Proof Planning for Automating Hardware Verification

Francisco J. Cantu-Ortiz

Ph.D.

University of Edinburgh

1997

Abstract

In this thesis we investigate the applicability of *proof planning* to automate the verification of hardware systems. Proof planning is a meta-level reasoning technique which captures patterns of proof common to a family of theorems. It contributes to the automation of proof by incorporating and extending heuristics found in the *Nqthm* theorem prover and using them to guide a tactic-based theorem prover in the search for a proof. We have addressed the automation of proof for hardware verification from a proof planning perspective, and have applied the strategies and search control mechanisms of proof planning to generate automatically customised tactics which prove conjectures about the correctness of many types of circuits.

The contributions of this research can be summarised as follows: (1) we show by experimentation the applicability of the proof planning ideas to verify automatically hardware designs; (2) we develop and use a methodology based on the concept of proof engineering using proof planning to verify various combinational and sequential circuits which include: arithmetic circuits (adders, subtracters, multipliers, dividers, factorials), data-path components (arithmetic logic units, shifters, processing units), and a simple microprocessor system; and (3) we contribute to the profiling of the *Clam* proof planning system by improving its robustness and efficiency in handling large terms and proofs.

In verifying hardware, the user formalises a problem by writing the specification, the implementation and the conjecture, using a logic language, and asks *Clam* to compose a tactic to prove the conjecture. This tactic is then executed by the *Oyster* prover. To compose a tactic, *Clam* uses a set of *methods* which implement heuristics that specify general-purpose tactics, and AI planning mechanisms. Search is controlled by a type of annotated rewriting called *rippling*, which controls the selective application of rewrites called *wave* rules. We have extended some of the *Clam*'s methods to verify circuits. The size of the proofs were orders of magnitude larger than the proofs that had been attempted before with proof planning, and are comparable with similar verification proofs obtained by other systems, but using fewer lemmas and less interaction.

Proof engineering refers to the application of formal proof for system design and verification. We propose a proof engineering methodology which consists of partitioning the automation of formal proof into three different kind of tasks: user, proof and systems tasks. User tasks have to do with formalising a particular verification problem and using a formal tool to obtain a proof. Proof tasks refer to the tuning of proof techniques (e.g. methods and tactics) to help obtain a proof. Systems tasks have to do with the modification of a formal tool system. By making this distinction explicit, proof development is more manageable. We

conjecture that our approach is widely applicable and can be integrated into formal verification environments to improve automation facilities, and be utilised to verify commercial and safety-critical hardware systems in industrial settings.

Acknowledgements

First and foremost I would like to express my deep gratitude to my supervisors Prof. Alan Bundy, Dr. Alan Smaill and Prof. David Basin. Their support, advice, friendship and encouragement, often beyond duty, have been invaluable. I would like to express my appreciation to Toby Walsh and James Molony for their comments and discussions on drafts of the thesis; to Andrew Ireland, Ian Green, Gordon Reid, Raul Monroy, Hugo Terashima and Santiago Negrete for all the support provided; and to the members of the Mathematical Reasoning Group who created such a productive and research environment. I thank Peter and Florence Sinclair through whom I met the superb Scottish hospitality. I would like to express my deep gratitude to Dr. Fernando J. Jaimes (ITESM) for his support, encouragement and patience during my graduate studies. I thank Leticia Rodriguez, Moraima Campbell, Manuel Valenzuela, Rogelio Soto, Jose Escamilla, Pablo Ramirez, Ramon Brena and the members of the Center for Artificial Intelligence at ITESM; I am indebted to my mother Raquel, my brother Humberto and my sisters Hilda, Maria de la Luz and Raquel; my parents-in-law Hector and Carmen and all the members of my family. I thank my examiners Prof. Michael Gordon, University of Cambridge and Prof. Michael Fourman, University of Edinburgh for their thoughtful comments and feedback on the thesis. Finally, I gracefully acknowledge the financial support of CONACYT and ITESM. God bless you all.

Dedication

I dedicate this thesis to Carmen, my wife, and to our children Francisco, Hector, Eduardo and Marycarmen. My deepest thanks for your love, patience, prayers and care, which made this dream a reality.

Declaration

I hereby declare that I composed this thesis entirely myself and that it describes my own research. Portions of the work described here have been published previously in [Cantu *et al* 96].

Francisco J. Cantu-Ortiz

Edinburgh

June 12, 1997

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Overview

This thesis is about the application of *proof planning* to the automation of formal verification in the hardware domain. Given an implementation of a circuit and a specification of its behaviour, formal verification shows that the implementation meets the specification. The specification and the implementation are expressed as formulae in a formal system, the relationship between the specification and the implementation is stated by a conjecture and the proof that the implementation meets the specification is obtained by using a calculus associated to the formal system.

Proof planning is a meta-level reasoning technique for the global control of search in automatic theorem proving. A proof plan captures common patterns of reasoning in a family of similar proofs and is used to guide the search for new proofs in the family. Proof planning combines two standard approaches to automated reasoning: the use of tactics and the use of meta-level control. The meta-level control is used to build large complex tactics from simpler ones and also abstracts the proof, highlighting its structure and the key steps. The main component of proof planning is a collection of methods. A *method* is a specification of a tactic. A tactic is a program that applies one or more rules of inference during

a proof. A method consists of an input formula (a sequent), preconditions, output formulae, postconditions, and a tactic. A method is *applicable* if the goal to be proved matches the input formula of the method and the method's preconditions hold. The preconditions, formulated in a meta-logic, specify syntactic properties of the input formula and contain heuristics to constrain search. Using the input formula and preconditions of a method, proof planning can predict if a particular tactic will be applicable without actually running it. The output formulae (there may be none) determine the new subgoals generated by the method and give a schematic description of the formulae resulting from the tactic application. The postconditions specify further syntactic properties of these formulae. For each method there is a corresponding general-purpose tactic associated to that method. Methods can be combined at the meta-level in the same way tactics are combined using tacticals. The process of reasoning about and combining methods is called *proof planning*. When planning is successful it produces a tree of methods, called a *proof plan*. A proof plan yields a composite tactic, which is built from the tactics associated with each method, custom designed to prove the current conjecture. Proof plan construction involves search. However, the planning search space is typically many orders of magnitude smaller than the object-level search space. One reason is that the heuristics represented in the preconditions of the methods ensure that backtracking during planning is rare. Another reason is that the particular methods used have preconditions which strongly restrict search, though in certain domains they are very successful in constructing proofs. There is of course a price to pay: the planning system is incomplete. However, this has not proved a serious limitation of the proof planning approach in general [Bundy *et al* 91] nor in our work where proof plans were found for all experiments we tried. The plan formation system upon which our work is built is called *Clam*. Methods in *Clam* specify tactics which build proofs for a theorem proving system called *Oyster*, which implements a type theory similar to *Nuprl's* [Bundy *et al* 90].

A number of methods have been developed in *Clam* for inductive theorem proving and we used these extensively to prove theorems about parameterised hardware designs. Induction is particularly difficult to automate as there are a

number of search control problems including selection of an induction rule, deciding a case split, possible generalisations and lemma speculation, etc. It turns out, though, that many induction proofs have a similar shape and a few tactics can collectively prove a large number of the standard inductive theorems. The induction strategy `ind_strat` is a method for applying induction and handling subsequent cases. After the application of induction, the proof is split into one or more base and step cases. The `sym_eval` method attempts to solve the base case using simplification and propositional reasoning. If necessary, another induction may be applied. The `step_case` method consists of two parts: *rippling* and *fertilisation*. The first part is implemented by the `ripple` method. Rippling is a kind of annotated rewriting where annotations are used to mark differences between the induction hypothesis and conclusion. Rippling applies annotated rewrite rules (called *wave-rules* which are applied with the `wave` method) which minimise these differences. Rippling is goal directed and manipulates just the differences between the induction conclusion and hypothesis while leaving their common structure preserved; this is in contrast to rewriting based on normalisation, which is used in other inductive theorem provers such as *Nqthm* [Boyer & Moore 79]. Rippling also involves little search, since annotations severely restrict rewriting. The second part of the step case, fertilisation, can apply when rippling has succeeded (e.g. when the annotated differences are removed or moved 'out of the way', for example, to the root of the term). The `fertilise` method then uses the induction hypothesis to simplify the conclusion.

The research reported in this thesis investigates how all these features of proof planning can be transported and extended to deal with the multiple problems that arise in automating hardware verification.

1.2 Motivation

Fabrication of hardware systems has become an increasingly difficult activity due to the growing complexity of the tasks the systems perform. Detecting errors after a commercial circuit has been fabricated may represent important economic losses for a hardware company. A recent example of this situation is the design error in the division algorithm of Intel's Pentium microprocessor that was detected after fabrication and when the product was in the market. Simulation, the traditional technique used to test circuit designs cannot validate all of the enormous amount of inputs that exist in an typical circuit because of the combinatorial explosion of the search space. Formal methods promise to overcome this problem by developing mathematical proofs of correctness which are independent of the size of the circuit, increasing in this way the confidence in the correctness of the hardware designs. For instance, formal verification has been used post hoc to detect the error in the division algorithm of the Pentium microprocessor [Moore *et al* 96,Rueß *et al* 96]. Although mathematical proof is a desirable feature, formal methods face their own difficulties which prevent them from being widely accepted and used by industry in a regular way [Saiedian 96]. Interest of industry in formal methods has centred mainly on 'push-button' systems based on model checking where weak decidable logics and efficient algorithms are used for problem specification and verification, but these systems are limited and cannot be applied to many important classes of problems such as those requiring hierarchical representations. On the other hand, systems based on more expressive logics are often resisted because translating the specification and the implementation requires expertise in logic, proof construction is not automated, and most companies are not willing to invest time and money in technologies which are still in an embrionic stage as in the case interactive theorem proving.

To increase the acceptance of formal methods in domains with undecidable verification problems, we must automate proof construction as much as is practically possible [Rushby 96]. One of the main problems to overcome is the large

search space for proofs. Even when semi-decision procedures like resolution are available, completely automated theorem proving is not viable because such techniques are too general and cannot exploit structure in the problem domain to restrict search. Heuristics, combined with user interaction to overcome incompleteness have become a useful resource. One example of this is the system *NQTHM* [Boyer & Moore 88], which uses a fixed set of heuristics to automate the construction of proofs by induction. Proof construction is automated but sometimes the user must interrupt the prover and suggest lemmas to stop it from exploring an unsuccessful branch. A second example is embodied by tactic-based proof development systems like *HOL*, *LAMBDA*, *PVS*, and *NUPRL* (described in section 2.4), where users themselves raise the level of automation by writing tactics for particular problem domains. Incompleteness is addressed by interactive proof construction, whereby, instead of writing a 'super-tactic' which works in all cases, users interactively combine tactics to solve the problem at hand and directly provide heuristics.

Our motivation comes from the desire to combine the best of these two approaches, providing automation comparable to systems like *NQTHM* and offering increased flexibility by supplying heuristics and new domain specific proof procedures like in the tactic approach. Proof planning attempts to automate decisions like the choice of induction scheme and variables, case splits, generalisations, lemma speculation, and the like, which are usually specified by the user when writing tactics, as well as the process of assembling a particular tactic for a given conjecture by using AI planning techniques. We will explore how these features of proof planning can be utilised to guide *Oyster*, a theorem prover similar to *NUPRL*, in verifying hardware.

1.3 Contributions

The main contribution of this thesis is in the area of proof automation for the hardware verification domain in a tactic-based setting: we demonstrate by experimentation the viability of using proof planning for guiding the automatic construction of customised tactics for the verification of hardware designs of small and medium size scale, and conjecture that proof planning can be scaled-up to verify more complex circuits of the type typically found in modern commercial applications. We develop a proof engineering methodology using proof planning to verify various combinational and sequential circuits which include: arithmetic circuits (adders, subtracters, multipliers, dividers), data-path components (arithmetic logic units, shifters, processing units), and a simple microprocessor system, using few lemmas compared to other systems. We also contribute to the profiling of the *Clam* proof planning system to improve its robustness and efficiency in handling large terms and proofs.

1.4 Organisation

The thesis is organised as follows: chapter 2 presents the background and a survey on formal methods for hardware verification; chapter 3 presents an overview of proof planning and its application to inductive theorem proving; chapter 4 uses two examples to introduce the basic ideas of proof planning for hardware verification; chapter 5 describes a methodology for hardware verification based on proof planning and the concept of proof engineering; chapter 6 describes the extensions to proof planning for verifying hardware; chapter 7 presents experiments for hardware verification, statistics of the experiments and an analysis of the statistics as well as a description of the experiments; chapter 8 describes related and future work; and chapter 9 presents conclusions.

Appendix A describes some basic elements for hardware verification: types, operations on types, conversion functions and conditional functions. The rest of the appendices describe the main experiments in verifying hardware with proof planning: appendix B describes non-recursive combinational circuits; appendix C explains a bit incrementer; appendix D describes a multiplier; appendix E describes the Gordon computer; and appendix F describes other circuits. Finally, appendix G displays the main methods used in the proofs.

