i_2, o) \equiv (o = (i_1 \wedge \neg i_2) \vee (\neg i_1 \wedge i_2))$$

or, in the case of SUM_{fun} and CARRY_{fun}, the functions:

$$\mathsf{OR}_{fun}(i_1, i_2) = (i_1 \vee i_2)$$

$$\mathsf{AND}_{fun}(i_1, i_2) = (i_1 \wedge i_2)$$

$$\mathsf{XOR}_{fun}(i_1, i_2) = (i_1 \wedge \neg i_2) \vee (\neg i_1 \wedge i_2)$$

As shown in the previous section, the two styles of expressing behaviour are sometimes closely connected. The similarity between the corresponding relational and functional models of the logical gates above is obvious. Slightly less obvious is the connection between the implementation definitions of SUM and CARRY. Consider the definitions for SUM. The relational definition based on the structures shown in Figure 3.2 is defined using a predicate $\mathsf{SUM\_IMP}_{rel}$:

$$\mathsf{SUM\_IMP}_{rel}(i_1, i_2, cin, out) \equiv \exists p.\, \mathsf{XOR}_{rel}(i_1, p, out) \wedge \mathsf{XOR}_{rel}(i_2, cin, p)$$

The functional definition, however, uses a function $\mathsf{SUM\_IMP}_{fun}$, where

$$\mathsf{SUM\_IMP}_{fun}(i_1, i_2, cin) = (\mathsf{let}\ p = \mathsf{XOR}_{fun}(i_2, cin)\ \mathsf{in}\ \mathsf{XOR}_{fun}(i_1, p))$$

The technique of hiding the internal line p is demonstrated in both definitions. In one case p is bound using an existential quantifier, while in the other case it is bound within a **let** statement on the right hand side of the equation. The definitions for CARRY follow a similar style:

$$\begin{aligned}
&\mathsf{CARRY_IMP}_{rel}(i_1, i_2, cin, cout) \equiv \\
&\quad \exists\, p_1\ p_2\ p_3\ p_4\ . \\
&\qquad \mathsf{AND}_{rel}(i_1, i_2, p_1) \wedge \mathsf{AND}_{rel}(i_1, cin, p_2) \wedge \\
&\qquad \mathsf{AND}_{rel}(i_2, cin, p_4) \wedge \mathsf{OR}_{rel}(p_1, p_2, p_3) \wedge \\
&\qquad \mathsf{OR}_{rel}(p_3, p_4, cout)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{CARRY_IMP}_{fun}(i_1, i_2, cin) = \\
&\quad \mathsf{let}\ p_1 = \mathsf{AND}_{fun}(i_1, i_2) \\
&\quad \mathsf{and}\ p_2 = \mathsf{AND}_{fun}(i_1, cin) \\
&\quad \mathsf{and}\ p_4 = \mathsf{AND}_{fun}(i_2, cin) \\
&\quad \mathsf{in} \\
&\quad \mathsf{let}\ p_3 = \mathsf{OR}_{fun}(p_1, p_2) \\
&\quad \mathsf{in} \\
&\quad \mathsf{OR}_{fun}(p_3, p_4)
\end{aligned}$$

From these definitions it is routine to prove the four theorems below [7].

$$\vdash \mathsf{SUM}_{rel}(i_1, i_2, cin, out) \equiv \mathsf{SUM\_IMP}_{rel}(i_1, i_2, cin, out)$$

$$\vdash \mathsf{CARRY}_{rel}(i_1, i_2, cin, cout) \equiv \mathsf{CARRY\_IMP}_{rel}(i_1, i_2, cin, cout)$$

$$\vdash \mathsf{SUM}_{fun}(i_1, i_2, cin) = \mathsf{SUM\_IMP}_{fun}(i_1, i_2, cin)$$

$$\vdash \mathsf{CARRY}_{fun}(i_1, i_2, cin) = \mathsf{CARRY\_IMP}_{fun}(i_1, i_2, cin)$$

An outline of the proof of correctness for the implementation of SUM using relational definitions is given in [7] and the proof for the implementation of CARRY is almost identical. The proofs for the functional implementation definitions of SUM and CARRY are also very similar. Below is an outline of the proof of the functional definitions of SUM.

1. The theorem we wish to prove is:

$$\mathsf{SUM}_{fun}(i_1, i_2, cin) = \mathsf{SUM\_IMP}_{fun}(i_1, i_2, cin)$$

2. Expanding using the definition of $\mathsf{SUM\_IMP}_{fun}$, gives:

$$\mathsf{SUM}_{fun}(i_1, i_2, cin) = (\mathsf{let}\ p = \mathsf{XOR}_{fun}(i_2, cin)\ \mathsf{in}\ \mathsf{XOR}_{fun}(i_1, p))$$

3. Expanding using the definition of XOR_{fun} yields:

$$\begin{array}{l}\mathsf{SUM}_{fun}(i_1, i_2, cin) = \\ \mathsf{let}\ p = ((i_2 \wedge \neg cin) \vee (\neg i_2 \wedge cin)) \\ \mathsf{in} \\ (i_1 \wedge \neg p) \vee (\neg i_1 \wedge p)\end{array}$$

4. Eliminating the let statement gives:

$$\begin{array}{l}\mathsf{SUM}_{fun}(i_1, i_2, cin) = (i_1 \wedge \neg((i_2 \wedge \neg cin) \vee (\neg i_2 \wedge cin))) \vee \\ \qquad\qquad\qquad\qquad (\neg i_1 \wedge ((i_2 \wedge \neg cin) \vee (\neg i_2 \wedge cin)))\end{array}$$

5. Expanding using the definition of SUM_{fun} we obtain:

$$\begin{array}{l}(\neg i_1 \wedge \neg i_2 \wedge cin) \vee (\neg i_1 \wedge i_2 \wedge \neg cin) \vee \\ (i_1 \wedge \neg i_2 \wedge \neg cin) \vee (i_1 \wedge i_2 \wedge cin) \\ = \\ (i_1 \wedge \neg((i_2 \wedge \neg cin) \vee (\neg i_2 \wedge cin))) \vee \\ (\neg i_1 \wedge ((i_2 \wedge \neg cin) \vee (\neg i_2 \wedge cin)))\end{array}$$

which can be proven by simple boolean algebra, hence proving the original term.

A full-adder, ADD1, can now be defined as a 'block' by combining the definitions for SUM and CARRY. Relationally, we obtain:

$$\mathsf{ADD1}_{rel}(i_1, i_2, cin, out, cout) \equiv \\ \mathsf{SUM}_{rel}(i_1, i_2, cin, out) \wedge \mathsf{CARRY}_{rel}(i_1, i_2, cin, cout)$$

and functionally:

$$\mathsf{ADD1}_{fun}(i_1, i_2, cin) = \\ \mathsf{let}\ out = \mathsf{SUM}_{fun}(i_1, i_2, cin) \\ \mathsf{and}\ cout = \mathsf{CARRY}_{fun}(i_1, i_2, cin) \\ \mathsf{in}\ (out, cout)$$

3.4 An n-bit Adder

We now outline the verification of an n-bit adder, showing how it can be modelled in HOL using both relational and functional definitions.

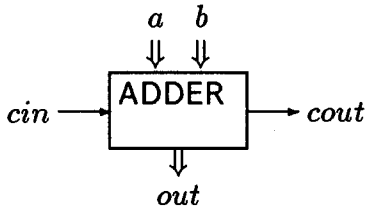


Figure 3.3: Specification of a Binary Adder

Figure 3.3 shows the specification diagram of the adder. It shows a device ADDER which takes three inputs: two n-bit words a and b, and a carry-in bit cin; and returns two outputs: the n-bit sum out, and the carry-out bit $cout$. The single bit values on cin and $cout$ are represented by booleans, and the n-bit word values on a, b and out are represented by functions of type $num \rightarrow bool$ (as explained in Section 3.2).

The relational specification of the behaviour of the adder given below uses val and $bitval$ to relate the outputs out and $cout$ to the inputs a, b and cin by showing that the integer representation of the outputs is equal to the sum of the integer representations of the inputs.

$$\mathsf{ADDER\_SPEC}_{rel}(n, a, b, cin, out, cout) \equiv \\ (2^{n+1} \times bitval(cout)) + val(n, out) = val(n, a) + val(n, b) + bitval(cin)$$

The functional specification for the adder, $\mathsf{ADDER\_SPEC}_{fun}$, can be defined as shown below:

$$
\begin{array}{l}
\mathsf{ADDER\_SPEC}_{fun}\ n\ a\ b\ cin = \\
\quad \mathsf{let}\ int\_sum = (val\ n\ a) + (val\ n\ b) + bitval(cin) \\
\quad \mathsf{in} \\
\quad \mathsf{let}\ sum = (boolval\ (n{+}1)\ int\_sum) \\
\quad \mathsf{in} \\
\quad (sum,\ sum(n{+}1))
\end{array}
$$

It uses *val* and *bitval* to compute the integer sum of a, b and cin, and converts the result to a binary word by using the inverse of *val*, *boolval*. The function $\mathsf{ADDER\_SPEC}_{fun}$ returns a pair of values: the first element *sum* is a function which represents the result of adding a, b and cin, and the second element is the carry value evaluated by $sum(n{+}1)$, the most significant bit of the n-bit addition of a, b and cin.

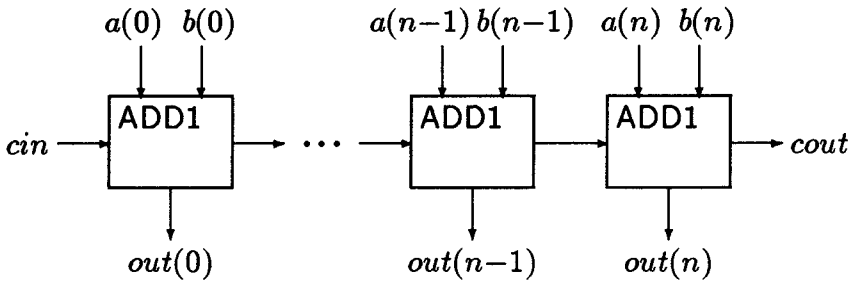


Figure 3.4: Implementation of a Binary Adder

Figure 3.4 above, showing iterated 1-bit adder slices, can be represented in logic in several ways [21]. The most straightforward way to achieve a relational model is to use a primitive recursive definition as follows:

$$
\begin{array}{l}
\mathsf{ADDER\_IMP}_{rel}(0, a, b, cin, out, cout) \equiv (cout = cin) \wedge \\
\mathsf{ADDER\_IMP}_{rel}(n{+}1, a, b, cin, out, cout) \equiv \\
\qquad \exists\, cn.\ \mathsf{ADDER\_IMP}_{rel}(n, a, b, cin, out, cn) \wedge \\
\qquad\quad \mathsf{ADD1}_{rel}(a(n), b(n), cn, out(n), cout)
\end{array}
$$

Here, if the value of n is 0 then no addition is performed but cin is wired straight through to $cout$ (i.e. we have a 0-bit adder). There is no value associated with out in the base case of the relational model shown above so the specification is only partial. This brings up a problem when defining the functional model. All functions in HOL must be total functions, and so it is not possible to partially specify behaviour as in the case of relations. The function must be defined in all cases—even in the *don't care* cases.

To solve this we could define a constant ARB_FUN which represents an arbitrary function of type *num→bool* in a way similar to the way that ARB_VAL was defined on page 35. The constant ARB_FUN is defined as follows:

$$\mathsf{ARB\_FUN} = \varepsilon\, f{:}num{\rightarrow}bool.\ \mathsf{T}$$

Arbitrary values, however, provide a commonly used technique for choosing *don't care* values, especially when converting partial specifications to total functions. It is therefore convenient to define an arbitrary value of polymorphic type α as follows:

$$\mathsf{ARB} = \varepsilon\, x{:}\alpha.\ \mathsf{T}$$

Separate definitions for ARB_FUN and ARB_VAL are hence no longer necessary, since the constant ARB can be used in either case with its type α instantiated to *num→bool* and *bool* respectively.

When defining the implementation of the adder functionally, now, the output *out* can be set to ARB in the case when n is 0, thus making the function $\mathsf{ADDER\_IMP}_{fun}$ total.

$$\begin{array}{l}\mathsf{ADDER\_IMP}_{fun}\ 0\ a\ b\ cin = (\mathsf{ARB}, cout) \wedge \\ \mathsf{ADDER\_IMP}_{fun}\ n{+}1\ a\ b\ cin = \\ \qquad\qquad \mathsf{let}\ (out, cn) = (\mathsf{ADDER\_IMP}_{fun}\ n\ a\ b\ cin)\ \mathsf{in} \\ \qquad\qquad \mathsf{let}\ (outn, cout) = \mathsf{ADD1}_{fun}(a(n), b(n), cn)\ \mathsf{in} \\ \qquad\qquad ((\lambda m.\ (m{=}n) \Rightarrow outn \mid out(m)), cout)\end{array}$$

The function $\mathsf{ADDER\_IMP}_{fun}$ is recursive and curried over its parameters. It computes a pair with the following elements.

- The first is a function which represents the n-bit output.
- The second is the carry-out (or overflow).

From the definitions of $\mathsf{ADDER\_IMP}_{rel}$ and $\mathsf{ADDER\_IMP}_{fun}$ it is evident that the function is not an obvious translation of the relation because the functional definition contains more information. In the base case, the function gives the value on *out* at time 0. In the recursive case, the output *out* is defined as a λ-expression:

$$\lambda m.\ (m{=}n) \Rightarrow outn \mid out(m)$$

which computes the entire n-bit wide bus output by an n-bit adder. A direct translation of the relation would not include such a definition of *out*, and although all n bits on *out* would be recursively computed, only the last value $out(n)$ would

be returned, but such a functional definition would not be an accurate model of the n-bit adder shown in Figure 3.4 where all n output lines are specified as external ports. The problem here is that in the relational definition it is enough to model the structure of the device by showing the way in which components are connected. In the functional definition, however, it is also necessary to ensure that the outputs computed by the function model exactly the outputs of the device.

Automatic translation from relations to functions is by no means straightforward and is dealt with in Chapter 6. Verification is needed to show that the functions and relations do model the same behaviour.

3.4.1 Formal Proof of Relational Specifications

To show that the implementation of an n-bit adder correctly performs binary addition, we prove the following theorem about the relational models of the adder.

$$\forall n.\ \forall a\ b\ cin\ out\ cout.$$
$$\mathsf{ADDER\_IMP}_{rel}(n{+}1, a, b, cin, out, cout) \supset$$
$$\mathsf{ADDER\_SPEC}_{rel}(n, a, b, cin, out, cout)$$

The correctness statement above, unlike those for the SUM and CARRY components on page 38 formulated as logical equivalences, is expressed as an implication as explained in Section3.2.

The proof of the relational models of the adder proceeds as follows by performing mathematical induction on n. The basis and step cases obtained are:

$$\forall a\ b\ cin\ out\ cout.$$
$$\mathsf{ADDER\_IMP}_{rel}(0{+}1, a, b, cin, out, cout) \supset$$
$$\mathsf{ADDER\_SPEC}_{rel}(0, a, b, cin, out, cout)$$

and

$$\forall a\ b\ cin\ out\ cout.$$
$$\mathsf{ADDER\_IMP}_{rel}(n{+}1, a, b, cin, out, cout) \supset$$
$$\mathsf{ADDER\_SPEC}_{rel}(n, a, b, cin, out, cout)$$
$$\supset$$
$$\forall a\ b\ cin\ out\ cout.$$
$$\mathsf{ADDER\_IMP}_{rel}(n{+}1{+}1, a, b, cin, out, cout) \supset$$
$$\mathsf{ADDER\_SPEC}_{rel}(n{+}1, a, b, cin, out, cout)$$

Proving the basis case is straightforward but the step case is tedious though not difficult. A brief outline of this proof is given below.

1. The proof of the basis case is done by repeatedly expanding using the definitions of $\mathsf{ADDER\_IMP}_{rel}$ and $\mathsf{ADDER\_SPEC}_{rel}$ to obtain:

$$(\exists\, cn.\,(cn = cin) \wedge \mathsf{ADD1}_{rel}(a(0), b(0), cn, out(0), cout)) \supset$$
$$(2^{0+1} \times bitval(cout)) + val(0, out) = val(0, a) + val(0, b) + bitval(cin)$$

   which is proved by eliminating the existential quantifier, expanding using the definitions of $\mathsf{ADD1}_{rel}$, SUM_{rel}, CARRY_{rel}, *val* and *bitval*, and using the excluded middle axiom to perform boolean case analysis on $a(0)$, $b(0)$ and *cin*.

2. The induction step is proved by expanding using the definitions of *val*, $\mathsf{ADD1}_{rel}$, SUM_{rel}, CARRY_{rel}, $\mathsf{ADDER\_SPEC}_{rel}$ and $\mathsf{ADDER\_IMP}_{rel}$, assuming the induction hypothesis, and performing modus-ponens with the assumptions to obtain:

$$\begin{array}{l}
\forall\, n\ a\ b\ cn\ out\ cout. \\
\quad (out(n{+}1) = (\neg a(n{+}1) \wedge \neg b(n{+}1) \wedge cn) \vee \\
\qquad\qquad (\neg a(n{+}1) \wedge b(n{+}1) \wedge \neg cn) \vee \\
\qquad\qquad (a(n{+}1) \wedge \neg b(n{+}1) \wedge \neg cn) \vee \\
\qquad\qquad (a(n{+}1) \wedge b(n{+}1) \wedge cn)) \wedge \\
\quad (cout = (a(n{+}1) \wedge b(n{+}1)) \vee (a(n{+}1) \wedge cn) \vee (b(n{+}1) \wedge cn)) \\
\quad \supset \\
\quad (2^{n+1} \times bitval(out(n{+}1))) + (2^{n+1+1} \times bitval(cout)) = \\
\quad (2^{n+1} \times bitval(a(n{+}1))) + (2^{n+1} \times bitval(b(n{+}1))) + \\
\quad (2^{n+1} \times bitval(cn))
\end{array}$$

   which can be proved by eliminating the universal quantifiers, assuming the hypothesis, rewriting using the assumptions, and repeated use of the excluded middle axiom.

3.4.2 Formal Proof of Functional Specifications

From the definitions of $\mathsf{ADDER\_SPEC}_{fun}$ (page 40) and *boolval* (page 34), it is clear that if n is the number of bits in a and b being added together then *sum* only evaluates to non-arbitrary values for arguments between 0 and $n{+}1$ inclusive. For example, looking back to the definition of $\mathsf{ADDER\_SPEC}_{fun}$ it can be seen that:

$$\mathsf{ADDER\_SPEC}_{fun}\ 4\ a\ b\ cin$$

returns a function *sum* and an overflow *sum*(5), where *sum* evaluates to non-ARB values for parameters 0, 1, ..., 5. Furthermore, only the first 4 values of *sum* correspond to the *out* bus shown in Figure 3.4 (i.e. the sum of a and b without the overflow) so we would like to abstract only the first four bits of data from *sum* in order to compute the value returned on *out*.

To do this we define a data abstraction function Data_Abs as follows:

$$\mathsf{Data\_Abs}\ n = (val\ n) \otimes I$$

where I is the identity function and $\otimes$ is an infix operator which takes two functions as arguments and applies them to the respective elements of a pair as defined below.

$$I\ x = x$$
$$(f \otimes g)(x, y) = ((f\ x), (g\ y))$$

When Data_Abs(n) is applied to $\mathsf{ADDER\_SPEC}_{fun}$ it applies $val(n)$ to *sum* and I to $sum(n+1)$ thus restricting the domain of *sum* to 0, 1, ..., n to represent *out*, and leaving the overflow *cout* represented by the second element of the pair untouched.

To show the correctness of the functional definitions of the adder, therefore, we prove the theorem:

$$\begin{array}{l} \forall n\ a\ b\ cin. \\ \quad \mathsf{Data\_Abs}\ n\ (\mathsf{ADDER\_IMP}_{fun}\ n{+}1\ a\ b\ cin) = \\ \quad \mathsf{Data\_Abs}\ n\ (\mathsf{ADDER\_SPEC}_{fun}\ n\ a\ b\ cin) \end{array}$$

The proof of this correctness statement is long and complicated, and makes use of several lemmas. A brief outline of this proof is shown below, but without stating any of the lemmas used. The outline is not intended to provide a thorough understanding of the proof, but merely to provide an idea of what is involved and how it proceeds.

Each of the functional definitions in the correctness statement above computes a pair of type $((num \rightarrow bool) \# bool)$. Therefore, each pair p say, can be rewritten as $(fst(p), snd(p))$, simplified using the definitions of Data_Abs, I and $\otimes$, and the resulting equality rewritten as a conjunction of two equalities using the theorem:

$$\vdash\ ((a, b) = (c, d)) = ((a{=}c) \wedge (b{=}d))$$

to obtain the statement consisting of two separate equations for the values on the output busses *out* and the carry lines *cout*, as shown below.

$$
\begin{array}{l}
\forall\, n\ a\ b\ cin. \\
\quad (val\ n\ (fst\,(\mathsf{ADDER\_IMP}_{fun}\ n{+}1\ a\ b\ cin)) = \\
\quad\ val\ n\ (fst\,(\mathsf{ADDER\_SPEC}_{fun}\ n\ a\ b\ cin))) \wedge \\
\quad (snd\,(\mathsf{ADDER\_IMP}_{fun}\ n{+}1\ a\ b\ cin) = snd\,(\mathsf{ADDER\_SPEC}_{fun}\ n\ a\ b\ cin))
\end{array}
$$

The proof proceeds by performing induction on n. The basis and step cases obtained are:

$$
\begin{array}{l}
\forall\, a\ b\ cin. \\
\quad (val\ 0\ (fst\,(\mathsf{ADDER\_IMP}_{fun}\ 0{+}1\ a\ b\ cin)) = \\
\quad\ val\ 0\ (fst\,(\mathsf{ADDER\_SPEC}_{fun}\ 0\ a\ b\ cin))) \wedge \\
\quad (snd\,(\mathsf{ADDER\_IMP}_{fun}\ 0{+}1\ a\ b\ cin) = snd\,(\mathsf{ADDER\_SPEC}_{fun}\ 0\ a\ b\ cin))
\end{array}
$$

and

$$
\begin{array}{l}
(\forall\, a\ b\ cin. \\
\quad (val\ n\ (fst\,(\mathsf{ADDER\_IMP}_{fun}\ n{+}1\ a\ b\ cin)) = \\
\quad\ val\ n\ (fst\,(\mathsf{ADDER\_SPEC}_{fun}\ n\ a\ b\ cin))) \wedge \\
\quad (snd\,(\mathsf{ADDER\_IMP}_{fun}\ n{+}1\ a\ b\ cin) = snd\,(\mathsf{ADDER\_SPEC}_{fun}\ n\ a\ b\ cin))) \\
\supset \\
(\forall\, a\ b\ cin. \\
\quad (val\ n{+}1\ (fst\,(\mathsf{ADDER\_IMP}_{fun}\ n{+}1{+}1\ a\ b\ cin)) = \\
\quad\ val\ n{+}1\ (fst\,(\mathsf{ADDER\_SPEC}_{fun}\ n{+}1\ a\ b\ cin))) \wedge \\
\quad (snd\,(\mathsf{ADDER\_IMP}_{fun}\ n{+}1{+}1\ a\ b\ cin) = \\
\quad\ snd\,(\mathsf{ADDER\_SPEC}_{fun}\ n{+}1\ a\ b\ cin)))
\end{array}
$$

The proof of the basis case is done by repeatedly expanding using the definitions of $\mathsf{ADDER\_IMP}_{fun}$, $\mathsf{ADDER\_SPEC}_{fun}$ and $\mathsf{ADD1}_{fun}$, by changing the **let** statements to λ-expressions, and by using β-conversion to eliminate the λ-expressions to obtain:

$$
\begin{array}{l}
\forall\, a\ b\ cin. \\
\quad (val\ 0\ (\lambda m.\ (m{=}0) \Rightarrow \mathsf{SUM}_{fun}(a(0), b(0), cin) \mid \mathsf{ARB}(m)) = \\
\quad\ val\ 0\ (boolval\ 0{+}1\ ((val\ 0\ a) + (val\ 0\ b) + bitval(cin)))) \wedge \\
\quad (\mathsf{CARRY}_{fun}(a(0), b(0), cin) = \\
\quad\ boolval\ 0{+}1\ ((val\ 0\ a) + (val\ 0\ b) + bitval(cin))\ 0{+}1)
\end{array}
$$

The above goal is a conjunction of two equations: the first conjunct compares the sum components, and the second conjunct compares the overflows. The goal is proved by specialisation of the universally quantified variables, expanding using the definitions of SUM_{fun}, CARRY_{fun}, *val*, *boolval* and *bitval*, using β-conversion to evaluate λ-expressions, simplifying using simple laws of exponentiation and boolean algebra, and using the excluded middle axiom to perform boolean case analysis on $a(0)$, $b(0)$ and *cin*.

The induction step is proved by assuming the induction hypothesis, expanding using the definitions of $\mathsf{ADDER\_IMP}_{fun}$, $\mathsf{ADDER\_SPEC}_{fun}$ and $\mathsf{ADD1}_{fun}$, changing the let expressions to λ-expressions, and applying β-conversion to obtain the goal below, once again separating the sum from the overflow.

$$
\begin{array}{l}
\forall\, a\ b\ cin. \\
\quad (val\ n{+}1 \\
\qquad (\lambda m.\ (m{=}n{+}1) \Rightarrow \\
\qquad\quad \mathsf{SUM}_{fun}(a(n{+}1), b(n{+}1), snd(\mathsf{ADDER\_IMP}_{fun}\ n{+}1\ a\ b\ cin)) \mid \\
\qquad\quad fst\,(\mathsf{ADDER\_IMP}_{fun}\ n{+}1\ a\ b\ cin)\ m) = \\
\quad val\ n{+}1 \\
\qquad (boolval\ n{+}1{+}1\ ((val\ n{+}1\ a) + (val\ n{+}1\ b) + bitval(cin)))) \wedge \\
\quad (\mathsf{CARRY}_{fun}(a(n{+}1), b(n{+}1), snd(\mathsf{ADDER\_IMP}_{fun}\ n{+}1\ a\ b\ cin)) = \\
\quad boolval\ n{+}1{+}1\ ((val\ n{+}1\ a) + (val\ n{+}1\ b) + bitval(cin))\ n{+}1{+}1)
\end{array}
$$

The goal is proved by expanding using the definitions of *val* and *boolval*, performing β-reduction and modus-ponens, and rewriting using the assumptions and several lemmas.

3.5 Proofs Relating Functions and Relations

In the previous sections we have shown that functional definitions sometimes carry more information than relational definitions, especially when the relational definitions are partial or when extra information is required to specify the outputs in the functional definitions.

It is clear, therefore, that in cases like the above it is not possible to prove the functional definitions equivalent to the corresponding relational definitions. When the two styles of definitions carry the same amount of information, it is possible to prove them equivalent; otherwise the following relationship will hold:

$$
\begin{array}{l}
\forall i_1 \ldots i_m\ o_1 \ldots o_n. \\
\quad ((o_1, \ldots, o_n) = \mathsf{Def}_{fun}(i_1, \ldots, i_m)) \supset \mathsf{Def}_{rel}(i_1, \ldots, i_m, o_1, \ldots, o_n)
\end{array}
$$

The two theorems relating the functional and relational definitions of the adder are:

$$
\begin{array}{l}
((out, cout) = (\mathsf{ADDER\_IMP}_{fun}\ (n{+}1)\ a\ b\ cin)) \supset \\
(\mathsf{ADDER\_IMP}_{rel}\ (n{+}1)\ a\ b\ cin\ out\ cout)
\end{array}
$$

and

$$
\begin{array}{l}
((out, cout) = (\mathsf{ADDER\_SPEC}_{fun}\ n\ a\ b\ cin)) \supset \\
(\mathsf{ADDER\_SPEC}_{rel}\ n\ a\ b\ cin\ out\ cout)
\end{array}
$$

The converses of the above implications do not hold because in both cases the functional definitions are more detailed.

For example, in the proof of the first correctness statement above concerning the implementation definitions of the adder, induction on n is performed, and the basis case subgoal is reduced to the following term:

$$((out, cout) = (\mathsf{ARB}, cin)) \supset (cout = cin)$$

which is easily shown to be true. The same term would clearly be false, however, if the implication were to be replaced by an equivalence since there is no information regarding *out* on the right hand side of the implication. This example provides a simple yet typical case to explain why the more detailed definitions imply the less detailed, but not vice-versa.

Once again, both proofs are rather long, tedious and complicated, needing many supporting lemmas. Induction on n is essential in both cases and several different techniques (such as rewriting, modus-ponens and β-reduction) are required to conduct the proofs. Automatic tools, such as a tautology checker and a partial decision procedure for Presburger arithmetic[1] were useful in conducting the final stages of the proofs where the theorems were reduced to instances of propositional logic or Presburger arithmetic. Both decision procedures work purely by inference so their application within a proof preserves the soundness of that proof.

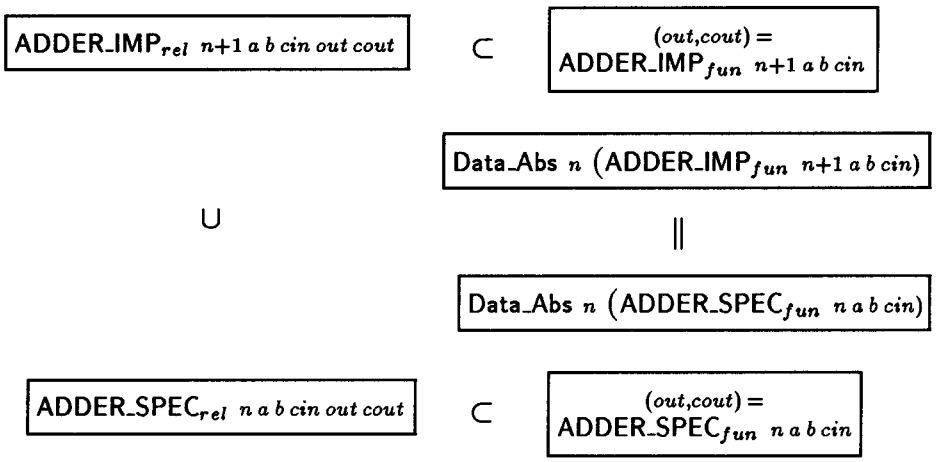


Figure 3.5: Correctness Theorems

[1]These automatic tools were developed and implemented by the author, but are at the moment still undocumented.

Figure 3.5 shows how the four adder definitions are related in proof, and how the functional definitions are found to be more detailed. It is interesting to note the directions of the implications in the theorems as shown in the diagram, where the definition containing least detail is shown to be the relational specification of the adder.

The proofs of all the above mentioned theorems and lemmas were carried out using the HOL theorem proving system.

Chapter 4

Executing Specifications

In Chapter 3 we discussed how the behaviour and structure of digital circuits can be expressed in the HOL logic. The example of an n-bit adder was used to illustrate how specifications are written and how they are used in correctness proofs.

Even in the relatively easy example of the adder, however, it was by no means straightforward to show that the specifications were satisfied. The process of writing down correct specifications involves much checking by hand; the specifications have to be simulated on paper over a range of cases until sufficient confidence is gained that the specifications are correct.

Simulating specifications is, therefore, unavoidable and a mechanical tool to conduct such simulations is of course a natural solution since complicated definitions are too tedious, or impossible, to check by hand.

4.1 ML as a Simulation Language

In this chapter we discuss the application of ML to simulation. In Chapter 1 various attempts to apply programming languages to hardware simulation were mentioned, but few of these were done in the context of a formal proof system. ML is of special interest to us because it is the meta-language of HOL and will therefore provide a natural target language for the execution of the HOL logic discussed in Chapter 6.

Since ML is primarily a functional programming language, we shall be concerned with executing functional specifications. With certain languages it is possible to execute relations, such as in [9] where William Clocksin has used PROLOG to simulate digital circuits using relations. With ML, however, it is necessary for the specifications to be in a functional form, where parameters are passed as inputs and values are calculated and returned as outputs.

In Chapter 6 we will show how ML simulation models can be automatically generated from relational HOL specifications. Often there are several ways of writing the same specification, but in this chapter we stick to one style only; a style that suits automatic translation. This style does not always provide the most straightforward model, but it is consistent in the sense that all the hardware features discussed can be expressed in the same manner. The reason for adhering to the same style of writing specifications, rather than writing specifications in an ad-hoc fashion is that we need a 1-1 correspondence between HOL relations and ML functions to facilitate automatic translation. The choice of style is not arbitrary; it is the style which provides the strongest resemblance to the HOL specifications. In the following sections we give a detailed explanation of this style, and with the aid of several examples we show how it can be used to model many aspects of hardware design.

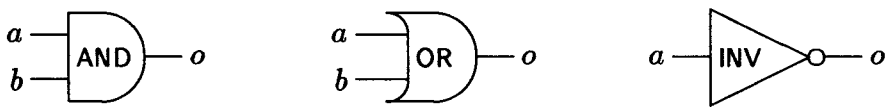
4.2 Combinational Circuits

There are many levels of detail at which hardware circuits may be described. The level at which circuits will be specified, simulated and discussed in this thesis is called the *register-transfer* level.

At the register-transfer level one makes use of the standard logical gates as primitives for building circuits by structuring these simple components into larger blocks. The flow of data between registers is modelled as *signals* which are approximated by sequences of digital values. At this level, time is treated as *discrete* (i.e. it is represented by integers in ML) and so these sequences constituting signals consist of values that are valid during successive clock cycles.

4.2.1 Modelling Logic Gates

Combinational devices such as the inverter are assumed to have no delay. Their outputs are computed as instantaneous results of applying functions to their inputs. The three basic logic operations, 'and', 'or' and 'not', can be performed by using the ML infix operators & and *or*, and the prefix operator *not* respectively. So the logic gates shown below:



in which a and b are the inputs and o is the output, can be modelled by defining

corresponding functions AND, OR and INV as follows:

let AND $a\ b$ = let $o = a\&b$ in o
let OR $a\ b$ = let $o = a\ or\ b$ in o
let INV a = let $o = not\ a$ in o

Immediately the style we adopt for writing definitions becomes obvious. Simpler and more natural definitions could have been used, such as:

let AND′ $a\ b = a\&b$

but instead we use a style in which a function modelling the behaviour of the device is applied to the inputs, and the result is bound to an output using a local let declaration. There are three main reasons for this:

- The adopted style provides a very descriptive model of the contemplated hardware circuit; it shows how the function describing the behaviour of the device is applied to the values on the input ports to evaluate the values that should appear on the output ports. In the alternative definition of AND′ shown above, this detail is somewhat obscured because there is no mention of an output port.

- Functional programs written in this style show an obvious relationship to relational specifications written in HOL. For example, the corresponding HOL relational model for a two-input AND gate is commonly written as:

  $$\mathsf{AND}(a, b, o) \equiv (o = a \wedge b)$$

  The functional definition written in the adopted style is closer to the above relation than the alternative function AND′, once again, mainly due to the modelling of the output ports. The difference is much more pronounced in more complicated examples.

- The adopted style is necessary to model certain features of hardware design, namely feedback in sequential circuits. Keeping to the same method for all models, therefore, makes automatic translation from HOL relations to ML functions easier. Examples on how sequential devices are modelled are presented later on in the chapter.

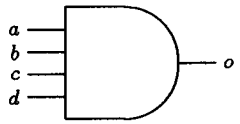
Other gate primitives such as NOR and NAND can be modelled by composition of the basic logic operations. For example, a two input NAND gate can be modelled by

let NAND $a\ b$ = let $o = not\ (a\ \&\ b)$ in o

while a two input NOR gate can be modelled by

$$\mathsf{let\ NOR}\ a\ b = \mathsf{let}\ o = not\,(a\ or\ b)\ \mathsf{in}\ o$$

Logic gates with more than two inputs can similarly be specified by simply performing the logical operations of the gate on two of the inputs and then repeatedly applying the operation on the result and the next input until the logic operation is performed on all inputs. For example, the four-input AND gate shown below:



can be specified as:

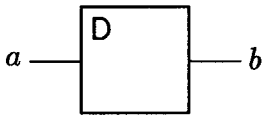
$$\mathsf{let\ AND}\ a\ b\ c\ d = \mathsf{let}\ o = ((a\ \&\ b)\ \&\ c)\ \&\ d\ \mathsf{in}\ o$$

Both & and *or* are commutative and associative on variables[1] so the order in which the operation is applied to the inputs in such cases is irrelevant.

4.2.2 Modelling Behaviour

Once we have established a technique for specifying logic gates, more complicated devices can be modelled by composing several gates together. Indeed, it soon becomes tedious to describe large circuits using their logic gate components and it becomes necessary to group together various components of a circuit into 'black-box' devices.

For example, consider a device D with input a and output b.



The behaviour of the above circuit can be modelled by using the ML let declaration:

$$\mathsf{let\ D}\ a = \mathsf{let}\ b = t[a]\ \mathsf{in}\ b$$

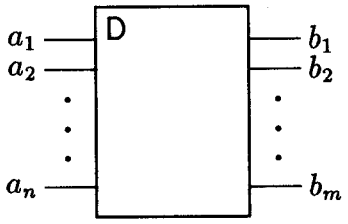
where the value on the output line b is set to the result of the expression $t[a]$ which models the behaviour of device D.

The previous models for logic gates, therefore, were merely special cases of this model for an arbitrary device. For example, in the model for the inverter, INV

[1]They are not, of course, commutative and associative for arbitrary expressions, e.g. $fail \,\&\, false$ is not equal to $false \,\&\, fail$.

can be seen as a particular case of D and the expression $t\,[a]$ which specifies its behaviour is the application $not(a)$.

In fact, in general, a device D may have several inputs and outputs as shown in the diagram below where $a_1, \ldots, a_n$ are the inputs and $b_1, \ldots, b_m$ are the outputs.



The general model for the above device is achieved by using tuples to represent the multiple input and output lines.

$$\mathsf{let}\ \mathsf{D}(a_1, \ldots, a_n) = \mathsf{let}\ (b_1, \ldots, b_m) = t[a_1 \ldots a_n]\ \mathsf{in}\ (b_1, \ldots, b_m)$$

Once again, the abbreviation:

$$\mathsf{let}\ \mathsf{D}(a_1, \ldots, a_n) = t[a_1 \ldots a_n]$$

is equivalent to the above definition; we use the former version for consistency and ease of translation (as explained on page 51).

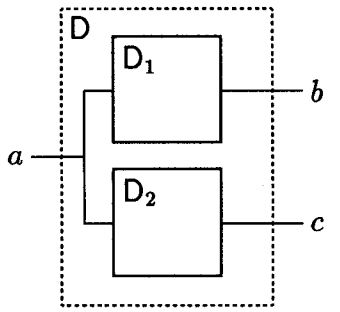
An optional (and equivalent) way to specify the behaviour of the above device is to curry the parameters of D to obtain:

$$\mathsf{let}\ \mathsf{D}\ a_1 \ldots a_n = \mathsf{let}\ (b_1, \ldots, b_m) = t[a_1 \ldots a_n]\ \mathsf{in}\ (b_1, \ldots, b_m)$$

The advantage of defining functions with curried parameters, of course, is that the functions can be partially applied, i.e. a function can be applied to one argument at a time, each time returning a function with one less argument. With the arguments in a cartesian product form, however, the function must be applied to all parameters at once.

4.2.3 Modelling Structure

Devices can be connected in parallel or in series. For example, consider a device D consisting of two devices, D_1 and D_2, connected as shown below:

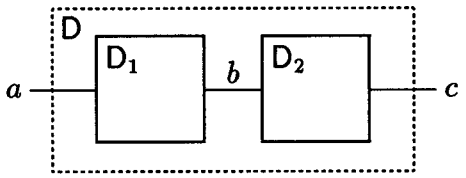


The structure of such a device can be specified using the **and** construct in a **let** statement which allows simultaneous declarations. This neatly captures the notion of devices connected in parallel where the values on the output lines are calculated as function applications of the input lines independent of one device to another. The order in which the output values are calculated in a simultaneous declaration is irrelevant.

```
let D a = let b = D1 a
          and c = D2 a
          in (b, c)
```

The **and** construct used in the above specification is part of the syntax of the ML **let** statement used for simultaneous declarations and must not be confused with the & operator used previously as a logical 'and' operation.

Devices connected in sequence, such as in the example below, are modelled by using the **in** construct to join **let** statements together.



The specification shown below uses two local declarative statements in sequence where the output value of the first device is evaluated and passed on as input to the second device. The order in which the declarative statements appear in the specification is, this time, obviously important.

```
let D a = let b = D1 a
          in
          let c = D2 b
          in c
```

Only external inputs to a device are parameterised in the functional specification definition. In the above example, b is an output with respect to D_1 and an input with respect to D_2, but it becomes an internal line once D_1 and D_2 are grouped to form a larger device D. It is, therefore, said to be *hidden* from the external representation of device D.

At this stage, the reasons for writing ML specifications in the particular style chosen begin to become obvious. The specifications reflect very closely the structure of the circuit; they show the functional specification of a device being applied to its inputs, and the result declared to the outputs.

Of course, the above programs for representing parallel and serial composition can be abbreviated to:

$$\mathsf{let\ D}\ a = ((\mathsf{D}_1\ a), (\mathsf{D}_2\ a))$$

and

$$\mathsf{let\ D} = \mathsf{D}_2 \circ \mathsf{D}_1$$

respectively. These abbreviated versions, however, do not offer such an obvious representation of structure.

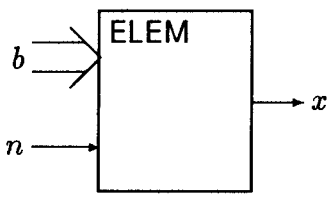
Furthermore, the specifications expressed in ML in the chosen style, and those expressed in HOL in Chapter 3 are very similar. The syntactic sugaring supported by the implementation of the HOL logic makes use of let statements, tuples, etc. much in the same way as ML, and specifications written in HOL are very similar to those written in ML in the described style. The similarity in the way circuits are expressed in HOL and in ML will be useful when translating HOL definitions into ML programs (discussed in Chapter 6) because it makes the mapping clean and straightforward.

4.2.4 Dealing with Partial Specifications

Very often, when writing a specification for a device, only certain features of its behaviour are of interest. One designs a device to behave in a certain way under certain conditions, but does not care about how it behaves under other conditions. A *partial specification* is therefore used to describe the required behaviour of the circuit; anything else remaining unspecified.

For simulation purposes, however, some result must be computed for all data, and so the functions used need to be total. Whenever only a partial specification of the behaviour of a device is available, therefore, it is necessary to construct a total function which models cases outside the scope of the partial specification as *don't care* cases. In ML, this can be done by using a conditional statement to branch off to a *fail* command in unspecified cases.

For example, consider the following device ELEM which takes two inputs: a four-bit wide bus b and an integer n, and returns one output: a bit x corresponding to the n^{th} bit of b.



Before describing the specification for this device, the ML representation of busses must be described. Busses can be modelled in several ways; one way is to use higher order variables as described in Chapter 3 for the representation of binary words in HOL. Thus, we can represent the bus b in the above diagram by a function of type $int \rightarrow bool$ such that $b(0)$ returns the first bit, $b(1)$ returns the second bit, and so on.

The informal specification of the device ELEM: *to select the n^{th} bit of bus b, and output the bit on line x*, is partial. It assumes that the input to the device will always be valid, i.e. for a bus b containing exactly four bits, n is always within the range $1 \leq n \leq 4$. Nothing is said about the behaviour of ELEM when the inputs do not comply to the above assumptions (e.g. if n is outside the specified range). In fact, one does not care about such cases; the device only 'makes sense' when presented with the right kind of input and one should ensure that the input is valid before operating the device.

If the partially specified device forms part of a larger circuit such that the inputs to the device are generated by the preceding circuitry, then the generation of invalid inputs is an indication of bad design or inaccurate modelling of the preceding circuit. In other words, if the circuit functions correctly, inputs should always be valid and failure should never occur.

The ML specification for the ELEM device, therefore, forces failure upon an attempt to evaluate the output line for unspecified cases.

$$\mathsf{let}\ \mathsf{ELEM}\ n\ b = \mathsf{let}\ x = (1 \leq n) \wedge (n \leq 4) \Rightarrow b(n-1) \mid \mathit{fail}\ \mathsf{in}\ x$$

The function checks that the bit selector n lies within the range 1 through 4 before attempting to compute the appropriate bit. The advantage of this technique is that it points out errors during simulation and it enables one to trace inaccuracies in the specifications.

Once again, the above definition could be abbreviated to:

$$\mathsf{let}\ \mathsf{ELEM}\ n\ b = (1 \leq n) \wedge (n \leq 4) \Rightarrow b(n-1) \mid \mathit{fail}$$

4.3 Sequential Circuits

The kind of circuits described so far have been straightforward to model. Very often, however, circuits are more complicated and it becomes necessary to model features like:

- delay—where the input signals take time to propogate to the output signals,

- clocks—where the flow of data through a circuit is controlled to ensure that events occur in the right order.
- feedback—where output lines are fed back in as inputs to a device,

4.3.1 Time, Delay and Clocks

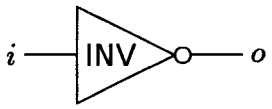
In order to model delay, feedback, clocks, etc., we need to incorporate the notion of time into our definitions. As mentioned earlier, time at the register-transfer level is treated as discrete and so it may be represented in ML by a type *time* represented by the integers.

typeabbrev *time* = *int*

Signals can then be represented as functions from time to values rather than just variables denoting particular values. Such functions with a domain of type *time* are often referred to as *history functions.*

So, for example, devices with no delay such as logic gates can be modelled by using higher order functions which take history functions for inputs and return corresponding history functions for outputs.

For example, the inverter,



can be represented as

let INV i = let $o\ t = not\,(i\ t)$ in o

where i and o are functions of type *time→bool*. If i had the following values for the first five time cycles,

time	0	1	2	3	4	...
i	T	F	F	T	F	...

then (INV i) would return a function, o say, which when mapped over the first five instants of time, the corresponding output values are returned.

time	0	1	2	3	4	...
o	F	T	T	F	T	...

Once again, there are many ways of writing down the same specification. The inverter above could have been specified as:

let INV′ $i = \lambda t.\ not\,(i\ t)$

or even:

$$\mathsf{let\ INV''}\ i = not \circ i$$

but the INV definition was chosen to conform with the style described earlier.

The use of history functions to represent signals makes it easy and natural to model delay. For example, a simple register REG with an input i and output o can act as a storage buffer of one time unit.

This can be modelled by the definition:

$$\mathsf{let\ REG}\ i\ initval = \mathsf{let}\ o\ t = (t{=}0) \Rightarrow initval \mid i(t{-}1)\ \mathsf{in}\ o$$

Three important points result from this specification:

- Delay is modelled by setting the value of the output at time t to some function of the input at time $t-1$, the previous time cycle.

- Time is represented using the non-negative subset of the integers, with time 0 denoting the starting point to halt recursive computations involving time. Hence, one must ensure that statements of the form $t-1$ do not result in a negative time representation (such as if $t{=}0$). In the above example a conditional statement is used to check for the special case when $t{=}0$ and in such a case it assigns an initial value, *initval*, to the output at time t=0. One can think of *initval* as the value present on the output line o when the register is in its initial state. The initial value *initval* is passed as a parameter to the external definition, thus enabling one to simulate the specification with different (or perhaps all) values of *initval*.

- The ML type checker infers that t is of type *time* from the subterm $t{=}0$, but is unable to determine the types of $o(t)$, $i(t-1)$ or *initval*. The variables i and o are, therefore, assigned types $time \rightarrow *$ where $*$ is a type variable. The advantage of not stating the types of i and o explicitly in the definition of REG is that by forcing the type checker to use type variables, one is able to define specifications of 'blocks' which can have different applications. For example, if i_1 is a variable of type $time \rightarrow bool$ and i_2 is a variable of type $time \rightarrow int$, then

$$\mathsf{REG}\ i_1\ false$$

denotes a register storing boolean values with an initial state of *false*, while

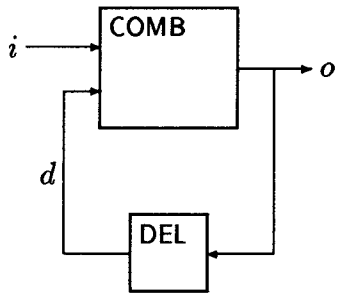
REG i_2 0

denotes a register storing integers with an initial state of 0.

Clocks are just a special kind of signal. They are represented in a very similar way to other signals, but to distinguish them, one can define a new type, *trigger* say, to have two possible values, *OFF* and *ON* (see the example definition and explanation on page 22). Clock lines can then be represented by functions of type *time→trigger*, and can be distinguished from boolean signals of type *time→bool*.

4.3.2 Feedback

The combination of history functions and recursion are used to model feedback in sequential circuits. Consider the following example of a general representation of a finite state machine to see how feedback can be modelled in ML.



The finite state machine consists of two devices:

- A combinational device COMB with no delay—the behaviour of this device can be defined as:

  let COMB i_1 i_2 = let $o(t)$ = $beh[i_1(t), i_2(t)]$ in o

  where i_1, i_2 and o are history functions, and $beh[i_1(t), i_2(t)]$ is some expression which models the behaviour of COMB.

- A delay device DEL which delays the propogation of the input to the output by one time unit—the behaviour of this device can be defined as:

  let DEL i $ival$ = let $o(t)$ = $(t{=}0) \Rightarrow ival \mid i(t{-}1)$ in o

where i and o are history functions modelling the input and output lines respectively, and *ival* is the initial value parameter corresponding to the value on o at time 0.

The formal definition of a finite state machine FSM, therefore, can be written as follows:

```
let FSM i ival =
    letrec o(t) = COMB i d t
    and d(t) = DEL o ival t
    in o
```

The initial state of FSM is parameterised via *ival* which also provides a basis value to the recursive definition, avoiding infinite recursion. At first glance, the above definition of FSM would seem to involve an infinite loop, since all functions are evaluated at time t. The application

```
DEL o ival t
```

however, is expanded to

$$(t{=}0) \Rightarrow ival \mid o(t{-}1)$$

and so $d(t)$ gets evaluated from $o(t{-}1)$, thus evaluating the signals at a previous time unit at each call to DEL until $t{=}0$ when the recursion is halted. The recursive definitions of history functions provide a good representation of feedback since one can evaluate the value of a function at any time prior to the present time t and use it to evaluate the value of a function at time t.

The style used in the above specification of FSM is the standard way for writing recursive definitions in ML. Other equivalent specifications may be possible, but in this case this is the most natural, simple and straightforward. The style used for writing the specifications presented earlier in the chapter is based on this syntax for recursive declarations.

4.4 Example of a Simple Counter

We now present the specification of a simple example which demonstrates several of the techniques for modelling the behaviour and structure of register-transfer level descriptions of hardware discussed in the preceding sections of this chapter.

The device we wish to specify is a counter with two input ports, *cntl* and *in*, and one output port *out*, as shown in Figure 4.1.

Figure 4.1: Specification Diagram of a Counter

The intended behaviour of COUNTER is that the value on *out* is incremented on each cycle unless a new value is input via *in* by setting *cntl* to *true*. We can model the input and output ports *cntl*, *in* and *out* by the history functions of type (*time*→*bool*), (*time*→*int*) and (*time*→*int*) respectively. An ML specification of the behaviour of COUNTER can then be written as:

```
let COUNTER in cntl countval =
    letrec out t = (t=0) ⇒ countval |
                   cntl(t−1) ⇒ in(t−1) |
                   (out(t−1)) + 1
    in out
```

In other words, when *cntl* is *true* at some time, then the input value at that time is passed on to the output line at the next time cycle. When *cntl* is *false*, however, the output value at that instant of time is set to the increment of the output value in the previous time cycle. The output port *out*, therefore, is modelled as a recursive function with an initial value *countval* at time 0.

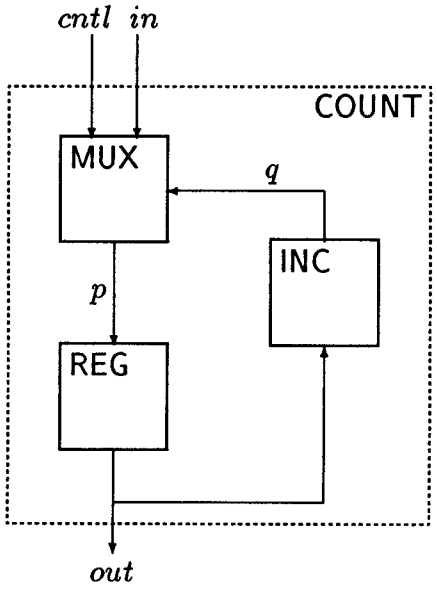


Figure 4.2: Implementation of a Counter

Figure 4.2 shows one possible way of implementing the COUNTER device. The implementation makes use of the same three external ports, *in*, *cntl* and *out*, and is made up of three device components: a multiplexor, a register and an incrementer.

The general behaviour of the multiplexor, MUX, with three input ports i_1, i_2 and sw, and an output *out*, is such that the value on the switch at time t determines the value on *out* at time t as either one of the values on i_1 or i_2.

$$\text{let MUX } i_1\ i_2\ sw = \text{let } out\ t = sw(t) \Rightarrow i_1(t) \mid i_2(t) \text{ in } out \tag{4.1}$$

The register REG merely stores data for one unit of time, so the output value at time t is set to the input value at the previous time unit, $t-1$. At time t=0, the output is in its initial state.

$$\text{let REG } in\ initval = \text{let } out\ t = (t{=}0) \Rightarrow initval \mid in(t{-}1) \text{ in } out \tag{4.2}$$

Like REG, the incrementer INC only has one input port and one output port. The purpose of the incrementer is to add one to the input value at time t and pass the result on to the output port.

$$\text{let INC } in = \text{let } out\ t = in(t) + 1 \text{ in } out \tag{4.3}$$

Having established specifications for the behaviour of MUX, REG and INC, we define the structure of the implementation of the counter, COUNT as:

```
let COUNT in cntl countval =
    letrec p t = (MUX in q cntl) t
    and out t = (REG p countval) t
    and q t = (INC out) t
    in out
```

The applications (MUX *in q cntl*), (REG *p countval*) and (INC *out*) are themselves history functions which represent the histories of values output by the multiplexor, the register and the incrementer respectively. The internal lines p and q are also represented by history functions in the same way as the external lines, but they are hidden from the external definition.

We have, therefore, defined a specification of the intended behaviour of the counter and of a possible implementation of it. The two definitions can now be executed using test data until it is demonstrated that the described implementation does model the device.

For example, if *in* and *cntl* are functions of type *time→int* and *time→bool* respectively, which evaluate to the values shown in the following table over the

first eleven time units, 0 through 10:

time	0	1	2	3	4	5	6	7	8	9	10	...
in	8	7	6	9	8	7	7	6	5	3	2	...
cntl	T	T	F	F	F	F	F	F	T	F	F	...

then if the definitions of COUNTER and COUNT are executed with the same initial values, 0 say:

```
let out = COUNT in cntl 0
let out' = COUNTER in cntl 0
```

the histories of *out* and *out'* can be examined, checking that the results are equal.

time	0	1	2	3	4	5	6	7	8	9	10	...
out	0	8	7	8	9	10	11	12	13	5	6	...
out'	0	8	7	8	9	10	11	12	13	5	6	...

Exhaustive simulation is impossible, however, even in cases as simple as this because there are infinitely many possible input sequences. One can never be absolutely certain of an implementation merely on the basis of inexhaustive simulation so verification of these definitions should be the next move. This is not yet possible because our definitions are only programs expressed in the functional programming language ML—they are not formal specifications expressed within the context of a formal proof system.

Our goal is, therefore, to execute the HOL logic specifications in a similar way to ML programs. One can then simulate and verify hardware using the same definitions. The implementation of an executable subset of the logic is presented in Chapter 6 and various examples of hardware devices specified and simulated in the logic are discussed in Chapters 7, 8 and 9.

4.5 ML or ELLA

The purpose of this chapter has not been merely to show that a general purpose language can be used for simulation, but to show that the meta-language of HOL can be used to model many aspects of hardware design in a style that is very similar to the way hardware is modelled in logic.

It is interesting to look at how ML compares with special purpose, register-transfer level simulators. To do this, we briefly compare ML to ELLA, a simulator designed by the Royal Signals and Radar Establishment and marketed by Praxis Systems plc (see Section 1.2.1). We use the example of a two-input half-adder to carry out the comparison. We do not give any introduction to ELLA (for this

see [55]), but simply explain the ELLA specification for a half-adder, and discuss how it compares with ML and HOL.

The implementation of a two-input half-adder is shown below, where i_1 and i_2 are the inputs, c and s are the carry and sum outputs respectively, and $p_1 \ldots p_4$ are internal lines. The logic gates are labelled for the purpose of the ELLA specification described later, where n_1 and n_2 are inverters, $a_1 \ldots a_3$ are and-gates, and o_1 is an or-gate.

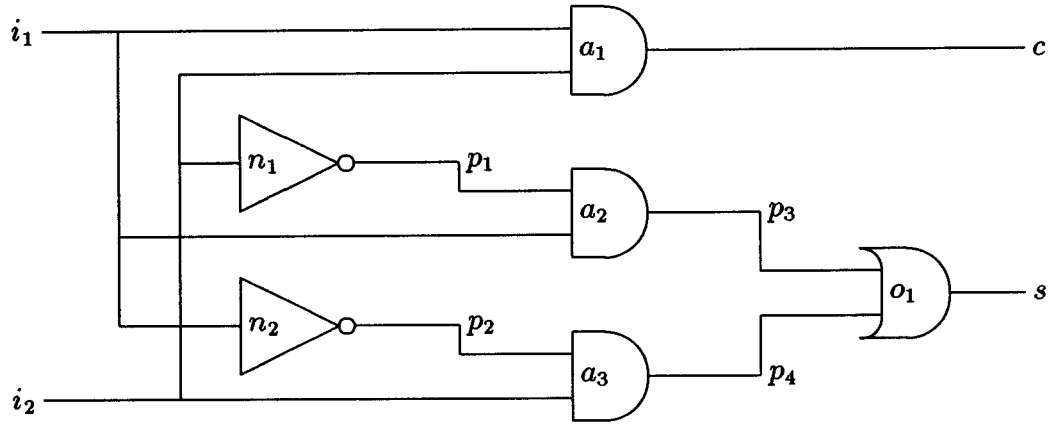


Figure 4.3: Implementation of a Half-Adder

In each of the three languages (HOL, ML and ELLA), there are many ways of writing specifications. Table 4.1 shows three specifications for the half-adder above: a HOL relational specification written in the conventional way, an ML functional specification written in the style described earlier, and an ELLA specification. In all three specifications, the logic gates are represented using functions or predicates with the names AND, OR and NOT. The definitions of these primitives are, of course, different for each specification; the same names are used in all the specifications only for the purpose of comparison.

The HOL and ML models should by now be familiar. In HOL, existential quantification is used for hiding internal lines, and conjunction is used to denote composition. In ML, local let declarations are used to hide internal lines, and constructs are used to denote parallel composition, and in constructs are used to denote serial composition.

In ELLA, the specification shown is referred to as a *function declaration.* In the first line, a node HA is specified to have two inputs of type *bool*, i_1 and i_2, and

HOL Relation	ML Function	ELLA Function
$\mathsf{HA}(i_1, i_2, s, c) \equiv$ $\exists p_1\ p_2\ p_3\ p_4.$ $\mathsf{AND}(i_1, i_2, c) \wedge$ $\mathsf{NOT}(i_2, p_1) \wedge$ $\mathsf{NOT}(i_1, p_2) \wedge$ $\mathsf{AND}(p_1, i_1, p_3) \wedge$ $\mathsf{AND}(p_2, i_2, p_4) \wedge$ $\mathsf{OR}(p_3, p_4, s)$	HA $i_1\ i_2$ = let c = AND $i_1\ i_2$ and p_1 = NOT i_2 and p_2 = NOT i_1 in let p_3 = AND $p_1\ i_1$ and p_4 = AND $p_2\ i_2$ in let s = OR $p_3\ p_4$ in (s, c)	FN HA = $(bool: i_1\ i_2) \rightarrow [2]bool:$ BEGIN MAKE NOT : $n_1\ n_2$, AND : $a_1\ a_2\ a_3$, OR : o_1. JOIN $(i_1, i_2) \rightarrow a_1$, $i_1 \rightarrow n_2$, $i_2 \rightarrow n_1$, $(i_1, n_1) \rightarrow a_2$, $(i_2, n_2) \rightarrow a_3$, $(a_2, a_3) \rightarrow o_1$. OUTPUT (o_1, a_1) END.

Table 4.1: Specifications of a Half-Adder in HOL, ML and ELLA

two outputs (no names specified) also of type *bool*.[2] In the rest of the function enclosed between BEGIN and END, the network is described as follows:

- a MAKE statement is used to declare instances of gates,
- a JOIN statement is used to explicitly interconnect the gates, and
- an OUTPUT statement is used to return the computed values that should appear on the output lines of the circuit.

The use of MAKE and OUTPUT is straightforward; their purpose is simply declarative. The interesting part of the function lies in the JOIN section, particularly in the way in which the connections are described. The statement uses expressions of the form $exp_1 \rightarrow exp_2$ to represent connections from a single node (or tuple of nodes) exp_1 to another node exp_2. A node in ELLA can be the name of any interconnecting device. External inputs are also treated as nodes, but ones which can only be used as inputs.

This way of representing structure in ELLA is different to that of HOL:

- In ELLA, structure is described as a connection of nodes. There is no mention of internal lines; the outputs of devices are labelled as nodes which them-

[2]The type *bool* is assumed to be previously declared for the purpose of the definition. For details on type declarations in ELLA see [55]

selves become inputs to other nodes. Instances of devices, therefore, have to be defined (by using the MAKE statement) in order to distinguish between connections with identical devices. The devices are labelled, and these labels denote the output of a node, or act as inputs to other nodes.

- In HOL, structure is represented by a conjunction of relations, the parameters of which describe how wires connect to the ports of a device. The wires are labelled in this case, not the devices, so the connections involving identical devices are distinguished by the wires they connect. General models for devices can therefore be used and no labelling of devices is necessary.

Three points emerge from the three specifications above:

- Of the two functional models, the ML model is closer to the HOL model. Structure is modelled in a very similar way by labelling internal lines, using them as inputs or outputs according to the appropriate connection, and hiding them from the external definition.

- No superior techniques were evident in the ELLA model which were lacking in the ML model. This suggests that ML is adequate for simulation of logic specifications, and a special purpose simulation language such as ELLA is therefore not necessary.

- Although the representation in ML is closer to HOL, the ELLA model is not too far off from the HOL and ML models. The concept of inter-connecting nodes is similar to connecting devices by labelling wires. This suggests that it could be possible to establish a mapping which translates from logic specifications to specifications of a special purpose simulator, if so required. Of course, the example considered is trivial and further research is necessary to explore the latter point by considering more complex circuits.

A tool for translating HOL specifications to a special purpose simulator like ELLA could enable more effective simulation of circuits via the HOL system. This is not the idea behind this thesis, however, and is an alternative topic suggested for future research. What we are interested in here is the execution of the logic specifications themselves, which can be achieved by translating into an identical executable representation. The ML language is not identical but seems to be extremely close in its representation of hardware, as shown in the earlier sections.

In the next chapters we show how ML can be used effectively for simulation; we show how HOL relational specifications can be automatically transformed to ML

programs; and we suggest in the concluding chapter how the transformation from logic to programs can be done with minimum risk of introducing inconsistencies.

We do not consider ELLA specifications any further. The reasons for choosing to use ML instead of ELLA as a simulation language can be summarised as follows:

- ML happens to be the meta-language of HOL. This makes it more convenient for the automatic translation of HOL specifications to an executable form, as discussed in Chapter 6.

- There is a strong resemblance between specifications written in the HOL logic to ones written in ML. This is important because it helps to minimise the risk of introducing inconsistencies in the specifications during translation.

- The way in which the subset of ML programs required for specifying hardware is executed is analogous to expanding definitions and using β-reduction in the logic. It is easy to see how an ML program is executed and straightforward to check that the execution of the program correctly models the operation of the intended circuit. Perhaps a connection could be found between an operational semantics for this subset of ML, and the inference rules in the logic.

All considered, the ML approach seems a step closer to the logic formalism than the ELLA approach.

Chapter 5

Faster Simulation Techniques

Before moving on to discuss the execution of the HOL logic, there are certain features in the simulation methods described in the previous chapter which will be explained further, showing how they can be modified to improve the performance of simulation.

5.1 Inefficiencies in Basic Method

The application of ML to hardware simulation, outlined in Chapter 4 is quite elegant. Most aspects of register-transfer level descriptions of hardware circuits can be modelled naturally using standard functional programming techniques.

When attempting to simulate large and complex circuits, however, a major problem is encountered: certain simulations are far too slow. While many circuits can be simulated successfully at a reasonable speed, others are totally unpractical.

The problem is tracked down to circuits with models that have the following properties:

- too many recursive calls on history functions (generally resulting from feedback)
- too many repeated computations (generally resulting from fanout)

In general, therefore, the problem arises to varying extents in most sequential devices.

Consider the example shown in Figure 5.1. The device is kept simple but includes features such as feedback and fanout, typical of the inefficient modelling mentioned above.

The device DEV is made up of four components D, E, F and G, and two external ports, an input *in* and an output *out*. The behaviour of the four components can

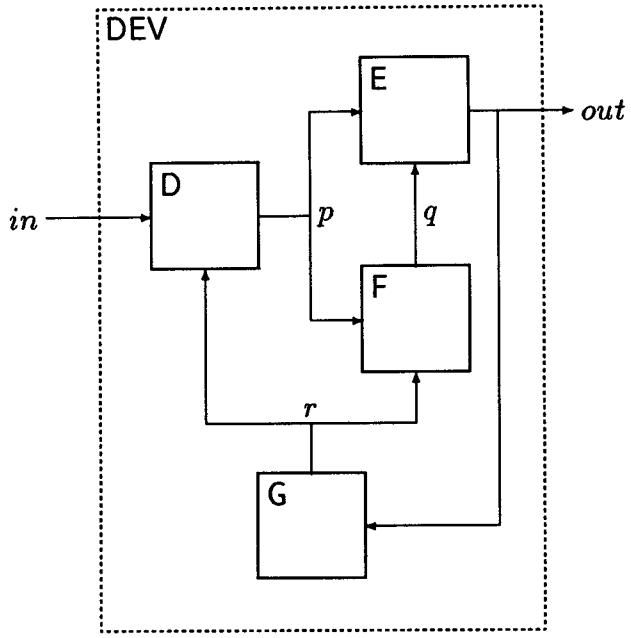


Figure 5.1: Example Device for Illustrating Inefficient Modelling

be specified as shown in the equations 5.1 through 5.4, using the style explained in the previous chapter.

$$\mathsf{let\ D}\ a\ b = \mathsf{let}\ o(t) = d[a(t), b(t)]\ \mathsf{in}\ o \tag{5.1}$$

$$\mathsf{let\ E}\ a\ b\ ival = \mathsf{let}\ o(t) = (t{=}0) \Rightarrow ival \mid e[a(t{-}1), b(t{-}1)]\ \mathsf{in}\ o \tag{5.2}$$

$$\mathsf{let\ F}\ a\ b = \mathsf{let}\ o(t) = f[a(t), b(t)]\ \mathsf{in}\ o \tag{5.3}$$

$$\mathsf{let\ G}\ a = \mathsf{let}\ o(t) = g[a(t)]\ \mathsf{in}\ o \tag{5.4}$$

In the equations above, $d[a(t), b(t)]$ and $f[a(t), b(t)]$ are some expressions involving occurrences of $a(t)$ and $b(t)$, $e[a(t{-}1), b(t{-}1)]$ is some expression involving $a(t{-}1)$ and $b(t{-}1)$, and $g[a(t)]$ is an expression involving $a(t)$.

The inputs and outputs of the above devices are modelled by history functions. The devices D, F and G have no delay; the output at a time t is a function of the inputs at time t. The device E, however, has a unit delay and the output at a time t is a function of the inputs at time $t{-}1$ unless $t{=}0$ in which case the output is in its initial state, parameterised by *ival*.

The structure of DEV can therefore be specified by the program below. A recursive declaration is used to compute the values on the output *out*, and internal

lines of the device, p, q and r. The initial state of the device is parameterised by *ival* and the recursion halts at time t=0.

$$
\begin{array}{l}
\mathsf{let}\ \mathsf{DEV}\ in\ ival = \\
\quad \mathsf{letrec}\ p\,t = (\mathsf{D}\ in\ r)\,t \\
\quad \mathsf{and}\ q\,t = (\mathsf{F}\ p\ r)\,t \\
\quad \mathsf{and}\ out\,t = (\mathsf{E}\ p\ q\ ival)\,t \\
\quad \mathsf{and}\ r\,t = (\mathsf{G}\ out)\,t \\
\quad \mathsf{in}\ out
\end{array}
$$

The inefficiency in the execution of the model is not immediately evident. If an attempt is made to investigate what is involved in evaluating various values on the output line, however, the problem becomes obvious.

For a given history function *in* and value *ival*, the expression

$$\mathsf{let}\ out = \mathsf{DEV}\ in\ ival$$

returns a history function *out*. In order to evaluate the value on *out* at a particular time, 5 say, the following computation is involved.

- From the definition of DEV:

$$
\begin{array}{lll}
out(5) = (\mathsf{E}\ p\ q\ ival)\,5 & & \\
\quad = (\lambda t.\ (t{=}0) \Rightarrow ival \mid e[p(t{-}1), q(t{-}1)])\,5 & & \ldots \text{from 5.2} \\
\quad = e[p(4), q(4)] & & \ldots \text{evaluation}
\end{array}
$$

  Hence, the values of $p(4)$ and $q(4)$ are required to compute $out(5)$.

- From the definition of DEV:

$$
\begin{array}{lll}
q(4) = (\mathsf{F}\ p\ r)\,4 & & \\
\quad = (\lambda t.\ f[p(t), r(t)])\,4 & & \ldots \text{from 5.3} \\
\quad = f[p(4), r(4)] & & \ldots \text{evaluation}
\end{array}
$$

  The value of $q(4)$, therefore, requires that of $p(4)$ and $r(4)$.

- From the definition of DEV:

$$
\begin{array}{lll}
p(4) = (\mathsf{D}\ in\ r)\,4 & & \\
\quad = (\lambda t.\ d[in(t), r(t)])\,4 & & \ldots \text{from 5.1} \\
\quad = d[in(4), r(4)] & & \ldots \text{evaluation}
\end{array}
$$

The value of $in(4)$ is known since it is an input value. The computation of $p(4)$, therefore, requires the value of $r(4)$.

- From the definition of DEV once again:

$$\begin{aligned} r(4) &= (\mathsf{G}\ out)\,4 \\ &= (\lambda t.\ g[out(t)])\,4 && \ldots \text{from 5.4} \\ &= g[out(4)] && \ldots \text{evaluation} \end{aligned}$$

Hence, all the values required for the calculation of $out(5)$, require the value of $out(4)$. This completes one cycle of the recursion and further recursive calls are performed until $out(0)$ terminates the recursion, allowing the value of $out(5)$ to be calculated.

A summary of the computation involved for one recursive call in the calculation of $out(5)$ is shown in Figure 5.2. The inefficiencies mentioned earlier are now

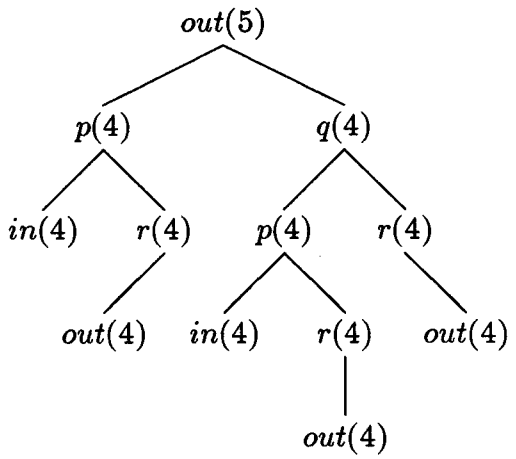


Figure 5.2: Tree Illustration of Inefficient Evaluation

obvious. The computation of $out(5)$ involves that of $p(4)$ and $q(4)$. After the value of $p(4)$ is evaluated, recursively from time 0, the process must be repeated in order to evaluate $q(4)$ since $q(4)$ also requires the value of $p(4)$. This kind of repetition is also involved in the calculation of $r(4)$, required by $p(4)$ and $q(4)$, and in the calculation of $out(4)$, required every time $r(4)$ is calculated.