Chapter 2

Background

Formal methods for hardware verification is an active area of research and efforts are being made to transfer its techniques to solve problems of industrial scale. In this chapter we present a summary of basic concepts and a brief survey of formal methods for hardware verification. More comprehensive and extended surveys are presented in [Yoeli 90] and [Gupta 91]. Section 2.1 describes formal verification; section 2.2 characterises the components of the verification problem; section 2.3 describes the approaches to formal hardware verification; section 2.4 describes the main hardware verification environments and the work done in them; and section 2.5 presents a summary of the chapter.

2.1 Verification

Verification consists of establishing a formal relationship between a specification and an implementation of a system. That is, showing that:

$$\forall x_1 : \tau_1 \ldots \forall x_n : \tau_n . \, Cond \rightarrow \mathcal{S}(x_1 \ldots x_n) \, \mathcal{R} \, \mathcal{I}(x_1 \ldots x_n)$$

where $x_1 \ldots x_n$ are variables of type $\tau_1 \ldots \tau_n$ respectively which represent inputs or outputs to the system. $\mathcal{R}$ is some mathematical relation like equivalence, implication or equality and some others. $Cond$ represent some conditions, $\mathcal{S}(x_1 \ldots x_n)$ represents the specification and $\mathcal{I}(x_1 \ldots x_n)$ the implementation. This is known

as the *verification problem*. Formal here means the use of a mathematical framework for describing and solving the problem. A formal system is a mathematical framework for reasoning about a problem and its possible solutions. It provides a mathematical description language, and a calculus (or proof system) for proving conjectures in the theory associated with the formal system. The verification problem is given as a conjecture where the specification $\mathcal{S}$, the implementation $\mathcal{I}$, and the relationship $\mathcal{R}$ between $\mathcal{S}$ and $\mathcal{I}$ are expressed using the language of the formal system, and the calculus is used to prove that the implementation and the specification satisfy the relationship. Two important aspects of the mathematical language are its syntax and its semantics. The former has to do with the rules for writing valid formulae, the latter is concerned with the meaning of formulae. A deductive calculus formed by a set of axioms and a set of inference rules, is the best well-known technique for proving conjectures in the formal system and has been widely applied to the verification problem.

2.1.1 Hierarchical Verification

System complexity is frequently dealt with by recursively decomposing the system into simpler interrelated systems yielding a hierarchy of components at different levels of abstraction. A specification and implementation of the system is given for each level. In this hierarchy, a system implementation at a certain level serves as the specification of the system at the next level in the hierarchy. The verifications of the system at two consecutive levels in the hierarchy can then be composed to give a verification of a system implementation with respect to a more abstract specification. This procedure can be extended to other levels to achieve a verification of the bottom level implementation with respect to the top level specification. In [Moore 89], J Moore presents such a methodology for the hierarchical verification of systems specified in the Boyer-Moore logic.

2.2 Components

There are three components of the verification problem: the specification, the implementation and the relationship between them. In this section we characterise each of these components.

2.2.1 Specification

The specification of a system describes *what* the system should do. It is an abstract description of its external behaviour. Details about the internal working of the system are ignored by the specification and left to the implementation. The specification must include external inputs and outputs which are relevant to the designer, and a relation between inputs and outputs. The following aspects which help the designer in understanding the specification will now be discussed: abstraction mechanisms, properties, verifiable specifications, executable specifications, and representation formalisms.

Abstraction

Abstraction mechanisms for hardware verification were first identified by Melham and are described in [Melham 88]. These include the following:

- *Structural.* This type of abstraction consists of hiding internal connections and components of a system, displaying just the external aspects;

- *Behavioural.* This type of abstraction allows the designer to write partial specifications of a system. This means that there are inputs of the system which are left undefined and treated as "don't cares";

- *Data.* This type of abstraction allows the designer to map one data type (e.g. binary numbers) into another data type (e.g. decimal numbers)

- *Temporal.* This type of abstraction allows the designer to map a time-scale (e.g. micro-operations time scale) into another time scale (e.g. programming instruction time scale)

System properties

The specification of a system describes properties of the system which are of particular interest. This includes *safety* properties, e.g. two devices cannot access simultaneously their bus; *liveness* properties, e.g. a device will eventually be allowed to access its bus; and *timing* properties, e.g. a device will access its bus within 5 seconds. These properties are modelled using logics to reason about time and events.

Verifiable specifications

A specification can either be assumed correct or can be verified with respect to given criteria. This defines a hierarchy of specifications, as these criteria can in turn be verified with respect to more abstract criteria, until we reach a criterion which is assumed correct.

Expressive specifications

Expressiveness is determined by the kind of formalism employed to write the specification. With some formal languages, we can express more facts than we can with others. As we will see in section 2.3, higher-order logic is more expressive than first-order logic, which is more expressive than propositional logic. High expressivity is an important feature, because it allows the designer to write compact and broad definitions, facilitating in this way the formalisation task from the user point of view, although transferring the verification effort to the machine.

Executable specifications

A specification can be described in a language which is executable, and tested on concrete input data, e.g. a set of definitions given in Lisp. This feature helps the designer in writing and debugging the specifications. But there is a trade-off since, in general, executable specifications tend to be less expressive than non-executable specifications.

Representation formalisms

Many representation formalisms exist for describing specifications. Among the most common are logical formulae (functional or relational), finite-state machines (either automata or state-transition graphs), and trace structures [Yoeli 90]. Which formalism is used determines the type of proof methods that can be used to reason about the specification.

2.2.2 Implementation

The implementation of a system indicates *how* the system does what it should do. For a hardware system this may be a design displaying its components, inputs and outputs, and connections between the components. Hierarchical layers, timing, and synchronicity among devices (that is, when to send or receive data from other device) are aspects of interest to the designer, and will now be discussed.

Hierarchical implementations

As we mentioned above, system complexity is usually tackled by decomposing the system into simpler components organised in hierarchical levels, treating each level separately, and composing the solutions of each layer into an integrated solution. This is commonly done for hardware systems. Layers spread from the modelling of physical characteristics of components, such as speed, capacitance, voltage, current, etc., all the way up to the program level. The following is a description of the main layers:

- *Physical.* The physical level deals with electrical properties (Current, voltage, speed, delays) of basic electronic components (transistors, resistors, diodes).

- *Switch.* The switch level uses the transistor as its basic element to build electronic systems. Systems verified at this level are electronic circuits made up of transistor, resistors, diodes and other components that implement some type of device, such as boolean gates.

- *Gate.* The gate level uses the gate as its basic element to build combinational and sequential circuits. Systems verified at this level are circuits which are made up of various kinds of gates (AND, OR, XOR, INVERTER, etc). Among these circuits are several kinds of combinational and sequential functions: adders, decoders, multiplexors, flip-flops, load-registers, counters, etc.

- *Register transfer.* The register transfer level uses the register as its basic element and micro-operations to process data in the registers. A register-transfer logic provides a language to describe valid operations among registers. This level is used to describe the implementation of microprocessor architectures.

- *Assembly program.* The assembly program level provides programming instructions to write assembly programs. A description of this level is given in terms of the semantics of the individual programming instructions. Each instruction is implemented by a set of micro-operations which are executed at the register-transfer level.

Combinational or sequential

A circuit can be either combinational or sequential. In a combinational circuit all the calculations are considered to occur instantly with no delays in data propagation. Combinational circuits use Boolean gates to compute Boolean functions. Sequential circuits use storage elements which can be either flip-flops or registers, combinational circuits which compute inputs to the flip-flops, and feedback loops.

Synchronous or asynchronous

A sequential circuit can be either synchronous or asynchronous. In a synchronous circuit signal processing is regulated by the pulses of a clock which is global to circuit components. Time is usually considered discrete, and outputs at time $t + 1$ are calculated from the inputs at time t.In an asynchronous circuit signal processing, transmission and storage can occur at any instant of time.

Representation issues

The following aspects of representation are relevant to the implementation: circuit representation by functions or relations, word representation by lists or functions, and parameterised representation of circuits. The first aspect is discussed in subsection 2.2.4.

- *Parameterised representation.* Hardware representations are sometimes parameterised by an attribute of the circuit. Word size is an example of an attribute that is frequently used as a parameter in representations of various hardware devices.

- *Word representation.* Words can be modelled using either lists or functions. Lists are commonly used by many verification tools for representing and manipulating words.

Hardware description languages

Hardware description languages (HDL) have been developed, standardised, and utilised by the hardware industry to describe and simulate hardware designs [IEEE 88]. These languages have become popular and are used by design engineers in a regular basis. However formal reasoning about the designs is very difficult because the semantics of the language is not clearly defined. Subsets of HDLs with a well established semantics have been defined for formal reasoning in verification environments, although their use remains limited [Gordon 95].

2.2.3 Relationship

Type of relationship

The relationships between the specification and the implementation may include the following:

- *equality.* The implementation and the specification are represented by functions: $imp = spec$

- *implication.* The implementation and the specification are represented by relations; the specification is a partial description of system behaviour, so the implementation which contains more information, implies the specification: $\forall x\ imp(x) \rightarrow spec(x)$

- *equivalence.* The implementation and the specification are represented as relations; both contain the same amount of information, so the implementation implies the specification and vice-versa: $\forall x\ imp(x) \equiv spec(x)$

- *subset.* The language representing the implementation is a subset of the language representing the specification. The language refers to the one accepted by a finite-state machine: $L(imp) \subset L(spec)$

- *logical implication* The implementation provides a semantic model with respect to which the specification is satisfied: $imp \models spec$

Proof methods

The proof methods used to establish the relationship may include: theorem proving, model checking, equivalence checking, and language containment. These methods are further explained in section 2.3.

2.2.4 Functional versus relational verification

The verification problem can be formalised using either a functional or a relational description. A full account of system verification using functional and relational representations is given in [Camilleri 88]. Here we present a summary of this topic because it is relevant for the experiments with hardware verification that we will present later.

Functional Verification

Functional verification is characterised as follows:

- circuit components are represented by total functions where the inputs of the component are the arguments of the function and the value of the function itself represents one of the possible outputs of the component.

- component interconnections also called internal wires as well as the overall structure of the circuit are represented by function composition, where a term is passed as an argument to another function.

Figure 2–1 describes these characteristics.

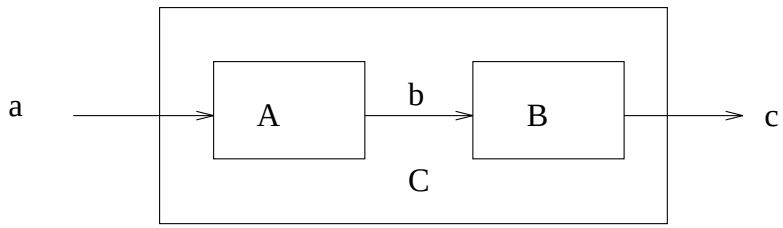

Figure 2–1: Modelling the structure of a device

n

The functional representation of this device is as follows:

$$C_{fun}(a) = B_{fun}(A_{fun}(a))$$

C_{fun} is the functional representation of the device with input a, output c, internal components A and B. A_{fun} and B_{fun} are functional representations of components A and B. Value b is computed by A and passed to B.

To prove that implementation I meets specification S we must prove a theorem of the form:

$$\forall i_1, \ldots, i_n. Abs_1(I_{fun}(i_1, \ldots, i_n)) = Abs_2(S_{fun}(Abs(i_1, \ldots, i_n)))$$

where Abs, Abs_1 and Abs_2 are data abstraction functions. Abs is required to convert data representations of the implementation to that of the specification. Furthermore, the function definitions need not be equal for all valid data values and so data abstractions Abs_1 and Abs_2 are also required to restrict and select the data for which the specification and implementation descriptions can be shown to be equal. The use of data abstractions to select and restrict the domain of a function is analogous to defining partial specifications when using relations. S_{fun} and I_{fun} are the specification and the implementation respectively given as functions.

Relational Verification

The relational representation approach is characterised as follows:

- circuit components are represented by predicates where the inputs and outputs of the component are the arguments of the predicate. The predicate constrains the values of these parameters so that the predicate is true.

- component interconnections also called internal wires are represented by shared existentially quantified variables.

- the overall structure of the circuit is represented by the conjunction of the predicates and shared variables.

Figure 2–1 also illustrates this modelling technique. The relational representation of this device is as follows:

$$C_{rel}(a,c) \equiv \exists b.A_{rel}(a,b) \wedge B_{rel}(b,c)$$

C_{rel} is the relational representation of the device with input a, output c, internal components A and B, and internal wire b. A_{rel} and B_{rel} are relational representations of components A and B. Value b is hidden by an existential quantifier.

To prove that implementation I meets specification S we must prove a theorem of the form:

$$\forall i_1,\ldots,i_n,o_1,\ldots,o_m.I_{rel}(i_1,\ldots,i_n,o_1,\ldots,o_m) \rightarrow S_{rel}(Abs(i_1,\ldots,i_n,o_1,\ldots,o_m))$$

where again, Abs is required to convert data representations of the implementation to that of the specification. S_{rel} is the specification given as a relation. Similarly I_{rel} is the implementation given as a relation.