In fact, in order to compute $out(5)$, one needs to compute $p(4)$ twice, $r(4)$ three times, $q(4)$ once, and $out(4)$ three times. Now each time a value is computed at time t, the computation is recursive all the way to time 0. So if the value of t is large (say 100 instead of 5) and many more internal lines are involved making

the recursive definition more complex, then the problem of recalculating several values recursively from time 0 is escalated and the effect is disastrous.

Worse still, once the value of $out(5)$ is successfully computed, it is not used in further computations of *out* at time values ≥ 5. If the value of $out(6)$ say, is required after the evaluation of $out(5)$, then $out(5)$ is still recalculated whenever required for the computation of $out(6)$.

The nature of the problem suggests the solution. For every value required in a computation, either

- it is evaluated and stored for possible later use, or
- evaluation is delayed until the last possible moment when the value is evaluated once and assigned to all instances where the value is required in the course of the computation.

Both optimisations are standard techniques and are presented in the following sections. The former technique is called *memoisation* while the latter is called *lazy evaluation.*

5.2 Memoisation

The first technique presented as an optimising solution to the inefficiency problems encountered in the simulation of circuits in ML, is called *memoisation.* The technique of memoisation was originally invented by Donald Michie and was used as an aid for improving performance of programs as well as an 'artificial learning mechanism' [41]. A more recent and detailed description of the memoisation algorithm (also called *tabulation*) and its implementation in LISP is found in [1].

5.2.1 The Algorithm

Memoisation can make a vast difference in the performance of a program. The idea is to define higher order functions called memo-functions which take an inefficient function as a parameter and return an optimised function.

A memo-function remembers all arguments it is applied to as well as the results computed from them by maintaining a table in which values of previous calls to the function are stored using the arguments that produced the values as keys.

When a memoised function is applied to a set of arguments to compute a value, it first checks the table to see if the function has already been applied to these arguments, and if so, it merely returns the previously computed value stored in

the table. If the function has not already been applied to the arguments before, however, then the new value is computed in the ordinary way, stored in the table with the function's arguments as keys, and returned as the result of the function.

The *Fibonacci* function is a classic example of the optimising effect memoisation can have upon inefficient algorithms. The function computes the n^{th} Fibonacci number and can be written in ML as follows:

```
letrec fib n =
        (n=0) ⇒ 0 |
        (n=1) ⇒ 1 |
        fib(n−1) + fib(n−2)
```

The function is recursive in n and each call of fib generates two recursive calls with smaller arguments. One way of writing the memoised version of this function is:

```
letrec memo_fib =
        memoise (λn. (n=0) ⇒ 0 |
                     (n=1) ⇒ 1 |
                     memo_fib(n−1) + memo_fib(n−2))
```

where *memoise* is the memoisation function which will be defined in the next section.

Consider now, the application $memo\_fib(3)$ in order to analyse the computation involved when using memoised functions.

1. The first call of the function is $memo\_fib(3)$. The table is checked, but no entry is found under key 3.

2. So the function is invoked. The conditional tests 'is $(n=0)$?' and 'is $(n=1)$?' are both false, hence the default $memo\_fib(n-1) + memo\_fib(n-2)$ is calculated.

3. The computation resumes with $memo\_fib(2)$. Checking the table is once again unsuccessful, resulting in the function being invoked with $n=2$ which requires $memo\_fib(1) + memo\_fib(0)$.

4. The function call $memo\_fib(1)$ fails once again to find an entry in the table but, the function this time evaluates to a result since the condition 'is $(n=1)$?' is true yielding an answer 1, terminating one recursion branch. The answer is now stored in the table under key 1 and the computation continues.

5. The remaining call to *memo_fib* from step 3, *memo_fib*(0), results in the value 0 in a similar way to the computation process in step 4. The value for n=0 is also stored in the table, and the computation of step 3 is ended by evaluating 1+0 which equals 1—the answer of *memo_fib*(2) (called from step 2) which is also stored in the table.

6. Back to step 2, finally, the value of *memo_fib*(1) is still required. This time, at last, the result is already stored in the table and is retrieved without recomputation. The result of *memo_fib*(3), therefore, is evaluated by the expression 1+1 resulting 2. This value also is stored, resulting in a table with Fibonacci numbers for $0 \leq n \leq 3$ after computing the value for *memo_fib*(3).

If, now, the next call to *memo_fib* is *memo_fib*(4), then the table lookup fails for n=4, the condition 'is (n=0)?' and 'is (n=1)?' are both false, but the recursive calls *memo_fib*(3) and *memo_fib*(2) both have their answers stored in the table (as a result of the previous computation), these being 2 and 1 respectively, and so the result of *memo_fib*(4) is evaluated without further calls to *memo_fib* and stored in the table as well.

It is obvious, therefore, that it is not enough to optimise the Fibonacci function by applying a memoisation function to the original function *fib* as in:

let $memo\_fib' = memoise\ fib$

since this allows *fib* to operate inefficiently, and only to store the result for the top-level parameter of the function call. Thus, with the above incorrect optimisation of *memo_fib'*, the computation *memo_fib'*(3) only memoises the result for argument 3, not the results for all arguments 0–3. The optimisation must be done at every recursive call to the Fibonacci function.

Three points stand out from the function *memo_fib*:

- The cost of computing the n^{th} Fibonacci number is reduced from exponential in n to linear in n, since *memo_fib*(n) is computed only once for each value of n. Some overheads such as the look-up time for accessing tables are involved, but these are negligible for efficiently implemented data structures and functions to access them.

- Subsequent requests for the n^{th} Fibonacci number are not recomputed and the value is returned immediately. In general, of course, this is the major advantage since recomputation does waste time.

- The structure of the algorithm for calculating the Fibonacci numbers (as viewed in the definition of fib) remains unaltered after memoisation. This is important because the optimisation of functional specifications can be kept clean—i.e. the structural representation of a circuit is not modified for the sake of performance, introducing the danger of incorrect modelling.

5.2.2 Implementation

A natural way to implement memo-functions in ML is to use *association lists* as tables. An association list is a list of pairs such that every element of the list has the form (a, b), where a is the key and b is the value stored associated with that key.

A function *make_table* can be defined to represent an empty list of pairs.

$$\textsf{let } make\_table = ([\,] : (\alpha \# \beta)\ list)$$

The type variable α will be instantiated by the type of the key and β will be instantiated by the type of the stored values. Whenever a new table is required, therefore, one makes declarations specifying the types of the pairs intended to be stored in that table. For example:

$$\textsf{letref } table1 = make\_table : (int \# int)\ list$$

declares a new table, *table*1, which stores pairs of integers, while

$$\textsf{letref } table2 = make\_table : (int \# bool)\ list$$

declares a new table, *table*2, in which the keys stored will be integers but their associated values will be booleans. Obviously, values of different types cannot be stored in the same list.

Once a data type for storing values is defined, we can define functions to manipulate this data type. In fact, only two functions are required for the purpose of memoisation.

- *lookup*—a function which checks whether there is an entry in a table for a particular key, and if so, returns the associated value:

  $$\textsf{let } lookup\ key\ table = snd\,(assoc\ key\ table)$$

  where *assoc* is a function which returns the first pair in a table, (a, b), with a matching the *key* required. If no entry for a key is found in the table, then *assoc* (and hence *lookup*) fails. Otherwise, *lookup* returns the second element of the pair returned by *assoc*.

- *insert*—a function to add on a new pair of values to the beginning of a list:

```
let insert key value table = (key, value).table
```

A memoisation function *memoise* (such as that used in the previous section to optimise the function for computing Fibonacci numbers), can now be defined using *lookup* and *insert*.

```
let memoise f =
    λx. (lookup x table)?
        (let result = f(x)
         in
         (table := (insert x result table);
          result))
```

The function *memoise* takes a function f as an argument and returns another function which, when applied to a value x, the value of $(lookup\, x\, table)$ is returned or, if that fails, the result of $f(x)$ is returned after being stored in the table.

Notice that *table* would have to be globally declared in an ML session and is not a parameter to the *memoise* function. This is because ML does not allow the operation := to operate on parameters of functions. It is of no use either, to declare the table locally in the *memoise* function as that way, a new table will be declared on every call to *memoise* (i.e. *table* is reset to [] each time).

5.2.3 Memoisation of the Counter

As an example of the use of memoisation in the execution of hardware specifications, consider once again the implementation of the counter described in Section 4.4 and illustrated in Figure 4.2.

To recap, the specification of the counter, COUNT, was:

```
let COUNT in cntl countval =
    letrec p t = (MUX in q cntl) t
    and out t = (REG p countval) t
    and q t = (INC out) t
    in out
```

The functions in this definition that need to be memoised are the history functions p, q and *out* which represent the outputs of MUX, INC and REG respectively. Now this would require three different tables in order to store values of p, q and *out* distinctly. Also, due to the way in which tables have to be globally declared, a memoisation function would have to be defined for each table since the use of

memo-functions is restricted to one table only and so, three memo-functions also have to be defined.

It does seem rather superfluous to have to declare a table and a memoisation function for every function to be memoised, especially if there are a large number of functions.

An improvement on the representation of tables suggested in the previous section, therefore, is to use one table to store all values of functions of the same type. This is achieved by using another key in the association list which represents a name that uniquely identifies which functions denote which values. This new form of table, therefore, is represented by an association list in which the keys are names denoting different functions and the associated values are themselves association lists representing the history of the named functions.

A function *make_table'* is defined in a similar way to *make_table*, this time represented by a list of lists:

$$\textsf{let } make\_table' = ([\,] : (\gamma \# (\alpha \# \beta)\, list)\, list)$$

In the example of COUNT, all functions to be memoised (p, q and out) are of type $int \rightarrow int$ (deduced from the definitions 4.1, 4.2 and 4.3 in Section 4.4). It is only necessary to declare one table, therefore, instead of the three required if the previous method were used:

$$\textsf{letref } count\_table = make\_table' : (string \# (int \# int)\, list)\, list$$

where the type *string* is an ML type for strings of characters which will be used to name the association lists.

The functions *lookup'*, *insert'* and *memoise'* are defined in a similar way to *lookup*, *insert* and *memoise*—they are only extensions to support the named association lists.

```
let lookup' name key table =
    snd(assoc key (snd(assoc name table)))

let insert' name key value table =
    (let (subtable, rest) = assoc_split name table
     in
     (name, ((key, value).(snd subtable))) . rest)
    ? ((name, [(key, value)]).table)
```

```
let memoise' name f =
    λx. (lookup' name x count_table) ?
        (let result = f(x)
         in
         (count_table := (insert' name x result count_table);
          result))
```

where *assoc_split* returns a pair whose elements are the table entry returned by *assoc* and the remainder of the table after removing that entry.

The memoised definition of COUNT, therefore, is written as:

```
let COUNT in cntl countval =
    letrec p t = memoise' 'p' (MUX in q cntl) t
    and out t = memoise' 'out' (REG p countval) t
    and q t = memoise' 'q' (INC out) t
    in out
```

The changes done to the definition do not upset the modularity in the structure of the definition.

Both the inefficient and the memoised models of COUNT were simulated for various test data, each time computing the values on the output function for 126 time cycles. The average run time taken for the inefficient definition was 84*secs*; the memoised version took only 38*secs*. The improvement in the performance of the memoised definition is evident even for this simple example. The improvement in performance can be much greater for larger and more complicated examples which can be impossible to simulate without optimisation, as shown in Chapters 7–9.

5.3 Lazy Evaluation

The second optimising solution presented in this thesis is the use of *lazy evaluation* [54]. Lazy evaluation is a method of computation where the evaluation of arguments to user-defined functions is delayed until their value is absolutely necessary. The required arguments are then evaluated once and assigned to all instances where they are used in the body of the function.

For example, the function f defined below takes two arguments a and b and computes the value $b+b$ if a is non-negative; returns the value of a otherwise.

$$f(a,b) = (a \geq 0) \Rightarrow b+b \mid a$$

If the function f is applied to two arguments, $g(x)$ and $h(x)$ say:

$$f(g(x), h(x))$$

then two kinds of optimisations are possible if the application is evaluated lazily.

- First, the value of parameter $h(x)$ is not evaluated in the event of $g(x)$ having a negative value since it is not involved in the result returned by f.
- Secondly, if the value of $g(x)$ is positive, the value of parameter $h(x)$ is evaluated only once in the computation of $h(x)+h(x)$, instead of twice, as would be the case with normal order evaluation.

Thus, lazy evaluation can improve the performance of certain programs by avoiding unnecessary computations of parameters.

The idea of using a lazy functional programming language for hardware simulation is not new. For example, Stephen Hill describes the application of the functional language MIRANDA [62] to simulation in [29]; his techniques rely heavily on the lazy nature of the language.

In this section we describe the techniques used to simulate hardware using LAZY ML—LML for short—a strongly typed, lazy, purely functional language developed by Lennart Augustsson and Thomas Johnsson at Chalmers University of Technology, Göteberg. No attempt is made here to describe the syntax or semantics of LML; for this see [2]. This section is merely intended to give a general description of the techniques used for simulation, comparing them to those used with ordinary ML.

5.3.1 General Simulation Principles

The use of LML to simulate combinational circuits is identical to that of ordinary ML:

- values on ports can be represented by the booleans *true* and *false*,
- logic gates can be represented using the boolean operators, and
- behaviour of devices can be represented using functions.

In fact, the syntax of the two languages are very similar and the same specification techniques used in ML to represent hiding, structure, etc. are used in LML. For this reason we do not go through explaining the role of LML in such cases.

The two languages do, however, differ in the approach needed to simulate sequential circuits (or any circuit in which time is taken into account). The difference lies in the data type chosen to represent signals. While it is perfectly possible to

use history functions to model signals in LML, they do not exploit the lazy nature of the language, and do not make any improvements over the performance of the ML programs.

The reason is that laziness only delays the evaluation of parameter identifiers, not of function applications in the body of a definition. For example, in the application:

$$(\lambda x.\, t[x,x])\, f(n)$$

where $t[x,x]$ represents some term t involving two free occurrences of x, the application $f(n)$ of some function f to an argument n is only computed once and substituted for each occurrence of x in $t[x,x]$. Laziness here successfully reduces the number of times $f(n)$ has to be computed.

In the application:

$$(\lambda x.\, t[g(x), g(x)])\, f(n)$$

however, although $f(n)$ is only computed once, the application of g to the value of $f(n)$ is still computed twice. Laziness does not reduce the number of times $g(x)$ is computed because it does not remember results of previous computations—it does not perform memoisation.

Likewise, in the recursive function

$$f(n) = t[f(n-1), f(n-2)]$$

results of applying the function to certain arguments are not remembered when the function is next applied to the same arguments. Thus, if $f(5)$ involves the computation of $f(4)$ and $f(3)$, the result of $f(3)$ computed in order to evaluate $f(4)$ is not re-used when next required in the computation of $f(5)$, but $f(3)$ has to be recomputed.

Laziness, therefore, only avoids superfluous evaluation of parameters. For this reason the lazy approach to the evaluation of history functions does not improve performance because it does not avoid the unnecessary recomputation of results of previous function applications.

In LML, therefore, signals are best considered as streams of bits represented by lists of booleans. The advantage of this form of representation is that the lists are infinite objects. In a lazy language, infinite data structures are allowed since they are not evaluated when defined.

The method of using infinite lists to represent signals is based on the ideas proposed by Stephen Hill in [29]. The approach to modelling hardware presented

in this thesis, however, differs from that of Hill's in several ways. For example, our specifications are based on a two-valued logic (instead of three-valued), and initial values of signals are parameterised (instead of being set to a value denoting an undefined state). A brief description of the way infinite lists are used to model hardware in LML is given below.

In LML, lists can be defined to be infinite as long as only finite portions of them are evaluated. For example, a clock line which is always ON can be defined as:

$$\mathsf{ON\_CLOCK} = ON\ .\ \mathsf{ON\_CLOCK}$$

and a clock line which toggles ON and OFF consecutively can be defined as:

$$\mathsf{CLOCK} = [ON; OFF]\ @\ \mathsf{CLOCK}$$

where . and @ are the infix 'cons' and 'append' operators.

If the first six values of CLOCK are required, say, then the built-in function *head* is used as follows:

$$head\ 6\ \mathsf{CLOCK}$$

which only evaluates the first six values on the clock:

$$[ON; OFF; ON; OFF; ON; OFF]$$

These lists can therefore be viewed as sequences of values sampled at discrete intervals of time. Two consecutive values a and b in a list $[\ldots; a; b; \ldots]$ can be seen as though a is a value at time t and b is the next value at time $t+1$.

The *map* function obviously plays a big part in writing specifications using lists. For example, if *Not* is the negation operator then an inverter INV can be defined as:

$$\mathsf{INV} = map\ Not$$

so that if i is the list

$$[true; true; false; true; false; \ldots]$$

then (INV i) returns the list

$$[false; false; true; false; true; \ldots]$$

Delays and feedback are easy to model. A delay of one time unit is effected on a list by putting an arbitrary value on the front of the list, which has the effect of 'shifting' all values in the list one place to the right. So if a list has values

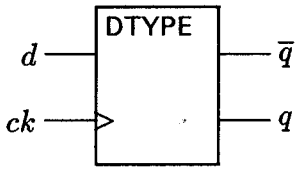
$$[a; b; c; d; \ldots]$$

a delay of one time unit is introduced to give the list:

$$[del\_val; a; b; c; d; \ldots]$$

If a is the value at time 0 in the original list, then in the second list it is the value at time 1. The delay value, *del_val*, can be parameterised in the specifications of circuits in the same way initial values were parameterised when using ML.

For example, consider the following register-transfer level description of a D-type flipflop to illustrate some of the above principles.



The D-type flip-flop shown above takes two inputs: a data line d and a clock ck, and returns two outputs: a data line q and its compliment $\overline{q}$. Informally, the specification of DTYPE is:

> If the clock at time t is high, then the value on q at time t+1 is set to the value of d at time t. Otherwise, if the clock is low at a time t, then the value on q at time t+1 is set to the previous value on q at time t. The value on $\overline{q}$ is the complement of that on q at all times.

The formal definition of DTYPE, therefore, can be written in LML using infinite lists as follows:

```
DTYPE d ck qval =
        let rec q =
            qval . (map3 (λa. λb. λc. (a = ON) ⇒ b | c) ck d q)
        and qbar = map Not q
        in
        (q, qbar)
```

where *map*3 is defined as:

$$map3\ f\ a\ b\ c = (f\ (hd\ a)\ (hd\ b)\ (hd\ c)).(map3\ f\ (tl\ a)\ (tl\ b)\ (tl\ c))$$

The value of q at time 0 is set to *qval* by putting *qval* at the start of the list. Subsequent values of q are then computed using the previous value of q in the list as well as the values of *ck* and d corresponding to a time previous to that in which q is computed.

The length of the delay, in fact, can be arbitrary since a delay of n units long can be represented by adding n delay values at the start of the list.

5.3.2 The Counter using Lazy Evaluation

For a slightly larger example, consider the simple counter presented in Section 4.4. The specification of the COUNTER device shown in Figure 4.1 can be written as:

```
COUNTER in cntl countval =
    let rec out =
        countval . (map3 (λa. λb. λc. a ⇒ b | (c+1)) cntl in out)
    in out
```

The next value of *out* in the list (say the value at position $n+1$) is equal to the increment of the last value in the list (the one at position n) if the value of *cntl* at position n is false; otherwise it is equal to the value of *in* at position n.

The implementation of COUNT shown in Figure 4.2, on the other hand, can be defined by:

```
COUNT in cntl countval =
    let rec p = MUX in q cntl
    and out = REG p countval
    and q = INC out
    in out
```

where the behaviours of the components MUX, REG and INC are specified as:

```
MUX = map3 (λa. λb. λc. a ⇒ b | c)

REG in initval = initval . in

INC = map (λa. a+1)
```

The definition of COUNT above is very similar to the one in ordinary ML. The only difference is that the recursion is over lists instead of history functions. Otherwise, the structural representation of the device is modelled in exactly the same style. The behavioural definitions of MUX, REG and INC, however, are very different to the ones in ML. The LML definitions are more compact, but this makes them slightly harder to understand.

The approach, therefore, has not been to optimise existing ML definitions (as with memoisation), but rather to choose a different data representation which can be efficiently manipulated using a lazy language. This of course leads to a different style of writing specifications.

5.4 Memo-functions or Infinite Lists

The LML definitions are simple though their way of expressing timing properties does not seem as natural as that of history functions. With history functions, time is expressed relative to a reference point t, and the instant of time at which a value is calculated in relation to t is immediately obvious from the definition. With lists, however, the definition has to be examined carefully before it becomes obvious as to how the signals are timed.

The advantage of using the lazy approach and infinite list representations is that the definitions are efficient to execute without requiring further optimisation. With the strict approach of ML, however, optimisation is essential if the execution of the definitions is to be practical.

In this thesis we do not investigate any further the role of lazy evaluation. We shall instead stick to the ordinary ML approach and show in the next section how the optimisations can be automated, making this approach more practical. The choice of using ML as opposed to LML as an execution language for the HOL logic (as described in the next chapter) is mainly due to the expressiveness of history functions and the strong resemblance between specifications written in the HOL logic to ones written in ML. The fact that the meta-language of HOL happens to be ML and not LML also makes it more practical to translate the logic into ML.

Chapter 6

Executing the HOL Logic

In the previous two chapters we have shown how ML can be used as a hardware simulator at the register-transfer level. The goal now is to show how these ideas can be applied to the execution of specifications written in the HOL logic.

The conventional approach to the hardware design process has been to use simulators and other informal design tools to demonstrate the correctness of circuits. If formal methods and verification are at all considered, the circuit often ends up being formally specified and verified in a completely different notation to that used by the simulators. The processes of simulation and verification, therefore, have been kept apart and the benefits of one not used to aid the other.

Their view as two separate issues is not cost effective since doing both separately is expensive. Furthermore, because of the many different notations used, one is never sure whether the simulation definitions and formal specifications model precisely the same circuit. It is desirable to be able to use the same specifications for both simulation and verification, and the necessity of executing specifications which form part of a formal proof system is increasing as chip design becomes more complex.

One example in chip design where several notations were used to attempt to demonstrate correctness is the design of the VIPER chip manufactured by the RSRE [13]. The chip was specified in LCF-LSM but the lack of simulation facilities in the proof system led to translating the specifications into ELLA [48], ML and ALGOL 68. With so many notations and languages involved, errors can be introduced into the models by inaccurate translation between the different notations. Aspects of the VIPER chip have now been specified and verified in HOL by Avra Cohn [11], but once again no simulation facilities to aid with the verification were available in the proof system.

With the facility of executing the HOL logic specifications, the processes of simulation and verification can be combined to demonstrate correctness. The danger of introducing inconsistencies in the models is also reduced when translating the HOL logic to executable code if:

- only a subset of the HOL logic which has a clean and clear representation in ML is used,
- the style of writing specifications is restricted to one in which the specifications can be expressed both in the logic and the meta-language in an almost identical manner,
- the executable code is automatically generated from the HOL logic, thus reducing the chance of manual errors in translation.

It has already been shown in Chapters 4 and 5 that ML is suitable as a simulation language at the register-transfer level and that a subset of it can be adequately used to write specifications in a style which is almost identical to functional specifications written using a subset of the HOL logic. In this chapter we describe in detail that subset of the HOL logic and the style of representation we must adhere to. We show that despite these restrictions in style and notation, most features in register-transfer level descriptions of hardware can be modelled, and we outline how the specifications can be automatically translated into executable code.

In Chapter 3 we showed that specifications in the HOL logic can be written as relations or functions. A program for translating HOL terms into executable code, therefore, must cope with both styles of representation. The algorithm for automatically defining ML programs corresponding to given suitable HOL specifications is discussed, therefore, showing how relations can be first automatically translated into functions which can then be mapped into ML programs.

6.1 From Relations to Functions

The first step towards executing specifications in the HOL logic is to derive functional definitions for them. This is done by either:

- writing functional specifications from the start, or
- writing relational specifications and later translating them into functional ones.

The second alternative may seem superfluous but is in fact very useful. There are two main reasons why the translation of relations into functions can be required: one may find it more natural to express properties as relations, or one may want to simulate circuits already specified using relations. It would be rather time consuming to have to manually translate existing relational specifications into a form in which they can be executed.

Of course, not all relations have a functional interpretation and, for those that do, it is not always possible to automatically translate them into functions. Furthermore, not all functions can be executed so the relations which we want to automatically translate to functions for the purpose of simulation are restricted to a subset of the HOL logic which can be interpreted as executable functions.

Relations written in the HOL logic can be divided into two sets: the set of relations which can be interpreted as functions and the set of those which have no functional interpretation. Figure 6.1 illustrates this classification of relations in the HOL logic and shows how they can be divided into further subsets.

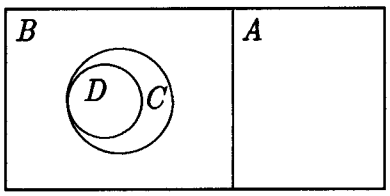


Figure 6.1: Relational Expressions in the HOL logic

In Figure 6.1:

- A represents the set of those relations which do not have a functional interpretation.
- B represents the set of all relations which can be interpreted as functions.
- C represents the subset of B which contains those relations which can be interpreted as executable functions.
- D represents a subset of C which contains relations which can be interpreted as executable functions and are commonly used for specifying implementations of hardware circuits.

The boundary enclosing set C in the above diagram is difficult to define and is subject to interpretation. The purpose of the diagram is not to present a precise

classification of the infinite number of relations that can be written in the HOL logic, but to explain which set of relations we are interested in translating (set D) and how that fits in with the other relations we do not translate.

Examples of relations which are used for specifying hardware but which are classified in sets A or B (i.e. either they do not have a functional translation or they have a non-executable functional translation) are presented in Section 6.3. In the next section we describe those relations classified in set D—those relations which we automatically translate into functions.

6.2 Automatically Translating Relations

The main syntactic difference between relational and functional specifications was explained in Chapter 3:

- In a relational model there is no distinction between inputs and outputs—all ports are parameterised.
- In a functional model, only the inputs are parameterised from which the function computes the values on the output ports.

The translation from relations to functions, therefore, can only be automated if information is given as to which parameters of the predicate are to be inputs of the function and what form the output should take. An ML function

$$\mathsf{Rel\_to\_Fun} \ : \ term \# term\ list \# term \rightarrow term$$

can be defined to translate relations to functions by:

$$\mathsf{Rel\_to\_Fun}\ (relation, input\_list, output) = function$$

The form of relations presented below can be automatically translated into functions using the function Rel_to_Fun, written in ML. Important algorithms used in the definition of Rel_to_Fun are explained informally throughout this section by showing how various relations translate into functions. The form of relations which can be translated by Rel_to_Fun is based on the style used for specifying hardware implementations, and in Chapters 7–9 a number of examples are presented to illustrate this. Of course the relations which Rel_to_Fun can translate do not need to describe hardware devices. We merely present hardware oriented examples since our theme of executing definitions is aimed as a simulation aid for hardware verification. A full grammar describing the relational definitions which can be automatically translated by Rel_to_Fun is given in Appendix A.

6.2.1 Translating Predicates

A relation

$$\mathsf{PRED}(x_1, x_2, x_3, x_4)$$

translates to

$$\mathsf{let}\ x_3 = \mathsf{FUN}\ x_1\ x_2\ x_4$$

by evaluating the application

$$\mathsf{Rel\_to\_Fun}\ \text{“}\mathsf{PRED}(x_1, x_2, x_3, x_4)\text{”}\ [\text{“}x_1\text{”}; \text{“}x_2\text{”}; \text{“}x_4\text{”}]\ \text{“}x_3\text{”}$$

To avoid restating the application of Rel_to_Fun to its arguments for each example, to indicate which parameters are required as inputs and which should form the output, we will use the identifier *inlist* to denote the list of input variables, and *output* to denote the required output. In the above example, therefore, one would refer to [“x_1”; “x_2”; “x_4”] as *inlist* and “x_3” as *output*. Furthermore, the notation PRED will be used to denote some predicate and FUN to denote its translation for a given *inlist* and *output*. Finally, the double quotes in the above expression are used to identify HOL terms. Using quotes whenever a HOL term is written, however, tends to clutter the text and so, they will only be used when it becomes necessary to distinguish HOL terms from ML expressions. Otherwise, all relations and functions described in this chapter should be taken to be HOL terms.

The way in which the parameters of a relation are ordered before translation is unimportant but the order is preserved for those parameters which represent inputs when generating the parameters of the functions. So, in the above example,

$$\mathsf{PRED}(x_4, x_3, x_2, x_1)$$

would have translated into

$$\mathsf{let}\ x_3 = \mathsf{FUN}\ x_4\ x_2\ x_1$$

A function must always have an output but the number of input variables is arbitrary. For example, a relation can model a device with no inputs which always generates a constant output. The relation

$$\mathsf{PRED}\ x$$

is translated to

$$\mathsf{let}\ x = \mathsf{FUN}$$

where FUN is a function with no arguments (called a 0-*ary function* or a constant). A function, of course, can only return a single output but this can represent several values by returning the output values in tuple form.

6.2.2 Simple Relational Definitions

In general, a relational definition takes the following form of an equivalence:

$$\mathsf{PRED}(x_1, \ldots, x_n) \equiv t[x_1, \ldots, x_n]$$

where $t[x_1, \ldots, x_n]$ is some boolean term t involving free occurrences of the variables $x_1, \ldots, x_n$. The left hand side represents the predicate being defined and the right hand side describes the properties which define the predicate.

A new function name FUN_j is generated to define a function corresponding to a relational definition with a name PRED_j. Also, once a function is generated, its name must be stored with the name of the corresponding relational definition so that whenever the predicate is used in other definitions, the same function is recalled each time.

The simplest of these definitions are relations of the form:

$$\mathsf{PRED}_1(i_1, \ldots, i_m, o_1, \ldots, o_n) \equiv \mathsf{PRED}_2(i_1, \ldots, i_m, o_1, \ldots, o_n)$$

which translate to:

$$\mathsf{let}\ \mathsf{FUN}_1(i_1, \ldots, i_m) = \mathsf{let}\ (o_1, \ldots, o_n) = \mathsf{FUN}_2(i_1, \ldots, i_m)\ \mathsf{in}\ (o_1, \ldots, o_n)$$

given that *inlist* contains $i_1, \ldots, i_m$ and $output = (o_1, \ldots, o_n)$. Of course, the above definition can be simplified to:

$$\mathsf{let}\ \mathsf{FUN}_1(i_1, \ldots, i_m) = \mathsf{FUN}_2(i_1, \ldots, i_m)$$

or even:

$$\mathsf{let}\ \mathsf{FUN}_1 = \mathsf{FUN}_2$$

The first definition, however, is easier to translate to when using a general algorithm. Apart from the reason of facilitating automatic translation, all relations are automatically translated to functions in this style for reasons that were explained in Chapter 4.

The right hand side of a definition can be a conjunction of predicates instead of just one predicate as above. One common use of conjunction is in the representation of structure in hardware circuits. For example, the relation

$$\begin{aligned}
&\mathsf{PRED}(i_1, \ldots, i_m, o_1, \ldots, o_n) \equiv \\
&\qquad \mathsf{PRED}_1(i_1, \ldots, i_m, o_1) \wedge \\
&\qquad \mathsf{PRED}_2(i_1, \ldots, i_m, o_2) \wedge \\
&\qquad \vdots \\
&\qquad \mathsf{PRED}_n(i_1, \ldots, i_m, o_n)
\end{aligned}$$

is translated to

$$
\begin{array}{l}
\mathsf{let}\ \mathsf{FUN}(i_1, \ldots, i_m) = \\
\quad \mathsf{let}\ o_1 = \mathsf{FUN}_1(i_1, \ldots, i_m) \\
\quad \mathsf{and}\ o_2 = \mathsf{FUN}_2(i_1, \ldots, i_m) \\
\quad \vdots \\
\quad \mathsf{and}\ o_n = \mathsf{FUN}_n(i_1, \ldots, i_m) \\
\quad \mathsf{in}\ (o_1, \ldots, o_n)
\end{array}
$$

All parameters to the functions above are inputs with known values so all the output values are computed simultaneously. When internal lines are involved, however, it is necessary to evaluate the outputs in the right order. In the general case, an existential quantifier on the right hand side of the equivalence binds the variables denoting internal lines. These lines act as outputs to some devices and inputs to others so it becomes necessary to know when an identifier denoting an internal line represents an input or an output within a set of parameters.

This is achieved by keeping an input-output specification of the parameters of every defined predicate such that later reference to the predicate will demand the same ordering in its parameters. This information can be stored in the form of a table with the predicate names as keys and parameter models as associated values. A parameter model takes the form of a list of values 1 and 0 such that, for every identifier denoting an input, a 0 is stored and, for every identifier denoting an output, a 1 is stored. For example, if $inlist = [x_1; x_2]$ and $output = x_3$, then

$$\mathsf{PRED}_1(x_1, x_2, x_3)$$

would generate a model

$$[0; 0; 1]$$

Hence, this model can be consulted in later uses of PRED_1 to determine whether internal lines represent inputs or outputs. For example, the relational definition

$$\mathsf{PRED}(i_1, i_2, i_3, o) \equiv \exists p.\ \mathsf{PRED}_1(i_1, p, o) \wedge \mathsf{PRED}_1(i_2, i_3, p)$$

makes use of PRED_1 twice. In the first instance, it can be deduced from the input=output specification that the parameters representing inputs are i_1 and p, and the parameter representing an output is o. Since the value of p is unknown, o cannot be computed yet and so, the second instance of PRED_1 is considered. Here, the inputs are i_2 and i_3 (both known values) so p can be computed by applying the function corresponding to PRED_1 to i_2 and i_3. Now the value of p is known and so the value of o can be computed from the inputs i_1 and p. The relational definition, therefore, translates to the function shown below.

$$
\begin{array}{l}
\mathsf{let}\ \mathsf{FUN}(i_1, i_2, i_3) = \\
\quad \mathsf{let}\ p = \mathsf{FUN}_1(i_2, i_3) \\
\quad \mathsf{in} \\
\quad \mathsf{let}\ o = \mathsf{FUN}_1(i_1, p) \\
\quad \mathsf{in}\ o
\end{array}
$$

In general, the algorithm for determining the order in which relations containing existentially quantified variables should be translated, is as follows:

1. Strip off the existentially quantified variables and form a list of all the conjuncts, called *conjl*.

2. Examine each conjunct in *conjl*, doing steps 3 and 4 for each relation.

3. Match the parameters of the predicate with their model, thus obtaining a list of parameters which are inputs and a list of parameters which are outputs. Call these lists *inl* and *outl* respectively.

4. If all parameters in the input list *inl* have known values, i.e. $inlist{-}inl = [\,]$, then the relation can be translated. Delete the relation from *conjl* and add it to a list called *known_inpl* (originally with a value [ ]).
   If $inlist{-}inl \neq [\,]$, i.e. some parameters in the input list do not have known values, then the relation cannot be translated yet and it is left in *conjl*.

5. Translate all relations in *known_inpl* using a single declarative statement of the form

   $$\mathsf{let}\ var = \mathsf{FUN}(args)\ \mathsf{and}\ \ldots\ \mathsf{in}$$

   leaving a trailing in at the end to join onto the rest of the translation. The list *known_inpl* is reset to [ ].

6. If there are any relations left in *conjl* go back to step 2. If not, then all relations have been translated into functions and the process terminates.

The relations on the right hand side of an equivalence in a relational definition are not always predicates applied to a set of parameters. As in the case for specifications of primitives, they are often equations of the form shown in the example below:

$$
\begin{array}{l}
\mathsf{PRED}(i_1, \ldots, i_m, o_1, \ldots, o_n) \equiv \\
\qquad o_1 = t_1[i_1, \ldots, i_m] \wedge \\
\qquad \vdots \\
\qquad o_n = t_n[i_1, \ldots, i_m]
\end{array}
$$

where $t_j[i_1, \ldots, i_m]$ denotes some term t_j involving free occurrences of $i_1, \ldots, i_m$, for $1 \leq j \leq n$.