Functional representations in general carry more information than relational representations, especially when the relational descriptions are partial or when extra information is required to specify the outputs in the functional case. Therefore, in these cases it is not possible to prove the equivalence between both representations, except when both representations carry the same amount of information. Otherwise, the following relationship will hold:

$$\forall i_1,\ldots,i_n,o_1,\ldots,o_m$$

$$((o_1,\ldots,o_m) = Rep_{fun}(i_1,\ldots,i_n)) \rightarrow Rep_{rel}(i_1,\ldots,i_n,o_1,\ldots,o_m)$$

Here Rep_{fun} and Rep_{rel} mean functional and relational representations where these representations can be either specifications or implementations. Camilleri has shown this theorem in verifying various types of combinational hardware [Camilleri 88].

The advantages and disadvantages of each representation can be summarised as follows:

- Relational representations allow the description of only the features of a device which are of interest, thus forming a partial specification.

- Relational representations allow the definition of bidirectional devices by merely defining relations between ports without distinguishing inputs from outputs.

- Functional representations require the definition of total functions, although in principle, partial functions could also be used but with more difficulty.

- Functional representations can be executed given a suitable interpreter.

- Functional representations are easier to reason about than their relational counterparts (e.g. mathematical induction).

- Relational representations are more expressive, while functional representations are computationally more efficient for system verification.

2.3 Approaches to Formal Verification

In this section we describe the main approaches to formal verification. These approaches are typically based on logic, and on other formalisms e.g. finite-state machines and trace structures.

2.3.1 Logic

Logic studies the principles of reasoning, and is a science that has as a wide range of applications in many disciplines which include computer science and artificial intelligence [Gallier 86,Genesereth & Nilsson 87].

There are two aspects of logic: its syntax and its semantics. The syntax of logic can be described in terms of a formal system where the language consists of a set of well-formed formulae made of symbols from an alphabet (constants, variables,

functions, predicates, connectives) and rules for constructing the formulae; and the calculus (or proof system) associated to the formal system consists of a set of axioms, and a set of rules of inference for deductive reasoning. A formula α which is derived from the axioms by a sequence of inference rules applications is called a theorem and is denoted by: $\vdash \alpha$. If α is derived from a set of formulae Γ we write $\Gamma \vdash \alpha$ which is read α is deduced from Γ.

The semantics of classical logic assigns meaning to the formulae by means of an interpretation. An interpretation consists of a domain and a mapping from elements, functions, and relations of the domain to constant, variables, function and predicate symbols in the formula. Constants, variables, and terms denote elements of the domain. A formula can take the values true or false. A model $\mathcal{M}$ is an interpretation that makes a formula α true and is denoted: $\mathcal{M} \models \alpha$, which is read "$\mathcal{M}$ satisfies α". A formula α is valid if it is true for every interpretation, and denoted by $\models \alpha$. A formula is satisfiable if it has a model and is a contradiction if it is false for every interpretation. A set of formulae Γ is satisfiable if there is an interpretation that satisfies every formula in Γ. The fact that, every interpretation that satisfies a set of formulae Γ also satisfies a formula α, is denoted by $\Gamma \models \alpha$ and read "Γ logically implies α".

Soundness and completeness are two attributes of a proof system with respect to semantics. Soundness means that any theorem deduced by the proof system is valid. Completeness means that any valid formula can be proved by the proof system. The completeness theorem establishes the relationship between deduction and logical implication:

$$\Gamma \models \alpha \leftrightarrow \Gamma \vdash \alpha$$

which establishes the equivalence between $\models$ and $\vdash$.

Theorem proving has to do with establishing that a formula is a theorem in a formal system. Automatic theorem proving is concerned with the mechanisation by computer of the deduction process. Theorem proving has been used to verify hardware. The specification and the implementation are given as formulae in logic and the relationship is either an implication or an equivalence. Alternatively, the

specification and the implementation can be given as terms with the relationship being an equality. Model checking has to do with establishing that an interpretation is a model of a formula, and has also been used to verify hardware. The specification is given as a formula, the implementation is given as an interpretation and the relationship is a logical implication. Model checking tries to establish that the implementation is a model of the specification.

Many types of logics can and have been derived, depending upon the constraints imposed on the syntax and semantics of the logic. Without being exhaustive, we survey some of the logics that have been used for hardware verification: propositional logic, first-order logic, higher-order logic, intuitionistic logic, modal-temporal logic.

2.3.2 Propositional Logic

Propositional logic consists of a set of logical symbols with a fixed meaning (connectives, parentheses), and a set of non-logical symbols with variable meaning (predicate symbols of arity zero called propositions) with which well-formed formulae (wffs) can be constructed. A truth assignment (or interrelation) assigns wffs one of the values true or false. Valid formulae are called tautologies. Propositional logic was one of the first logics used to model digital systems and represent Boolean functions, and is well known among design engineers. Propositional logic is a convenient way of representing and reasoning about gate-level combinational circuits but is inappropriate for modelling time and feedback loops. Propositional logic is decidable and there are tautology checkers for establishing the validity of a formula (e.g. truth tables). However, the problem of satisfiability for propositional logic is known to be NP-complete. Finding a model which satisfies a formula is known as the SAT problem [Gallier 86].

Binary Decision Diagrams

Binary decision diagrams (BDDs) are a data structure for the efficient manipulation of Boolean functions which include testing for validity of formulae and equi-

valence of functions. BDDs are represented by acyclic graphs which can represent Boolean functions in a concise way [Akers 78]. R. Bryant introduced a restricted form of BDDs called Ordered Binary Decision Diagrams (OBDDs), found a canonical representation of functions, and developed efficient algorithms for validity and equivalence operations [Bryant 86]. Some theorem provers (e.g. PVS) and model checkers (e.g SMV) have implemented a variation of BDDs for tautology checking.

2.3.3 First-Order Logic

Universal and existential quantification runs just over elements of the domain which can be any set. A term denotes an element of the domain, predicates can take one of the values true or false. First-order logic is more expressive than propositional logic and digital systems can be better described. Proof systems for first-order logic have been developed which are sound and complete, although when one introduces particular theories (e.g. lists, arithmetic), a proof system may be incomplete. Proof systems in general are semi-decidable, which means that if a formula is valid there are proof methods which will establish this fact, but if the formula is not valid, these same methods may run forever without detecting this fact. First-order logic has found many types of application including program and hardware verification.

2.3.4 Boyer-Moore Computational Logic

The Boyer-Moore Computational Logic is a subset of first-order predicate logic with implicit universal quantification, equality, and the inference rule for a type of Noetherian induction, developed for reasoning about computations. The logic exploits the duality between inductive reasoning and recursive definitions. It also includes the *shell principle* for introducing and axiomatising new inductively defined types by defining a recogniser function, a constructor function, and an accessor function for elements of the type. The *principle of definition* allows the user to define new functions and avoid possible inconsistencies either by defin-

ing non-recursive functions in terms of pre-defined functions or by making sure that a well-founded order exists on a measure of the arguments that decreases on each recursive call. The logic includes built-in shells that axiomatise the natural numbers, integers, lists, and strings. The proof system associated with the logic comprises a pre-defined proof strategy consisting of the consecutive application of pre-defined basic proof techniques which are inference rules supported by powerful heuristics [Boyer & Moore 79].

Mathematical induction is required for reasoning about object and events containing repetition. Since repetition is ubiquitous in many application areas and inductive reasoning is very difficult to automate, research has focussed on the automation of proof for mathematical induction. Inductive proofs are characterised by the application of induction rules such as Peano induction and structural induction on lists and other data structures. All these forms of induction are subsumed by a single, general schema of well-founded induction:

$$\frac{\forall x : \tau.\, (\forall y : \tau.\, x \succ y \rightarrow P(y)) \rightarrow P(x)}{\forall x : \tau.\, P(x)}$$

where $\succ$ is some well-founded relation on the type τ, *i.e.* there is no infinite descending sequence: $a_1 \succ a_2 \succ a_3 \succ \ldots$. The data-structure, control flow, time step, *etc.*, over which induction is to be applied, is represented by the type τ. The inductive proof is formalised in a many-sorted or typed logical system.

Success in proving a conjecture, P, by well-founded induction is highly dependent on the choice of x, τ and $\succ$. For many types, τ, there is an infinite variety of possible well-orderings, $\succ$. Thus choosing an appropriate induction rule to prove a conjecture is one of the most challenging search problems to be solved in automating inductive inference.

The automation of inductive inference raises a number of unique difficulties in search control:

Synthesis of induction rules: To prove a theorem by induction, one of the infinite number of possible induction rules must be synthesised and the induction variables chosen;

Conjecturing lemmata: Sometimes a lemma required to complete the proof is not already available and must be conjectured and then proved;

Generalisation of induction formulae: Sometimes a theorem cannot be proved without first being generalised on one of its terms.

In addition to these special search problems all the standard problems also manifest themselves, *e.g.* deciding when to make a case split, determining the witness for an existential quantifier, etc.

2.3.5 Higher-Order Logic

In Higher-order logic (HOL) variables range over predicates and functions as well as over individuals of the domain. This means that predicates and functions can be quantified, given as arguments of other predicates or functions, and be outputs of other predicates and functions as well. This makes higher-order logic very expressive, mathematically elegant, and permits a concise description of complex problems. But this increased complexity makes reasoning more difficult as the logic becomes undecidable, inconsistencies can be introduced, and proof systems also become incomplete. To alleviate these problems, the domain is usually constrained to have decidable or semi-decidable sub-classes, a type discipline is used to avoid inconsistencies (e.g. Russell's paradox) by introducing hierarchies among the elements of the domain, and heuristics are developed to cope with incompleteness. With these amendments that retain its advantages, higher-order logic has become very popular for describing and verifying hardware systems [Gordon 86].

One restricted higher-order logic is monadic second-order logic, a decidable logic for reasoning about strings. It has been used for verifying parameterised hardware, as well as generalising standard BDDs hardware verification capabilities [Basin & Klarlund 95].

2.3.6 Intuitionistic Logic

In intuitionistic logic the fact that a formula is either true or false does not hold in general. To prove an existential conjecture it is not sufficient to negate the conjecture and arrive at a contradiction, we need to construct explicitly an element (witness) which satisfies the conjecture. This constructive approach has been found very useful in program synthesis and has also been applied to the synthesis of circuits [Basin 91].

2.3.7 Modal Logic

Modal logic extends the scope of predicate logic with the notion of change and introduces new modal operators to express variability [Hughes & Cresswell 90]. In propositional logic predicates (propositions) are either true or false; in predicate logic the truth of a predicate depends on the elements of the domain within a static world; in modal logic it is possible to change from a world into another world by means of an accessibly relationship, where the truth of a predicate also depends on the world in which it is applied. Worlds can be seen as states in which a system can be. The accessibility relationship is characterised by modal operators which describe properties which remain true when going from one state to another state. The basic modal operators are the necessity operator $\Box P$ which means that $\Box P$ is true in a state s if P is true in all states accessible from s; the possibility operator $\Diamond P$ which means that $\Diamond P$ is true in a state s if P is true in some state accessible from s; the next operator $\bigcirc P$ which, for linear logics, means that $\bigcirc P$ is true in a state s if P is true in the next state from s; and the until operator $P \cup Q$ which means that $P \cup Q$ is true in a state s if either Q is true in s or it is true in some other state accessible from s with P being true in every intermediate state [Galton 87].

Temporal Logic

Temporal logic is a type of modal logic. It was developed for reasoning about time in the verification of concurrent programs and has found a wide variety of applica-

tions in studying time-dependent systems [Pnuelli 77]. The four modal operators are referred as the Always, Sometimes, Next-time, and Until operators respectively in temporal logic terminology. Correctness properties of time-dependent systems can easily be expressed in temporal logic. These include Safety properties which state that nothing bad happens $\models \Box P$ (e.g. two devices do not access the bus simultaneously); liveness properties which state that eventually something good happens $\models P \rightarrow \Diamond Q$ (e.g. a device will eventually access the bus); and precedence properties which grants orders of events $\models P \cup Q$ (e.g. two devices access the bus in the order their requests are done).

Temporal logic can be seen from different perspectives yielding different types of temporal logics. In one view the truth of a formula can be defined with respect to the interval between two states or with respect to a state. In the first case, we have interval temporal logic (ITL) [Moszkowski 85]. In the second case, time can be seen from two different views. In one case, time is a linear sequence of events which results in linear-time temporal logic (LTTL) [Manna & Pnuelli 81]. In the second case time has a branching structure of events which results in what is called branching-time temporal logic (BTTL). Computation tree logic (CTL) is a version of BTTL that has been applied extensively to the formal verification of sequential circuits [Clarke & Emerson 81].