Here, of course, the translation is easier as the outputs are already separated from the expressions which will evaluate their values. Hence, the functional definition is obtained by inserting let statements in the appropriate places. In the example above if $inlist$=$[i_1; \ldots; i_m]$ and $output$=$(o_1, \ldots, o_n)$, then the following definition is obtained:

$$
\begin{array}{l}
\mathsf{let}\ \mathsf{FUN}(i_1, \ldots, i_m) = \\
\qquad \mathsf{let}\ o_1 = t_1(i_1, \ldots, i_m) \\
\qquad \vdots \\
\qquad \mathsf{and}\ o_n = t_n(i_1, \ldots, i_m) \\
\qquad \mathsf{in}\ (o_1, \ldots, o_n)
\end{array}
$$

6.2.3 Primitive Recursive Definitions

Primitive recursive definitions are often used to describe the structure of several identical hardware devices iteratively connected together, such as in the relational definition of the n-bit adder presented in Chapter 3.

In the HOL logic, primitive recursive relational definitions are expressed as a conjunction of two relations defining the base and recursive cases as follows:

$$
\begin{array}{l}
\mathsf{PRED}\ 0\ (i_1, \ldots, i_m, o_1, \ldots, o_r) \equiv t_0[0, i_1, \ldots, i_m, o_1, \ldots, o_r] \wedge \\
\mathsf{PRED}\ n{+}1\ (i_1, \ldots, i_m, o_1, \ldots, o_r) \equiv t_1[n, i_1, \ldots, i_m, o_1, \ldots, o_r]
\end{array}
$$

where $t_j[n, i_1, \ldots, i_m, o_1, \ldots, o_r]$ denotes some boolean term involving free occurrences of $n, i_1, \ldots, i_m, o_1, \ldots, o_r$.

Relational definitions of the above form can be translated into recursive functions of the form:

$$
\begin{array}{l}
\mathsf{letrec}\ \mathsf{FUN}\ n\ i_1 \ldots i_m = \\
\qquad \mathsf{let}\ (o_1, \ldots, o_r) = (n{=}0) \Rightarrow t_0'[0, i_1, \ldots, i_m] \mid t_1'[n{-}1, i_1, \ldots, i_m] \\
\qquad \mathsf{in}\ (o_1, \ldots, o_r)
\end{array}
$$

where $inlist = [i_1; \ldots; i_m]$, $output = (o_1, \ldots, o_r)$, $t_1'[n{-}1, i_1, \ldots, i_m]$ is the functional translation of $(t_1[n, i_1, \ldots, i_m, o_1, \ldots, o_r])[n{-}1/n]$ and t_0' represents the functional translation of t_0. The notation $A[B/C]$ means "B is substituted for all free occurrences of C in A".

6.2.4 Relations Involving History Functions

Finally, consider relational specifications which make use of history functions to describe timing features in sequential circuits. One commonly used form of such relations is:

$$\begin{aligned}&\mathsf{PRED}\ (i_1, \ldots, i_m, o_1, \ldots, o_n) \equiv \\ &\qquad \forall t.\ o_1(t) = t_1[i_1, \ldots, i_m, t] \wedge \\ &\qquad \vdots \\ &\qquad \forall t.\ o_n(t) = t_n[i_1, \ldots, i_m, t]\end{aligned}$$

The same rules for translating relations not involving history functions apply. The universal quantification is removed and declarative statements are inserted. If the values of *inlist* and *output* are $[i_1; \ldots; i_m]$ and $(o_1, \ldots, o_n)$ respectively, the above relation translates to:

$$\begin{aligned}&\mathsf{let\ FUN}\ (i_1, \ldots, i_m) = \\ &\quad \mathsf{let}\ o_1(t) = t_1[i_1, \ldots, i_m, t] \\ &\quad \vdots \\ &\quad \mathsf{and}\ o_n(t) = t_n[i_1, \ldots, i_m, t]\end{aligned}$$

However, extra checks must be made for detecting feedback. If o_j occurs free in the corresponding term t_j, then feedback occurs and o_j must be declared recursively.

For example, in the relation

$$\mathsf{PRED}(i, o) \equiv \forall t.\ o(t{+}1) = t'[i(t), o(t)]$$

the value of the function o at time $t{+}1$ is defined in terms of some expression t' involving the values of i and o at time t. The function o is, therefore, recursive and a letrec statement must be used to declare o as a function. Moreover, recursive functions have to be defined at time t rather than $t{+}1$ so all values of $t{+}1$ in the expression are replaced by t and all values of t are replaced by $t{-}1$ during the translation. The last problem is that the expression

$$o(t{+}1) = t'[i(t), o(t)]$$

is only a partial specification of o; it does not define a value for o at time 0. The translation, therefore, must introduce a new identifier (which does not already occur in the overall definition) to parameterise the initial value of o. A conditional statement is used in the definition of o such that the initial value is returned at time $t{=}0$ and the value $t'[i(t{-}1), o(t{-}1)]$ is returned otherwise. The initial value is then added on to the list *inlist* which contains the list of variables which will be parameterised when defining the overall function. The functional translation

of the relational definition of PRED above, therefore, is:

$$\mathsf{let\ FUN}\ i\ oval = \mathsf{letrec}\ o\ t = (t{=}0) \Rightarrow oval \mid t'[i(t{-}1), o(t{-}1)]$$

Information on the number of initial values parameterised when generating functional definitions and their types must be stored in relation to the names of their corresponding functions. Thus, whenever the name of a pre-translated predicate forms part of another definition and its functional translation is found to require initial value parameters by consulting the stored information suggested above, identifiers denoting initial values are generated according to the types required and are inserted in the functional application and parameterised in the overall new definition.

When internal lines are involved, the same algorithm on page 94 used for determining the order in which relations should be translated can also be used, but a slight modification is necessary. Consider the example:

$$\mathsf{PRED}(i, o) \equiv \exists p.\ \mathsf{PRED}_1(i, o, p) \wedge \mathsf{PRED}_2(p, o)$$

where $inlist{=}[i]$, $output{=}o$ and the parameter models of PRED_1 and PRED_2 are $[0; 0; 1]$ and $[0; 1]$ respectively.

According to the algorithm on page 94, the existential quantifiers are first stripped off from the right hand side of the equivalence and the two conjuncts are put in a list called $conjl$.

$$conjl = [\mathsf{PRED}_1(i, o, p); \mathsf{PRED}_2(p, o)]$$

The parameters of each relation in the list are matched against their models:

- for the first relation, PRED_1, the model is $[0; 0; 1]$, yielding inputs i and o, and output p. Now o does not have a known value so PRED_1 cannot be translated yet and the relation is left in $conjl$.

- for the second relation, PRED_2, the model is $[0; 1]$, indicating that p should act as input and o as output. Once again, the relation cannot be translated yet since p is an unknown and so the second relation is also left in $conjl$.

So after one pass over the list $conjl$, no relations are found to be ready for translation, therefore the list which should contain the new relations for translation, $known\_inpl$ is empty and step 5 of the algorithm fails. In models of combinational circuits, feedback should not occur, so the failure of the algorithm is justified since any attempt to model feedback in the absence of timing parameters would result in an infinite recursion.

The algorithm should therefore be modified such that step 5 reads:

5. If *known_inpl* $\neq$ [ ] then translate all relations in *known_inpl* using a single declarative statement of the form

   $$\textsf{let } var = \textsf{FUN}(args) \textsf{ and } \ldots \textsf{ in}$$

   leaving a trailing in at the end to join onto the rest of the translation. The list *known_inpl* is reset to [ ].
   Otherwise, if *known_inpl* = [ ], a recursion is detected and all relations still in *conjl* are translated using a recursive declaration of the form

   $$\textsf{letrec } var(t) = \textsf{FUN } (args)\, t \textsf{ and } \ldots \textsf{ in}$$

   leaving a trailing in at the end to join onto the rest of the translation. The list *conjl* is set to [ ].

The relational definition in the current example translates to:

$$\begin{array}{l}\textsf{let FUN } i = \\ \quad \textsf{letrec } p(t) = (\textsf{FUN}_1\ i\ o)\, t \\ \quad \textsf{and } o(t) = (\textsf{FUN}_2\ p)\, t \\ \quad \textsf{in } o\end{array}$$

The relations which can be automatically translated into functions as described above are relations of a form commonly used for specifying the structure of hardware and the behaviour of simple components. Chapters 7 through 9 demonstrate that translating such types of relations is enough to generate functional specifications which can be used for simulating complex hardware devices.

Relations describing the abstract behaviour of a device, however, are often in a form which does not have a functional translation. Examples of such relations and of relations which translate into non-executable functions are dealt with in the next section.

6.3 Relations Not Automatically Translated

Relations cannot always be translated into functions and are often preferred for writing definitions simply because the behaviour being modelled cannot be expressed functionally.

In this section we present a few examples to explain the problem. Since it is impossible to give a general example which models the infinite number of relations

which cannot be translated, the examples below are specific and are only intended to give an idea of some of the reasons why certain relations have no functional translation.

6.3.1 Relations with No Functional Interpretation

One problem with functions is that they must have an output. Often, a definition is composed of a series of conditions which, if they hold, then they imply some boolean facts. For example, consider the definition of a predicate STABLE given below:

$$\forall t_1\ t_2\ sig.\ \mathsf{STABLE}\ (t_1, t_2)\ sig = \forall t.\ (t_1 < t) \wedge (t < t_2) \supset (sig\ t) = (sig\ t_1)$$

This definition is commonly used in the context of hardware specification and verification [7,21,39,60]. The parameters of STABLE are two identifiers t_1 and t_2 of type *time* and a signal *sig* of type *time→bool*. An informal interpretation of the definition is:

> STABLE (t_1, t_2) *sig* holds if for all times t between time t_1 and time t_2, where t_1 is the earlier time, the value of *sig* remains constant, equal to the value at t_1.

In other words, the value of *sig* does not change between the times t_1 and t_2. So, for example, if $t_1 = 1$, $t_2 = 8$ and $sig(t_1) = \mathsf{F}$, then the value of *sig* will also be F for times 2 through 7 if the STABLE condition above is to hold.

The definition of STABLE has no notion of output whatsoever and for this reason it is not possible to model it as a function. The definition is merely a set of boolean conditions which define when the value on a signal can be assumed to be stable. The problem of not obtaining a function here is not in the automation process discussed in the previous section but in the fact that none of the parameters can be output. There is nothing to compute.

Another simple example is the definition of NEXT below which is also commonly used in the behavioural descriptions of hardware circuits.

$$\begin{array}{l}\forall\ sig\ t_1\ t_2.\\ \quad \mathsf{NEXT}\ (t_1, t_2)\ sig = \\ \qquad\qquad (t_1 < t_2) \wedge \\ \qquad\qquad sig(t_2) \wedge \\ \qquad\qquad (\forall t.\ (t_1 < t) \wedge (t < t_2) \supset \neg(sig\ t))\end{array}$$

The definition of NEXT says that:

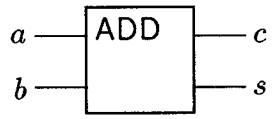
> NEXT (t_1, t_2) *sig* holds if for any two instants in time, t_1 and t_2, and any signal *sig*, t_2 occurs after t_1, *sig* is *true* at t_2 and *sig* is *false* at all times between t_1 and t_2. NEXT is *true* if t_2 is the first time after t_1 at which *sig* is *true*.

One possible interpretation of NEXT as a function would be to attempt to compute t_2, i.e. to compute the next time the signal *sig* becomes *true*. There is not much use for such a function, however, because it can only compute a value if *sig* does become *true* at some time. The function does not make sense if *sig* is never *true*. Once again, therefore, no functional translation is possible since none of the parameters can be sensibly interpreted as outputs.

6.3.2 Relations Requiring Normalisation

Certain relations pose an entirely different problem. Often the problem is not in choosing which parameters should be the outputs but in finding the right function that computes the outputs. This kind of problem is generally associated with equations in a non-definitional form.

For example, consider a two-bit binary adding device ADD shown below which adds two input bits a and b and returns the result as a sum bit s and a carry bit c.



The behaviour of the device can be specified relationally as follows:

$$\mathsf{ADD}(a, b, s, c) \equiv \mathit{bitval}(s) + 2 \times \mathit{bitval}(c) = \mathit{bitval}(a) + \mathit{bitval}(b)$$

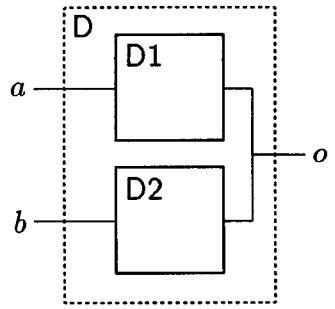
where *bitval* is defined on page 34. The parameters which should be treated as outputs in a functional specification are obviously s and c which represent the result of adding a to b. The problem here, however, is that s and c cannot be computed from the above equation. The specification would have to be rewritten either using two equations such that only the identifiers s and c appear on the left hand side of each equation or, using a single equation with the pair (s, c) on the left hand side. Equations like the above cannot be automatically manipulated to compute the value of an output unless information regarding appropriate inverse functions is given. This would make the task of translating relations to functions very complicated. Inverse functions do not always exist and when they do, if one

is to take the trouble to find and define them, one might as well restructure the specification to the form where only the identifiers appear on the left hand side.

For this reason, the relations in the form of equations that were presented in the previous section to show the way in which they were translated were all of the form *outputs* = *expression*, where *expression* is an executable expression (composed of primitive recursive or boolean functions). Presently, equations in any other form cannot be translated although it might be worth while exploring the possibility of introducing some automatic basic equation manipulation for simple arithmetic.

6.3.3 Relations with Non-Executable Interpretations

Certain relations can have a functional translation which is not executable. A feature in hardware specification which results in the translation to non-executable functions at a register-transfer level is the joining of two or more output lines. Consider for example, a device D composed of two devices D1 and D2 connected as shown in the diagram below.



The relational specification of D is:

$$\mathsf{D}_{rel}(a, b, o) \equiv \mathsf{D1}_{rel}(a, o) \wedge \mathsf{D2}_{rel}(b, o)$$

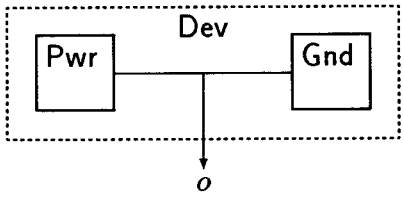
which translates to the functional specification:

$$\begin{array}{l}\mathsf{let}\ \mathsf{D}_{fun}\ a\ b = \\ \quad \mathsf{let}\ o = \mathsf{D1}_{fun}\ a \\ \quad \mathsf{and}\ o = \mathsf{D2}_{fun}\ b \\ \quad \mathsf{in}\ o\end{array}$$

The problem with the functional specification above is that the output identifier o is declared more than once. An attempt to parse D_{fun}, therefore, would fail and the following error message would be produced:

multiple occurrence of a variable in left hand side of a definition

At the register-transfer level, joining of wires is seldom modelled since it is this feature that most commonly gives rise to the 'false implies everything' problem [7] by using false relational models of implementation designs. A simple example is a device in which a power source is connected to ground:



The relational definitions of Pwr and Gnd are:

$$\mathsf{Pwr}_{rel}(o) \equiv (o = \mathsf{T})$$
$$\mathsf{Gnd}_{rel}(o) \equiv (o = \mathsf{F})$$

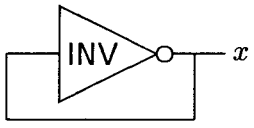
The relational definition of Dev, therefore, is:

$$\mathsf{Dev}_{rel}(o) \equiv \mathsf{Pwr}_{rel}(o) \wedge \mathsf{Gnd}_{rel}(o)$$

which is obviously always false since the above definitions of Pwr and Gnd assert that the value on the output o should be both *true* and *false* simultaneously. Attempting to execute the functional translation of Dev_{rel} would result in a similar error message as the one above given for any general output join.

A simulation tool at the register-transfer level, therefore, traps instances of *fan-in* in circuit designs. These fan-in features can give rise to false implementations at the register-transfer level and should be avoided. The notion of using simulation to track down occurrences of false implementation definitions is very much in line with the main theme of this thesis and in Figure 1.1 it was shown how one should backtrack from simulation to design until confident that the definitions are correct before moving on to verification.

Another feature which can cause relations to translate into non-executable functions is feedback that results in an infinite recursion. For example, the diagram below shows an inverter INV with the output connected to its input. Since logic gates are modelled without delay at the register-transfer level, such a connection cannot be allowed.



The relational definition for the above device is:

$$\mathsf{DEV}_{rel}(x) \equiv \mathsf{INV}_{rel}(x, x)$$

where INV_{rel} is a relational model of the inverter defined as:

$$\mathsf{INV}_{rel}(a, b) \equiv b = \neg a$$

By expanding the definition of INV_{rel} in that of DEV_{rel} we obtain:

$$\mathsf{DEV}_{rel}(x) \equiv x = \neg x$$

which is clearly false for all x.

The functional translation of the above definition should be:

$$\mathsf{DEV}_{fun} = \mathsf{let}\ x = \neg x\ \mathsf{in}\ x$$

but this cannot be defined since the same identifier x appears on both sides of the declaration $\mathsf{let}\ x = \neg x$. An identifier can only appear on both sides of an equation in a recursive definition. In such a case, however, the identifier must denote a recursive function instead of a value. Of course, a recursive declaration does not provide a solution in this case since the methods we use only deal with primitive recursion. Thus, the recursive definition:

$$\mathsf{letrec}\ x\ t = \neg(x\ t)$$

loops indefinitely.

6.4 Automatic Translation of HOL to ML

Having covered which forms of relations we are interested in translating into functions in Section 6.2, we outline in this section the method used for translating these relations into executable functions.

We wish to translate terms in the HOL logic to some form of executable code representing ML programs, optimised as explained in Chapter 5. To do this, it is convenient to translate the HOL terms to some intermediate form which is simple to manipulate. This intermediate form, itself a HOL term, can then be translated into a code representing the corresponding ML program.

The structure chosen for representing the intermediate form is based on the idea of *constructor functions*. In the HOL framework there are certain ML functions called constructor functions that can be used to build up HOL terms [22]. They are used to input HOL terms and types to the ML system as an alternative to quotation. For example, the ML expression:

$$mk\_type(\text{`}bool\text{`}, [\,])$$

returns the HOL type " : *bool*", and the expression:

mk_var('*x*', " : *bool*")

returns the HOL term "*x*", where *x* is a HOL variable of type *bool*.

Obviously, using these constructor functions to input HOL expressions is extremely tedious when one can merely type in the quoted HOL expression directly. The functions are useful, however, for building HOL expressions in a non-interactive manner, such as in programs which generate HOL terms.

What is needed, therefore, for representing ML programs is a set of HOL constructor functions (instead of the ML ones mentioned above) which can be used in HOL terms to form an intermediate representation of ML programs as discussed above. For example, if "*mk_ml_type*" is the HOL function for constructing atomic types in ML, the HOL term:

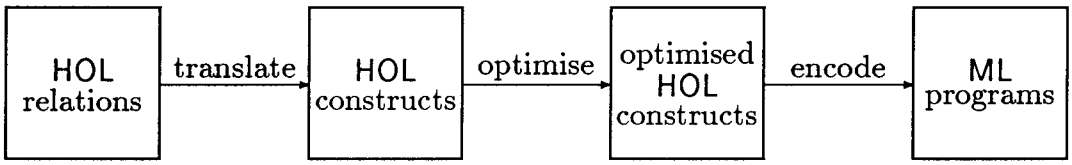
"*mk_ml_type*('*bool*', [])"

would represent the ML type *bool*, and if "*mk_ml_var*" is the HOL function for constructing ML variables, the term:

"*mk_ml_var*('*x*', *mk_ml_type*('*bool*', []))"

would represent an ML variable of type *bool*. These representations of ML programs can then be transformed to optimised ML programs which can be evaluated.

The process of translating HOL relations into ML programs can be summarised using the following diagram:



The first branch in the above diagram represents the transformation of HOL relations to an intermediate HOL term, which consists of HOL constructors that represent ML programs. The second branch in the diagram shows that this representation of an ML program is converted into a representation of the efficient version of the program (a second intermediate form), and the third branch represents the final stage at which the optimised HOL constructs are converted into the ML program.

The intermediate forms represented by HOL constructs are usually long and over detailed. Even simple relations explode into an almost incomprehensible

term. This form of coding terms, however, is simple to manipulate mechanically. It only forms an intermediate stage in the transition of translating HOL relations to ML programs and it is not output by the program unless specifically requested. Because these intermediate forms are so incomprehensible, no examples of them are presented here.

The second stage of the transformation is to pass through the constructs produced from the first stage and check for instances where optimisation is required. The construct is optimised at every recursive declaration.

The optimisation stage produces yet another construct which replaces the inefficient subterm in the overall term. This makes use of another constructor function, used for constructing memo-functions. The constructor function's arguments consist of the necessary information for memoisation in the final stage of the process. The first argument is an automatically generated alphanumeric string which denotes the name of the memoised function; the second argument is the type of the function used for determining which table to store results in; and the third argument is the inefficient construct.

An optimisation construct is inserted for every recursively declared function. If memoisation constructs are introduced in a definition, a naming parameter for the overall definition must be introduced. The reason for this is that if a memoised function is used in more than one place to compute output using different data, the output must be stored in, and retrieved from, different locations. Otherwise, the results of one function will overwrite those of another.

Consider as an example a circuit composed of two identical D-type flip-flops. The functional definitions of the D-types are recursive and so they are memoised. If one requests the output of a particular D-type, A say, at time t, the memo-function computes the result and stores it in a table. If, then, the output of D-type B at time t is requested, then if the same function models both D-types, the result previously computed for D-type A is retrieved from the table and returned as the output of D-type B. This is obviously wrong and the problem is solved as suggested above by parameterising a name to memoised functions.

Therefore, at the optimising stage, not only must memoisation constructs be introduced when recursive declarations are encountered but also:

- an extra string parameter must be added to the overall definition if memoisation constructs are used;
- the extra appropriate string parameters must be added on to the calls of memoised functions encountered in the body of a definition.

The concept of naming to provide unique addressing for the memoisation functions is further explained and illustrated in the examples presented in Chapters 8–9.

The final stage in the transformation from HOL relations to ML functions is to generate ML code that defines a function corresponding to the optimised construct produced in the previous stage. In general, the program that goes straight from the HOL relations to the ML programs is invoked; Appendix B gives an example interactive session to illustrate how the translator can be used. Other programs can be used, however, which return the output at intermediate stages in the transformation in case this information is required.

The final parse that encodes the ML programs involves the translation of predefined operators and constants. For example, the symbol ¬ denoting negation in HOL is translated to the corresponding ML function *not*. Of course there is no reason why the same symbols cannot be declared to denote the same constant or operator as long as they do not already denote something. The existing parser, however, makes use of predefined symbols and translates from HOL symbols to ML symbols when these differ. Other examples of differing constant symbols are T and F in HOL and *true* and *false* in ML.

The HOL constant ARB (defined in Chapter 3) is translated as a special case. There is no corresponding ML constant since ARB cannot be executed. It represents any arbitrary value and this cannot be evaluated. The constant ARB, however, is only used in partial specifications where part of a definition is undefined. The definition should not be executed under those conditions which generate ARB, therefore, since the result is undefined. Thus, the constant ARB is translated into the ML *fail* command which fails the program if it is executed under undefined conditions.

Finally, there is one side effect which takes place when parsing constructs involving optimised functions. Whenever a memoisation construct is encountered, the parser checks whether a memo-function and table have been defined corresponding to the type of the function being memoised. If such a memo-function and storing table had previously been defined, then the memoisation construct is parsed with no side effects. Otherwise, a memo-function and table of the appropriate types are defined as a side effect before resuming with the parsing.

The following chapters present examples of hardware devices which have been specified using HOL relations and whose specifications are transformed into ML programs using the methods described in this chapter. The resulting programs are executed using some example test cases.

Chapter 7

Simulating a Factorial Machine

In this chapter, and in the following two chapters, examples of hardware devices are presented which have been previously specified in HOL without intentions of simulating the definitions. The simulation of these devices offers evidence of the extent and generality of the automatic translating techniques explained in Chapter 6; inventing new examples introduces the danger of writing specifications in a style which suits the automatic translation, resulting in a biased idea of how versatile and effective the techniques are.

The first example is a sequential device which computes the factorial function. The design of the device is the same as in [17] where the factorial machine was first specified and verified by Mike Gordon using a formalism based on denotational semantics. The same device was later specified and mechanically verified in HOL by Tom Melham [7].

This chapter shows how the existing HOL relational definitions of the device, described in [7], are automatically translated into executable ML functions. The derived programs are later executed to demonstrate the behaviour of the device.

The ML programs presented in this chapter are the final optimised versions derived automatically from HOL relations. At first glance, this final product can look rather different from the original relations. On closer examination, however, one can distinguish the 'naive' ML functions embedded in the larger optimised functions. The strong similarity between the naive (inefficient) ML functions and the corresponding HOL functions was shown in Chapter 4. The optimised functions are achieved by simple and clear transformations performed on the inefficient functions as described in Chapters 5–6. With the inefficient version of the function in mind, therefore, the optimised result becomes more obvious.

Since the translations presented in this chapter (and in Chapters 8–9) are from HOL terms to ML programs, it is necessary to distinguish between the two

notations. In these chapters, therefore, HOL terms are enclosed within double quotes whereas ML expressions are not.

7.1 The Specification

The factorial machine, as presented in [7], is a device Factorial which has one input line *in* and two output lines *ready* and *out* as shown in Figure 7.1.

Figure 7.1: A Factorial Machine

Its behaviour is defined as follows:

> At any time t, if the value on the boolean line *ready* is true, then the integer value, n say, on the input line *in* at time t is read by the device. The factorial of n is calculated and output on line *out* at time $t+n+1$. During the time taken for the factorial of n to be calculated, i.e. from time t to time $t+n$, the value on the output line is maintained at 0. Also, the value of *ready* becomes *false* over times $t+1$ through $t+n+1$ so that no further input is loaded into the device while the factorial of n is being calculated. The factorial value is displayed on the output line for one unit of time (at time $t+n+1$) and at time $t+n+2$ the value on the *ready* line becomes *true* once again, ready to read the next input.

Formally, the above specification can be written as a relation as follows:

$$
\begin{array}{l}
\text{"Factorial}(ready, in, out) \equiv \\
\quad \forall n\ t.\ ready(t) \wedge n{=}in(t) \supset \\
\qquad \mathsf{NEXT}(t, t{+}n{+}2, ready) \wedge \\
\qquad \forall t'.\ (t{\leq}t' \wedge t'{<}t{+}n{+}1) \supset (out(t'){=}0) \wedge \\
\qquad out(t{+}n{+}1) = \mathsf{FACT}(n)\text{"}
\end{array}
$$

where t is a value of type *time* (modelled by integers), n is an integer, *in* and *out* are history functions mapping time to integers, and *ready* is a history function from time to booleans. NEXT is the relation defined on page 99, and FACT is the factorial function defined below.

"FACT(0) = 1 ∧
FACT(n+1) = (n+1) × FACT(n)"

Unfortunately, this specification, as presented in [7], is a relation which cannot be translated to a function. The behaviour of the device is defined using implications and conjunctions of boolean conditions which together cannot be executed. For example, one of these conditions involves the predicate NEXT discussed in Section 6.3.1. Since this is a relation with no functional interpretation, it is impossible for the overall definition of Factorial to be interpreted as a function.

Of course, it may be possible to find an equivalent specification which has a functional interpretation but this is beyond the scope of this chapter. Here we are interested in translating existing definitions to find out how effective our techniques for translating relations to functions really are, and writing definitions which have a functional interpretation whenever some are found which do not is not relevant.

In this chapter, therefore, we do not investigate further the simulation of the specification, but that of the implementation which can be used to check that the specification conditions hold.

7.2 The Implementation

In this section we investigate the translation of the existing relational definitions which model the implementation of the factorial device, simulate the functional definitions over sample data and check that the results of the simulation conform with the specification conditions.

The design of the factorial device is presented in a top-down manner. The implementation is defined in terms of the definitions of three smaller devices Down, Mult and Test connected as shown in Figure 7.2 to form the device Fact.

In [7], the devices Down, Test and Mult were treated as primitives because the proofs of the implementations of these devices were not relevant to the paper. Their behaviour is defined using the following relational definitions.

"$\mathsf{Down}_{rel}(i, ready, l_1) \equiv$
$\forall t.\ l_1(t{+}1) = ready(t) \Rightarrow i(t) \mid l_1(t){-}1$"

"$\mathsf{Mult}_{rel}(ready, l_1, l_2) \equiv$
$\forall t.\ l_2(t{+}1) = ready(t) \Rightarrow 1 \mid l_2(t){\times}l_1(t)$"

"$\mathsf{Test}_{rel}(ready, l_1, l_2, out) \equiv$
$(\forall t.\ ready(t{+}1) = ((l_1(t) = 0) \wedge \neg(ready(t)))) \wedge$
$(\forall t.\ out(t) = ((l_1(t) = 0) \wedge \neg(ready(t))) \Rightarrow l_2(t) \mid 0)$"

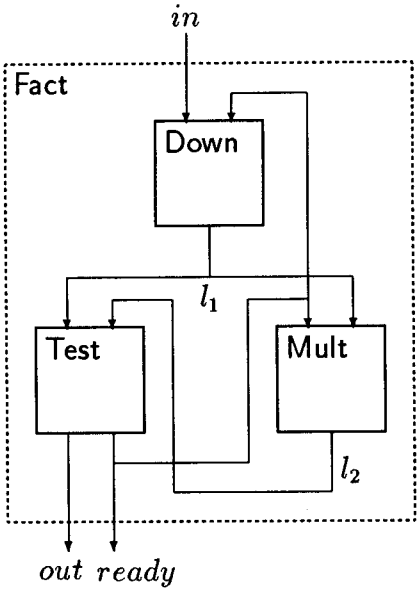


Figure 7.2: Implementation of Factorial Device

The definition of Down_{rel} states that the value on l_1 at time t+1 is set to the value of i at time t, if the value on *ready* at time t is *true*; it is set to the decrement of its previous value otherwise. The definition of Mult_{rel} is similar: the value on l_2 at time t+1 is set to 1 if *ready* is *false* at time t; the product of the values of l_2 and l_1 at time t otherwise. In the definition of Test_{rel}, if the value of l_1 at time t is 0 and the value of *ready* at time t is *false*, then *ready* at time t+1 becomes *true* and *out* at time t is set to the value of l_2 at time t; otherwise, *ready* at time t+1 is set to *false*, and *out* at time t is set to 0.

The behavioural specifications above are in a form that can be translated to functions since the devices are defined in terms of equations of a definitional form. The inputs and outputs of each device are easy to identify from the specifications and so, with the appropriate information on inputs and outputs, the above relations can be automatically translated to ML programs.

From the specification of Down_{rel} it can be seen that the device Down requires two inputs, *in* and *ready*, and returns one output, l_1. With this information, the definition of Down_{rel} is translated to:

```
let Down_fun name i ready l1_val =
   letrec l1(t') =
      (memo_num
          (name ^ 'down_l1')
          (λt. (t = 0) ⇒ l1_val | ready(t−1) ⇒ i(t−1) | l1(t−1)−1)) t'
   in l1
```

The above program Down_{fun} which models the device Down is an optimised translation of the definition of Down_{rel}. The program returns a history function l_1 which is computed recursively. The presence of a recursive computation triggers the automatic translator to optimise the calculation by introducing memoisation. If no memo-function has yet been defined for the type of l_1, a new memo-function and table are defined as a side effect before Down_{fun} is defined. The function l_1 maps time to numbers and so the function *memo_num* in the above program represents a memo-function which stores number values.

The relational definition Down_{rel} is only a partial specification; it does not define the function l_1 at time t=0. An initial value l_1_val is therefore automatically introduced in the functional definition to halt the recursion at time t=0. This initial value is a parameter of the overall definition.

Finally, since memoisation is used to store values computed by the program which models the Down device, a further parameter is introduced to represent a name for each occurrence of such a device. By using the infix concatenate operator, ^, this parameter of type *string* is used to prefix further names generated for uniquely identifying memoised functions.

Automatically generated names involve special sequences of symbols which are unlikely to be input interactively and in any case, they are always checked to ensure they do not already denote some value. Since such automatically generated sequences are often long and unreadable, they are replaced by shorter names in this thesis.

The relational definitions of Mult_{rel} and Test_{rel} are translated in the same way. In the translation of Mult_{rel}, *ready* and l_1 are specified as inputs and l_2 as output to obtain the function Mult_{fun} below.

$$
\begin{array}{l}
\mathsf{let}\ \mathsf{Mult}_{fun}\ name\ ready\ l_1\ l_2\_val = \\
\quad \mathsf{letrec}\ l_2(t') = \\
\quad\quad (memo\_num \\
\quad\quad\quad (name\ \hat{}\ \text{`}mult\_l2\text{`}) \\
\quad\quad\quad (\lambda t.\ (t = 0) \Rightarrow l_2\_val \mid ready(t{-}1) \Rightarrow 1 \mid l_2(t{-}1){\times}l_1(t{-}1)))\ t' \\
\quad \mathsf{in}\ l_2
\end{array}
$$

Because the history function to be memoised, l_2, is of the same type as l_1 used in the definition of Down_{fun}, the same memoisation function used in the definition of Down_{fun}, namely *memo_num*, is used in the definition of Mult_{fun}.

The translation of the definition of Test_{rel}, shown below, makes use of two memoisation functions: one for storing and retrieving number values, *memo_num*, and

the other for storing and retrieving booleans, *memo_bool*. From the parameters of Test_{rel}, l_1 and l_2 are selected as inputs while *out* and *ready* are the outputs.

$$
\begin{array}{l}
\mathsf{let}\ \mathsf{Test}_{fun}\ name\ l_1\ l_2\ ready\_val = \\
\quad \mathsf{letrec}\ ready(t') = \\
\quad\quad (memo\_num \\
\quad\quad\quad (name \mathbin{\hat{}} \text{`}test\_ready\text{`}) \\
\quad\quad\quad (\lambda t.\ (t = 0) \Rightarrow ready\_val \mid (l_1(t{-}1) = 0)\ \&\ not(ready(t{-}1)))))\ t' \\
\quad \mathsf{and}\ out(t') = \\
\quad\quad (memo\_bool \\
\quad\quad\quad (name \mathbin{\hat{}} \text{`}test\_out\text{`}) \\
\quad\quad\quad (\lambda t.\ (l_1(t) = 0)\ \&\ not(ready(t)) \Rightarrow l_2(t) \mid 0))\ t' \\
\quad \mathsf{in}\ (out, ready)
\end{array}
$$

The structure of the device Fact is defined relationally in the standard way by joining together the definitions of Down, Test and Mult using conjunction and by hiding the internal lines using existential quantifiers.

$$
\begin{array}{l}
\text{“}\mathsf{Fact}_{rel}(i, out, ready) \equiv \\
\quad\quad \exists l_1\ l_2.\ \mathsf{Down}_{rel}(i, ready, l_1) \wedge \\
\quad\quad\quad\quad \mathsf{Mult}_{rel}(ready, l_1, l_2) \wedge \\
\quad\quad\quad\quad \mathsf{Test}_{rel}(ready, l_1, l_2, out)\text{”}
\end{array}
$$

This is translated to the function Fact_{fun} which makes use of the previously defined functions Down_{fun}, Mult_{fun} and Test_{fun}.

$$
\begin{array}{l}
\mathsf{let}\ \mathsf{Fact}_{fun}\ name\ i\ ival_1\ ival_2\ ival_3 = \\
\quad \mathsf{letrec}\ l_1(t) = \\
\quad\quad\quad (memo\_num \\
\quad\quad\quad\quad\quad (name \mathbin{\hat{}} \text{`}fact\_l1\text{`}) \\
\quad\quad\quad\quad\quad (\mathsf{Down}_{fun}\ (name \mathbin{\hat{}} \text{`}fact\_down\text{`})\ i\ ready\ ival_1))\ t \\
\quad \mathsf{and}\ l_2(t) = \\
\quad\quad\quad (memo\_num \\
\quad\quad\quad\quad\quad (name \mathbin{\hat{}} \text{`}fact\_l2\text{`}) \\
\quad\quad\quad\quad\quad (\mathsf{Mult}_{fun}\ (name \mathbin{\hat{}} \text{`}fact\_mult\text{`})\ ready\ l_1\ ival_2))\ t \\
\quad \mathsf{and}\ out(t) = \\
\quad\quad\quad (memo\_num \\
\quad\quad\quad\quad\quad (name \mathbin{\hat{}} \text{`}fact\_out\text{`}) \\
\quad\quad\quad\quad\quad (fst\ (\mathsf{Test}_{fun}\ (name \mathbin{\hat{}} \text{`}fact\_test\text{`})\ l_1\ l_2\ ival_3)))\ t \\
\quad \mathsf{and}\ ready(t) = \\
\quad\quad\quad (memo\_bool \\
\quad\quad\quad\quad\quad (name \mathbin{\hat{}} \text{`}fact\_ready\text{`}) \\
\quad\quad\quad\quad\quad (snd\ (\mathsf{Test}_{fun}\ (name \mathbin{\hat{}} \text{`}fact\_test\text{`})\ l_1\ l_2\ ival_3)))\ t \\
\quad \mathsf{in}\ (out, ready)
\end{array}
$$

The translator detects that the functions Down_{fun}, Mult_{fun} and Test_{fun} require extra parameters to those identified as inputs in the translation of the predicates Down_{rel}, Mult_{rel} and Test_{rel}. The appropriate parameters are generated which denote initial values and unique string names. All identifiers denoting initial values are parameterised in the overall program. It is only necessary to parameterise one naming identifier which will be used to name different instances of the program modelling multiple occurrences of the same device. This parameter is concatenated to a uniquely generated string to supply a different name for each device requiring such identification and for each memo-function.