2.3.8 Mu Calculus

The Mu Calculus extends the expressiveness of propositional temporal logic by adding operators for denoting fixed points of predicate transformers (i.e functions from relations to relations). It was developed independently of temporal logic and various versions have been studied in the context of program verification and has also been applied to the hardware domain [Emerson & Lei 86].

2.3.9 Other Approaches to Hardware Verification

There are other approaches for hardware verification which traditionally are not regarded as part of logic like finite-state machines, although they keep a close

connection with logic. For instance, the notion of regularity which is observed in certain types of parameterised circuits (e.g. adders, ALUs, counters) and which have been formalised using automata for which decision procedures for a type of second-order logic on strings exist.

Finite-state machines

An automaton is a mathematical structure for accepting or rejecting input sequences of symbols which correspond to words in a language. Starting at an initial state, the automaton applies a transition for each symbol of the input sequence giving new states. When the last input symbol is read, if the resulting state corresponds to a designated final state then the input sequence is accepted, otherwise it is rejected. This assumes a finite automaton where the input sequence is finite. Some concurrent/reactive systems may require automata over infinite sequences. A finite-state machine is an automaton extended with an output alphabet to form output sequences rather than just an accept/reject condition. Given a state, if a transition produces a single new state, then the machine is deterministic, otherwise it is non-deterministic. Moore and Mealy machines are commonly studied in automata theory. In a Moore machine the output sequence is a function of a state. In a Mealy machine the output is a function of the state and the input [Hopcroft & Ullman 79]. Finite-state machines have been used to represent both the specification and the implementation of the verification problem, and several techniques have been developed to establish the relationship between the two which include:

- *Machine equivalence.* Establishes the equivalence of two finite-state machines by composing a machine which accepts the exclusive-or of the two corresponding languages. If the composite language accepts just the empty language then the two machines are equivalent. Since this algorithm is computationally expensive, other techniques based on extracting a state-transition graph from each machine, composing the graphs, and developing efficient algorithms for its manipulation have been attempted [Devadas *et al* 87].

- *Language containment.* The relationship between the implementation and the specification is that of language subset rather that equivalence. Here it is shown that the language generated from the machine that describes the implementation is contained in the language generated by the machine that describes the specification [Kurshan 87]

- *Trace theory.* A trace is a sequence of transitions in a finite-state machine. A trace structure consists of a set of input and output wires, a set of traces representing a circuit's behaviour on legal inputs, and a set of traces representing a circuit's behaviour on invalid inputs. Trace theory represents the behaviour of a system as a set of traces and has been used to model asynchronous systems [Hoare 78], delay-insensitive circuits, and speed-independent circuits [Dill 89]. The specification and the implementation of speed-independent circuits are described by trace structures, and the relationship corresponds to the concept of safe substitution. An implementation is correct if it preserves the correctness of a larger context when substituted for the specification in that context. A context is an expression in trace theory containing a free variable and is denoted by $\epsilon[]$. A trace structure $\mathcal{T}$ is said to conform to another structure $\mathcal{M}$, denoted by $\mathcal{T} \leq \mathcal{M}$, if for every context $\epsilon[]$, if $\epsilon[\mathcal{M}]$ is failure-free then so is $\epsilon[\mathcal{T}]$. A trace structure is failure-free if its failure set is empty. Since it is not possible to test an infinite number of contexts, a worst-case context called the *mirror* of $\mathcal{M}$ can be defined, such that $\mathcal{T} \leq \mathcal{M}$ holds if and only if the composition of $\mathcal{T}$ and the mirror of $\mathcal{M}$ is failure-free. Thus, the mirror of a trace structure represents the strictest environment conditions under which the trace structure is expected to operate correctly. Then, if an implementation operates correctly when composed with the mirror of the specification, then it is a safe substitution under all environments.

  The main advantage of trace structures is the ability to perform hierarchical verification by using the same representation formalism (trace structures) of specifications and implementations, and by defining hierarchical operations on traces such as hiding, composition, and renaming. Its main disadvantage,

is the need for the explicit construction of a state-transition graph associated with the composite trace structure.

2.4 Verification Environments

In this section we present some of the environments used for hardware verification and the type of systems they have been applied to. There are many ways in which these environments could be classified, depending upon features of interest. One such classification considers the type of proof construction method employed, grouped into theorem proving, and model/equivalence checkers. This grouping also reflects a current trend in the use of proof environments in both academic and industrial organisations nowadays.

2.4.1 Theorem provers

There are many proof construction methods in the theorem proving arena. Three popular approaches based on seminal works from the 60s and 70s are resolution [Robinson 65], tactics [Gordon *et al* 79], as well as approaches that emphasise a particular sort of inference, like mathematical induction [Boyer & Moore 79]. To raise the level of automation of equational and tactic-based provers, meta-level reasoning methods that reason about equations or tactic formation, have also been developed [Bundy 88,Silver 85].

The *NQTHM* theorem prover is based on the Boyer&Moore computational logic to automate proof by mathematical induction [CLINC 96]. It uses a subset of Lisp as the specification language of the designs, thus making the the specifications executable. It uses a fixed set of proof techniques, namely, simplification, destructor elimination, cross-fertilisation, generalisation, elimination of irrelevance and induction. The techniques are tried in the order listed on each remaining formula with hardly no backtracking, until all formulae have been proved or all the techniques fail. Users can guide the prover by providing lemmas or suggesting sets

of rewrite rules [Boyer & Moore 88]. *NQTHM* has been used to verify a net-list implementation of the FM9001, a 32-bit general-purpose microprocessor which was fabricated as a CMOS gate array by LSI Logic Inc., and which serves as the basis of the *CLINC* stack [Bishop *et al* 95]. The elements of the stack which include a simple applications program in a high-level language a compiler, an assembler, a linker, and a simple multitasking operating system have also been verified using *NQTHM* [Moore 89]. The system is a stack in the sense that each upper layer is an abstract machine implemented on top of the lower layer. The stack is short only by comparison to systems of practical interest. The *ACL2* theorem prover is an industrial-strength implementation of the Boyer-Moore logic and a descendant of *Nqthm* that has been used to verify the correctness of the kernel of the $AMD5_K86^{tm}$ floating-point division algorithm [Moore *et al* 96].

The *OTTER* theorem prover is designed to prove theorems stated in first-order logic with equality. Otter's inference rules are based on resolution and paramodulation, and it includes facilities for term rewriting, term orderings, Knuth-Bendix completion, weighting, and strategies for directing and restricting searches for proofs. Otter can also be used as a symbolic calculator and has an embedded equational programming system. Otter is a fourth-generation Argonne National Laboratory deduction system whose ancestors (dating from the early 1960s) include the TP series, NIUTP, AURA, and ITP. Currently, the main application of Otter is research in abstract algebra and formal logic. Otter and its predecessors have been used to answer many open questions in the areas of finite semi-groups, ternary Boolean algebra, logic calculi, combinatory logic, group theory, lattice theory, and algebraic geometry [OTTER: 96].

Tactic-based theorem provers

Theorem provers like *HOL*, *LAMBDA*, *VERITAS*, *ISABELLE* , *NUPRL* and *PVS* are representative of a family of systems that have derived from the *LCF* system [Gordon *et al* 79].

HOL is a tactic-based, interactive theorem prover based on classical higher-order logic, developed by M. Gordon and his group at Cambridge University [Gordon 88]. The logic is derived from Church's Simple Type Theory with the addition of polymorphism in types, and embedded in ML, a functional programming language [Gordon 85]. Tactics and tacticals are derived from the Edinburgh LCF system [Gordon 83]. A tactic is a program that applies a set of inference rules in a single proof step. The *HOL* system has been extensively used for hardware verification since the early 80s. Currently, it has a wide installed base in academic and industrial institutions [HOL 96]. *HOL* has been used to verify microprocessors with pipe-line architectures [Windley & Coe 94] and commercial circuits like *VIPER*, a commercially available microprocessor designed by the British Ministry of Defence for use in life-critical applications. The formal verification of two layers of the implementation of the VIPER's design was done by Avra Cohn [Cohn 88]. Early work include the verification of the TAMARACK microprocessor [Joyce 90].

LAMBDA is the first commercial, formal methods-based, computer aided-design tool, specially tailored to problems of synthesis and verification of hardware designs [Mayger & Harris 91]. *LAMBDA* automatically applies formal verification during the design process, so that each completed design is effectively correct by construction, yet the user is isolated from much of the complexity of the underlying mathematical proof techniques involved [Fourman & Hexsel 91]. *LAMBDA*, as well as other model/equivalence checking products have been developed and commercialised by *Abstract Hardware Limited.* Clients of *LAMBDA*, which include Large Systems Integration, Avionics, Telecommunications and High Integrity Systems, apply *LAMBDA* in the design and verification safety-critical systems [LAMBDA 96]. The *LAMBDA* tool set includes the DIALOG design environment [Mayger 91] , which enables the power of *LAMBDA* to be effectively utilised by design engineers, the ANIMATOR, a behavioural simulator for effective specification analysis, and the BROWSER, a friendly user interface to the theorem prover core, which can be used effectively for proving properties of specifications and for the development of new proofs by using graphics and menus based on schematics, extending the design automation capability. The theorem prover is based on

classical higher-order logic, tactics and natural deduction. The library contains hundreds of rules of inference and tactics. *LAMBDA* contains a powerful rewriting system [Francis *et al* 92].

PVS is a Prototype Verification System developed at Stanford Research Institute for system verification [Owre *et al* 94]. It is based on classical higher-order logic, tactics, and uses efficient decision procedures for arithmetic reasoning. It has been used for the verification of commercial circuits. PVS was used to verify some aspects of the *AAMP5*, a pipelined microprocessor built by Collins Commercial Avionics, a division of Rockwell International, using almost half a million transistors. An implementation of the microcode at the register-transfer level was verified with respect to a representative subset of instructions [Srivas & Miller 95]. It has been extended with model checking techniques and applied to the verification of commercial systems [Havelund & Shankar 96].

Other tactic based provers include:

VERITAS, designed by Hanna and Daeche who first proposed the use of higher-order logic for hardware verification [Hanna & Daeche 86] and proposed a technique for the synthesis of circuits augmented with elements of type theory [Hanna *et al* 89b,Hanna *et al* 89a];

ISABELLE, an interactive tactic-based theorem prover developed by Larry Paulson and his group [ISABELLE 96]. Logics are encoded in a meta-logic by declaring a theory, which consists of a signature and a set of axioms. *ISABELLE*'s theory of higher-order logic extended with theories of sets, well founded recursion, natural numbers and higher-order resolution and unification has been used for the synthesis of circuits [Basin & Friedrich 96];

NUPRL, a proof development system developed by R.L. Constable and his group [Constable *et al* 86]. *NUPRL* is based upon a constructive type theory, similar to Martin-Lof's polymorphic type theory and is intended as a foundation for constructive mathematics. Mantissa Adjuster and Exponent Calculator (MAEC) is an image processor developed by P. DelVecchio [DelVecchio 90] under contract for NASA. MAEC is a section of a floating-point adder unit, and was partially

formally verified by D. Basin and P. DelVecchio [Basin & DelVecchio 89] using *NUPRL*.

A meta-level reasoning theorem prover

The *Clam-Oyster* system was developed by Alan Bundy and the Mathematical Reasoning Group at the University of Edinburgh. *Oyster* is a Prolog implementation of the *Nuprl* system. *Clam* is a meta-level reasoning proof planner which sits on top of the *Oyster* prover to guide it in the search for a proof. A more detailed description of this tool is given in section 3.4.

2.4.2 Model/equivalence checkers

Model checking was first proposed by Clarke and Emerson [Clarke & Emerson 81] as an efficient proof technique for CTL. One of its serious limitations is the use of an explicit state-transition graph representation of the hardware to be verified, which makes expressiveness limited (i.e. hierarchical or parameterised circuits cannot be modelled) and the number of states increases exponentially with the number of elements in the design, resulting in a state explosion problem. Symbolic model checking is an extension to model checking that uses symbolic Boolean representations for states and transitions functions of a sequential system, in order to avoid building its global state-transition graph explicitly. Efficient symbolic Boolean manipulation techniques are then used to evaluate the truth of temporal logic formulae with respect to these models.

SMV is a model checker developed by Kenneth McMillan for the CTL temporal logic. It has been used to verify the cache coherence protocol described in the IEEE Futurebus+ standard finding a number of previously undetected designs errors [McMillan 92].

CheckOff-M is an AHL's product which performs model checking to find errors in a design and to verify critical properties, such as the absence of deadlocks and whether the design performs specified functions. Given a set of logical properties

which the design should possess, *CheckOff-M* determines if the design omits required behaviour or includes unwanted behaviour. For example, it can check that mutually exclusive events cannot occur concurrently, and that a desired event will occur at a specific time. *CheckOff-M* can model check both complex combinatorial and sequential designs, and can be used at all stages of the design flow as it can check behavioural, register-transfer level and gate-level designs. *CheckOff-M* works with designs made using hardware description languages like VHDL, EDIF and Verilog [CHECKOFF-M 96].

CheckOff-E is an AHL's product which checks that two circuit designs are behaviourally identical. *CheckOff-E* reduces design times by replacing long simulation runs for validation with much faster equivalence checking, and increases design confidence by guaranteeing to find any differences between designs. *CheckOff-E* can compare both complex combinatorial and sequential designs, even if their architectures or state encodings are different. *CheckOff-E* can also be used at all stages of the design flow and is compatible with VHDL and EDIF designs [CHECKOFF-E 96].

MONA is a tool developed by Nils Klarlund and his group at the University of Aarhus, Denmark. *MONA* translates formulae into finite-state automata. The formulas may express search patterns, temporal properties of reactive systems, or parse tree constraints. *MONA* is based on weak monadic second-order logic of the natural numbers, which is a decidable fragment of arithmetic. *MONA* has been applied to the verification of combinational and sequential circuits exhibiting regularities [Basin & Klarlund 95].

COSMOS is a symbolic simulator developed by Randy Bryant and co-workers for modelling low-level circuit behaviour with a three-valued logic (adding don't care values X) [Bryant *et al* 87].

Other formal method tools include VERIFY [Barrow 84a], VOSS [Seger 93], VIS [Brayton 96], CIRCAL [Milne 85], and LDS [Madre & Billon 88].

2.5 Summary

We have presented a summary and a survey of formal methods for hardware verification. Most of the approaches to formal verification are based on logic and employ various types of proof methods to establish the verification relationship between the specification and the implementation. Model-checking has developed very efficient techniques for the automatic verification of complex circuits, but they are difficult to use in hierarchical verification. Theorem proving seems more appropriate for hierarchical verification, but its generality creates difficult problems of search control. User interaction with a theorem prover is not always affordable because this is a time-consuming task and demands experts in logic. Automation is called for, and there, search control is the problem to overcome. Meta-level reasoning techniques are a useful resource to control search and raise the level of automation in tactic-based settings.

Chapter 3

Proof Planning

Proof planning uses *methods* to reason about tactics in searching for a proof. In this chapter we explain proof planning as developed by the *Mathematical Reasoning Group* (MRG) at the University of Edinburgh, and the way it has been used in automating inductive theorem proving. Section 3.1 explains methods, the main component of proof planning; section 3.2 describes inductive proof planning and its main heuristics: rippling, rippling analysis, and fertilisation; section 3.3 presents an example of proof planning and domains for which proof planning has been applied; section 3.4 describes the *Clam-Oyster* system; and finally section 3.5 presents a summary of the chapter.

3.1 Methods

The *Oyster* proof development system reasons backwards from the conjecture to be proved, using an intuitionistic higher-order logic and a sequent calculus proof formalism. The search for a proof must be guided either by a user or by a tactic. The Oyster search space is very big even by theorem proving standards. There is a big number of inference rules, and some rules like induction have an infinite branching rate, therefore careful search is very important if a combinatorial explosion is to be avoided. The intention of proof planning is to guide as much of the search for a proof as possible, thus relieving the user of a tedious and error-prone

burden. The core of proof planning is a set of heuristics to apply flexibly a set of pre-defined general tactics to maximise the chances of proving a conjecture. The tactics incorporate extensions to many of the heuristics embedded in *Nqthm*. Each tactic is partially specified by a *method*, which is a description of the preconditions and effects of the tactic. Artificial intelligence planning techniques (e.g. depth-first, breadth-first, best-first, etc.) are used to construct a custom made proof plan for the conjecture. This plan is then executed by *Oyster* to derive a proof for the conjecture. A method describes important aspects of a tactic without representing all of its details. A method is represented as a frame with 6 components: a name, an input formula (a sequent), preconditions, output formulae, postconditions, and a tactic. The preconditions and postconditions are expressed in a meta-logic where a subset of Prolog is used as a meta-language. Figure 3–1 shows the components.

```
method(Name(...Args...),
       H==>G,
       [...Preconditions...],
       [...Postconditions...],
       [...Outputs...],
       tactic(...Args...)
      ).
```

Figure 3–1: Method components

- the name component `Name(...Args...)` corresponds to the name and the arguments of the method;

- the input component `H==>G` is a sequent with hypothesis H and goal G. H must unify with the hypothesis list of the current goal, which is also a sequent, and G must unify with the goal of the current goal;

- the preconditions component `[...Preconditions...]` is a list of formulae that specify properties that the input sequent must meet. A method is said to be *applicable* if the input formula unifies with the input sequent and

the preconditions are satisfied. These preconditions specify heuristics that constrain the search space of applicable methods;

- the postconditions component `[...Postconditions...]` is a list of properties that will hold after the method has been applied successfully;

- the output component `[...Outputs...]` is a list of new subgoals generated by the method which remain to be proved. If the list is empty then the method is said to be *terminating*, otherwise it becomes a branching point if it has more that one output formulae. Each output is represented by a sequent formula;

- the tactic component `tactic(...Args...)` is the name of a general-purpose tactic with the same arguments as the method. Using the input formula and preconditions of a method, proof planning can predict if this tactic will be applicable without actually running it.

Methods can be composed at the meta-level in the same way tactics are composed using tacticals. The operators that combine methods are called methodicals. For instance, the methodical *then* in *M1 then M2* combines sequentially the two methods $M1$ and $M2$, by providing as input to $M2$ the output sequents of $M1$. Methods can call other methods by a single invocation or by a repeated invocation of a sequence of methods. A method called by another method is called a sub-method. A method can be both a method or a sub-method and the difference is made explicit. The call is typically made from within the preconditions of the calling method. The process of recursively composing methods at the meta-level is called *proof planning*, and the resulting tree of methods is called a *proof plan*, which is just a composite tactic customised for the current conjecture, built-out of general-purpose tactics. *Clam*, the plan formation system, receives a conjecture and looks for the first method which is applicable. Methods are stored in a data-base and are tried one by one in the order in which they appear. If a method fails to apply, then *Clam* tries the next one, until one is applicable. If no method is applicable, then *Clam* reports failure, otherwise, the output of the applicable

method yields the new subgoals. If two or more methods are applicable, they are tried one by one. This process is applied recursively to each of the resulting subgoals until all the leaves of the tree contain terminating methods, that is, methods that produce no subgoals. This means that *Clam* can backtrack from any node in the tree, should a method failed to apply.

The standard data-base of methods in the *Clam* system is shown in figure 3–2. A procedural interpretation of this set of methods is as follows:

Methods:

1. sym_eval	simplifies symbolic expression
a. elementary	applies simple propositional reasoning
b. equal	applies an equality in the hypothesis
c. eval_def	applies rewriting
d. existential	handles existential quantification
2. generalise	generalises common term in a formula
3. normalise	normalises a formula
4. induction_strategy	applies induction strategy
a. induction	selects induction scheme and induction variables
b. sym_eval	applies symbolic evaluation to the base case
c. step_case	applies ripple and fertilise to the step case
i. ripple	applies rippling heuristic
- wave	applies wave-rules
- casesplit	unblocks rippling
- unblock	does case split
ii. fertilize	simplifies induction conclusion with induction hypothesis
- strong_fertilise	applies induction hypothesis directly
- weak_fertilise	rewrites induction conclusion with induction hypothesis

Figure 3–2: Data base of methods

- try to solve the conjecture by symbolic evaluation. If successful, finish, if there are remaining subgoals, try the set of methods again;

- try to solve the conjecture by induction, generalising or normalising any terms, if necessary. Solve the base case by applying symbolic evaluation and solve the step case by applying rippling and fertilisation. In either case, if there are remaining subgoals, try the set of methods again.

A brief description of these methods follows:

`sym_eval`: this method simplifies a formula with symbolic evaluation which consists of the repeated application of the sub-methods `elementary`, `equal`, `eval_def`, and `existential`.

`generalise`: this method generalises a formula by substituting a variable for a common sub-term. If the formula is an equation, and there is a common sub-term on both sides of the equation, then this sub-term is replaced by a variable.

`normalise`: this method applies various kinds of normalisation operations to a formula like removing universal quantifiers, moving the antecedent of an implication into the hypotheses, and replacing a conjunctive hypothesis by its conjuncts.

`ind_strat`: this method defines a strategy for proof by mathematical induction. It applies the induction sub-method, which yields two or more subgoals: the base case and the step case. The base case is simplified by the sub-method `sym_eval`. The step case is simplified by the `step_case` sub-method. The induction strategy is further explained in section 3.2.

The sub-methods called by these methods are the following:

`elementary`: this sub-method removes universally quantified variables if there are any, applies simple propositional reasoning to the goal, and solves simple well-formedness goals.

`equal`: this sub-method checks if there is any equality among the hypotheses and uses it to rewrite the goal.

`eval_def`: this sub-method normalises the goal by the exhaustive application of rewrite rules from a terminating rewrite system.

`existential`: this sub-method deals with existentially quantified variables in equalities.

`induction`: the preconditions of this sub-method implement a heuristic called rippling analysis for suggesting an induction scheme and induction variables by a look-ahead analysis into the rippling process. This heuristic is described in subsection 3.2.1. The output of the method is a list of base cases and step cases.

`step_case`: this sub-method repeatedly applies the sub-methods `ripple` and `fertilise`.

`ripple`: this sub-method is used to reduce the difference between the induction conclusion and the induction hypothesis by the repeated application of the sub-methods `wave`, `case-split`, and `unblock`. The Rippling heuristic is described in subsection 3.2.1.

`wave`: this sub-method repeatedly applies *wave rules* to the current goal. A wave rule is a kind of rewrite with annotations that control search. Annotations include boxes to wrap up terms called *wave fronts*, underlines to denote terms called *wave holes*, and upwards and downwards arrows to indicate the direction in which wave fronts are moved by wave rules. A precise meaning of annotations is given in section 3.2.1.

`casesplit`: this sub-method introduces a case split in a proof, based on the notion of complementary sets.

`unblock`: this sub-method unblocks rippling either by removing wave holes annotations, or by applying *meta-rippling*, which consists of the manipulation of annotations with rewrite rules.

`fertilise`: this sub-method utilises the induction hypothesis to simplify the induction conclusion. It applies the sub-methods `strong_fertilise` or `weak_fertilise`. This heuristic is explained in subsection 3.2.2.

`strong_fertilise`: this sub-method appeals to the induction hypothesis to prove the rippled induction conclusion.

`weak_fertilise`: this sub-method uses the induction hypothesis as a rewrite rule to simplify the rippled induction conclusion.

As an example, figure 3–3 displays the method `eval_def`, which applies rewrite rules from base and recursive equations which are not *wave* rules. The current goal

```
method(eval_def{Pos,[Rule,Dir]),
        H==>G,
        [matrix(Vars,Matrix,G),
         wave_fronts(_,[],Matrix),
         exp_at(Matrix,Pos,Exp),
         \+ Pos = [0|_],
         not metavar(Exp),
         reduction_rule(Exp,NewExp,Cond,Dir,Rule,_),
         polarity_compatible(Matrix,Pos,Dir),
         elementary(H==>Cond,_),
        [replace(Pos,NewExp,Matrix,NewMatrix),
         matrix(Vars,NewMatrix,NewGoal)],
        [H==>NewGoal],
        eval_def(Pos,[Rule,Dir])).
```

Figure 3–3: Method eval_def

is matched with the sequent `H==>G`. The preconditions split the goal G into universal variables and the matrix, checks that the matrix contains no *wave* front annotations, and locates an expression `Exp` within the matrix that is not meta-variable, and which can be rewritten using the conditional rule $Cond \rightarrow Exp \Rightarrow NewExp$ where $\Rightarrow$ indicates rewriting. `Rule` may be based on implication, equality or equivalence and must had been proven to be measure decreasing under a well-founded termination order. The polarity conditions ensures that the rule is terminating and the hypotheses must imply the conditions `Cond` of the rule. The postconditions replace the expression `Exp` by the new expression `NewExp` and form the new goal `NewGoal`. The output is the new sequent formed with the hypotheses and the new goal `H==>NewGoal`. There is a tactic called `eval_def` with arguments `Pos,[Rule,Dir]` associated to the method, which applies the rewriting.

Proof plan construction involves search. However, the planning search space is typically many orders of magnitude smaller than the object-level search space. The heuristics represented in the preconditions of the methods ensure that backtracking during planning is rare. So the search space for a plan is computationally very cheap. The cost of this dramatic pruning of the object-level search space is that the planning system is incomplete. Fortunately, this has not proved a serious limitation [Bundy *et al* 91].

3.2 Inductive Proof Planning

In this section we explain in more detail the induction strategy and its main heuristics: *rippling analysis*, *rippling*, and *fertilisation*. A pictorial representation of this strategy is given in figure 3–4.