The functions *fst* and *snd* are used to select the appropriate members of the tuple returned by the function call to Test_{fun} since in ML the declarations of $out(t)$ and $ready(t)$ have to be made separately and it is not possible to declare tuples of recursive functions in the following way:

$$\mathsf{letrec}\ \ldots\ (out(t), ready(t)) = \mathsf{Test}_{fun}\ \ldots$$

It is also possible to translate the relational definition of Fact if the applications of Down_{rel}, Mult_{rel} and Test_{rel} are first expanded in the definition of Fact_{rel}. Such an expansion results in the definition Fact'_{rel}:

$$\begin{array}{l}
\text{“}\mathsf{Fact}'_{rel}(i, out, ready) \equiv \\
\quad \exists l_1\ l_2. \\
\qquad (\forall t.\ l_1(t{+}1) = ready(t) \Rightarrow i(t) \mid l_1(t){-}1) \wedge \\
\qquad (\forall t.\ l_2(t{+}1) = ready(t) \Rightarrow 1 \mid l_2(t){\times}l_1(t)) \wedge \\
\qquad (\forall t.\ ready(t{+}1) = ((l_1(t) = 0) \wedge \neg(ready(t)))) \wedge \\
\qquad (\forall t.\ out(t) = ((l_1(t) = 0) \wedge \neg(ready(t))) \Rightarrow l_2(t) \mid 0)\text{”}
\end{array}$$

which can be translated to the program Fact'_{fun} below, using techniques similar to those used in translating Fact_{rel}, Down_{rel}, Mult_{rel} and Test_{rel}. Of course,

$$\mathsf{Fact}_{rel} \equiv \mathsf{Fact}'_{rel}$$

and

$$\mathsf{Fact}_{fun} = \mathsf{Fact}'_{fun}$$

The complexity of the optimised programs derived so far supports the necessity for automatic translation. Example simulations of these programs are shown in Section 7.4 together with those for the programs derived in the next section.

$$
\begin{array}{l}
\mathsf{let}\ \mathsf{Fact}'_{fun}\ name\ i\ ival_1\ ival_2\ ival_3 = \\
\quad \mathsf{letrec}\ l_1(t') = \\
\quad\quad (memo\_num \\
\quad\quad\quad (name \mathbin{\hat{}} \text{`}fact'\_l1\text{`}) \\
\quad\quad\quad (\lambda t.\ (t = 0) \Rightarrow l_1\_val \mid ready(t{-}1) \Rightarrow i(t{-}1) \mid (l_1(t{-}1){-}1)))\ t' \\
\quad \mathsf{and}\ l_2(t') = \\
\quad\quad (memo\_num \\
\quad\quad\quad (name \mathbin{\hat{}} \text{`}fact'\_l2\text{`}) \\
\quad\quad\quad (\lambda t.\ (t = 0) \Rightarrow l_2\_val \mid ready(t{-}1) \Rightarrow 1 \mid l_2(t{-}1){\times}l_1(t{-}1)))\ t' \\
\quad \mathsf{and}\ out(t') = \\
\quad\quad (memo\_num \\
\quad\quad\quad (name \mathbin{\hat{}} \text{`}fact'\_out\text{`}) \\
\quad\quad\quad (\lambda t.\ (t = 0) \Rightarrow ready\_val \mid (l_1(t{-}1) = 0)\ \&\ not(ready(t{-}1))))\ t' \\
\quad \mathsf{and}\ ready(t') = \\
\quad\quad (memo\_bool \\
\quad\quad\quad (name \mathbin{\hat{}} \text{`}fact'\_ready\text{`}) \\
\quad\quad\quad (\lambda t.\ (l_1(t) = 0)\ \&\ not(ready(t)) \Rightarrow l_2(t) \mid 0))\ t' \\
\quad \mathsf{in}\ (out, ready)
\end{array}
$$

7.3 Implementation Using Simpler Primitives

In this section we consider the translations of relational definitions which model the implementations of Down, Mult and Test using simpler devices for primitives. Nine primitives are used altogether:

- Two devices, And and Not, to perform the logical operations of conjunction and negation. The relational definitions for these are:

  $$\text{“}\mathsf{Not}_{rel}(i, o) \equiv \forall t.\ o(t) = \neg(i(t))\text{”}$$

  $$\text{“}\mathsf{And}_{rel}(i_1, i_2, o) \equiv \forall t.\ o(t) = i_1(t) \wedge i_2(t)\text{”}$$

  which are translated to the functions:

  $$\mathsf{let}\ \mathsf{Not}_{fun}\ i = \mathsf{let}\ o(t) = not(i(t))\ \mathsf{in}\ o$$

  and

  $$\mathsf{let}\ \mathsf{And}_{fun}\ i_1\ i_2 = \mathsf{let}\ o(t) = i_1(t)\ \&\ i_2(t)\ \mathsf{in}\ o$$

  respectively.

- Two devices, Zero and One, to generate the constants 0 and 1. The relational definitions for these are:

  $$\text{“}\mathsf{Zero}_{rel}(o) \equiv \forall t.\ o(t) = 0\text{”}$$

  $$\text{“}\mathsf{One}_{rel}(o) \equiv \forall t.\ o(t) = 1\text{”}$$

  which translate to:

  $$\mathsf{let}\ \mathsf{Zero}_{fun} = \mathsf{let}\ o(t) = 0\ \mathsf{in}\ o$$

  $$\mathsf{let}\ \mathsf{One}_{fun} = \mathsf{let}\ o(t) = 1\ \mathsf{in}\ o$$

- A decrementer Dec which subtracts one from the input. The relation:

  $$\text{“}\mathsf{Dec}_{rel}(i, o) \equiv \forall t.\ o(t) = i(t){-}1\text{”}$$

  translates to:

  $$\mathsf{let}\ \mathsf{Dec}_{fun}\ i = \mathsf{let}\ o(t) = i(t){-}1\ \mathsf{in}\ o$$

  The subtraction used in the HOL term ranges over natural numbers such that $0{-}n = 0$. Since the predefined ML subtraction operator ranges over integers, a natural number subtraction operator was defined and used in the definition of Dec_{fun} shown above. To avoid the use of two different symbols to denote subtraction, only the symbol '$-$' is used in this chapter and it represents natural number subtraction.

- A multiplier Multiply which multiplies two natural numbers. The relational definition:

  $$\text{“}\mathsf{Multiply}_{rel}(i_1, i_2, o) \equiv \forall t.\ o(t) = i_1(t){\times}i_2(t)\text{”}$$

  translates to:

  $$\mathsf{let}\ \mathsf{Multiply}_{fun}\ i_1\ i_2 = \mathsf{let}\ o(t) = i_1(t){\times}i_2(t)\ \mathsf{in}\ o$$

- A multiplexor Mux to select which input to pass on to the output. The relational definition:

  $$\text{“}\mathsf{Mux}_{rel}(sw, i_1, i_2, o) \equiv \forall t.\ o(t) = sw(t) \Rightarrow i_1(t) \mid i_2(t)\text{”}$$

translates to:

$$\mathsf{let}\ \mathsf{Mux}_{fun}\ sw\ i_1\ i_2 = \mathsf{let}\ o(t) = sw(t) \Rightarrow i_1(t) \mid i_2(t)\ \mathsf{in}\ o$$

- A device Eqzero for testing equality with zero. The relational definition:

  "$\mathsf{Eqzero}_{rel}(i, o) \equiv \forall t.\ o(t) = i(t)=0$"

  translates to:

  $$\mathsf{let}\ \mathsf{Eqzero}_{fun}\ i = \mathsf{let}\ o(t) = i(t)=0\ \mathsf{in}\ o$$

  If the input is zero, Eqzero returns *true*; otherwise *false*.

- A register Reg for storing values. The relational definition:

  "$\mathsf{Reg}_{rel}(i, o) \equiv \forall t.\ o(t+1) = i(t)$"

  translates to:

  $$\mathsf{let}\ \mathsf{Reg}_{fun}\ i\ oval = \mathsf{let}\ o(t) = (t=0) \Rightarrow oval \mid i(t-1)\ \mathsf{in}\ o$$

  The definitions are polymorphic thus modelling registers for storing values of any type.

None of the above definitions require recursive computation so no optimisation is introduced. The definition of Reg_{rel} is only partial and so an initial value is introduced in the functional definition.

The devices Down, Mult and Test can be implemented using the primitives described above. The implementation of Down is composed of a decrementer Dec, a multiplexor Mux and a register Reg as shown in Figure 7.3.

The implementation of Down is very similar to that of the counter described in Chapter 4. The only difference is that the input of the counter is incremented whereas that of Down is decremented. The Mux device checks if the value on the *ready* line at time t is *true*. If it is, then the value on the input line i at time t is passed on to Reg. Otherwise, 1 is subtracted from the value on the output line l_1 at time t by the device Dec and the result is passed on by Mux to Reg. The output of Down at time $t+1$ is equal to the value input to Reg at time t.

The structure of the implementation of Down can be defined relationally by:

$$\begin{aligned}
&\text{“}\mathsf{Down_Imp}_{rel}(i, ready, l_1) \equiv \\
&\quad \exists p_1\ p_2. \\
&\qquad \mathsf{Dec}_{rel}(l_1, p_1) \wedge \\
&\qquad \mathsf{Mux}_{rel}(ready, i, p_1, p_2) \wedge \\
&\qquad \mathsf{Reg}_{rel}(p_2, l_1)\text{”}
\end{aligned}$$

which is translated to the program:

$$\begin{aligned}
&\mathsf{let}\ \mathsf{Down_Imp}_{fun}\ name\ i\ ready\ regval = \\
&\quad \mathsf{letrec}\ p_1(t) = \\
&\qquad (memo_num\ (name \hat{}\ \text{`}down\_p1\text{`})\ (\mathsf{Dec}_{fun}\ l_1))\ t \\
&\quad \mathsf{and}\ p_2(t) = \\
&\qquad (memo_num\ (name \hat{}\ \text{`}down\_p2\text{`})\ (\mathsf{Mux}_{fun}\ ready\ i\ p_1))\ t \\
&\quad \mathsf{and}\ l_1(t) = \\
&\qquad (memo_num\ (name \hat{}\ \text{`}down\_l1\text{`})\ (\mathsf{Reg}_{fun}\ p_2\ regval))\ t \\
&\quad \mathsf{in}\ l_1
\end{aligned}$$

The program computes the value of l_1 recursively and is therefore optimised using appropriate memo-functions.

The implementation of Mult consists of four components: Multiply, One, Mux and Reg, connected as shown in Figure 7.4. The multiplexor Mux tests the value on the *ready* line at time t. If this is *true* then the constant 1 computed by One is input to the register Reg. If it is false, however, the result of multiplying the

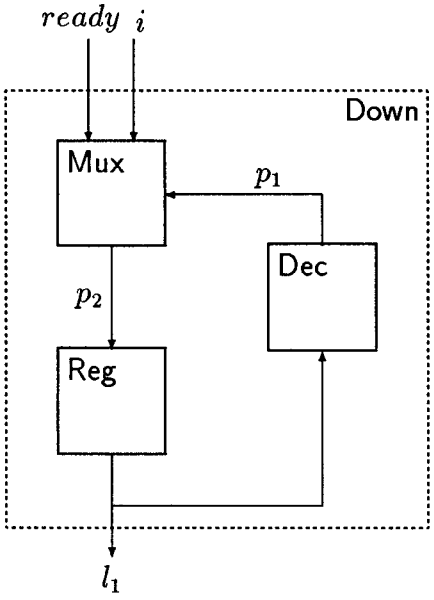


Figure 7.3: Implementation of Down

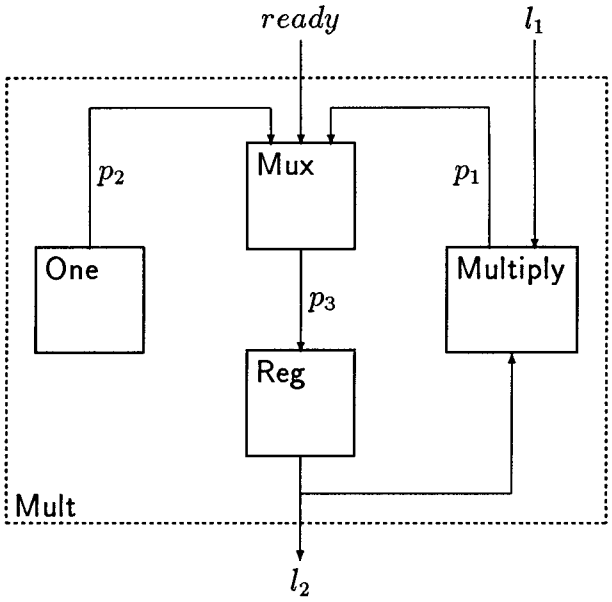


Figure 7.4: Implementation of Mult

values on the input l_1 and output l_2 computed by Multiply is input to Reg. The value input to Reg at time t is output one time unit later at time $t+1$; this also being the output of the overall Mult device.

The structure of the implementation of Mult can be defined relationally as:

$$
\begin{aligned}
&\text{“Mult\_Imp}_{rel}(ready, l_1, l_2) \equiv \\
&\quad \exists p_1\ p_2\ p_3 . \\
&\qquad \text{Multiply}_{rel}(l_2, l_1, p_1) \wedge \\
&\qquad \text{One}_{rel}(p_2) \wedge \\
&\qquad \text{Mux}_{rel}(ready, p_2, p_1, p_3) \wedge \\
&\qquad \text{Reg}_{rel}(p_3, l_2)\text{”}
\end{aligned}
$$

which is translated to the program:

```
let Mult_Imp_fun name ready l1 regval =
    let p2 = One_fun
    in
    letrec p1(t) =
            (memo_num (name ^ 'multi_p1') (Multiply_fun l2 l1)) t
    and p3(t) =
            (memo_num (name ^ 'multi_p3') (Mux_fun ready p2 p1)) t
    and l2(t) =
            (memo_num (name ^ 'multi_l2') (Reg_fun p3 regval)) t
    in l2
```

The computation of the values on line p_2 is not recursive and so, there is no need for it to be optimised along with the rest of the computation. It is declared separately

from the rest of the lines which are declared recursively.

Finally consider the implementation of Test; the largest of the three original devices. This has six components: Eqzero, Not, And, Reg, Mux and Zero connected as shown in Figure 7.5.

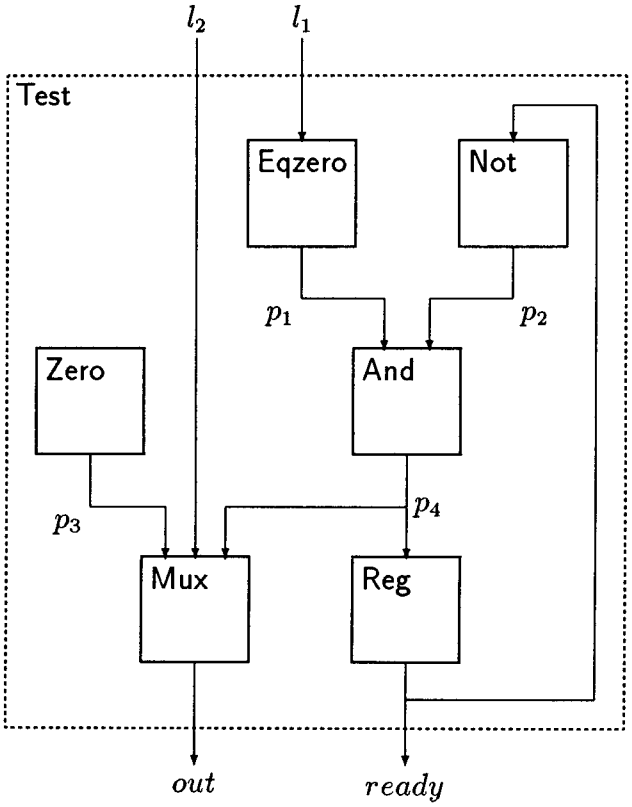


Figure 7.5: Implementation of Test

The value on the input l_1 at time t is tested by Eqzero. If the value is equal to zero, then the negation of the value on *ready* at time t computed by Not is passed through And to p_4. This value is output via the register Reg as the next value on *ready* at time $t+1$. The value on p_4 at time t is also used to compute the value on *out* at time t. The Mux tests the value on p_4; if it is *true* then the value on the input line l_2 at time t is output, otherwise the constant 0 computed by Zero is output.

The relational definition describing the structure of Test is as follows:

$$
\begin{aligned}
&\text{“}\mathsf{Test\_Imp}_{rel}(ready, l_1, l_2, out) \equiv \\
&\quad \exists p_1\ p_2\ p_3\ p_4 . \\
&\qquad \mathsf{Eqzero}_{rel}(l_1, p_1) \wedge \\
&\qquad \mathsf{Not}_{rel}(ready, p_2) \wedge \\
&\qquad \mathsf{And}_{rel}(p_1, p_2, p_4) \wedge \\
&\qquad \mathsf{Zero}_{rel}(p_3) \wedge \\
&\qquad \mathsf{Reg}_{rel}(p_4, ready) \wedge \\
&\qquad \mathsf{Mux}_{rel}(p_4, l_2, p_3, out)\text{”}
\end{aligned}
$$

This translates to:

$$
\begin{aligned}
&\mathsf{let}\ \mathsf{Test\_Imp}_{fun}\ name\ l_1\ l_2\ regval = \\
&\quad \mathsf{let}\ p_1 = \mathsf{Eqzero}_{fun}\ l_1 \\
&\quad \mathsf{and}\ p_3 = \mathsf{Zero}_{fun} \\
&\quad \mathsf{in} \\
&\quad \mathsf{letrec}\ p_2(t) = \\
&\qquad (memo\_bool\ (name \mathbin{^\wedge} \text{`}testi\_p2\text{`})\ (\mathsf{Not}_{fun}\ ready))\ t \\
&\quad \mathsf{and}\ p_4(t) = \\
&\qquad (memo\_bool\ (name \mathbin{^\wedge} \text{`}testi\_p4\text{`})\ (\mathsf{And}_{fun}\ p_1\ p_2))\ t \\
&\quad \mathsf{and}\ ready(t) = \\
&\qquad (memo\_bool\ (name \mathbin{^\wedge} \text{`}testi\_ready\text{`})\ (\mathsf{Reg}_{fun}\ p_4\ regval))\ t \\
&\quad \mathsf{and}\ out(t) = \\
&\qquad (memo\_num\ (name \mathbin{^\wedge} \text{`}testi\_out\text{`})\ (\mathsf{Mux}_{fun}\ p_4\ l_2\ p_3))\ t \\
&\quad \mathsf{in}\ (out, ready)
\end{aligned}
$$

In the above definitions of Down_{fun} and Mult_{fun}, the function Reg_{fun} is used to model registers for storing *numbers* whereas in Test_{fun} it is used to model a register for storing *booleans*. This multiple use of the program Reg_{fun} to model repeated occurrences of the device Reg does not require naming to uniquely identify the particular devices since the program in this case is not involved in memoisation and there is no storage of values which need to be labelled to the device from which they were computed. The same applies for the multiple occurrences of Mux.

The overall implementation of Fact can now be redefined using the implementation definitions of Down, Mult and Test. The definitions are identical to those of Fact_{rel} and Fact_{fun} except that the implementation predicates and functions $\mathsf{Down\_Imp}_{rel}$, $\mathsf{Down\_Imp}_{fun}$, etc. replace the behavioural ones, Down_{rel}, Down_{fun}, etc. as shown in the definitions of $\mathsf{Fact\_Imp}_{rel}$ and $\mathsf{Fact\_Imp}_{fun}$ below.

$$
\begin{aligned}
&\mathsf{Fact\_Imp}_{rel}(i, out, ready) \equiv \\
&\quad \exists l_1\ l_2 .\ \mathsf{Down\_Imp}_{rel}(i, ready, l_1) \wedge \\
&\qquad \mathsf{Mult\_Imp}_{rel}(ready, l_1, l_2) \wedge \\
&\qquad \mathsf{Test\_Imp}_{rel}(ready, l_1, l_2, out)
\end{aligned}
$$

```
let Fact_Imp_fun name i ival_1 ival_2 ival_3 =
  letrec l_1(t) =
      (memo_num
          (name ^ 'fact_l1')
          (Down_Imp_fun (name ^ 'fact_down') i ready ival_1)) t
  and l_2(t) =
      (memo_num
          (name ^ 'fact_l2')
          (Mult_Imp_fun (name ^ 'fact_mult') ready l_1 ival_2)) t
  and out(t) =
      (memo_num
          (name ^ 'fact_out')
          (fst (Test_Imp_fun (name ^ 'fact_test') l_1 l_2 ival_3))) t
  and ready(t) =
      (memo_bool
          (name ^ 'fact_ready')
          (snd (Test_Imp_fun (name ^ 'fact_test') l_1 l_2 ival_3))) t
  in (out, ready)
```

The definitions of $\mathsf{Down\_Imp}_{rel}$, $\mathsf{Mult\_Imp}_{rel}$ and $\mathsf{Test\_Imp}_{rel}$ can be expanded in the definition of $\mathsf{Fact\_Imp}_{rel}$ as was done with the definitions of Down_{rel}, Mult_{rel} and Test_{rel} in the definition of Fact_{rel} in Section 7.2. This results in an expanded form of $\mathsf{Fact\_Imp}_{rel}$ which can be easily translated in exactly the same manner as Fact_{rel} and $\mathsf{Fact\_Imp}_{rel}$ were translated. The functional translation, however, is long and presents nothing of interest which has not already appeared in the previous translations. For this reason, the expanded cases of the Fact_Imp definitions are not presented here.

7.4 Example Simulations

The automatically derived programs described in the previous sections can be executed to show that the implementations model the intended behaviour of the devices. Only example simulations of the overall factorial machine are presented here. An example session to show how the ML programs can be generated from the HOL relations that model the implementation of the factorial machine, is given in Appendix B.

For example, take the first ten values on the input line i of the factorial machine to be:

$$4, 2, 5, 6, 9, 1, 3, 7, 8, 10$$

The functions:

- Fact_{fun}—the function modelling the implementation of Fact using the behavioural definitions of Down, Mult and Test as primitives;
- $\mathsf{Fact\_Imp}_{fun}$—the function modelling the implementation of Fact using the implementation definitions of Down, Mult and Test;

can be simulated by evaluating the expressions:

$$\mathsf{let}\ (out, ready) = \mathsf{Fact}_{fun}\ `Fact`\ i\ 0\ 0\ \mathsf{T}$$
$$\mathsf{let}\ (out', ready') = \mathsf{Fact\_Imp}_{fun}\ `Fact\_Imp`\ i\ 0\ 0\ \mathsf{T}$$

respectively. The initial values are consistent in both simulations, in this case 0, 0 and T. The results of the above simulations are sampled over the relevant time units as shown in the table below.

time	0	1	2	3	4	5	6	7	8	9	10	11	...
i	4	2	5	6	9	1	3	7	8	10			...
out	0	0	0	0	0	24	0	0	0	0	6	0	...
out'	0	0	0	0	0	24	0	0	0	0	6	0	...
$ready$	T	F	F	F	F	F	T	F	F	F	F	T	...
$ready'$	T	F	F	F	F	F	T	F	F	F	F	T	...

As expected, the values computed by both definitions are identical. Furthermore, the simulations show that the results conform to the expected behaviour of the factorial machine as specified in Section 7.1:

- $ready(0)$ is $true$, i.e. $t = 0$
- $i(0)$ is loaded, i.e. $n = 4$
- 4! is computed and appears on out at time 5, i.e. at $t+n+1$
- the value on out is 0 for times 0 through 4, i.e. t to $t+n$
- the value on $ready$ is F for times 1 through 5, i.e. $t+1$ to $t+n+1$
- $ready(6)$ is $true$ once again, i.e. at $t+n+2$
- the cycle repeats itself for $i(6)$

The times taken for the above computations are tabulated below alongside those for identical computations using the corresponding non-optimised functions. The times below are the *cpu* times (measured in seconds) taken on a SUN 3 computer to compute the first twelve values of the history functions *out*, *out'*, *ready* and *ready'*. The effectiveness of the optimisation is obvious from the results.

Runtime Statistics		
Output	Memoised	Non-Memoised
out	1.8*s*	14.3*s*
ready	0.2*s*	12.2*s*
Total	2.0*s*	26.5*s*
out'	10.6*s*	26.2*s*
ready'	0.6*s*	22.5*s*
Total	11.2*s*	48.7*s*

The computations were carried out in the same order as presented in the table, with the topmost statistics representing the first computation. The order is important only to explain the large difference in the times taken to compute the values of *out* and *ready*, and *out'* and *ready'* in the memoised computations. This is because several values of *ready* and *ready'* are calculated during the computation of *out* and *out'* respectively, and are memoised for later use. Thus, when calculating the values of *ready* and *ready'*, only those not already memoised need to be computed.

Chapter 8

Simulating a Computer

In this chapter we describe how the relational specifications of a simple computer are automatically translated into executable programs and show how these derived programs can be used to simulate the mechanisms of the computer.

The design of this general-purpose computer was invented by Mike Gordon in [17] where it was specified and verified using a formalism based on denotational semantics. Commonly referred to as 'Gordon's computer', it became a classic example in hardware specification and verification due to its appeal as a simple yet sufficiently realistic circuit. It has been specified and verified in LCF-LSM by Mike Gordon [19], in VERIFY by Harry Barrow [4] and in HOL by Jeffrey Joyce [34]. Martin Richards has written specifications of Gordon's computer in BSPL [56] while Daniel Weise has written specifications in a LISP-like language of a modified version of the computer [63]. Gordon's computer became the first formally verified computer to be fabricated when an 8-bit version was implemented as a 5000 transistor CMOS microchip as part of a project conducted by Jeffrey Joyce [35] at Calgary and Xerox Parc.

The relational definitions translated in this chapter are based on the HOL specifications presented in [34]. A brief description of Gordon's computer is first presented, followed by an account of various data types which are set up in ML to enable a direct translation of the HOL definitions which make use of such types. Both the specification models and the implementation models are then translated and the simulation of an example program is presented in the final section.

One of the main features of this chapter is the definition of a data path DATA presented in Section 8.4 which provides an excellent example of why non-memoised functional translations can be so inefficient. To explain this, of course, all the definitions used in the definition of DATA have to be explained. In fact, most of the definitions in both the specification and the implementation of the computer

are presented in this chapter. Many translation techniques have already been illustrated in the previous chapter but are discussed once again in the light of a bigger example.

8.1 Description of Gordon's Computer

A detailed description of Gordon's computer is given in [34]. A brief outline is presented below, however, to enable a full understanding of the formal specifications and their translations presented in the rest of the chapter.

At the register-transfer level, the target computer contains two registers, a 13-bit program counter and a 16-bit accumulator, and a random access memory which is addressed by 13-bit words each pointing to a 16-bit memory space.

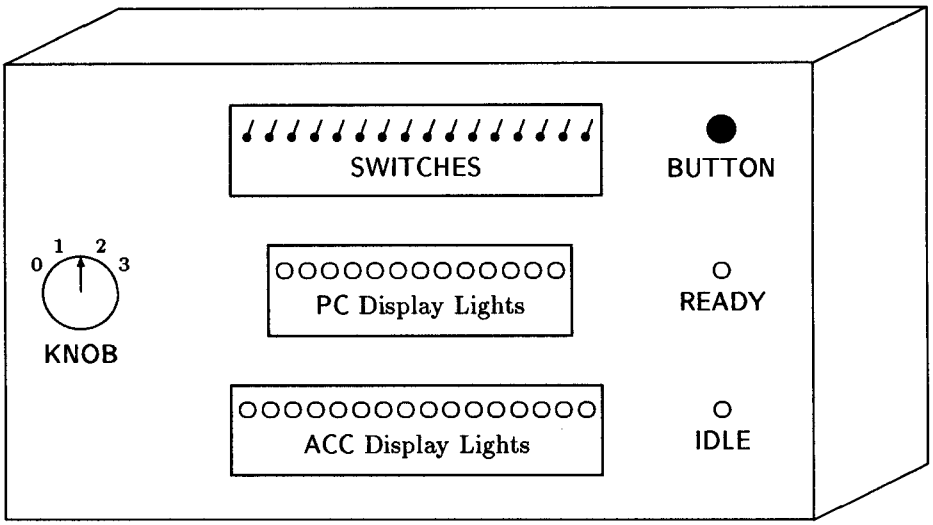


Figure 8.1: Front Panel of Gordon's Computer

Externally it has four sets of lights used to display output, and a set of buttons and switches which are used for input. Figure 8.1 is an illustration of the front panel of the computer which shows the four sets of output lights, namely:

- a set of 13 lights to display the contents of the program counter (PC).
- a set of 16 lights to display the contents of the accumulator (ACC).
- a light which goes on to indicate when the computer is idle (IDLE).
- a light which goes on to indicate the completion of a major state transition (READY).

and the three input mechanisms:

- a set of 16 switches to load data into the program counter or the accumulator (SWITCHES).
- a knob to select the type of instruction to execute (KNOB).
- a button used to interrupt the computer during program execution and make it idle, or if the computer is already idle, to execute the instruction selected by the knob (BUTTON).

Each switch or button can be either on or off; these two states are represented by the booleans *true* and *false* respectively. Thus, a sequence of switches or lights in some combination of on and off states is used to model sequences of binary digits.

The types of instructions to be executed are determined by the KNOB being in one of positions 0, 1, 2 or 3.

- position 0 = load PC
- position 1 = load ACC
- position 2 = store ACC at PC
- position 3 = start execution at PC

When the button is pushed and the computer is idle, if the knob is in position 0, the rightmost thirteen bits indicated by the switches are loaded into the program counter. If the knob is in position 1 then all sixteen bits are loaded into the accumulator instead. When the knob is in position 2, no input is read but the current contents of the accumulator are stored in memory at the location indicated by the contents of the program counter. The knob in position 3 starts the execution of a program (loaded in memory) at the location indicated by the contents of the program counter. During the execution of a program, the idle light remains off indicating that the computer is busy; the light going back on when execution of the program is terminated. If the button is pressed during the execution of the program then the program is interrupted and the idle light comes on again.

Programs are written using the following eight microinstructions:

- HALT—Terminates execution
- JMP x—Jump to address x

- JZR x—Jump to address x if ACC=0
- ADD x—Add contents of address to ACC
- SUB x—Subtract contents of address from ACC
- LD x—Load contents of address into ACC
- ST x—Store contents of ACC in memory at address
- SKIP—No operation

Each instruction consists of sixteen bits; the leftmost three denote the opcode, and the rightmost thirteen denote the address. For example,

001 000 000 000 0111

denotes the instruction

JMP 7

where 001 is the opcode for JMP and the address field has value 7. Further details of all the instructions are presented in [34].

Before showing the formal HOL specifications that represent the above behaviour and deriving their executable translations, it is necessary to explain how the representation of the HOL data types used to write definitions describing Gordon's computer can be represented in ML. Keeping the representations of data types consistent in the two formalisms enables a cleaner and easier translation.

8.2 Setting Up *wordn* Types in ML

Values denoting sequences of bits are represented in the HOL specifications as values of type *wordn*, where n is the number of bits represented by the particular type. For example, values stored in the program counter are represented by the type *word*13 and values stored in the accumulator are represented by the type *word*16.

The *wordn* types are defined in HOL as primitive data types and a number of axioms and theorems are defined and proved which describe their properties. The types are not, however, formally axiomatised; they are introduced in a rather ad-hoc manner with only certain properties being axiomatised, those required to conduct the correctness proofs of the computer [34]. A description of how *wordn* types can be formally axiomatised in the HOL logic is given in [40].

The HOL definitions modelling Gordon's computer use the following data types to represent n-bit words: *word*2, *word*3, *word*5, *word*13, *word*16 and *word*30, and the data types: *mem*5_30 and *mem*13_16 to represent the ROM (explained in Section 8.4) and the RAM respectively. Furthermore, there is also a *tri_wordn* data type defined for every *wordn* which allows the values on data lines to be floating.

In the rest of the chapter, values of type *wordn* are sometimes referred to as words while values of type *tri_wordn* are sometimes referred to as triwords.

The above mentioned data types must be translated into ML, along with certain functions that manipulate them. Unfortunately such translations cannot be done automatically while the type definitions are not properly defined because no regular procedure can be identified in their definitions. With a systematic formalism as presented in [40], however, it would be interesting to investigate whether an algorithm can be found to automatically translate the data type specifications into ML. At the time of this research, the proper *wordn* definitions were not yet implemented in HOL and so only the ad-hoc definitions in [34] were available.

Certain strategies were used, however, to define the data types manually. The type representation is very attractive and the strategy used to define them is promising for future research in the automatic translation of types. Below is a description of how the above data types were set up in ML.

Primitive HOL data types can be neatly modelled using abstract data types in ML. For example, the type of words handled by the accumulator, *word*16, can be defined as follows:

```
abstype word16 = bool list
with VAL16 w = val 16 (rep_word16 w)
and WORD16 n = abs_word16 (int_to_list 16 n)
and ARB16 = abs_word16 []
and BITS16 = (bits 16) o rep_word16
and NOT16 w = abs_word16 (bits 16 (map not (rep_word16 w)))
and OR16 v w =
    abs_word16 (bits 16 (word_or (rep_word16 v) (rep_word16 w)))
and AND16 v w =
    abs_word16 (bits 16 (word_and (rep_word16 v) (rep_word16 w)))
```

This type declaration introduces a new type *word*16 represented by the type *bool list*. It makes use of the two locally available functions *abs_word*16 and *rep_word*16 (see Chapter 2) to define a set of primitive functions for manipulating the new data type.

The function VAL16 takes an argument of type *word*16 and uses a previously defined function *val* to compute the integer representation of the 16-bit word. The function WORD16 takes an integer argument and converts it to a 16-bit word using a pre-defined function *int_to_list*, which fails if the integer represents a word larger than 16-bits long. The constant ARB16 represents an undefined value of type *word*16, and the function BITS16 simply returns the boolean list representation of a 16-bit word. The functions NOT16, OR16 and AND16 are functions for performing the logical operations of negation, disjunction and conjunction on the boolean representations of the bits making up 16-bit words.

The function (*bits n*) is used to check that the list representation of a *wordn* value contains n bits; it returns the list of bits if n bits are present and fails otherwise. Thus, since the undefined value ARBn should never be output, any attempt to evaluate it will fail. Its use is shown in the type definitions of triwords and in Section 8.4 when the implementation of the computer is described.