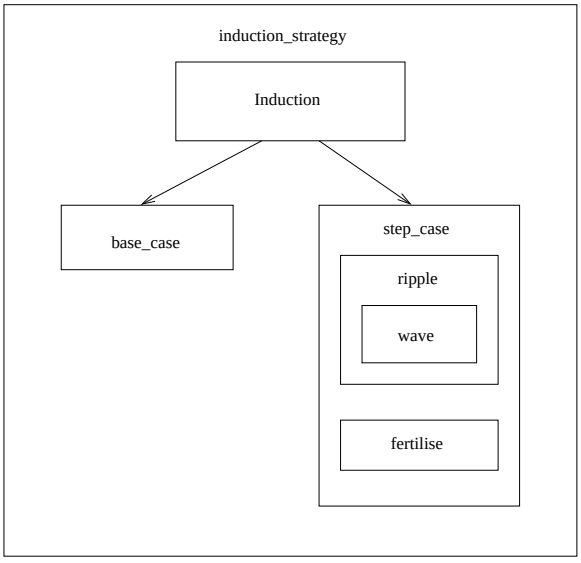

Figure 3–4: A Proof Plan for Induction

Each of the boxes represents a method. The nesting of the boxes represents the nesting of methods, *i.e.* an inner method is a sub-method of the one immediately outside it.

3.2.1 Rippling

The purpose of rippling is to complete an inductive proof by rewriting and appeal to the induction hypothesis, once an induction scheme and an induction variable have been chosen. The idea behind rippling is that the induction conclusion will be an image of the induction hypothesis except that constructor terms such as the successor function s, will be wrapped around the induction variables. Rippling attempts to move the constructor function out through the term, leaving behind an exact image of the induction hypothesis within the induction conclusion by annotating the differences with special annotations [Bundy *et al* 93]. For example, suppose we have a conjecture about the associativity of addition:

$$\forall x, y, z : Natural\ (x + y) + z = x + (y + z) \tag{3.1}$$

If we prove this conjecture by induction then we have the induction hypothesis:

$$(X + Y) + Z = X + (Y + Z) \tag{3.2}$$

and the induction conclusion with annotations:

$$(\boxed{s(\underline{x})}^{\uparrow} + y) + z = \boxed{s(\underline{x})}^{\uparrow} + (y + z) \tag{3.3}$$

The annotations are explained as follows: the box around the s constructor is called a *wave-front*; the arrow indicates that the term s is moving outwards to dominate the term in the left-hand side; the underlined term within the box is called the *wave-hole*; Deleting the arrow and everything in the box which is not underlined gives the *skeleton* which is identical to the induction hypothesis and is preserved during rippling; the induction conclusion without the annotations is called the *erasure*.

The selection of induction schemes and induction variables is supported by rippling analysis. The movement of wave-fronts is supported by various types of rippling. Both are described next.

Rippling Analysis

Rippling Analysis is a heuristic that helps to suggest an induction variable and an induction scheme by a look-ahead analysis of the rippling process [Bundy *et al* 89].

Rippling analysis first suggests candidate induction schemes, and then tests their validity. Suggested inductions are generated by considering all permutations of all the universally quantified variables with all possible infinitely many induction terms, severely restricted to those suggested by wave-rules. Induction terms are generated subject to the criterion that they can be rippled by some wave-rule present in the data-base. Each occurrence of each variable is tagged with a collection of candidate inductions which describe:

- the induction term
- annotations of the induction term
- any variables that must be sinks (explained below)
- any simultaneous inductions required on some other variables
- how well this suggestion fares for the other occurrences of that variable

An occurrence for which a wave-rule is applicable with this induction term is said to be *unflawed*, otherwise it is said to be *flawed*. Flawed occurrences are to be avoided since there is no wave-rule to move one or more initial wave-fronts introduced by the induction.

All this information is then processed by the following heuristics to rank the various suggestions:

- prefer inductions on a variable which minimises the number of flawed occurrences;
- prefer inductions on a variable which minimises the depth on the term tree of all flawed occurrences;
- prefer inductions on a variable that minimises the number of unflawed occurrences

For instance, suppose we have the conjecture for the associativity of addition 3.1, the Peano axiom:

$$\forall U, V : Natural\, U = V \leftrightarrow s(U) = s(V) \tag{3.4}$$

and the definition of addition in terms of 1-step recursion:

$$\begin{aligned} 0 + V &= V \\ s(U) + V &= s(U + V) \end{aligned} \tag{3.5}$$

The induction scheme dual to 1-step recursion is 1-step induction on the natural numbers with induction term $s(x)$ and induction variable x:

$$\frac{P(0) \wedge \forall x : Natural\, P(x) \rightarrow P(s(x))}{\forall x : Natural\, P(x)}$$

Each of the three universally quantified variables in the conjecture x, y, z, are analysed with respect to the axiom and the definition of $+$. Both occurrences of the variable x are in the recursive position of $+$: e.g. $x + \ldots$; one occurrence of the variable y is in the recursive position of $+$: e.g. $y + \ldots$; and no occurrence of the variable z is in a recursive position. The variables x is unflawed, variables y and z are flawed. The following table summarises this analysis:

Var.	Def.	Ind.	Ind.	Occurrences			Status
		term	scheme	Total	unflawed	flawed	
x	+	s(x)	1-step	2	2	0	unflawed
y	+	s(y)	1-step	2	1	1	flawed
z	none	none	none	2	0	2	flawed

In this case a 1-step induction on x is chosen. The step-case becomes:

$$\forall x, y, z : Natural,\ (x + y) + z = x + (y + z) \vdash (s(x) + y) + z = s(x) + (y + z)$$

to which we can apply three rewrites based on the definition of + and finish the proof successfully.

An extension to rippling analysis which uses meta-variables to instantiate unknown induction schemes is called middle out reasoning [Hesketh 91].

Types of rippling

There are several types of rippling. We describe rippling-out, rippling-in, rippling sideways, conditional rippling and rippling with multiple wave-rules. Other types of rippling exist and are described in [Bundy *et al* 93].

Rippling out

Rippling out moves wave-fronts outwards to match the induction hypothesis using *wave-rules*. For instance, wave-rules corresponding to equations 3.5 and 3.4 are as follows:

$$\boxed{s(\underline{x})}^{\uparrow} + y \quad \Rightarrow \quad \boxed{s(\underline{x+y})}^{\uparrow} \tag{3.6}$$

$$\boxed{s(\underline{x})}^{\uparrow} = \boxed{s(\underline{y})}^{\uparrow} \quad \Rightarrow \quad x = y \tag{3.7}$$

If we apply wave-rule 3.6 to the left-hand side of the induction conclusion (3.3) we obtain:

$$\boxed{s(\underline{x+y})}^{\uparrow} + z = \ldots$$

one more application of the rule gives:

$$\boxed{s(\underline{(x+y)+z})}^{\uparrow} = \ldots$$

the wave-front s is now completely rippled out in the left-hand side. We can now rewrite the right-hand side with the same wave-rule:

$$\ldots = \boxed{s(\underline{x+(y+z)})}^{\uparrow}$$

the wave-front s is also completely rippled-out in the right-hand side too and we obtain:

$$\boxed{s(\underline{(x+y)+z})}^{\uparrow} = \boxed{s(\underline{x+(y+z)})}^{\uparrow} \tag{3.8}$$

we can now use wave-rule 3.7 to obtain:

$$(x+y)+z = x+(y+z)$$

which matches the induction hypothesis finishing the proof.

Rippling in

Rippling in moves wave-fronts in the opposite direction, that is, inwards. Wave rules that do rippling in can be obtained from equations that rewrite the right-hand side of the equation into the left-hand side. For instance, an inwards wave rule for equation 3.5 is:

$$\boxed{s(\underline{x+y})}^{\downarrow} \Rightarrow \boxed{s(\underline{x})}^{\downarrow} + y \tag{3.9}$$

the inwards direction is indicated by the arrow in the box.

Rippling sideways

Rippling sideways is formed by combining rippling out and rippling in to move a wave-front within an expression towards an argument of that expression where there is a universally quantified variable. Initially the wave-front goes outwards, and when it is in the appropriate position, moves inwards to the position of the universally quantified variable, which will 'absorb' the wave-front. These variables are called *sinks*. The idea is that in the induction conclusion there must be another universally quantified variable that will be unified with the wave-front that was absorbed by the sink. Sinks are represented by the symbol: $\lfloor \ldots \rfloor$. For instance, consider the following conjecture about list reversal:

$$\forall L, M : Natural\ list,\ rev(L) <> M = qrev(L, M)$$

where rev is the naive list reversal function, $qrev$ is the tail recursive reversal function and $<>$ is the function that appends two lists. These primitive recursive functions are defined by:

$$\begin{aligned} rev(nil) &= nil \\ rev((H :: U) &= rev(U) <> (H :: nil) \end{aligned} \tag{3.10}$$

$$\begin{aligned} qrev(nil, V) &= V \\ qrev(H :: U, V) &= qrev(U, H :: V) \end{aligned} \tag{3.11}$$

$$nil <> V = V$$
$$(H :: U) <> V = H :: (U <> V) \quad (3.12)$$

where :: is the list constructor operator and nil is the empty list. The following wave-rules are derived from the recursive equations 3.10, 3.11 and 3.12:

$$rev(\boxed{H :: \underline{U}}^{\uparrow}) \Rightarrow \boxed{\underline{rev(U)} <> (H :: nil)}^{\uparrow} \quad (3.13)$$
$$qrev(\boxed{H :: \underline{U}}^{\uparrow}, V) \Rightarrow qrev(U, \boxed{H :: \underline{V}}^{\downarrow}) \quad (3.14)$$
$$\boxed{(H :: \underline{U}) <> V}^{\uparrow} \Rightarrow \boxed{H :: (\underline{U <> V})}^{\uparrow} \quad (3.15)$$

Wave rule 3.14 is an example of a side-ways (also called transverse) wave rule. Additionally, we can show that append is associative and derive the following wave-rule which is also a transverse wave-rule:

$$(\boxed{\underline{U <> V}}^{\uparrow}) <> W \Rightarrow U <> (\boxed{\underline{V <> W}}^{\downarrow}) \quad (3.16)$$

Applying induction, we get the hypothesis:

$$rev(l) <> M = qrev(l, M)$$

and the conclusion:

$$\boxed{rev(h :: \underline{l})}^{\uparrow} <> \lfloor m \rfloor = qrev(\boxed{h :: \underline{l}}^{\uparrow}, \lfloor m \rfloor)$$

To make the induction conclusion match the induction hypothesis we will ripple the two wave-fronts sideways so that each surrounds an $\lfloor m \rfloor$. Applying 3.14 to the right-hand side we get:

$$\boxed{rev(h :: \underline{l})}^{\uparrow} <> \lfloor m \rfloor = qrev(l, \lfloor h :: \underline{m} \rfloor)$$

We draw the sink annotation outside the wave-front to show that it has absorbed the wave-front. We now apply 3.13 to ripple the wave-front out in the left-hand side:

$$(\boxed{\underline{rev(l)} <> (h :: nil)}^{\uparrow}) <> \lfloor m \rfloor = qrev(l, \lfloor h :: \underline{m} \rfloor)$$

Wave-rule 3.16 is used to ripple the wave front sideways to the left-hand $\lfloor m \rfloor$:

$$(rev(l) <> (\left\lfloor \boxed{(h :: nil) <> \underline{m}}^{\downarrow} \right\rfloor)) <> \lfloor m \rfloor = qrev(l, \lfloor h :: \underline{m} \rfloor)$$

Normalisation is now applied to simplify the wave-fronts in the sinks and make them identical. This permits the use of the induction hypothesis which matches the induction conclusion if we instantiate M to $h :: m$, finishing the proof.

Conditional rippling

Conditional rippling uses conditional wave-rules, which are wave-rules that are applied when a condition is provable:

$$Condition \rightarrow \alpha \Rightarrow \beta$$

where $\alpha \Rightarrow \beta$ is a wave rule and $Condition$ is a formula. If a condition of a wave-rule is provable from the current hypotheses, then we can use the rule. But even if the condition is not currently provable, we can still use the rule, provided we divide the proof into two cases, using the condition and its negation. A problem to solve is when to use the condition to split the current case into two sub-cases, and when to try to prove the condition within the current case. A partial solution to this problem is to store related conditional rules in complementary sets of the form:

$$\begin{aligned} Condition_1 &\rightarrow \alpha \rightarrow \beta_1 \\ &\ldots \\ Condition_n &\rightarrow \alpha \rightarrow \beta_n \end{aligned}$$

where $Condition_1, \ldots, Condition_n$ form a partition. The following is a hypothetical example of a complementary set of wave-rules:

$$\begin{aligned} x = false \wedge y = false &\rightarrow bus(\boxed{s(\underline{t})}^{\uparrow}, x, y) \Rightarrow \boxed{memory(t, \underline{bus(t, x, y)})}^{\uparrow} \\ x = false \wedge y = true &\rightarrow bus(\boxed{s(\underline{t})}^{\uparrow}, x, y) \Rightarrow \boxed{acc(t, \underline{bus(t, x, y)})}^{\uparrow} \\ x = true \wedge y = false &\rightarrow bus(\boxed{s(\underline{t})}^{\uparrow}, x, y) \Rightarrow \boxed{pc(t, \underline{bus(t, x, y)})}^{\uparrow} \\ x = true \wedge y = true &\rightarrow bus(\boxed{s(\underline{t})}^{\uparrow}, x, y) \Rightarrow \boxed{buffer(t, \underline{bus(t, x, y)})}^{\uparrow} \end{aligned}$$

which says that a bus loads at time $t+1$ from either the memory, the accumulator, the program counter, or the buffer register, depending upon the values of x and y.

Rippling with multiple wave-fronts

This kind of rippling ripples out more than one wave-front at once. Wave rules that do this type of rippling are called multi-wave rules. Examples of multi-wave

rules are the following:

$$\begin{aligned} \forall U, V : Natural \; \boxed{s(\underline{U})}^{\uparrow} = \boxed{s(\underline{V})}^{\uparrow} &\Rightarrow U = V \\ \boxed{\underline{U} + \underline{V}}^{\uparrow} = \boxed{\underline{X} + \underline{Y}}^{\uparrow} &\Rightarrow \boxed{\underline{U = X} \wedge \underline{V = Y}}^{\uparrow} \\ \boxed{(\underline{U} + \underline{V})}^{\uparrow} * W &\Rightarrow \boxed{\underline{U * W} + \underline{V * W}}^{\uparrow} \end{aligned}$$

Wave rules are derived from axioms, recursive definitions, and lemmas. Annotations of *well annotated terms* (wats) on wave rules and induction conclusions are obtained by the recursive application of the unary functions *wfout*, *wfin*, and *wh* which place an outwards box, and inwards box, or an underline in an un-annotated term, respectively. These functions can be represented schematically as follows:

$$\begin{aligned} wfout(\alpha(x)) \quad gives \quad & \boxed{\alpha(x)}^{\uparrow} \\ wfin(\alpha(x)) \quad gives \quad & \boxed{\alpha(x)}^{\downarrow} \\ wh(\alpha(x)) \quad gives \quad & \alpha(\underline{x}) \end{aligned}$$

This way, an outwards wave-front like $wfout(\alpha(wh(\mu_1), \ldots, wh(\mu_n)))$ is represented schematically by $\boxed{\alpha(\underline{\mu_1}, \ldots, \underline{\mu_n})}^{\uparrow}$, where $n > 0$. The arguments μ_i can also be well annotated terms. The function *skel* takes a well-annotated term *wat* and returns a set of terms which form the skeleton of *wat*.

It has been shown that rippling with well-annotated terms has the following properties [Basin & Walsh 96b]:

well-formedness: if s is a *wat*, and s ripples to t, then t is also a *wat*.

skeleton preservation: if s ripples to t, then $skel(s) \subseteq skel(t)$;

correctness: if s ripples to t, then $skel(t)$ rewrites to $skel(t)$ in the un-annotated theory;

termination: rippling terminates because it guarantees a measure reduction in a well-founded order. The position and orientation of wave-fronts define this measure, and rippling applies wave rules that reduce this measure in an unchanging skeleton.

Notice that rippling reasons backwards, from the conjecture to the axioms. Therefore, to apply the implication $\phi \rightarrow \psi$ it uses the rewrite rule $\psi \Rightarrow \phi$. *Clam* uses a general notion of polarity to determine in which orientation and to which sub-expressions a wave-rule can be legally applied. For instance, for wave-rules formed from implications, the polarity of a sub-formula is the parity of the number of implicit or explicit negations within which it is contained, e.g in $\neg p \wedge q \rightarrow r$, p and r are of positive polarity and q is of negative polarity. For wave-rules formed from inequalities, the polarity can be calculated from the monotonicity of the functions containing the sub-expression. The annotations in the induction conclusion and the wave-rules are calculated automatically in *Clam* by an algorithm implemented in a wave-rule parser.

3.2.2 Fertilisation

The use of the induction hypothesis to simplify the induction conclusion is called fertilisation. It is applied when no further rippling is possible. In the previous example we made a direct appeal to the induction hypothesis to complete the proof. This is called *strong fertilisation*. Alternatively, the induction hypothesis can also be used as a rewrite rule to simplify the induction conclusion. This is called *weak fertilisation*. For instance, in the example about the associativity of addition, instead of using wave rule 3.7, we can use the induction hypothesis (3.2) to rewrite either side of the conclusion (3.8). Using the hypothesis 3.2 left-to-right we get:

$$s(x + (y + z)) = s(x + (y + z))$$

which is trivially true, finishing the proof.

Weak fertilisation is useful when rippling gets blocked in one side of the induction conclusion. Consider the following conjecture:

$$\forall x : Natural\ half(x + x) = x$$

where half is defined by:

$$half(0) \quad = \quad 0$$

$$\begin{aligned} half(s(0)) &= 0 \\ half(s(s(u))) &= s(half(u)) \end{aligned} \tag{3.17}$$

from which *Clam* derives the wave-rule:

$$half(\boxed{s(s(\underline{U}))}^{\uparrow}) \Rightarrow \boxed{s(\underline{half(U)})}^{\uparrow}$$

Assume that rippling analysis chooses a 1-step induction on x. The step case becomes:

$$half(x+x) = x \vdash half(\boxed{s(\underline{x})}^{\uparrow} + \boxed{s(\underline{x})}^{\uparrow}) = \boxed{s(\underline{x})}^{\uparrow}$$

The wave-rule from the recursive definition of plus applied to the left-hand side of the induction conclusion gives:

$$half(\boxed{s(\underline{x + \boxed{s(\underline{x})}^{\uparrow}})}^{\uparrow}) = \boxed{s(\underline{x})}^{\uparrow}$$

at this point no further wave-rules are applied and the rippling is blocked. Note that the wave-rule required to unblock this rippling is:

$$U + \boxed{s(\underline{V})}^{\uparrow} \Rightarrow \boxed{s(\underline{U+V})}^{\uparrow}$$

This is a commuted version of addition, but let us assume that this wave-rule is not available. However, the right-hand side is trivially fully rippled, so we can apply weak fertilisation using the induction hypothesis right-to-left to get:

$$half(s(x+s(x))) = \boxed{s(\underline{half(x+x)})}^{\downarrow}$$

After strong fertilisation, wave-fronts are removed since they have completed their job. After weak fertilisation they may still have a role to perform on the unblocked side of the equation, e.g. the right-hand side, so the annotations are left in place in this side. Their role is to be rippled-in by applying wave-rules derived from equations in the opposite direction, that is, from right to left. This is indicated by annotating the wave-front with the direction in which it is going to be rippled: inwards. Thus, the inwards wave-rule obtained from equation 3.17 is:

$$\boxed{s(\underline{half(U)})}^{\downarrow} \Rightarrow half(\boxed{s(s(\underline{U}))}^{\downarrow})$$

applying it to the right-hand side gives:

$$half(s(x + s(x))) = half(\boxed{s(s(\underline{x + x}))}^{\downarrow})$$

after which no more rippling in is possible. The wave-fronts are now removed and the outermost function symbols are cancelled to give:

$$x + s(x) = s(x + x)$$

which is solved by generalisation and induction.

Note that this subgoal is an instance of the missing wave-rule. This is not a coincidence, from the above analysis, we can see that if weak fertilisation and rippling succeed, then the subgoal they leave is an instance of the missing wave rule, which can be generalised into the missing wave-rule, stored, and used again when needed.

3.3 An example

We now give an example to illustrate the concepts explained. Consider the conjecture that says that the length of two lists appended is equal to the sum of the lengths of each list:

$$\forall x, y : \tau\ list\ length(x <> y) = length(x) + length(y) \tag{3.18}$$

where x and y are two lists of elements of type τ, and $length$ calculates the length of a list. The recursive definition of $length$ is:

$$\begin{aligned} length(nil) &= 0 \\ length(H :: U) &= s(length(U)) \end{aligned} \tag{3.19}$$

From which the following outwards wave-rule is derived:

$$length(\boxed{U :: \underline{V}}^{\uparrow}) \Rightarrow \boxed{s(\underline{length(V)})}^{\uparrow} \tag{3.20}$$

$$\boxed{(H :: \underline{U}) <> V}^{\uparrow} \Rightarrow \boxed{H :: (\underline{U <> V})}^{\uparrow} \tag{3.21}$$

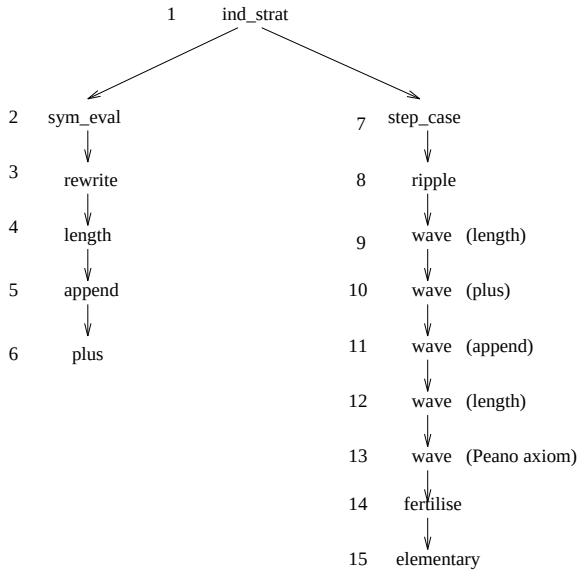

Figure 3–5: Structure of proof plan

The structure of the proof plan which proves this conjecture is displayed in figure 3–5. The proof plan is obtained automatically by *Clam* using the depth-first planner. Method and sub-method applications are marked with step numbers in the nodes of the tree and are as follows:

- methods in the data base are tried one by one. When the method `ind_strat` is tried, (in this case the other methods will have failed), we have that rippling analysis and the definition of *append* suggest a list induction with x as the induction variable (step 1)

- the base case is

  $$length(nil <> y) = length(nil) + length(y)$$

  to which symbolic evaluation is applied (step 2)

- rewriting with the base-case definitions of *append*, *length*, and *plus*, and propositional reasoning solves the base case (steps 3-6)

- the step case is

  $$length(x <> y) = length(x) + length(y)$$

$$\vdash length(\boxed{a :: \underline{x}}^{\uparrow} <> \lfloor y \rfloor) = length(\boxed{a :: \underline{x}}^{\uparrow}) + length(\lfloor y \rfloor)$$

to which the `step_case` sub-method is applied (step 7)

- the `ripple` sub-method looks for wave-rules to apply (step 8)
- the `wave` sub-method applies wave-rule 3.20 (the recursive definition of length) on the right-hand side of the induction hypothesis (step 9):

$$length(\boxed{a :: \underline{x}}^{\uparrow} <> \lfloor y \rfloor) = \boxed{s(\underline{length(x)})}^{\uparrow} + length(\lfloor y \rfloor)$$

- the `wave` sub-method applies wave-rule 3.6 (the recursive definition of plus) in the right-hand side (step 10):

$$length(\boxed{a :: \underline{x}}^{\uparrow} <> \lfloor y \rfloor) = \boxed{s(\underline{length(x) + length(\lfloor y \rfloor)})}^{\uparrow}$$

- the `wave` sub-method applies wave-rule 3.15 (the recursive definition of append) in the left-hand side (step 11):

$$length(\boxed{a :: \underline{x <> \lfloor y \rfloor}}^{\uparrow}) = \boxed{s(\underline{length(x) + length(\lfloor y \rfloor)})}^{\uparrow}$$

- the `wave` sub-method applies wave-rule 3.20 (the recursive definition of length) in the left-hand side (step 12):

$$\boxed{s(\underline{length(x <> \lfloor y \rfloor)})}^{\uparrow} = \boxed{s(\underline{length(x) + length(\lfloor y \rfloor)})}^{\uparrow}$$

- the `wave` sub-method applies wave-rule 3.7 to obtain (step 13):

$$length(x <> \lfloor y \rfloor) = length(x) + length(\lfloor y \rfloor)$$

- this matches the induction hypothesis and strong fertilisation finishes the proof applying the sub-method `elementary`

Proof planning has been applied to other domains in addition to inductive theorem proving. These include summing series [Walsh *et al* 92], hardware configuration [Lowe 91], program synthesis [Kraan *et al* 93], bridge game playing [Frank *et al* 92], and data-type transformation [Richardson 95].

3.4 *Clam-Oyster*

The *Clam-Oyster* system has been developing since 1988 by the Mathematical Reasoning Group at Edinburgh, incorporating previous experience in meta-level and equational reasoning obtained from the *PRESS* system during the late 70s and early 80s [Sterling *et al* 82,Silver 83]. A description of the *Clam-Oyster* system is given in [Bundy *et al* 90] and here, we give a brief description, highlighting the features which will be relevant for understanding hardware verification with proof planning.