The properties of *word*16 are therefore captured in a single declaration by means of the representation of the type and the definitions of primitive functions. Type declarations similar to the one above are made for the rest of the *wordn* types required for specifying the computer. The corresponding *tri_wordn* data types are defined in a similar fashion. The type declaration for *tri_word*16 is:

abstype *tri_word*16 = *word*16 + *void*
with FLOAT16 = *abs_tri_word*16 (*inr* ())
and MK_TRI16 w = *abs_tri_word*16 (*inl* w)
and DEST_TRI16 w = (*outl* (*rep_tri_word*16 w)) ? ARB16
and U16 t_1 t_2 = *abs_tri_word*16
tri_word_union (*rep_tri_word*16 t_1)(*rep_tri_word*16 t_2)

where the new type is represented by the disjoint union of the types *word*16 and *void*.

The declaration defines FLOAT16 by injecting *void* into the right summand of the type representation and MK_TRI16(w) by injecting w into the left summand. Thus, a value of type *tri_word*16 can be created by FLOAT16 if the value is undefined or by MK_TRI16 if the value is a 16-bit word.

The function DEST_TRI16 attempts to project a defined *word*16 value out of the left summand representing the triword. If this fails (i.e. the triword is FLOAT16) then the undefined word ARB16 is returned. The reason why *fail* cannot be used instead of ARB16 is that the values on certain components of the computer (namely the bus) are sometimes floating during the execution of certain microinstructions. This would not be allowed if *fail* is used at this stage, because

the entire simulation will fail outright when the first floating value is encountered. The value ARB16 is useful because it represents an arbitrary value which will cause failure only if an attempt to evaluate it is made.

Finally, the function U16 takes the union of two triwords by using a function *tri_word_union*. If either of the triwords is floating, the result is the other triword; if the two triwords are equal then the result is either one of the values; and if the two triwords are both non-floating unequal values then the function fails.

The functions defining conjunction, disjunction and union of words ANDn, ORn and Un are infixed to make the notation easier.

The memory data types are also defined using abstract type definitions. For example, the random access memory is defined as a data structure of type *mem*13_16 which is defined by:

```
abstype mem13_16 = word13→word16
with STORE13 (w1:word13) (w2:word16) m =
        abs_mem13_16 (λa. (a=w1) ⇒ w2 | (rep_mem13_16 m a))
and  FETCH13 = rep_mem13_16
and  EMPTY13_16 = abs_mem13_16 (λa:word13. WORD16 0)
```

Once again, a set of primitive functions are defined along with the new type represented by functions from values of type *word*13 to values of type *word*16. The constant EMPTY13_16 defines a memory with all contents set to zero. The function (FETCH13 m *addr*) evaluates the function representing the memory m at the location specified by *addr*. The function STORE13 takes three arguments: an address (of type *word*13), a value to be stored (of type *word*16) and a memory in which the value is to be stored (of type *mem*13_16). The function STORE13 simulates storing values in memory locations by 'updating' the function which represents the memory in such a way that the stored value is returned (the new contents) when the function is evaluated for the particular argument (the address). In this way, values stored in memory can be overwritten since the function evaluates the first result matching an address, which corresponds to the last value stored.

The type *mem*5_30 is defined in an identical way using *word*5 instead of *word*13 and *word*30 instead of *word*16. In fact, all primitive HOL types used to specify the computer which were not already defined in ML are straightforward to define using abstract type declarations.

Of course, other functions which act on values of these new types can now be defined. For example, a function to increment a 13-bit word can be defined by:

```
let INC13 w = WORD13 ((VAL13 w) + 1)
```

and a function for testing whether a 16-bit word is equal to zero can be defined as:

$$\mathsf{let\ ISZERO16}\ w = (\mathsf{VAL16}\ w) = 0$$

Non-primitive HOL data types are much easier defined using ML type abbreviations. For example, the type declarations:

$$\mathsf{typeabbrev}\ time = int$$
$$\mathsf{typeabbrev}\ sig = time \rightarrow bool$$

define a type *time* represented by integers and a type *sig* represented by functions from time to booleans.

Having covered how the HOL types can be defined in ML, the next section presents the formal specifications of the target machine described in Section 8.1 along with their derived programs.

8.3 The Target Machine

The behaviour of Gordon's computer described in Section 8.1 is formalised by the definition of the predicate COMPUTER shown below.

$$\begin{array}{l}
\text{“}\forall\ knob\ button\ switches\ mem\ pc\ acc\ idle.\\
\quad \mathsf{COMPUTER}_{rel}(knob, button, switches, mem, pc, acc, idle) \equiv\\
\quad\quad (mem(t{+}1), pc(t{+}1), acc(t{+}1), idle(t{+}1)) =\\
\quad\quad\quad (idle(t) \Rightarrow\\
\quad\quad\quad\quad (button(t) \Rightarrow\\
\quad\quad\quad\quad\quad ((\mathsf{VAL2}\ (knob(t)) = 0) \Rightarrow\\
\quad\quad\quad\quad\quad\quad (mem(t), \mathsf{CUT16_13}(switches(t)), acc(t), \mathsf{T})\ |\\
\quad\quad\quad\quad\quad (\mathsf{VAL2}\ (knob(t)) = 1) \Rightarrow\\
\quad\quad\quad\quad\quad\quad (mem(t), pc(t), switches(t), \mathsf{T})\ |\\
\quad\quad\quad\quad\quad (\mathsf{VAL2}\ (knob(t)) = 2) \Rightarrow\\
\quad\quad\quad\quad\quad\quad (\mathsf{STORE13}\ pc(t)\ acc(t)\ mem(t)), pc(t), acc(t), \mathsf{T})\ |\\
\quad\quad\quad\quad\quad\quad (mem(t), pc(t), acc(t), \mathsf{F}))\ |\\
\quad\quad\quad\quad\quad (mem(t), pc(t), acc(t), \mathsf{T}))\ |\\
\quad\quad\quad\quad (button(t) \Rightarrow (mem(t), pc(t), acc(t), \mathsf{T})\ |\\
\quad\quad\quad\quad\quad\quad\quad \mathsf{EXECUTE}(mem(t), pc(t), acc(t))))\text{”}
\end{array}$$

The parameters of the specification denote the memory, the input ports, and the output ports of the computer; all represented by history functions as listed in the table overleaf with their corresponding types. The definition recursively equates the values at time $t{+}1$ of the memory, the program counter, the accumulator, and the idle light with an expression involving the values at time t of *knob*, *button*, *switches*, *mem*, *pc*, *acc*, and *idle*.

Representation of Input and Output Ports		
Port	History Function	Type
memory	*mem*	*time*→*mem*13_16
BUTTON	*button*	*time*→*bool*
IDLE	*idle*	*time*→*bool*
SWITCHES	*switches*	*time*→*word*16
ACC	*acc*	*time*→*word*16
PC	*pc*	*time*→*word*13
KNOB	*knob*	*time*→*word*2

This definition is a straightforward model of the behaviour of the computer. It uses a conditional statement to define the different actions of the computer for the different situations determined by the values of *idle*, *button* and *knob*. Each terminal branch of the conditional represents a single target machine operation.

The function VAL2 returns the integer value of a two-bit word and is used to check whether the value on *knob* at time t is set to 0, 1, 2 or 3. The function CUT16_13 returns a thirteen-bit word consisting of the thirteen least significant bits of a sixteen-bit word. It is used to load the thirteen rightmost bits set up on the switches into the program counter when the knob is set to zero.

When the value of *idle* at time t is F (i.e. the computer is executing a program) and the value of *button* is F (i.e. the computer is not interrupted) then the next values of *mem*, *pc*, *acc* and *idle* are determined by a function EXECUTE which describes the execution of a single target level instruction. The definition of EXECUTE is as follows:

"$\forall$ *mval pcval accval.*
EXECUTE(*mval*, *pcval*, *accval*) =
let *op* = VAL3(OPCODE(FETCH13 *mval pcval*)) in
let *addr* = CUT16_13(FETCH13 *mval pcval*) in
(*op*=0) ⇒ (*mval*, *pcval*, *accval*, T) |
(*op*=1) ⇒ (*mval*, *addr*, *accval*, F) |
(*op*=2) ⇒ ((VAL16 *accval*)=0 ⇒ (*mval*, *addr*, *accval*, F) |
(*mval*, INC13 *pcval*, *accval*, F)) |
(*op*=3) ⇒ (*mval*, INC13 *pcval*, ADD16 *accval* (FETCH13 *mval addr*), F) |
(*op*=4) ⇒ (*mval*, INC13 *pcval*, SUB16 *accval* (FETCH13 *mval addr*), F) |
(*op*=5) ⇒ (*mval*, INC13 *pcval*, FETCH13 *mval addr*, F) |
(*op*=6) ⇒ (STORE13 *addr accval mval*, INC13 *pcval*, *accval*, F) |
(*mval*, INC13 *pcval*, *accval*, F)"

The function FETCH13 is used to fetch the contents of the memory stored at the location specified by the 13-bit program counter. The opcode *op* and the

address *addr* are extracted from the 16-bit word fetched from memory and are used to evaluate the next values of memory, program counter, accumulator and idle. An eight branch conditional is used, one branch for every possible value of the opcode. Each conditional evaluates the instruction corresponding to the particular opcode. For example, when the value of *op* is 0 the instruction to be executed is the HALT command, and so the current values in the memory, the program counter, and the accumulator are retained while the value on the *idle* line is changed to T to indicate the computer has finished executing the program.

The two definitions above fully model the behaviour of the target machine at the register-transfer level. Since EXECUTE is already defined as a HOL function (rather than a relation), there is little difficulty in translating it to the corresponding ML function. In fact, since EXECUTE has no recursion, no optimisation is needed and so the ML function looks almost identical to the HOL function. For this reason, the ML definition of EXECUTE is not presented here and will merely be referred to as EXECUTE′ when required.

The COMPUTER definition is translated using techniques similar to those presented in Chapter 7. The inputs are *knob*, *button* and *switches*, and the outputs are *mem*, *pc*, *acc* and *idle*.

$$
\begin{array}{l}
\mathsf{let}\ \mathsf{COMPUTER}_{fun}\ name\ knob\ button\ switches\ tpl\_val = \\
\quad \mathsf{letrec}\ mem(t') = (memo_{mem}\ (name\ \hat{}\ \text{`c\_mem`})\ (\lambda t.\ (fst\ \mathsf{exp})))\ t' \\
\quad \mathsf{and}\ pc(t') = (memo_{wd13}\ (name\ \hat{}\ \text{`c\_pc`})\ (\lambda t.\ (fst\ (snd\ \mathsf{exp}))))\ t' \\
\quad \mathsf{and}\ acc(t') = (memo_{wd16}\ (name\ \hat{}\ \text{`c\_acc`})\ (\lambda t.\ (fst\ (snd\ (snd\ \mathsf{exp})))))\ t' \\
\quad \mathsf{and}\ idle(t') = (memo_{bool}\ (name\ \hat{}\ \text{`c\_idle`})\ (\lambda t.\ (snd\ (snd\ (snd\ \mathsf{exp})))))\ t' \\
\quad \mathsf{in}\ (mem, pc, acc, idle)
\end{array}
$$

The notation exp used in the definition above is an abbreviation for the expression:

$$
\begin{array}{l}
(t{=}0) \Rightarrow \\
\quad tpl\_val\ \mid \\
\quad (idle(t{-}1) \Rightarrow \\
\quad\quad (button(t{-}1) \Rightarrow \\
\quad\quad\quad ((\mathsf{VAL2}\ (knob(t{-}1)) = 0) \Rightarrow \\
\quad\quad\quad\quad (mem(t{-}1), \mathsf{CUT16\_13}(switches(t{-}1)), acc(t{-}1), true)\ \mid \\
\quad\quad\quad\ (\mathsf{VAL2}\ (knob(t{-}1)) = 1) \Rightarrow \\
\quad\quad\quad\quad (mem(t{-}1), pc(t{-}1), switches(t{-}1), true)\ \mid \\
\quad\quad\quad\ (\mathsf{VAL2}\ (knob(t{-}1)) = 2) \Rightarrow \\
\quad\quad\quad\quad (\mathsf{STORE13}\ pc(t{-}1)\ acc(t{-}1)\ mem(t{-}1)), pc(t{-}1), acc(t{-}1), true)\ \mid \\
\quad\quad\quad\quad (mem(t{-}1), pc(t{-}1), acc(t{-}1), false))\ \mid \\
\quad\quad\quad (mem(t{-}1), pc(t{-}1), acc(t{-}1), true))\ \mid \\
\quad\quad (button(t{-}1) \Rightarrow (mem(t{-}1), pc(t{-}1), acc(t{-}1), true)\ \mid \\
\quad\quad\quad\quad\quad\quad \mathsf{EXECUTE}'(mem(t{-}1), pc(t{-}1), acc(t{-}1))))
\end{array}
$$

The function $\mathsf{COMPUTER}_{fun}$ recursively computes the output functions which represent the output mechanisms and the random access memory. Although the memory is an internal mechanism, it is included among the outputs of the definition, since this makes it possible for the contents of the memory to be examined when simulating the functional definitions. Optimisation is introduced via four different memo-functions; one for each type of recursive function. The selectors *fst* and *snd* are once again used to split the tuple of recursive functions into separate computations.

An example simulation of the target computer is given in Section 8.5 using the definitions above. These are compared with simulations of the implementation of the computer specified in the next section.

8.4 The Host Machine

The implementation of Gordon's computer at the register-transfer level, referred to as the 'host machine', is shown in Figure 8.2. In addition to the random access memory, the program counter and the accumulator, it has a number of other registers, a bus, several bus drivers, a read-only memory, an arithmetic and logic unit, and a decoder.

To model the various registers of the computer two registers are first defined as primitives, one to store thirteen-bit words and one to store sixteen-bit words. Their HOL definitions are as follows:

$$\text{“}\mathsf{REG13}_{rel}(i, ld, o) \equiv \forall t.\ o(t{+}1) = (ld(t) \Rightarrow \mathsf{CUT16\_13}(i(t)) \mid o(t))\text{”}$$

and

$$\text{“}\mathsf{REG16}_{rel}(i : num{\rightarrow}word16, ld, o) \equiv \forall t.\ o(t{+}1) = (ld(t) \Rightarrow i(t) \mid o(t))\text{”}$$

These two specifications are partial and recursive. In both cases, the $word16$ value on i is sampled at time t. In the case of the thirteen-bit register, the least significant thirteen bits are extracted using the function CUT16_13 and output on o at time $t{+}1$ if the value on ld at time t is true. In the case of the sixteen-bit register, the entire input word is passed on to the output o at time $t{+}1$ if the value on ld at time t is true. In both cases, the output at time $t{+}1$ is set to its previous value at time t when the value on ld at time t is false.

The translations of the above two relations use techniques which were detailed in previous chapters. The functions $\mathsf{REG13}_{fun}$ and $\mathsf{REG16}_{fun}$ shown below, are

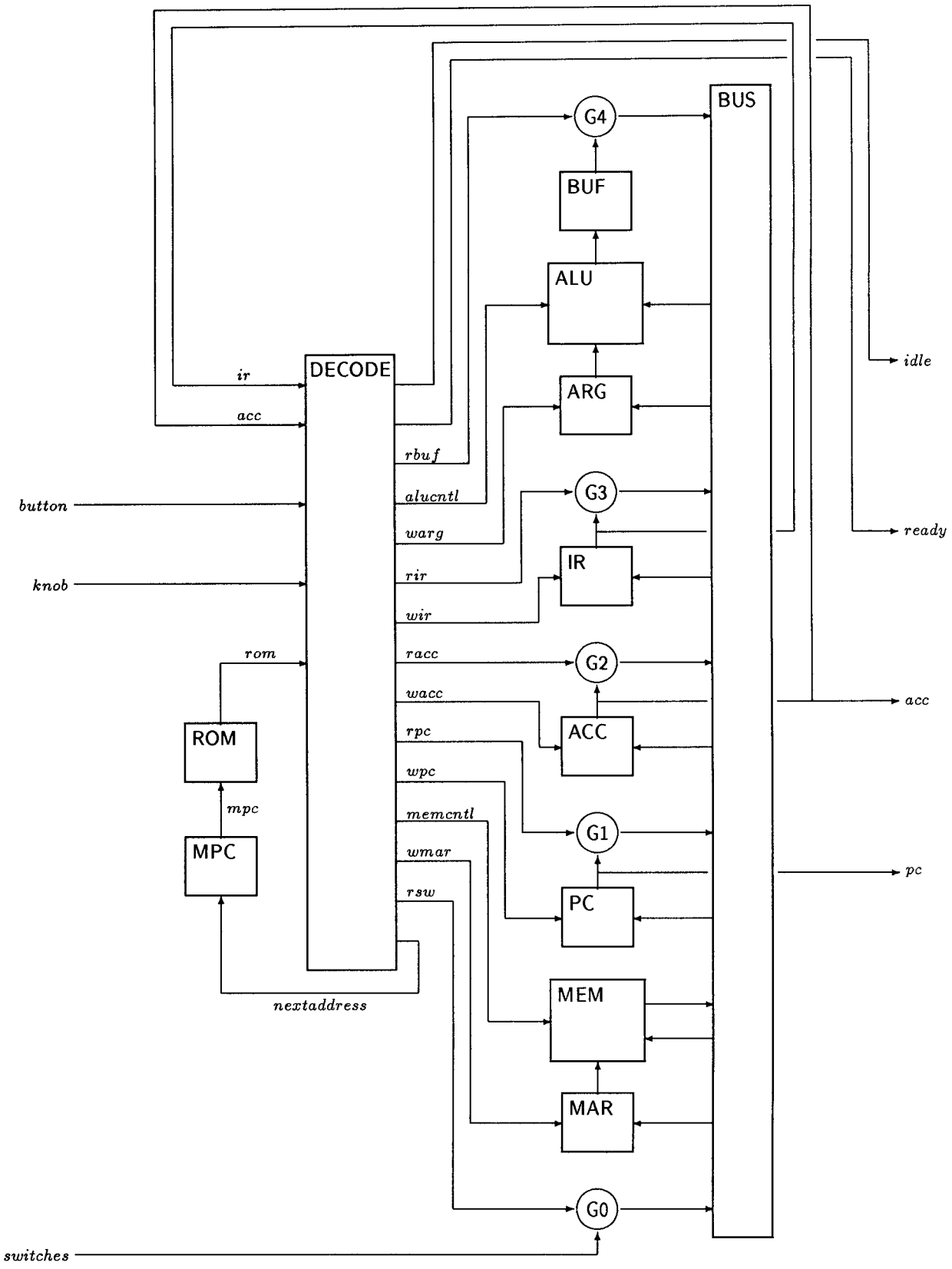


Figure 8.2: Implementation of Gordon's Computer

optimised and make use of initial values since the relations are only partially defined.

$$\begin{array}{l}\mathsf{let}\ \mathsf{REG13}_{fun}\ name\ i\ ld\ o\_val = \\ \quad \mathsf{letrec}\ o(t') = \\ \qquad (memo_{wd13} \\ \qquad\quad (name\ \hat{}\ \text{`}reg13\_o\text{`}) \\ \qquad\quad (\lambda t.\ (t{=}0) \Rightarrow o\_val \mid ld(t{-}1) \Rightarrow \mathsf{CUT16\_13}(i(t{-}1)) \mid o(t{-}1)))\ t' \\ \quad \mathsf{in}\ o\end{array}$$

$$\begin{array}{l}\mathsf{let}\ \mathsf{REG16}_{fun}\ name\ i\ ld\ o\_val = \\ \quad \mathsf{letrec}\ o(t') = \\ \qquad (memo_{wd16} \\ \qquad\quad (name\ \hat{}\ \text{`}reg16\_o\text{`}) \\ \qquad\quad (\lambda t.\ (t{=}0) \Rightarrow o\_val \mid ld(t{-}1) \Rightarrow i(t{-}1) \mid o(t{-}1)))\ t' \\ \quad \mathsf{in}\ o\end{array}$$

The implementation of the computer has two thirteen-bit registers: the program counter PC and the memory address register MAR, and three sixteen-bit registers: the accumulator ACC, the instruction register IR and a register ARG used for storing arguments to be processed by the arithmetic and logic unit. These registers are defined as instances of the thirteen and sixteen bit registers defined above. The relational definitions and their functional translations are similar for all the registers and so only the specifications of the program counter PC are shown here.

$$\text{“}\mathsf{PC}_{rel}(i, ld, o) \equiv \mathsf{REG16}_{rel}(i, ld, o)\text{”}$$

$$\mathsf{let}\ \mathsf{PC}_{fun}\ name\ i\ ld\ oval = \mathsf{let}\ o = (\mathsf{REG16}_{fun}\ (name\ \hat{}\ \text{`}pc\_o\text{`})\ i\ ld\ oval)\ \mathsf{in}\ o$$

The register models are good examples of cases in which naming of devices is required. Since they all share the same definitions of $\mathsf{REG13}_{fun}$ and $\mathsf{REG16}_{fun}$, these registers must be distinguished when storing and retrieving values from respective memo-tables.

Another register BUF is defined slightly differently in that it has no selector input to determine whether the register retains its current value or loads a new input. The register BUF merely acts as a delay mechanism used to store the output of the ALU for one unit of time.

Thus, BUF is defined relationally as:

$$\text{“}\mathsf{BUF}_{rel}(alu, buf) \equiv \forall t.\ buf(t{+}1) = alu(t)\text{”}$$

which is translated to:

$$\mathsf{let}\ \mathsf{BUF}_{fun}\ alu\ buf\_val = \mathsf{let}\ buf(t) = (t{=}0) \Rightarrow buf\_val \mid alu(t{-}1)\ \mathsf{in}\ buf$$

There are five tri-state bus drivers G0, G1, G2, G3 and G4 used to control the data that goes onto the sixteen-bit wide bus. Once again, two primitive gates are defined to model *word*13 input gates and *word*16 input gates. These gates, GATE13 and GATE16, convert thirteen and sixteen bit words to the corresponding 16-bit triwords.

$$\text{“}\mathsf{GATE13}_{rel}(i, cntl, o) \equiv$$
$$\forall t.\ o(t) = cntl(t) \Rightarrow \mathsf{MK\_TRI16}\ (\mathsf{PAD13\_16}\ i(t)) \mid \mathsf{FLOAT16}\text{”}$$

$$\text{“}\mathsf{GATE16}_{rel}(i, cntl, o) \equiv$$
$$\forall t.\ o(t) = cntl(t) \Rightarrow \mathsf{MK\_TRI16}\ i(t) \mid \mathsf{FLOAT16}\text{”}$$

In the case of GATE13, the *word*13 input is padded up to a 16-bit word using the function PAD13_16. In both cases, the 16-bit word is converted to a triword and output if the value on *cntl* is true. If the value on *cntl* is false, the undefined value FLOAT16 is output. There is no delay modelled in these gates. The derived programs are therefore:

$$\mathsf{let}\ \mathsf{GATE13}_{fun}\ i\ cntl =$$
$$\quad \mathsf{let}\ o(t) = (cntl(t) \Rightarrow \mathsf{MK\_TRI16}\ (\mathsf{PAD13\_16}\ i(t)) \mid \mathsf{FLOAT16})\ \mathsf{in}\ o$$

and

$$\mathsf{let}\ \mathsf{GATE16}_{fun}\ i\ cntl =$$
$$\quad \mathsf{let}\ o(t) = (cntl(t) \Rightarrow \mathsf{MK\_TRI16}\ i(t) \mid \mathsf{FLOAT16})\ \mathsf{in}\ o$$

The gates G0, G2, G3 and G4 are defined as instances of GATE16 above while G1 is the only instance of GATE13. Once again, the definitions are all very similar and so only the definitions for G0 are given below.

$$\text{“}\mathsf{G0}_{rel}(i, cntl, o) \equiv \mathsf{GATE16}_{rel}(i, cntl, o)\text{”}$$

$$\mathsf{let}\ \mathsf{G0}_{fun}\ i\ cntl = \mathsf{let}\ o = (\mathsf{GATE16}_{fun}\ i\ cntl)\ \mathsf{in}\ o$$

The memory device is modelled by a device MEM which takes three inputs: a two-bit wide control signal *memcntl*, the output of the memory address register *mar*, and the value on the bus *bus*. MEM returns one output line *mout*, which writes the fetched contents from memory directly to the bus. In addition, the actual memory representation *mem* is parameterised and is also treated as an output. The relational definition of MEM is shown below.

```
"MEM_rel(mem, mar, bus, memcntl, mout) ≡
  ∀t.
   (mout(t) =
     (VAL2 memcntl(t)) = 1 ⇒ MK_TRI16 (FETCH13 mem(t) mar(t))
                           | FLOAT16) ∧
   (mem(t+1) =
     (VAL2 memcntl(t)) = 2 ⇒ (STORE13 mar(t) bus(t) mem(t))
                           | mem(t))"
```

Once again, the definition is recursive. The contents of the memory are changed only when the value on *memcntl* is equivalent to 2. In this case, the value on the bus is stored in memory at the address specified by the memory address register. In other states, the memory stays the same. When the value on *memcntl* is equivalent to 1, the value stored in memory at the address specified by *mar* is fetched and written to the bus as a triword via *mout*. In other cases, FLOAT16 is written to the bus. The derived program modelling MEM is shown below:

```
let MEM_fun name mar bus memcntl mem_val =
  letrec mem(t') =
       (memo_mem
          (name ^ 'mem_m')
          (λt. (t=0) ⇒ mem_val |
               (VAL2 memcntl(t−1)) = 2 ⇒
                    (STORE13 mar(t−1) bus(t−1) mem(t−1)) |
               mem(t−1))) t'
  and mout(t') =
       (memo_tri16
          (name ^ 'mem_out')
          (λt. (VAL2 memcntl(t)) = 1 ⇒
                    MK_TRI16 (FETCH13 mem(t) mar(t)) |
                    FLOAT16)) t'
  in (mem, mout)
```

The arithmetic and logic unit is also defined as a primitive at this level of description. Three arithmetic functions are performed by the ALU on sixteen-bit words, namely incrementation, addition, and subtraction. The two-bit input control line *alucntl* determines which function is performed on the data present on lines *arg* and *bus*. The result of an ALU computation is output on line *alu* at the same instant of time. The relational definition is shown below:

```
"ALU_rel(arg, bus, alucntl, alu) ≡
     ∀t. alu(t) = (VAL2 alucntl(t)) = 0 ⇒ bus(t) |
                  (VAL2 alucntl(t)) = 1 ⇒ INC16 bus(t) |
                  (VAL2 alucntl(t)) = 2 ⇒ ADD16 arg(t) bus(t) |
                                          SUB16 arg(t) bus(t)"
```

The specification is total and non-recursive. The functional translation is therefore straightforward.

$$
\begin{array}{l}
\mathsf{let}\ \mathsf{ALU}_{fun}\ arg\ bus\ alucntl = \\
\quad \mathsf{let}\ alu(t) = (\mathsf{VAL2}\ alucntl(t)) = 0 \Rightarrow bus(t)\ \mid \\
\qquad\qquad\qquad\quad (\mathsf{VAL2}\ alucntl(t)) = 1 \Rightarrow \mathsf{INC16}\ bus(t)\ \mid \\
\qquad\qquad\qquad\quad (\mathsf{VAL2}\ alucntl(t)) = 2 \Rightarrow \mathsf{ADD16}\ arg(t)\ bus(t)\ \mid \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{SUB16}\ arg(t)\ bus(t) \\
\quad \mathsf{in}\ alu
\end{array}
$$

All the devices described so far, except BUF, either read from or write to the BUS. The five selectively loadable registers and the ALU read sixteen-bit words from the BUS, the five tri-state bus drivers write sixteen-bit triwords to the BUS, and the memory both reads from and writes to the BUS. Hence, the BUS device takes six triwords of type $triword16$ as inputs and returns a word of type $word16$ as output. Relationally, BUS is modelled as:

$$
\begin{array}{l}
\text{“}\mathsf{BUS}_{rel}(mout, g_0, g_1, g_2, g_3, g_4, bus) \equiv \\
\quad \forall t.\ bus(t) = \\
\qquad \mathsf{DEST\_TRI16} \\
\qquad\quad (mout(t)\ \mathsf{U16}\ g_0(t)\ \mathsf{U16}\ g_1(t)\ \mathsf{U16}\ g_2(t)\ \mathsf{U16}\ g_3(t)\ \mathsf{U16}\ g_4(t))\text{”}
\end{array}
$$

where U16 is the infix function for merging two triwords (see Section 8.2). The derived program is almost identical:

$$
\begin{array}{l}
\mathsf{let}\ \mathsf{BUS}_{fun}\ mout\ g_0\ g_1\ g_2\ g_3\ g_4\ = \\
\quad \mathsf{let}\ bus(t) = \\
\qquad \mathsf{DEST\_TRI16} \\
\qquad\quad (mout(t)\ \mathsf{U16}\ g_0(t)\ \mathsf{U16}\ g_1(t)\ \mathsf{U16}\ g_2(t)\ \mathsf{U16}\ g_3(t)\ \mathsf{U16}\ g_4(t)) \\
\quad \mathsf{in}\ bus
\end{array}
$$

The data path of the implementation of the computer can be specified by structuring all the devices described so far as shown in Figure 8.2. The relational definition which models the data path is shown overleaf, together with its functional translation.

The functional translation of DATA_{rel} is a good example for demonstrating the effect of memoisation. The function DATA_{fun} takes several inputs and returns a tuple of recursive functions as outputs. If no memoisation is done the function is extremely inefficient because it involves too many repetitions of recursive calculations.

For example, the evaluation of the value on the bus at time t, $bus(t)$, requires $mout(t)$ and $g_0(t), \ldots, g_4(t)$. Now the computations of g_1, g_2, g_3 and g_4 at time t require the computations of $pc(t)$, $acc(t)$, $ir(t)$ and $buf(t)$ respectively, each requiring, among other data, the value for $bus(t-1)$. Furthermore, $mout(t)$

requires $mem(t)$ and $mar(t)$, which in turn also require $bus(t-1)$. Thus, to compute $bus(t)$, $bus(t-1)$ is calculated six times. At time t, therefore, $bus(t-n)$ is calculated a maximum of 6^n times, i.e. $bus(t-10)$ could be calculated well over 60.5 million times!

$$
\begin{array}{l}
\text{“}\mathsf{DATA}_{rel}(switches, rsw, wmar, memcntl, wpc, rpc, wacc, racc, wir, \\
\qquad rir, warg, alucntl, rbuf, mem, mar, pc, acc, ir, arg, buf) \equiv \\
\quad \exists\, g_0\ g_1\ g_2\ g_3\ g_4\ mout\ alu\ bus. \\
\qquad \mathsf{MEM}_{rel}(mem, mar, bus, memcntl, mout) \wedge \\
\qquad \mathsf{MAR}_{rel}(bus, wmar, mar) \wedge \\
\qquad \mathsf{PC}_{rel}(bus, wpc, pc) \wedge \\
\qquad \mathsf{ACC}_{rel}(bus, wacc, acc) \wedge \\
\qquad \mathsf{IR}_{rel}(bus, wir, ir) \wedge \\
\qquad \mathsf{ARG}_{rel}(bus, warg, arg) \wedge \\
\qquad \mathsf{BUF}_{rel}(alu, buf) \ \wedge \\
\qquad \mathsf{G0}_{rel}(switches, rsw, g_0) \wedge \\
\qquad \mathsf{G1}_{rel}(pc, rpc, g_1) \wedge \\
\qquad \mathsf{G2}_{rel}(acc, racc, g_2) \wedge \\
\qquad \mathsf{G3}_{rel}(ir, rir, g_3) \wedge \\
\qquad \mathsf{G4}_{rel}(buf, rbuf, g_4) \wedge \\
\qquad \mathsf{ALU}_{rel}(arg, bus, alucntl, alu) \wedge \\
\qquad \mathsf{BUS}_{rel}(mout, g_0, g_1, g_2, g_3, g_4, bus)\text{”}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{let}\ \mathsf{DATA}_{fun}\ name\ switches\ rsw\ wmar\ memcntl\ wpc \\
\qquad\qquad rpc\ wacc\ racc\ wir\ rir\ warg\ alucntl\ rbuf \\
\qquad\qquad memval\ marval\ pcval\ accval\ irval\ argval\ bufval = \\
\quad \mathsf{let}\ g_0 = (\mathsf{G0}_{fun}\ switches\ rsw)\ \mathsf{in} \\
\quad \mathsf{letrec}\ mem(t) = \\
\qquad memo_{mem}\ \text{`}n_1\text{`}\ (fst\ (\mathsf{MEM}_{fun}\ \text{`}n_{15}\text{`}\ mar\ bus\ memcntl\ memval))\ t \\
\quad \mathsf{and}\ mout(t) = \\
\qquad memo_{tri16}\ \text{`}n_2\text{`}\ (snd\ (\mathsf{MEM}_{fun}\ \text{`}n_{15}\text{`}\ mar\ bus\ memcntl\ memval))\ t \\
\quad \mathsf{and}\ mar(t) = memo_{wd13}\ \text{`}n_3\text{`}\ (\mathsf{MAR}_{fun}\ \text{`}n_{16}\text{`}\ marval\ bus\ wmar)\ t \\
\quad \mathsf{and}\ pc(t) = memo_{wd13}\ \text{`}n_4\text{`}\ (\mathsf{PC}_{fun}\ \text{`}n_{17}\text{`}\ pcval\ bus\ wpc)\ t \\
\quad \mathsf{and}\ acc(t) = memo_{wd16}\ \text{`}n_5\text{`}\ (\mathsf{ACC}_{fun}\ \text{`}n_{18}\text{`}\ accval\ bus\ wacc)\ t \\
\quad \mathsf{and}\ ir(t) = memo_{wd16}\ \text{`}n_6\text{`}\ (\mathsf{IR}_{fun}\ \text{`}n_{19}\text{`}\ irval\ bus\ wir)\ t \\
\quad \mathsf{and}\ arg(t) = memo_{wd16}\ \text{`}n_7\text{`}\ (\mathsf{ARG}_{fun}\ \text{`}n_{20}\text{`}\ argval\ bus\ warg)\ t \\
\quad \mathsf{and}\ buf(t) = memo_{wd16}\ \text{`}n_8\text{`}\ (\mathsf{BUF}_{fun}\ bufval\ alu)\ t \\
\quad \mathsf{and}\ g_1(t) = memo_{tri16}\ \text{`}n_9\text{`}\ (\mathsf{G1}_{fun}\ pc\ rpc)\ t \\
\quad \mathsf{and}\ g_2(t) = memo_{tri16}\ \text{`}n_{10}\text{`}\ (\mathsf{G2}_{fun}\ acc\ racc)\ t \\
\quad \mathsf{and}\ g_3(t) = memo_{tri16}\ \text{`}n_{11}\text{`}\ (\mathsf{G3}_{fun}\ ir\ rir)\ t \\
\quad \mathsf{and}\ g_4(t) = memo_{tri16}\ \text{`}n_{12}\text{`}\ (\mathsf{G4}_{fun}\ buf\ rbuf)\ t \\
\quad \mathsf{and}\ alu(t) = memo_{wd16}\ \text{`}n_{13}\text{`}\ (\mathsf{ALU}_{fun}\ arg\ bus\ alucntl)\ t \\
\quad \mathsf{and}\ bus(t) = memo_{tri16}\ \text{`}n_{14}\text{`}\ (\mathsf{BUS}_{fun}\ mout\ g_0\ g_1\ g_2\ g_3\ g_4)\ t \\
\quad \mathsf{in}\ (mem, mar, pc, acc, ir, arg, buf)
\end{array}
$$

Without memoisation, each time a value is computed at a time t, the computation is recursive all the way down to time 0. For a large t, the effect of recalculating the same functions over the entire time scale is disastrous. Memoisation provides a good solution to the problem since it enables recursive functions to 'remember' previously computed values.

In the function DATA_{fun} above, the parameter *name* is a value of type *string*, the parameters ending in *val* represent initial values, and the rest of the parameters are history functions representing the various input data lines. The strings $`n_1` \ldots `n_{20}`$ are abbreviations for names obtained by concatenating the value of *name* to a uniquely generated string.

The three devices in the implementation of Gordon's computer shown in Figure 8.2 which have not yet been specified are MPC, ROM and DECODE.

The microcode program counter MPC is another non selectively-loadable register and is defined in almost the same way as BUF. The only difference is that the inputs and outputs to MPC are represented by functions of type $time \rightarrow word5$. The register is used to store a 5-bit address for the read-only memory.

The read-only memory ROM outputs the 30-bit word stored in the microcode at the address specified by the MPC. The microcode is addressed by 5-bit words, each pointing to a 30-bit instruction. The HOL representation of the microcode is generated by a function as a set of axioms which relate the contents of the memory to their corresponding address [34]. In ML the microcode is defined as a function which takes an argument of type $word5$ (the address) and returns a value of type $word30$ (the contents). The function is parameterised in the definition of ROM which makes use of the primitive function FETCH5 to fetch the contents from the microcode. The definition of ROM and its translation are as follows:

$$\text{“}\mathsf{ROM}_{rel}\ mcode\ (mpc, rom) \equiv \forall t.\ rom(t) = \mathsf{FETCH5}\ mcode\ mpc(t)\text{”}$$

$$\begin{array}{l}\mathsf{let}\ \mathsf{ROM}_{fun}\ mcode\ mpc = \\ \quad \mathsf{let}\ rom(t) = \mathsf{FETCH5}\ mcode\ mpc(t)\ \mathsf{in}\ rom\end{array}$$

The decode unit DECODE reads in the instruction from ROM and decodes it into the relevant signals which control the operation of the data path DATA. It also computes the next address to index the ROM by examining the values on *knob*, *button*, *acc* and *ir*.

The specification of DECODE and its translation are rather long and are therefore omitted here. As can be seen from [34], though, the definition is purely combinational and so its translation is straightforward. In the rest of this section,

the relational and functional specifications will be referred to as DECODE_{rel} and DECODE_{fun} respectively.

The three devices MPC, ROM and DECODE can now be grouped together to form a top-level component of the computer implementation. This component is called the control unit and is relationally specified using the predicate $\mathsf{CONTROL}_{rel}$ below:

$$
\begin{array}{l}
\text{“}\mathsf{CONTROL}_{rel} \\
\quad mcode \\
\quad (knob, button, acc, ir, rsw, wmar, memcntl, wpc, rpc, \\
\quad\ wacc, racc, wir, rir, warg, alucntl, rbuf, ready, idle) \equiv \\
\quad \exists\, mpc\ rom\ nextaddress. \\
\quad\ \mathsf{ROM}_{rel}\ mcode\ (mpc, rom) \wedge \\
\quad\ \mathsf{MPC}_{rel}\ (nextaddress, mpc) \wedge \\
\quad\ \mathsf{DECODE}_{rel} \\
\quad\quad (rom, knob, button, acc, ir, nextaddress, rsw, wmar, memcntl, \\
\quad\quad\ wpc, rpc, wacc, racc, wir, rir, warg, alucntl, rbuf, ready, idle)\text{”}
\end{array}
$$

The overall specification of the host machine is obtained by joining together the control unit and the data path. The usual techniques for modelling structure are used, namely conjunction and existential quantification. The relational specification of the computer implementation is defined using the predicate HOST_{rel}.

$$
\begin{array}{l}
\text{“}\mathsf{HOST}_{rel}(knob, button, switches, mem, pc, acc, ready, idle) \equiv \\
\quad \exists\, ir\ rsw\ wmar\ memcntl\ wpc\ rpc\ wacc\ racc \\
\quad\ wir\ rir\ warg\ alucntl\ rbuf\ mar\ arg\ buf. \\
\quad\ \mathsf{CONTROL}_{rel} \\
\quad\quad \mathsf{microcode} \\
\quad\quad (knob, button, acc, ir, rsw, wmar, memcntl, wpc, rpc, wacc, racc, \\
\quad\quad\ wir, rir, warg, alucntl, rbuf, ready, idle) \wedge \\
\quad\ \mathsf{DATA}_{rel} \\
\quad\quad (switches, rsw, wmar, memcntl, wpc, rpc, wacc, racc, wir, \\
\quad\quad\ rir, warg, alucntl, rbuf, mem, mar, pc, acc, ir, arg, buf)\text{”}
\end{array}
$$

The value microcode in the definition of HOST_{rel} is a constant which represents the microcode.

The translations of $\mathsf{CONTROL}_{rel}$ and HOST_{rel} are straightforward and involve the same techniques used to translate models which describe the structure of other devices already presented. The programs $\mathsf{CONTROL}_{fun}$ and HOST_{fun} are extremely long due to the large amount of tupled recursive functions, each of which has to be defined separately using the appropriate combinations of the tuple selectors *fst* and *snd*.

8.5 Executing Programs on Gordon's Computer

In the previous two sections we derived functional specifications to model the computer at the target level as well as at the host level. All HOL definitions used by the target and host definitions of the computer were successfully automatically translated to ML programs. The only two features not automatically translated required to simulate the computer were the microcode constant and the data types. In both cases this was due to a rather ad-hoc representation in HOL—it could be possible to automatically translate such features if a suitable methodology for their representation was used in HOL.

The program which models the target machine, $\mathsf{COMPUTER}_{fun}$, takes eight arguments: a name, three history functions modelling the input mechanisms, and four initial values for the memory, the program counter, the accumulator and the idle button. It computes four history functions modelling the values displayed in the memory, the program counter, the accumulator and the idle button.

At the host level, the specification HOST_{fun} takes twelve arguments. Besides a name and three history functions modelling the input mechanisms, eight initial values are needed to initialise the various registers as well as the memory. In addition to the four history outputs modelling the memory, the program counter, the accumulator and the idle light, another history function is also returned to model the ready light. This last output is not included in the target level description of the computer.

These specifications can be used to simulate the execution of programs by the computer. For example, consider the following interaction to add two integers:

> Starting with an empty memory, store an integer a in location 0 of the memory and store an integer b in location 1. From location 3 of the memory onwards, store the instructions which compute $a+b$ and store the result in location 2 of the memory.

Figure 8.3 illustrates the layout of the contents of the memory: the data in the first two locations, the program starting at location 3 and the result in location 2.

The simulation of the computer running this program is performed by setting up the appropriate values on the input mechanisms to load it into memory. In order to do this, the program must be expressed using the microinstructions of page 127. Figure 8.4 uses an assembler style notation to code the sequence of events loading the program and data, and executing the program. In fact, one can

0	a
1	b
2	$a+b$
3	program ↓

Figure 8.3: Representation of Memory for Program to Add Two Integers

define a small assembler language for the computer which would set the values on the input history functions automatically.

In the program of Figure 8.4, the integers 40960, 24577, 49154 and 0 on line numbers 3, 6, 9 and 12 respectively are the integer representations of the 13-bit instruction codes explained in the adjacent comments.

Both the specifications $\mathsf{COMPUTER}_{fun}$ and HOST_{fun} were used to simulate this program running on Gordon's computer. Figure 8.5 shows a table of the input values for the first 25 time cycles used to execute the target level specification $\mathsf{COMPUTER}_{fun}$. The figure also shows the output values for the corresponding 25 time cycles. The initial values on the program counter and the accumulator are set to zero, the initial value of the idle light is set to T and the memory is cleared using the primitive constant EMPTY13_16. In this example, the values of the numbers a and b added together are set to 54 and 85 respectively.

The table shows the different stages in the execution of the program. The outputs displayed at time $t+1$ are the results of the instruction executed at time t. For example, at time 0, all outputs are set to their initial values. Then at time 1, the value 3 is loaded in the program counter and at time 2, the number 40960, which codes the instruction for loading the accumulator with the contents of location 0 in memory, is loaded in the accumulator. At time 3 no change is apparent in the tabulated outputs because at this stage the contents of the accumulator are stored in memory at the location specified by the program counter. The loading of the program and the data in memory proceeds until time 19. Execution of the stored program begins at time 20 when the idle light goes off. The result of $a+b$ is shown in the accumulator at time 22, when it is stored in memory. The halt command is executed at time 23 and the idle light goes back on at time 24.

The program HOST_{fun} produces a similar but much longer table of outputs when simulated over the same example. In fact, 96 time cycles are necessary to show the entire output history. This is because each instruction is split into

```
1                          ; Load Program
2    LD PC   3
3    LD ACC  40960         ; Instruction code for "Load ACC with a"
4    ST ACC  PC
;
5    LD PC   4
6    LD ACC  24577         ; Instruction code for "Add b to ACC"
7    ST ACC  PC
;
8    LD PC   5
9    LD ACC  49154         ; Instruction code for "Store ACC at Location 2"
10   ST ACC  PC
;
11   LD PC   6
12   LD ACC  0             ; Instruction code for "HALT"
13   ST ACC  PC
;
14                         ; Load Data
15   LD PC   0
16   LD ACC  a
17   ST ACC  PC
;
18   LD PC   1
19   LD ACC  b
20   ST ACC  PC
;
21                         ; Execute
22   LD PC   3
23   EX PC
```

Figure 8.4: Representation of Program Executed on Gordon's Computer

several microinstructions at this level of abstraction and so an instruction executed in one time unit at the target level takes several time units at the host level. The tabulated results of the host machine simulation are not presented here.

The results shown over 24 time units in Figure 8.5 are computed by the optimised function $\mathsf{COMPUTER}_{fun}$ in an overall time of 288 seconds cpu time. The results over 96 time units (including those for the extra line *ready*) computed by the optimised function HOST_{fun} for the same example took 623 seconds cpu time. The simulations were carried out on an 8-megabyte SUN 3 machine running UNIX.

time	*knob*	*button*	*switches*	*pc*	*acc*	*idle*	Comment
0	0	T	3	0	0	T	
1	1	T	40960	3	0	T	
2	2	T	40960	3	40960	T	
3	0	T	4	3	40960	T	
4	1	T	24577	4	40960	T	
5	2	T	24577	4	24577	T	Program
6	0	T	5	4	24577	T	is loaded
7	1	T	49154	5	24577	T	
8	2	T	49154	5	49154	T	
9	0	T	6	5	49154	T	
10	1	T	0	6	49154	T	
11	2	T	0	6	0	T	
12	0	T	0	6	0	T	
13	1	T	54	0	0	T	
14	2	T	54	0	54	T	Data
15	0	T	1	0	54	T	is loaded
16	1	T	85	1	54	T	
17	2	T	85	1	85	T	
18	0	T	3	1	85	T	Go to location 3
19	3	T	3	3	85	T	Execute program
20	3	F	3	3	85	F	
21	3	F	3	4	54	F	
22	3	F	3	5	139	F	
23	3	F	3	6	139	F	
24	3	F	3	6	139	T	Computer is idle

Figure 8.5: Table Displaying Stages of Simulation

The times are certainly acceptable, especially when compared with the performance of the non-optimised versions of the specifications. The unoptimised version of $\mathsf{COMPUTER}_{fun}$ took over two hours of cpu time to terminate, and that of HOST_{fun} was allowed to run for over 24 hours on an ATLAS 10 mainframe computer—after which it was still not finished. The bulk of the inefficiency was traced to the large amount of repetitive recursive calculations involved in the definition of the data path DATA. Memoisation avoids recalculation and transforms highly inefficient functions to relatively efficient ones.

Chapter 9

Simulating the ECL Chip

The final example presented in this thesis is the ECL chip, one of the two components of the Cambridge Fast Ring [30]. The chip provides the interface between the ring and the slower access logic in other ring components. It performs modulation and demodulation of data, transforms serial data packets on the ring to 8-bit wide parallel packets for the slower logic, and does the reverse transformation for 8-bit wide packets from the slower logic.

The ECL chip was designed by Andrew Hopper and was later formally specified and verified by John Herbert using LCF-LSM and HOL [28]. In this chapter we present a brief account of the techniques involved in translating the HOL specification and implementation definitions, mainly concentrating on techniques for translating models of iterative structure since these were not discussed in the previous two examples of Chapters 7 and 8.

With a complexity of about 360 gates, the ECL chip is by far the largest of the three examples presented in this thesis and, since it is a real circuit, the success of translating its specifications and simulating them is especially significant. Full details of the ECL chip and its interface with the Cambridge Fast Ring are presented in [30] and [28]. Only a brief description of the ECL chip is presented in this chapter, enough to explain the operations of the chip before presenting its implementation and the results of example simulations.

9.1 The Specification

The ECL chip is shown in Figure 9.1 as a device with twenty nine pins (the two busses being eight bits wide): fourteen used for input, and fifteen used for output. The signals on these pins are represented by history functions of type $time \rightarrow bool$ (abbreviated as type *data*) except for the clock line $ck8$, represented by a function

of type *time→trigger*, and the bus lines *lin* and *lout* represented by functions of type *num→data*. The type *trigger*, used in the representation of clocks, ranges over two values *ON* and *OFF*. The bus lines are represented by curried functions where the first argument denotes the position of the particular data line in the bus.

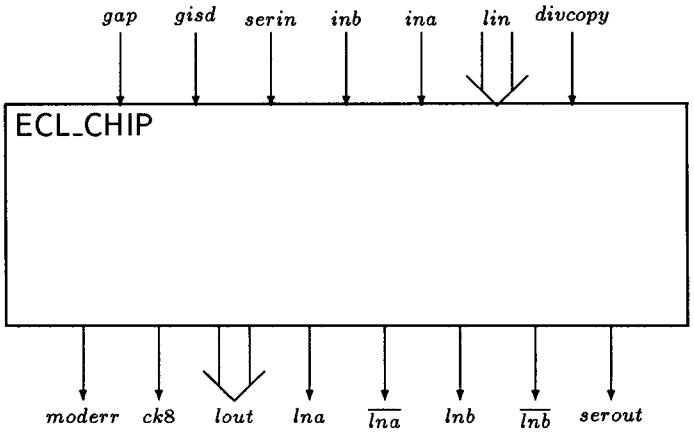


Figure 9.1: The ECL Chip

The ECL chip has several operating modes controlled by signals on the input ports *divcopy* and *gisd*. The *gisd* port is used to select between the modulated data inputs *ina* and *inb*, and the serial data input *serin*. If the value on *gisd* is high then the data on the input port *serin* is passed on to the serial data output port *serout*. When *divcopy* is asserted the chip is in its normal operating mode, receiving data being output to the ring from the slower logic chip (the CMOS chip). Packets of eight bits are input in parallel from the ring via *lin*, converted to a serial stream of data, and modulated to be output to the ring via *lna*, *lnb*, $\overline{lna}$ and $\overline{lnb}$. Modulated data read in from the ring via pins *ina* and *inb* is demodulated to a single serial stream of data which is output in packets of eight parallel bits via the bus *lout*. If a modulation error is detected in the modulated data received from the ring, the line *moderr* is asserted. Finally, the remaining two ports *gap* and *ck*8 are used to indicate the end of gap between packets on the ring, and to clock the data output by the ECL chip when read by the slower access logic, respectively.

The HOL specification of the behaviour of the ECL chip is given in [28], where it is explained in detail. The specification is long and consists mainly of boolean conditions which state timing relations between inputs and outputs. Most of these conditions cannot be automatically translated to functions and so the overall HOL

specification of the ECL chip cannot be automatically translated to an ML program and simulated. Simulations of the programs describing the implementation of the ECL chip discussed in the next section, however, can be used to demonstrate that the implementation satisfies the conditions in the behavioural specification presented in [28].

9.2 The Implementation

The top-level register-transfer implementation of the ECL chip is shown in Figure 9.2. It is built from six devices:

- a demodulator DEMOD
- a delay device SHIFT
- a device DETGAP for detecting the end of a gap between packets on the ring
- a device COUNT for generating clock signals
- a set of shift registers SHIFTREGS
- a modulator MOD

Each of these devices is built from smaller primitive devices, namely inverters, nor-gates and D-type flip-flops. Diagrams showing the structure of the implementation of each of these devices are presented in [28] and are omitted here.

The behavioural definitions of the primitive devices (inverters, nor-gates and D-type flip-flops) are of a definitional form and so they are translated using techniques already explained and illustrated in previous chapters. The structural definitions of the devices DEMOD, SHIFT, DETGAP, COUNT and MOD are also translated using techniques shown in previous chapters. These translations are therefore omitted here.

The only device in the implementation of the ECL chip shown in Figure 9.2 whose definitions require translation techniques not yet illustrated in the examples of Chapters 7 and 8 is the device SHIFTREGS. In the rest of this section we explain the implementation of SHIFTREGS and show how its specifications are translated.

The implementation of SHIFTREGS, shown in Figure 9.3, consists of five devices: CK, TOP_REGS, BOT_REGS, LOUT_BUS and DOUT. The device CK is used to generate two clock lines *ckl* and *ckr*, and two data lines *right* and *left*. These outputs are used to control the flow of data through the two shift registers

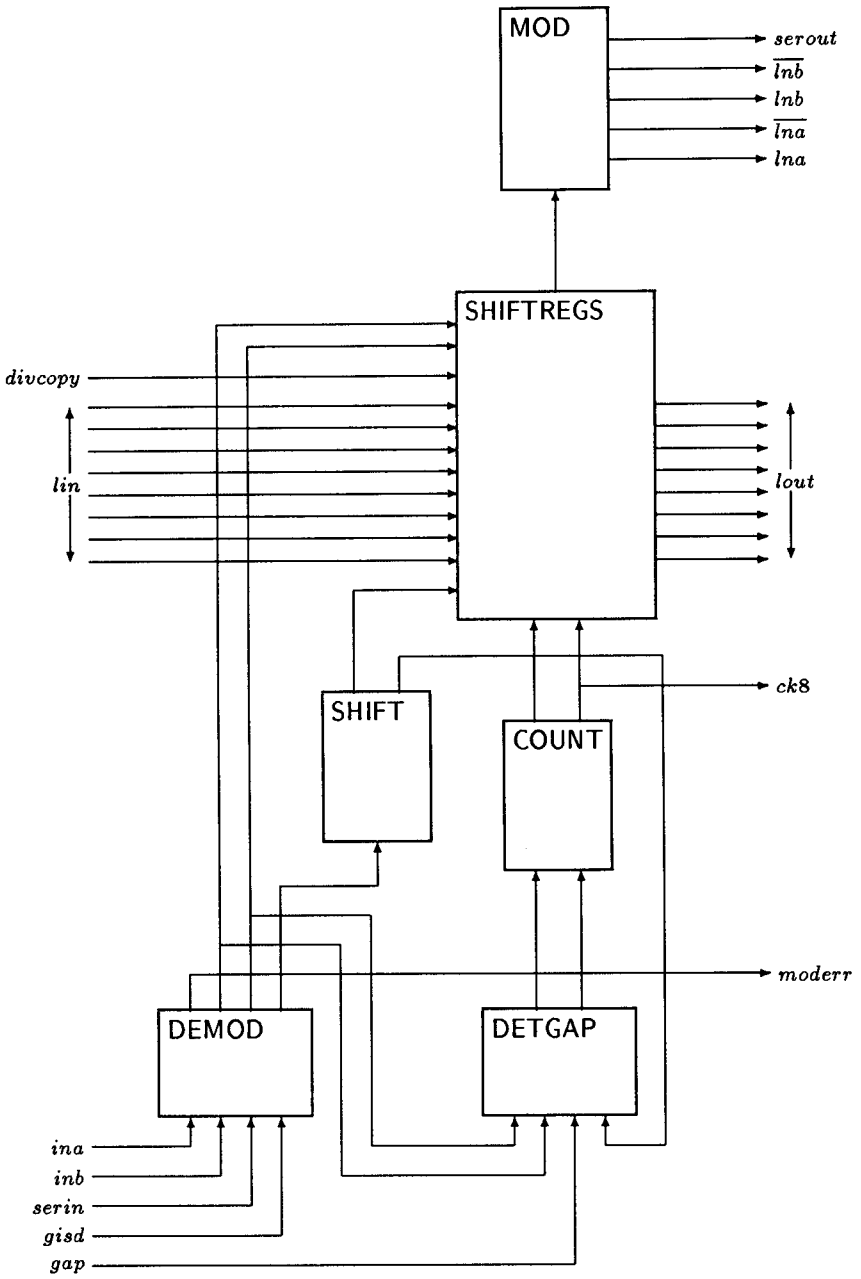


Figure 9.2: Top-level Implementation of ECL Chip

TOP_REGS and BOT_REGS, and the bus selector device LOUT_BUS. The shift registers are used to read in data serially from *d4*, shift it through the registers and, when the registers are full, output the data in parallel via a bus, and read in data in parallel from the bus *lin* and shift the data out of the registers via a serial output. The device LOUT_BUS selects between the busses *top* and *bot* for output

on the *lout* bus, and DOUT uses *divcopy* to select between the serial outputs from the registers and the external inputs *di*0 and *di*1.

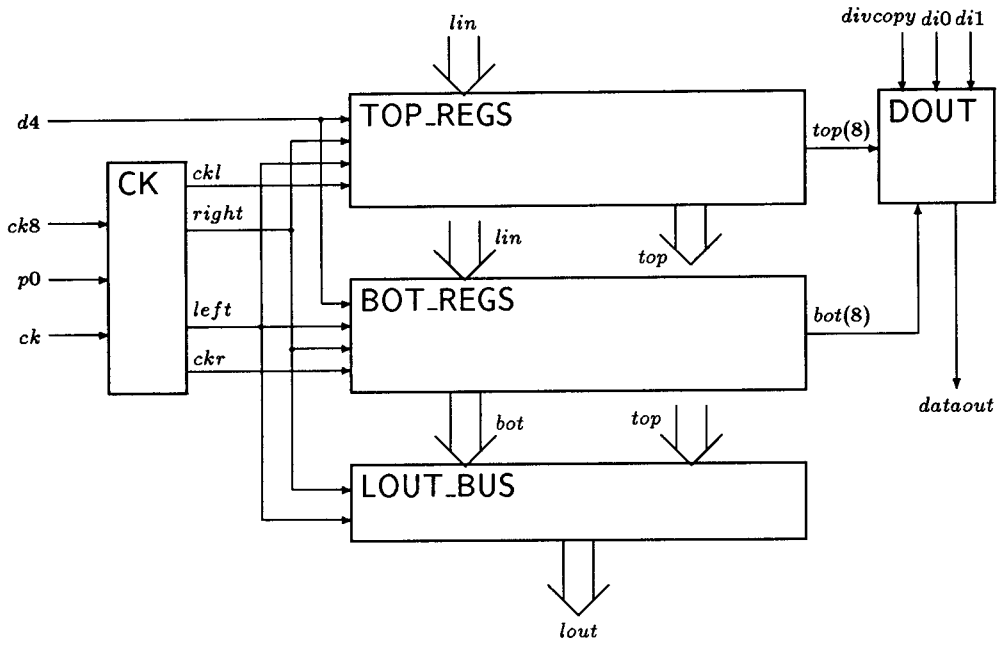


Figure 9.3: Implementation of the SHIFTREGS Device

As with all the other devices used in the implementation of the ECL chip, the above five devices used to build SHIFTREGS consist of inverters, nor-gates and D-type flip-flops. The implementations of LOUT_BUS and DOUT are purely combinational while that of CK is sequential. The translations of their relational structural definitions are straightforward. The implementations of the shift registers TOP_REGS and BOT_REGS are identical. They are built from an iteration of eight register cells, which is modelled recursively using primitive recursive definitions. The translation of such primitive recursive definitions has not yet appeared in the previous two examples and is therefore presented in this section. The definitions of the other devices are omitted.

The implementation of the shift register consists of eight register cells REG connected together in sequence as shown in Figure 9.4. Each of the bus lines $lin(0)$ through $lin(7)$ is input to a register which also receives input from a clock *ckl*, two data lines *left* and *right*, and the output of the previous register in the iteration. The serial data input to the first register is $d4$. The output of the device is an 8-bit wide bus *top* formed by the output of each register cell in order.

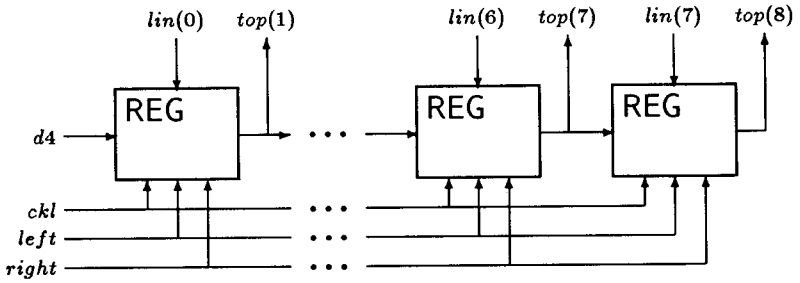


Figure 9.4: Implementation of a Shift Register

The implementation of each register cell REG is combinational and consists of three two-input nor-gates and a D-type flip-flop connected as shown in Figure 9.5.

The behavioural definitions of these primitive components are modelled relationally using the predicates NOR_{rel} and DTYPE_{rel} as shown below:

"$\mathsf{NOR}_{rel}(i_1, i_2, out) \equiv \forall t.\ out(t) = \neg(i_1(t) \vee i_2(t))$"

"$\mathsf{DTYPE}_{rel}(d, ck, q) \equiv \forall t.\ q(t{+}1) = ((ck(t){=}ON) \Rightarrow d(t) \mid q(t))$"

The corresponding derived functions are:

$$\mathsf{let}\ \mathsf{NOR}_{fun}\ i_1\ i_2 = \mathsf{let}\ out(t) = not(i_1(t)\ or\ i_2(t))\ \mathsf{in}\ out$$

$$\begin{array}{l}\mathsf{let}\ \mathsf{DTYPE}_{fun}\ name\ d\ ck\ qval = \\ \quad \mathsf{letrec}\ q(t') = \\ \quad\quad (memo\_data \\ \quad\quad\quad (name\ \hat{}\ \text{`}dtype\_q\text{`}) \\ \quad\quad\quad (\lambda t.\ (t{=}0) \Rightarrow qval \mid (ck(t{-}1){=}ON) \Rightarrow d(t{-}1) \mid q(t{-}1)))\ t' \\ \quad \mathsf{in}\ q\end{array}$$

The D-type flip-flop is used as a storage device. Data is loaded into the flip-flop whenever the value on the clock is ON, and data remains stored in the flip-flop while the value on the clock is OFF.

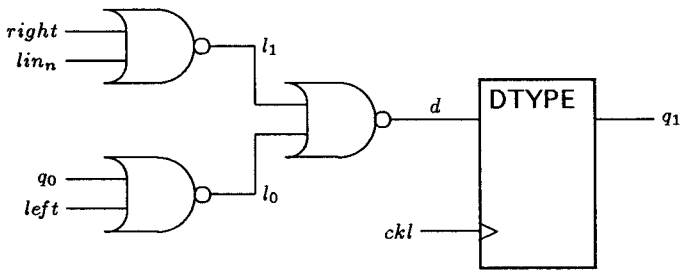


Figure 9.5: Implementation of a Register Cell

The structure of REG, therefore, is defined relationally as:

$$
\begin{array}{l}
\text{“}\mathsf{REG}_{rel}(lin_n, ckl, q_0, q_1, right, left) \equiv \\
\quad \exists\, l_0\ l_1\ d. \\
\quad\quad \mathsf{NOR}_{rel}(left, q_0, l_0) \wedge \\
\quad\quad \mathsf{NOR}_{rel}(right, lin_n, l_1) \wedge \\
\quad\quad \mathsf{NOR}_{rel}(l_0, l_1, d) \wedge \\
\quad\quad \mathsf{DTYPE}_{rel}(d, ckl, q_1)\text{”}
\end{array}
$$

and translated to the function:

$$
\begin{array}{l}
\mathsf{let}\ \mathsf{REG}_{fun}\ name\ lin_n\ ckl\ q_0\ right\ left\ ival = \\
\quad \mathsf{let}\ l_0 = \mathsf{NOR}_{fun}\ left\ q_0 \\
\quad \mathsf{and}\ l_1 = \mathsf{NOR}_{fun}\ right\ lin_n \\
\quad \mathsf{in} \\
\quad \mathsf{let}\ d = \mathsf{NOR}_{fun}\ l_0\ l_1 \\
\quad \mathsf{in} \\
\quad \mathsf{let}\ q_1 = \mathsf{DTYPE}_{fun}\ (name\ ^\wedge\ \text{`}reg\_q1\text{`})\ d\ ckl\ ival \\
\quad \mathsf{in}\ q_1
\end{array}
$$

The overall delay effected in REG is equivalent to one time unit, caused by the delay of the D-type flip-flop.

The structure of the shift register is represented in [28] using a higher order function to generate a conjunction of n register cells. The problem with this definition is that it requires ad-hoc translating techniques. Below is an alternative way of representing the structure of the shift register which uses primitive recursion to define the general case for an arbitrary number of cells. This definition is more in the HOL style of representation described in this thesis than the definition presented in [28]. The technique can be used to model iterative structure in general and the algorithm for automatically translating such definitions is straightforward and based on principles similar to the ones used so far.

$$
\begin{array}{l}
\text{“}(\mathsf{SH\_REG}_{rel}\ 0\ (lin, ckl, q, right, left, d4) \equiv \\
\quad\quad (q = \lambda m.\ (m{=}0) \Rightarrow d4 \mid \mathsf{ARB})) \wedge \\
\quad (\mathsf{SH\_REG}_{rel}\ (n{+}1)\ (lin, ckl, q, right, left, d4) \equiv \\
\quad\quad \exists\, q'\ qn.\ \mathsf{SH\_REG}_{rel}\ n\ (lin, ckl, q', right, left, d4) \wedge \\
\quad\quad\quad \mathsf{REG}_{rel}\ (lin(n), ckl, q'(n), qn, right, left)) \wedge \\
\quad\quad\quad (q = \lambda m.\ (m{=}n{+}1) \Rightarrow qn \mid q'(m)))\text{”}
\end{array}
$$

The recursive definition is very similar to that of the n-bit adder presented in Chapter 3. In fact primitive recursive definitions in HOL are a standard representation for iterative structure so techniques for translating them are essential.

The base case of the definition describes a shift register with no register cells, i.e. there is really no shift register but merely a wire through. A 1-bit wide bus

is defined equal to the input $d4$. The value ARB defined in Chapter 3 is used to express the idea that the function q is only defined for position 0, i.e. q only represents a 1-bit wide bus.

The recursive case defines a shift register with $n+1$ register cells where the output of the first n registers q', is input to the $n+1^{th}$ register to calculate the $n+1^{th}$ data line of the output bus qn. The output bus q is represented by a λ-expression which uses its first argument to select the corresponding register cell output.

The relational definition $\mathsf{SH\_REG}_{rel}$ can be automatically translated to the following recursive function:

$$
\begin{array}{l}
\mathsf{letrec}\ \mathsf{SH\_REG}_{fun}\ n\ name\ lin\ ckl\ right\ left\ d4\ ival = \\
\quad memo\_data \\
\qquad (name\ \hat{}\ \text{`}shreg\text{`}\ \hat{}\ (str\_of\_int\ n)) \\
\qquad ((n{=}0) \Rightarrow (\mathsf{let}\ q = (\lambda m.\ (m{=}0) \Rightarrow d4 \mid fail)\ \mathsf{in}\ q) \mid \\
\qquad\ (\mathsf{let}\ q' = \mathsf{SH\_REG}_{fun}\ (n{-}1)\ name\ lin\ ckl\ right\ left\ d4\ ival \\
\qquad\ \ \mathsf{in} \\
\qquad\ \ \mathsf{let}\ qn = \mathsf{REG}_{fun}\ (name\ \hat{}\ \text{`}shreg\_reg\text{`}\ \hat{}\ (str\_of\_int\ n)) \\
\qquad\qquad\qquad\qquad lin(n{-}1)\ ckl\ q'(n{-}1)\ right\ left\ ival \\
\qquad\ \ \mathsf{in} \\
\qquad\ \ \mathsf{let}\ q = (\lambda m.\ (m{=}n) \Rightarrow qn \mid q'(m)) \\
\qquad\ \ \mathsf{in}\ q))
\end{array}
$$

The definition $\mathsf{SH\_REG}_{fun}$ memoises the outputs of each register cell. The entire recursive function, therefore, is memoised in the same way as history functions.

The importance of naming memoised definitions is highlighted in the program above. Each register cell must be given a different name so that the memoisation functions can distinguish which register cell in the sequence to access. The individual naming of the cells is achieved by tagging the string representation of n (obtained by the function str_of_int) to the end of the name generated to represent the shift register.

Having defined a model for an n-bit shift register, the shift register TOP_REGS can be defined as a specialisation of the general definition. The relational and functional definitions are:

$$
\begin{array}{l}
\text{“}\mathsf{TOP\_REGS}_{rel}(lin, ckl, top, right, left, d4) \equiv \\
\ \mathsf{SH\_REG}_{rel}\ 8\ (lin, ckl, top, right, left, d4)\text{”}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{TOP\_REGS}_{fun}\ name\ lin\ ckl\ right\ left\ d4\ ival = \\
\quad \mathsf{let}\ top = \mathsf{SH\_REG}_{fun}\ 8\ name\ lin\ ckl\ right\ left\ d4\ ival \\
\quad \mathsf{in}\ top
\end{array}
$$

The definitions of BOT_REGS are similar. The structure of the SHIFTREGS device is modelled using the shift register definitions in a conjunction with the rest of the devices. The translation of the overall structural definition is straightforward and as explained in previous chapters.

9.3 Example Simulations

The overall derived program which models the implementation of the ECL chip was simulated over various samples of data. Below we present an example of one such simulation by showing waveform representations of the inputs to the chip together with the outputs obtained. In practice, a program was written to transform history functions to waveforms such as the ones shown in Figures 9.6 and 9.7 below. Figure 9.6 shows the inputs used in the simulation and Figure 9.7 shows the outputs.

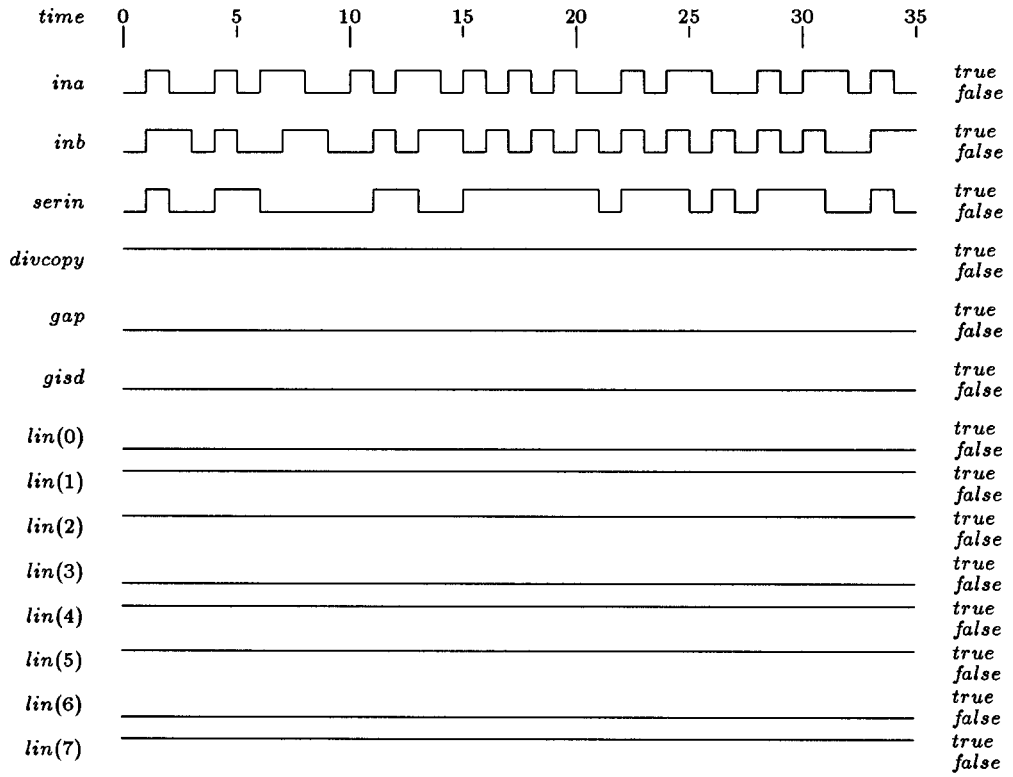


Figure 9.6: Waveforms Showing Inputs to ECL Chip

In the above example, the inputs *gap*, *gisd* and *divcopy* are set such that the chip is in its normal operating mode. Thus, *gap* is always set to *false* to indicate

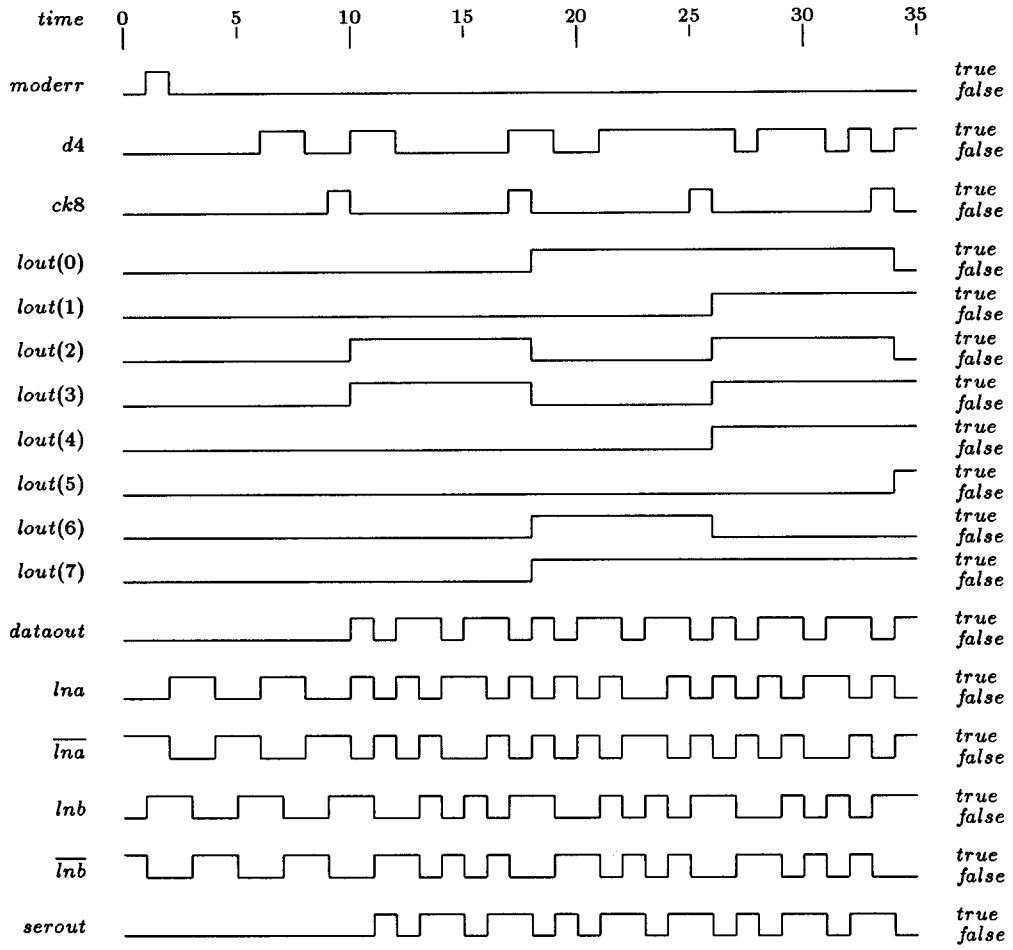


Figure 9.7: Waveforms Showing Outputs from ECL Chip

there are no gaps in the ring data, *gisd* is set to *false* so that the chip selects the modulated data inputs rather than the serial data input *serin*, and *divcopy* is asserted *true* throughout the computation to indicate that data received from the slower logic chips will be output to the ring.

The output moderr is always *false* except for a glitch at the beginning when the values on the lines are still being initialised. This indicates that no modulation errors are found on the inputs *ina* and *inb* when they are demodulated.

The waveform representing the values on the data line *d*4, although not of an external output, is included to show the demodulated data that is output in parallel via the output bus *lout*, while the input bus *lin* is output serially via *dataout*, another internal line. The waveforms of the two internal lines *d*4 and *dataout* are shown to illustrate how the data is processed by the shift registers.

The clock line $ck8$ goes high every eight cycles. If the clock is high at time t, the eight parallel bits of data on *lin* at time $t-1$ appear on *dataout* over times $t+1$ to $t+8$, and the eight serial bits of data on $d4$ over times $t-7$ to t appear on *lout* at time $t+1$. The transformation of data from parallel to serial and vice-versa is done by the shift register in a first-in first-out manner. Thus, $d4(t-7)$ is output on $lout(7)(t+1)$, $d4(t)$ is output on $lout(0)(t+1)$, $lin(7)(t)$ is output on $dataout(t+1)$, and $lin(0)(t)$ is output on $dataout(t+8)$. This effect is only noticed after the tenth time cycle when the clock goes on for the first time. The delay is due to the various D-type flip-flops in the implementation through which data must propogate.

The internal data line *dataout* is modulated and output via lines *lna* and *lnb*, and their inverses $\overline{lna}$ and $\overline{lnb}$ respectively.

The entire simulation shown in the example of Figures 9.6 and 9.7 was computed in approximately 123 seconds of cpu time on a SUN 3 computer. This involves the computation of all external output lines and the two internal lines $d4$ and *dataout* over 35 time cycles. Without memoisation, the computation does not get past nine time cycles after running for several hours—the inefficiency being mainly due to the recursion involved in the computation of the shift registers.

Chapter 10

Conclusions and Future Research

In this chapter we evaluate the research described in this thesis. We present a summary of the results achieved, discuss the problems arising from the research, and suggest solutions to these problems as well as ideas for future research.

10.1 Summary of Thesis

The goal of the research described in this thesis was to show that a subset of higher order logic could be used to specify hardware in a way which is almost identical to the notation of the programming language ML. This facilitated automatic translation from abstract definitions to executable programs with minimum risk of introducing inconsistencies. Furthermore it enabled one to perform simulation within a formal verification environment, with the aim of obtaining correct specifications earlier in the verification process.

The research was done using the HOL theorem proving system and can be summarised as follows:

- It was first shown that hardware can be effectively described in the HOL logic at the register-transfer level using both relations and functions. The two notations are similar, but not necessarily equivalent; relations can express partial specifications, but functions cannot.

- The verification of an n-bit adder using relational and functional definitions was used as an example to illustrate the general form of correctness statements that can be proved to demonstrate correctness of the relational and functional models.

- The general purpose functional programming language ML was used to simulate hardware. This was done by writing specifications as programs which

could then be executed. It was shown that many features of hardware design at the register-transfer level could be expressed in ML.

- The ML programs were slow to execute. Two optimising solutions were investigated, and one of them, memoisation, was implemented. The transformation from naive programs to efficient ones is clean and clear since the optimisation merely involves the application of higher order memo-functions to inefficient functions. The performance of the optimised functions was a vast improvement over the unoptimised ones.

- It was shown that the syntax of the HOL logic relations conventionally used for modelling structure and behaviour of digital systems closely resembles the syntax of the corresponding unoptimised ML programs. An algorithm for automatically translating these HOL relations into executable programs was presented, showing how the optimisation process could also be automated.

- It is not necessary to write hardware specifications in a special form to suit automatic translations; it was shown that the existing style of writing relational specifications in HOL is suitable to be automatically transformed into ML programs. To demonstrate this, relational definitions modelling three hardware systems previously specified and verified in HOL, namely a factorial machine, a microcomputer, and a communications chip, were automatically translated into executable functions and simulated over sample data. In the three cases, all the implementation specifications were successfully translated and executed, but only behavioural specifications written in a definitional form could be translated. This is, therefore, one restriction: if the behavioural specifications are to be simulated, behaviour must be defined in terms of functions of the inputs to the device, computing the outputs.

- In the three case studies, the relational specifications that were automatically transformed to functional programs were satisfactorily simulated at an acceptable efficiency.

Some of the points above are discussed further in the following sections, suggesting how future research can proceed to develop and improve the ideas.

10.2 Specifying Behaviour

One of the problems encountered above lies in the execution of behavioural specifications; they often contain boolean relations (such as timing conditions) which cannot be executed. A possible solution to this could be to omit these conditions from the specification and include them in the correctness statement instead. Thus, a general behavioural definition could be written for a device, simulated to establish a certain degree of correctness and confidence, and used to determine those conditions required to restrict the behaviour of the device. These conditions would then be included in the correctness statement of the contemplated device in the verification stage of the design process.

It is necessary to investigate further hardware examples. This would throw more light on the versatility of the algorithm for automatic translation and on the practicality of writing behavioural specifications strictly in a definitional form, determining whether the inclusion of non-executable conditions in correctness statements rather than in specifications is possible.

10.2.1 Automatic Manipulation of Specifications

To make the style of writing behavioural definitions more versatile, it would be convenient to have the specification automatically transformed to a definitional form. As discussed in Section 6.3.2, this is not always possible; and, even when it is possible, information regarding inverse functions is required to manipulate equations.

One possible extension to the current translating tools, is to introduce basic automatic equation manipulation for simple arithmetic and boolean algebra. Such a tool would have built-in information on inverse functions for certain arithmetic and boolean operators, and all manipulations performed to transform a specification to a definitional form could be done by the application of inference rules, such as rewriting, substitution, etc. Thus, specifications would only be transformed to equivalent alternatives.

10.3 Executing Logic Specifications

In Chapter 6, the algorithm for automatically generating programs from formal specifications was described. The translating process was broken down into intermediate stages, each stage being useful independent of the overall translation

process. The stages in the translation process are:

- translating HOL relations to HOL functions;
- translating HOL functions to ML functions; and
- optimising the ML functions.

These stages form three independent automatic translations which can be used separately or combined, depending on the intent of the user. Below we suggest some ideas worth investigating which could improve each of the automatic translations highlighted above.

10.3.1 From HOL Relations to HOL Functions

Presently, the automatic tool for translating relations to functions does not use verification strategies to perform the transformation. The tool is merely an ML program which parses HOL relations and generates HOL functions. An interesting improvement on this is to perform the translation by automatic formal proof using the HOL inference rules.

As described in Chapter 3, the relational definitions are not necessarily equivalent to their corresponding functions. Automatically proving that the generated function corresponds to the relation could therefore be hard. One possibility is to attempt to prove a theorem of the following form when the relation is not a partial specification:

$$\vdash (outputs = \mathsf{Fun}(inputs)) \equiv \mathsf{Rel}(inputs, outputs)$$

where Fun is the function which corresponds to the relation Rel. Otherwise, if Rel is some partial specification, the theorem:

$$\vdash (outputs = \mathsf{Fun}(inputs)) \supset \mathsf{Rel}(inputs, outputs)$$

could be proved, since Fun would be a total function which makes use of arbitrary values to represent unspecified values, and cannot be equivalent to Rel.

An alternative way to prove that a derived total function corresponds to a given partial relation is to first generate a total relational specification by using arbitrary values to denote unspecified cases in the same way as in forming total functions. Thus, one would first prove the theorem:

$$\vdash \mathsf{Rel}'(inputs, outputs) \supset \mathsf{Rel}(inputs, outputs)$$

where Rel′ is a total relation derived from the partial relation R. The verification could then proceed to prove that:

$$\vdash (outputs = \mathsf{Fun}(inputs)) \equiv \mathsf{Rel}'(inputs, outputs)$$

where Fun is the functional translation of Rel′.

The translation of logic relations to logic functions by automatic theorem proving is important because it helps to eliminate the possibility of inconsistencies being introduced when generating functions for simulation, and because it is a useful tool in itself to enable automatic proofs between relational and functional models of circuits.

10.3.2 From HOL Functions to ML Functions

One of the strong points of the simulation tool described in this thesis is that the restricted syntax used for writing formal specifications closely resembles the syntax of the derived, unoptimised ML programs. In fact, with sufficient extension of the HOL logic syntax, the two notations could be made identical.

Work to achieve this is recommended because the elimination of the necessity for using separate notations for simulation and verification again avoids the introduction of inconsistencies. We would like to be certain that the derived ML programs are precise models of the HOL functions. If the notations are identical for the subsets of expressions used, the possibility of syntactic error is eliminated. Combined with the automatic proof described in the previous section for transforming relations to functions, the derivation of ML programs from HOL relations in such a manner is robust.

10.3.3 Optimising ML Functions

Unfortunately, the ML programs obtained in the above stage are very inefficient and it is therefore impossible to use them effectively in large simulation examples. In this thesis we showed how the programs could be optimised to enable simulation by using the technique of memoisation.

One good point about memoisation is that the transformation from naive functions to efficient ones is clean and clear, so the risk of introducing inconsistencies in the process of optimising programs is kept to a minimum in this way. Furthermore, this optimisation method is completely automated so ad-hoc errors in translation are avoided. It is uncertain at this stage whether theorem proving techniques can

be used to make this optimising process more reliable, and it could be that the reliability of this transformation cannot be improved much further.

Future work in this area could concentrate on further improving the efficiency of the optimisations. This could be done by developing faster techniques for retrieving and storing computed information, and by using other data structures for storing which are more compact than lists.

10.4 Automatically Translating Types

Another area worth investigating is the automatic translation of HOL types to ML types. In the hardware case studies described in this thesis, type translation was performed manually. This was mainly because the HOL types used in the examples were set up in an ad-hoc manner so no general algorithm for type translation could be established.

In the process of manual translation, however, several repeated patterns were found when translating the logical types to the meta-language types. These commonly occurring features give reason to believe that an algorithm for translating types is possible. Further work on the matter should be based on a rigorous and formal approach to the axiomatisation of types in HOL such as that described by Tom Melham in [40]. Melham describes how higher order logic types can be automatically defined and axiomatised. It is possible that an algorithm parallel to this can be found for automatically defining the corresponding meta-language types.

10.5 Simulation at a Lower Level

Finally, if simulation is to aid verification, it must do so at all levels of description. Once the techniques for automatically generating efficient executable specifications from logic specifications of hardware circuits described at the register-transfer level are refined and made robust, the issue must be investigated for circuits described at a lower-level.

It could be that the same tools can be successfully applied to translating most of the aspects of hardware specifications at lower levels, with little or no extensions to the algorithms. Certain features, however, will definitely pose problems, and research in tackling these problems will be very valuable.

An example of such problematic issues concerns the modelling of bidirectionality at the switch level of description. This is one aspect where a relational model

seems most natural because no notion of inputs and outputs is desired; behaviour can be specified merely as a relationship between ports. With functions, however, the input and output ports must be identified, and once this distinction is made, they cannot be interchanged.

It is not yet certain as to what approach should be taken for finding a solution to problems like the above. William Clocksin uses the technique of executing relations in PROLOG to model bidirectional circuits at low levels of description [9]. Perhaps a relationship can be found between relational specifications in HOL and specifications in PROLOG, and an automatic translation developed for transforming low level, relational descriptions of circuits in HOL into a form which can be interpreted as PROLOG programs.

The task of investigating the execution of specifications of hardware at low levels of description is certainly the most challenging of the future research topics proposed in this chapter. It may be possible that certain issues cannot be resolved, but even if techniques are established that facilitated simulation of a subset of descriptions at lower levels, this will aid the verification process considerably.

Bibliography

[1] Abelson H., Sussman G. J. with Sussman J., *Structure and Interpretation of Computer Programs*, The MIT Electrical Engineering and Computer Science Series, MIT Press, 1985.

[2] Augustsson L., 'Compiling Lazy Functional Languages, Part II', Ph.D. Thesis, Department of Computer Sciences, Chalmers University of Technology, Göteborg, Sweden, 1987.

[3] Backus J., 'Can Programming be Liberated from the Von Neumann Style?', *Communications of the ACM*, August 1978, Vol. 21, No. 8, pp. 613–641.

[4] Barrow H. G., 'VERIFY: A Program for Proving Correctness of Digital Hardware Designs', *Artificial Intelligence*, 1984, Vol. 24, pp. 437–491.

[5] Boyer R., Moore J. S., *A Computational Logic*, Academic Press, New York, 1979.

[6] Bryant R. E., 'An Algorithm for MOS Logic Simulation', *Lambda Magazine*, Fourth Quarter 1980, pp. 46–53.

[7] Camilleri A., Gordon M., Melham T., 'Hardware Verification using Higher-Order Logic', in *From H.D.L. Descriptions to Guaranteed Correct Circuit Designs*, Borrione D. (editor), North-Holland, Amsterdam, 1987, pp. 43–67, Proceedings of the IFIP WG 10.2 International Working Conference, Grenoble, France, 9–11 September 1986.

[8] Church A., 'A Formulation of the Simple Theory of Types', *The Journal of Symbolic Logic*, 1940, Vol. 5, pp. 56–58.

[9] Clocksin W. F., 'Logic Programming and Digital Circuit Analysis', *The Journal of Logic Programming*, 1987, Vol. 4, pp. 59–82.

[10] Clocksin W. F., Mellish C. S., *Programming in* PROLOG, Springer-Verlag, Germany, 1987, 3^{rd}, Revised and Extended Edition.

[11] Cohn A. J., 'A Proof of Correctness of the VIPER Microprocessor: The First Level', in *VLSI Specification, Verification and Synthesis*, Birtwistle G. and Subrahmanyam P. A. (editors), Kluwer Academic Publishers, Boston, 1988, pp. 27–71, Proceedings of the Hardware Verification Workshop, Calgary, Canada, 12–16 January 1987.

[12] Cousineau G., Huet G., Paulson L., 'The ML Handbook', INRIA, 1986.

[13] Cullyer W. J., 'Implementing Safety-Critical Systems: The VIPER Microprocessor', in *VLSI Specification, Verification and Synthesis*, Birtwistle G. and Subrahmanyam P. A. (editors), Kluwer Academic Publishers, Boston, 1988, pp. 1–25, Proceedings of the Hardware Verification Workshop, Calgary, Canada, 12–16 January 1987.

[14] Eveking H., 'Formal Verification of Synchronous Systems', in *Formal Aspects of VLSI Design*, Milne G.J. and Subrahmanyam P. A. (editors), North-Holland, Amsterdam, 1986, pp. 137–159, Proceedings of the 1985 Edinburgh Conference on VLSI.

[15] Fourman M., Private Communication, 1988.

[16] Gordon M. J., Milner A. J., Wadsworth C. P., *Edinburgh* LCF*: A Mechanised Logic of Computation*, Lecture Notes in Computer Science, Vol. 78, Springer-Verlag, 1979.

[17] Gordon M. J. C., 'A Model of Register Transfer Systems with Applications to Microcode and VLSI Correctness', Internal Report CSR-82-81, University of Edinburgh, Department of Computer Science, March 1981.

[18] Gordon M. J. C., 'LCF-LSM', Technical Report No. 41, University of Cambridge, Computer Laboratory.

[19] Gordon M. J. C., 'Proving a Computer Correct', Technical Report No. 42, University of Cambridge, Computer Laboratory.

[20] Gordon M. J. C., 'HOL—A Machine Oriented Formulation of Higher Order Logic', Technical Report No. 68, University of Cambridge, Computer Laboratory, July 1985.

[21] Gordon M. J. C., 'Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware', in *Formal Aspects of VLSI Design*, Milne G. J. and Subrahmanyam P. A. (editors), North-Holland, Amsterdam, 1986, pp. 153–177, Proceedings of the 1985 Edinburgh Conference on VLSI.

[22] Gordon M. J. C., 'HOL—A Proof Generating System for Higher-Order Logic', in *VLSI Specification, Verification and Synthesis*, Birtwistle G. and Subrahmanyam P. A. (editors), Kluwer Academic Publishers, Boston, 1988, pp. 73–128, Proceedings of the Hardware Verification Workshop, Calgary, Canada, 12–16 January 1987.

[23] Hale R. W. S., 'Modelling a Ring Network in Interval Temporal Logic', in *Proceedings of EuroMicro 85*, Brussels, Belgium, September 1985, North-Holland, Amsterdam, pp. 77–84.

[24] Hanna F. K. and Daeche N., 'Specification and Verification using Higher-Order Logic: A Case Study', in *Formal Aspects of VLSI Design*, Milne G.J. and Subrahmanyam P. A. (editors), North-Holland, Amsterdam, 1986, pp. 179–213, Proceedings of the 1985 Edinburgh Conference on VLSI.

[25] Hanna F. K. and Daeche N., 'An Algebraic Approach to Computational Logic', in *Proceedings of the Workshop on Programming Logic*, Dybjer P., Nordström B., Petersson K., Smith J. (editors), Marstrand, Sweden, 1–4 June 1987, Programming Methodology Group, Göteborg, October 1987, pp. 301–326.

[26] Hatcher W., *The Logical Foundations of Mathematics*, Pergamon Press, 1982.

[27] Herbert J. M. J., 'The Application of Formal Specification and Verification to a Hardware Design', in *Proceedings of the Seventh International Conference on Computer Hardware Description Languages and their Applications*, Koomen C. J. and Moto-oka T. (editors), Tokyo 1985, North-Holland, Amsterdam, 1986, pp. 434–451.

[28] Herbert J. M. J., 'Application of Formal Methods to Digital System Design', University of Cambridge, Computer Laboratory, Ph.D. Thesis, 1987.

[29] Hill S. A., 'Simulating Digital Circuits in MIRANDA', Internal Report No. 42, Computer Laboratory, University of Kent at Canterbury, Revision 1, February 1987.

[30] Hopper A. and Needham R. M., 'The Cambridge Fast Ring Networking System (CFR)', Technical Report No. 90, University of Cambridge, Computer Laboratory, June 1986.

[31] Hunt W. A., Jr., 'FM8501: A Verified Microprocessor', Ph.D. Thesis, Technical Report No. 47, Institute for Computing Science, The University of Texas at Austin, December 1985.

[32] Johnson S. D., *Synthesis of Digital Designs from Recursion Equations*, Ph.D. Thesis, Indiana University, ACM Distinguished Dissertation 1983, MIT Press, 1984.

[33] Jones G., *Programming in* OCCAM, Prentice Hall International, 1987.

[34] Joyce J., Birtwistle G., Gordon M., 'Proving a Computer Correct in Higher Order Logic', Technical Report No. 100, University of Cambridge, Computer Laboratory, December 1986.

[35] Joyce J., 'Formal Verification and Implementation of a Microprocessor', in *VLSI Specification, Verification and Synthesis*, Birtwistle G. and Subrahmanyam P. A. (editors), Kluwer Academic Publishers, Boston, 1988, pp. 129–157, Proceedings of the Hardware Verification Workshop, Calgary, Canada, 12–16 January 1987.

[36] Leisenring A., *Mathematical Logic and Hilbert's ε-Symbol*, Macdonald & Co. Ltd. London, 1969.

[37] Martin-Löf P., 'Constructive Mathematics and Computer Programming', *Philosophical Transactions of the Royal Society of London*, Series A, 1984, Volume 312, pp. 501–518.

[38] May D., Shepherd D., 'Formal Verification of the IMS T800 Microprocessor', Internal Report, Inmos Ltd., 1987.

[39] Melham T. F., 'Abstraction Mechanisms for Hardware Verification', in *VLSI Specification, Verification and Synthesis*, Birtwistle G. and Subrahmanyam P. A. (editors), Kluwer Academic Publishers, Boston, 1988, pp. 267–291, Proceedings of the Hardware Verification Workshop, Calgary, Canada, 12–16 January 1987.

[40] Melham T. F., Ph.D. Thesis, University of Cambridge, Computer Laboratory, Forthcoming 1988.

[41] Michie D., 'Memo Functions and Machine Learning', *Nature*, 6 April 1968, Vol. 218, pp. 19–22.

[42] Milne G., 'The Representation of Communication and Concurrency', Technical Report No. 4088, Department of Computer Science, California Institute of Technology, 1980.

[43] Milne G., 'Simulation and Verification Related Techniques for Hardware Analysis', Technical Report CSR-174-84, University of Edinburgh, Department of Computer Science, November 1984.

[44] Milne G., 'CIRCAL and the Representation of Communication, Concurrency and Time', *ACM Transactions on Programming Languages and Systems*, April 1985, Vol. 7, No. 2, pp. 270-298.

[45] Milner R., *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, Vol. 92, Springer-Verlag, 1980.

[46] Milner R., 'Calculi for Synchrony and Asynchrony', Technical Report CSR-104-82, Department of Computer Science, University of Edinburgh, 1982.

[47] Mishra B., Clarke E. M., 'Automatic and Hierarchical Verification of Asynchronous Circuits using Temporal Logic', Technical Report CMU-CS-83-155, Department of Computer Science, Carnegie-Mellon University, September 1983.

[48] Morison J. D., Peeling N. E., Thorp T. L., 'The Design Rationale of ELLA, A Hardware Design and Description Language', in *Proceedings of the Seventh International Conference on Computer Hardware Description Languages and their Applications*, Koomen C. J. and Moto-oka T. (editors), Tokyo 1985, North-Holland, Amsterdam, 1986, pp. 303–320.

[49] Moszkowski B., 'Reasoning about Digital Circuits', Ph.D. Thesis, Technical Report STAN-CS-83-970, Department of Computer Science, Stanford University, 1983.

[50] Moszkowski B., *Executing Temporal Logic Programs*, Cambridge University Press, 1986.

[51] Nagel L. W., 'SPICE2: A Computer Program to Simulate Semiconductor Circuits', ERL Memo No. ERL-M520, University of California, Berkeley, May 1975.

[52] O'Donnell J. T., 'Hardware Description with Recursion Equations', in *Proceedings of the Eighth International Conference on Computer Hardware Description Languages and their Applications*, Barbacci M. R. and Koomen C. J. (editors), Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1987.

[53] Paulson L., *Logic and Computation*, Cambridge University Press, 1987.

[54] Peyton Jones S. L., *The Implementation of Functional Languages*, Prentice Hall International, 1987.

[55] Praxis Systems plc, 'The ELLA System Overview', Praxis Systems plc, Bath, England, 1985.

[56] Richards M., 'BSPL—A Language for Describing the Behaviour of Synchronous Hardware', Technical Report No. 84, University of Cambridge, Computer Laboratory, April 1986.

[57] Shahdad M., et. al., 'VHSIC Hardware Description Language', *IEEE Computer*, February 1985, Volume 18, N0. 2, pp. 94–103.

[58] Sheeran M., 'μFP, An Algebraic VLSI Design Language', Ph.D. Thesis, Technical Monograph PRG–39, Oxford University, Computer Laboratory, Programming Research Group, November 1983.

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

[60] Subrahmanyam P. A., 'Towards a Framework for Dealing with System Timing in Very High Level Silicon Compilers', in *VLSI Specification, Verification and Synthesis*, Birtwistle G. and Subrahmanyam P. A. (editors), Kluwer Academic Publishers, Boston, 1988, pp. 159–215, Proceedings of the Hardware Verification Workshop, Calgary, Canada, 12–16 January 1987.

[61] Traub N., 'A Formal Approach to Hardware Analysis', Ph.D. Thesis, Technical Report CST-43-87, Department of Computer Science, University of Edinburgh, March 1987.

[62] Turner D. A., 'MIRANDA: A Non-Strict Functional Language with Polymorphic Types', in *Proceedings of the IFIP Conference on Functional Programming Languages and Computer Architecture*, Springer Verlag, 1985.

[63] Weise D. W., 'Formal Multilevel Hierarchical Verification of Synchronous MOS VLSI Circuits', Ph.D. Thesis, Massachusetts Institute of Technology, August 1986.

[64] Wikström Å., *Functional Programming Using Standard* ML, Prentice Hall International, 1987.

[65] Winston P. H., Horn B. K. P., LISP, Addison-Wesley Publishing Company, 1984, 2^{nd} Edition.

Appendix A

Grammar for Parsing Relations

relational_term ::= predicate $\equiv$ relation |
predicate = relation |
$\forall$ variable* . relational_term |
$\exists$ variable* . relational_term |
relational_term $\wedge$ relational_term

relation ::= $\exists$ variable* . relation | $\forall$ variable* . relation |
relation $\wedge$ relation | boolean

boolean ::= equation | predicate | variable | true | false

equation ::= variable = term | function = term

function ::= functor [parameter]*

predicate ::= predicator [parameter]*

functor ::= constant | variable

predicator ::= constant | variable

parameter ::= variable[:type] | tuple | arith_expr

tuple ::= variable[:type] | (variable[:type], tuple)

arith_expr	::=	constant | variable | function | arith_expr infix arith_expr | (arith_expr)
infix	::=	any declared HOL infix operator
constant	::=	any declared HOL constant
variable	::=	alphanumeric
type	::=	alphanumeric
term	::=	as in HOL term
alphanumeric	::=	as in HOL alphanumeric

NOTES:
[*word*] indicates that word is optional.
*word** indicates a sequence of one or more *words*.

Appendix B

Example Interactive Session

In this appendix, we present an example session with the HOL system in which the relational implementation definitions of the Factorial Machine (described in Chapter 7) are translated into ML functions, and simulated over sample data.

The session does not show any intermediate stages in the translation; it shows how the overall translating program *rel_to_fun* can be used to interpret relational definitions as optimised ML functions. The aim of this session is to give some idea of what is involved in generating these programs, and how they can be used for simulation.

Throughout the session, HOL terms and types are enclosed within double quotes. The HOL types "*:sig*" and "*:bus*" are used to abbreviate the types "*:time→bool*" and "*:time→num*" respectively, where the HOL type "*:time*" is represented by the type of natural numbers "*:num*". The ML types *sig* and *bus* are used to denote the types *time→bool* and *time→int* respectively; the ML type *time* being represented by the type of integers *int*.

The user's input in each case consists of the application of the ML function *rel_to_fun* to a HOL relation, a list of input symbols, and an output term. The user's input follows the ML prompt # and is terminated by a sequence of two semi-colons. The system's response then follows, giving the names and types of any functions and tables it defines. Functions are printed as '—' followed be their type. When the system's output is finished, a new prompt is printed and the user's input follows once more. Comments are included throughout the session to explain the translations. The session begins overleaf.

We first derive programs corresponding to the relational specifications of the primitive devices.

```
# rel_to_fun
    "AND (i1, i2, o) = ∀t:time. o(t) = i1(t) ∧ i2(t)"
    ["i1:sig"; "i2:sig"] "o:sig";;
AND' = - : sig → sig → sig

# rel_to_fun
    "DEC (i, o) = ∀t:time. o(t) = i(t)−1"
    ["i:bus"] "o:bus";;
DEC' = - : bus → bus

# rel_to_fun
    "EQZERO (i, o) = ∀t:time. o(t) = i(t)=0"
    ["i:bus"] "o:sig";;
EQZERO' = - : bus → sig

# rel_to_fun
    "MULTIPLY (i1, i2, o) = ∀t:time. o(t) = i1(t) × i2(t)"
    ["i1:bus"; "i2:bus"] "o:bus";;
MULTIPLY' = - : bus → bus → bus

# rel_to_fun
    "MUX (sw, i1:bus, i2:bus, o) = ∀t:time. o(t) = sw(t) ⇒ i1(t) | i2(t)"
    ["sw:sig"; "i1:bus"; "i2:bus"] "o:bus";;
MUX' = - : sig → bus → bus → bus

# rel_to_fun
    "NOT (i, o) = ∀t:time. o(t) = ¬i(t)"
    ["i:sig"] "o:sig";;
NOT' = - : sig → sig

# rel_to_fun
    "ZERO (o) = ∀t:time. o(t) = 0"
    [] "o:bus";;
ZERO' = - : bus

# rel_to_fun
    "ONE (o) = ∀t:time. o(t) = 1"
    [] "o:bus";;
ONE' = - : bus

# rel_to_fun
    "REG (i:time→α, o) = ∀t:time. o(t+1) = i(t)"
    ["i:time→α"] "o:time→α";;
REG' = - : (int→*) → * → int→*
```

The above translations are straightforward. No optimisation is performed since no recursive definitions are present. In the translation of the register model REG, an initial value of type $*$ is parameterised since delay is modelled in the definition.

The next definitions describe the implementations of DOWN, MULT and TEST.

$$
\begin{array}{l}
\#\ rel\_to\_fun \\
\quad \text{“}\mathsf{DOWN}\,(i, ready, l_1) = \\
\qquad \exists p_1\ p_2. \\
\qquad\quad \mathsf{DEC}\,(l_1, p_1) \wedge \\
\qquad\quad \mathsf{MUX}\,(ready, i, p_1, p_2) \wedge \\
\qquad\quad \mathsf{REG}\,(p_2, l_1)\text{”} \\
\quad [\text{“}i\text{:}bus\text{”}; \text{“}ready\text{:}sig\text{”}]\ \text{“}l_1\text{:}bus\text{”};;
\end{array}
$$

$$
\begin{array}{l}
memo\_int\_table = [\,] : (string \,\#\, (time \,\#\, int)\ list)\ list \\
memo\_int = - : string \rightarrow bus \rightarrow bus \\
\mathsf{DOWN}' = - : string \rightarrow bus \rightarrow sig \rightarrow int \rightarrow bus
\end{array}
$$

The relational definition DOWN is recursive so its corresponding program is optimised. Before the function DOWN′ is defined, a memoisation table for storing integers *memo_int_table*, and a memo-function *memo_int*, are defined as a side effect. The function DOWN′ takes four parameters: a name, the inputs i and *ready*, and an initial integer value; it returns the output l_1.

Next we consider MULT.

$$
\begin{array}{l}
\#\ rel\_to\_fun \\
\quad \text{“}\mathsf{MULT}\,(ready, l_1, l_2) = \\
\qquad \exists p_1\ p_2\ p_3. \\
\qquad\quad \mathsf{MULTIPLY}\,(l_2, l_1, p_1) \wedge \\
\qquad\quad \mathsf{ONE}\,(p_2) \wedge \\
\qquad\quad \mathsf{MUX}\,(ready, p_2, p_1, p_3) \wedge \\
\qquad\quad \mathsf{REG}\,(p_3, l_2)\text{”} \\
\quad [\text{“}l_1\text{:}bus\text{”}; \text{“}ready\text{:}sig\text{”}]\ \text{“}l_2\text{:}bus\text{”};; \\
\mathsf{MULT}' = - : string \rightarrow sig \rightarrow bus \rightarrow int \rightarrow bus
\end{array}
$$

The function MULT′ is also optimised, but does not define any new memoisation functions or tables; it uses the ones already defined since it only optimises functions of type *bus*.

```
# rel_to_fun
    "TEST(ready, l1, l2, out) =
        ∃p1 p2 p3 p4.
          EQZERO(l1, p1) ∧
          NOT(ready, p2) ∧
          AND(p1, p2, p4) ∧
          ZERO(p3) ∧
          REG(p4, ready) ∧
          MUX(p4, l2, p3, out)"
    ["l1:bus"; "l2:bus"] "((ready:sig), (out:bus))";;

memo_bool_table = [] : (string # (time # bool) list) list
memo_bool = - : string → sig → sig
TEST' = - : string → bus → bus → bool → (sig # bus)
```

Apart from the memoisation of functions of type *bus*, TEST′ requires the memoisation of functions of type *sig*, so a new table and function are defined for type *sig*.

The factorial definition can now be translated. The optimisations required for its translation involve the previous definitions of memo-functions and tables.

```
# rel_to_fun
    "FACT(i, out, ready) =
        ∃l1 l2.
          DOWN(i, ready, l1) ∧
          MULT(ready, l1, l2) ∧
          TEST(ready, l1, l2, out)"
    ["i:bus"] "((out:bus), (ready:sig))";;
FACT' = - : string → bus → int → int → bool → (bus # sig)
```

The function FACT takes five arguments: a name (of type *string*), an input signal i (of type *bus*), and three initial values; it computes a pair *out* and *ready* of types *bus* and *sig*, respectively.

We are now ready to simulate the factorial machine. We first set up some test data by means of an ML function *mk_sig* defined below. The function takes a list of values as an argument and returns the corresponding history function.

```
# letrec mk_sig l n = (n=0) ⇒ hd(l) | mk_sig (tl l) (n-1);;
mk_sig = - : * list → (time→*)
```

Test data is set up on history function i for the first ten time units.

```
# let i = mk_sig [4; 2; 5; 6; 9; 1; 3; 7; 8; 10];;
i = - : bus
```

The function FACT′ is executed for test data i and initial values 0, 0 and T, to compute the output history functions *out* and *ready*.

```
# let (out, ready) = (FACT' 'fact' i 0 0 T);;
out = - : bus
ready = - : sig
```

The outputs are evaluated for the first twelve instants in time, and the results can be checked to determine whether the factorial machine is performing to specification.

```
# map out [0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11];;
[0; 0; 0; 0; 0; 24; 0; 0; 0; 0; 6; 0] : int list

# map ready [0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11];;
[T; F; F; F; F; F; T; F; F; F; F; T] : bool list
```