3.4.1 *Oyster*

Oyster is a tactic-based interactive proof editor for goal directed, backward chaining proofs. It is based on intuitionistic higher-order logic and type theory. It uses Prolog as a meta-language. It includes the types and constructors for natural numbers, integers, lists, disjoint union, functions, dependent functions, product, dependent product, quotient, subsets and recursion. A detailed description of the system is given in [Horn 95]. The user can create definitions and theorems which are proved using the inference rules of the logic and of the various types. For instance, the definition of addition over the natural numbers which we will be using in the following chapters, is given by:

```
plus(x,y)<==>p_ind(x,y,[p,r,s(r)])
```

where *p_ind* is the inductive constructor for the natural numbers, x is the induction variable, y is the value for the base case, and $[p, r, s(r)]$ describes the recursive computation of the function: $s(r)$ is the value at s(p) where r is the value at p. From this definition, we have derived a primitive recursive function of addition, given by equations 3.5, which must be justified as theorems in the theory of the

natural numbers [1]. We may also define addition using the extract term from a synthesis theorem in which the *p_ind* term is introduced.

```
plus(x,y)<==>term_of (plus) of x of y
```

This sort of definition facilitates the type-finding procedure during proof. All the object-level definitions as well as the equations and rewrite rules derived from the equations and used by *Clam* for proof planning, are treated in a similar way.

3.4.2 *Clam*

Clam implements the idea of proof planning. The main components of the *Clam* proof planner are: the logical objects, the method language, the planners, the library mechanism, and the wave-rule parser. The method language which consists of predicates and connectives, also uses Prolog as its meta-language. The library mechanism allow the user to manipulate the logical objects. The planners implement various search procedures like depth-first search, breadth-first search, iterative deepening search, and best-first-search to apply methods. The wave-rule parser generates automatically annotations of wave-rules and induction conclusions. These annotations are created dynamically when the wave-rules are required. A detailed description of the system is given in [vanHarmelen & group 96].

Clam has various types of logical objects and uses commands to handle them. For instance, an *Oyster* definition for addition is recognised by the command `def(plus)`. The base and recursive equations of addition are handled by giving them identifiers made from the name of the definition and a consecutive number, i.e., `plus1`, and `plus2`, and handling them by the commands `eqn(plus1)` and `eqn(plus1)` respectively. Theorems like the associativity of addition, given by formula 3.1 are named by an identifier, e.g. `assp` and handled by the command `thm(assp)`. Wave rules derived like 3.6 are identified by the name of the

[1] We have used a infix notation to represent arithmetic and list operations

formula from which they are derived and recognised by the command `wave` like in `wave(plus2)`. Methods, sub-methods, induction schemes, and proof plans are handled by commands `mthd`, `smthd`, `scheme`, and `plan` respectively, in a similar way.

Dependencies among logical objects are declared in a file called `needs` For instance, the logical objects needed by theorem 3.18 (called len_sum) are declared by the following entry in the needs file:

```
needs(thm(len_sum) [def(plus),def(length),def(append),wave(cnc_s)])
```

Logical objects can be loaded from and saved into the library with commands like `lib_load(T)` and `lib_save(T)`, respectively. For instance, by saying

```
lib_load(thm(len_sum))
```

Clam will load the theorem `len_sum` along with all the logical objects declared in the `needs` file.

3.5 Summary

We have presented a review of the proof planning technique as developed by the *MRG* group. We have described the notion of methods, which is the main component of proof planning. We have also described inductive proof planning and its main heuristics and search control mechanisms: *rippling analysis*, *rippling*, and *fertilise*, and have illustrated inductive proof planning with an example, along with a description of the *Clam-Oyster* proving system. This description provides us the background knowledge of proof planning required to explore now its use in verifying hardware.

Chapter 4

Hardware Verification

Proof planning can be applied to automate hardware verification. This is the main hypothesis of this thesis. In this chapter we illustrate the use of proof planning for hardware verification by applying it to the verification of two simple circuits, and introduce the basic ideas. Section 4.1 describes basic elements for hardware verification: types, operations on types, and conversion operations between types; section 4.2 explains the proof planning of a non-recursive circuit: a full adder; section 4.3 explains the proof planning of a recursive circuit: an n-bit incrementer; and section 4.4 presents a summary or the chapter.

4.1 Basic elements

In this section we describe the basic elements that we will need for our hardware verification task: types, operations on types, and conversion functions between types. Circuits are represented using a functional representation as described in section 2.2.4 and computer words are formalised using lists.

4.1.1 Types

The main types we will be using include: Booleans, words, natural numbers, signals, and states. Other types that we may use will be introduced when needed.

Booleans

The type *Boolean* is represented as a disjunction of two unary types, where *true* is the left injection and $false$ is the right injection of the disjoint type.

This definition is influenced by the fact that we are using an intuitionistic logic, type theory system (i.e. *Oyster*) at the object level, but the results are independent of initial design decisions like this.

Words

Words (or bit-vectors) are represented as lists of booleans:

$$word \quad = \quad bool\ list$$

The elements of the type (τ *list*) are *nil* and terms of the form $h :: t$, where *nil* stands for the empty list, :: is the list constructor, h is an element of type τ, and t is a list of type τ *list*. Inductive term constructors are supplied (e.g. `list_ind`), from which primitive recursive functions defined over lists, like the append of two lists, the reverse of a list, and others, can be obtained. Most significant bits can be either to the left or to the right of the list. The latter representation is called *big-endian*, the former is called *little-endian*.

We can parameterise words on their length using the following set type:

$$wordn(n) = \{w : word | length(w) = n\}.$$

The output of a hardware device is given as a word containing a finite number of Boolean values, so we will frequently use these types to formalise the output of circuits like adders, arithmetic logic units, shifters, memories and other devices.

Computer memory will be represented by a list of words of length n:

$$memn(n) \quad = \quad wordn(n)\ list$$

Natural numbers

The type *pnat* supplies the natural numbers. The elements of *pnat* are 0 and terms of the form $s(\ldots)$, where s is the successor function on natural numbers. Inductive

definition terms are supplied (e.g. `p_ind`), from which primitive recursive functions defined over pnat, like addition, subtraction, division and multiplication, can be obtained.

This type will be frequently used to formalise the specification of a hardware device, abstracting the internal working of the device.

Signals

Signals arise in modelling sequential circuits. A signal is represented by a function type from time (represented by the natural numbers) to words:

$$signal \quad = \quad time \rightarrow word$$

The output of data-path components which are transmitted by a bus, are usually formalised using signals. Control signals are represented by another function type $flag$ from time to Booleans:

$$flag \quad = \quad time \rightarrow bool$$

Signals generated by a control unit, usually are of type $flag$.

States

States arise when modelling the operation of a computer system. A state is represented by a product type where the types of the elements of the product depend on the kind of state being represented:

$$state \quad = \quad type_1 \# \ldots \# type_n$$

where # is the product type constructor. Elements of this type are formed by the operator &, e.g. $t_1 \& \ldots \& t_n$ is an element of the type $state$. For instance, a computer state at the instruction level ma be represented by the contents of the memory, the program counter, the accumulator, and a status flag which indicates whether the computer is idle or running:

$$inst\_state \quad = \quad memn(16) \# wordn(13) \# wordn(16) \# bool$$

where $memn(16)$, $wordn(13)$ and $wordn(16)$ are types for representing a memory of 16-bit words, words of 13 bits (e.g. the program counter), and words of 16 bits (e.g. the accumulator register).

A computer state at the register transfer level is often represented by the contents of the memory and various registers of different lengths like the program counter (13 bits), the accumulator (16 bits), the status flag (1 bit) , the buffer (16 bits), the memory address register (13 bits), the instruction register (16 bits), the argument register (16 bits), the microcode program counter (5 bits) and the ready flag (1 bit). Its type is given by:

$$rt\_state \quad = \quad memn(16)\#wordn(13)\#wordn(16)\#\ldots\#bool$$

We will see in the next sections and in chapter 5 how all these types are used to formalise and verify combinational and sequential hardware.

4.1.2 Operations

We define some operations on types, the ones we will be using in modelling hardware. A type operation is a function that takes elements of the type and produces another element of the same type.

Boolean logic

A Boolean logic comprises the type Boolean and operators on this type which are defined in terms of a built-in decision operator. For instance, the *and* operator applied to two variables U and V is defined by:

$$and(U,V) = \textit{if } U \textit{ is true then } V \textit{ else false}$$

From this definition we can derive properties on booleans like the *and* operation which is given by:

$$\begin{aligned} and(true, V) &= V \\ and(false, V) &= false \end{aligned}$$

These functions correspond to theorems which are justified from the primitive definition of *and*. From these equations, rewrite rules are obtained for proof planning. The other Boolean operators are defined in a similar way: *not*, *or*, *xor*, *imp*, and *equiv*. We these Boolean functions we will be able to formalise Boolean gates like AND, OR, INVERTER, XOR, NAND, etc., as well as basic hardware devices such as an adder which sums two bits.

Word operations

Primitive recursive functions defined over lists like *append*, *length* and *reverse* defined in chapter 3, apply also to words. The bit-wise *and* operation is defined by:

$$\begin{aligned} and\_w(nil, nil) &= nil \\ and\_w(H1 :: U, H2 :: V) &= and(H1, H2) :: and\_w(U, V) \end{aligned} \tag{4.1}$$

Other bit-wise Boolean operations, like *not*, *or* and *xor*, are defined in a similar way and will be useful in specifying the behaviour of an arithmetic logic unit.

Natural number operations

The following operations will be useful in specifying the behaviour of arithmetic circuits like multipliers, dividers, exponentiators and factorial circuits. The following are primitive recursive function defined on the natural numbers. Multiplication is defined by:

$$\begin{aligned} 0 * V &= 0 \\ s(U) * V &= U * V + V \end{aligned}$$

Exponentiation is defined by:

$$\begin{aligned} 2^0 &= s(0) \\ 2^{s(U)} &= 2 * 2^U \end{aligned}$$

where 2 is an abbreviation for $s(s(0))$. Factorial is defined by:

$$\begin{aligned} fact(0) &= s(0) \\ fact(s(U)) &= s(U) * fact(U) \end{aligned}$$

And division on the natural numbers (quotient and remainder) are defined by:

$$\begin{aligned} U/0 &= 0 \\ U < V \rightarrow U/V &= 0 \\ (U+V)/V &= s(U/V) \end{aligned}$$

$$\begin{aligned} rem(U,0) &= 0 \\ U < V \rightarrow rem(U,V) &= U \\ rem((U+V),V) &= rem(U,V)) \end{aligned}$$

4.1.3 Conversion functions

Functions that convert from one type into another type are necessary for comparing the specification and the implementation of a circuit, which are frequently defined in different types, as opposed to operations on a type, whose application produces elements of the same type. For instance, if the specification of a circuit is given by an equation in the natural numbers and the circuit gives a word as output, we need to convert the word into a natural number to make it comparable with the specification.

The function $bool2nat$ converts a Boolean into a natural number:

$$\begin{aligned} bool2nat(false) &= 0 \\ bool2nat(true) &= s(0) \end{aligned}$$

The primitive recursive function $word2nat$ converts a word in big endian format into a natural number:

$$\begin{aligned} word2nat(nil) &= 0 \\ word2nat(H :: U) &= bool2nat(H) + 2 * word2nat(U) \end{aligned} \tag{4.2}$$

$word2nat\_l$ converts a word in little endian format into a natural number:

$$\begin{aligned} word2nat\_l(nil) &= 0 \\ word2nat\_l(H :: U) &= bool2nat(H) * 2^{length(U)} + word2nat\_l(U) \end{aligned}$$

Sometimes we need the inverse operation, that is, converting a natural number into a word of a certain length. This is obtained by the following primitive recursive function:

$$\begin{aligned} nat2word(0, N) &= 0 \\ nat2word(s(len), N) &= nat2bool(N) :: nat2word(len, half(N)) \end{aligned}$$

where $nat2bool$ is a primitive recursive function that converts the number 0 into $false$ and the number $s(0)$ into $true$.

The function abs_imp converts a register-transfer level state into an instruction level state:

$$\forall x_1 : memn(16), x_2 : wordn(13), x_3 : wordn(16), x_4 : bool, x_5 : wordn(16),$$

$$x_6 : wordn(13), x_7 : wordn(16), x_8 : wordn(16), x_9 : wordn(5), x_{10} : bool$$

$$abs\_imp(x_1 \& x_2 \& x_3 \& x_4 \& x_5 \& x_6 \& x_7 \& x_8 \& x_9 \& x_{10}) = (x_1 \& x_2 \& x_3 \& x_4)$$

4.2 A non-inductive proof

We describe now, the verification of a *full-adder* using a non-inductive proof plan. The full adder is a non-recursive combinational circuit which serves as a building-block of more complex circuits. Basically, it is a 1-bit adder which takes as input two bits a, b and a carry input bit c and produces a sum bit fa_sum and a carry output bit fa_carry.

4.2.1 Formalisation

The formalisation of the full adder is given by formulae that describe the specification, the implementation and the verification conjecture.

The specification of a full adder is given by the equation:

$$full\_adder\_spec(a, b, c) = bool2nat(a) + bool2nat(b) + bool2nat(c)$$

An implementation of the full adder is displayed in figure 4–1. The sum fa_sum

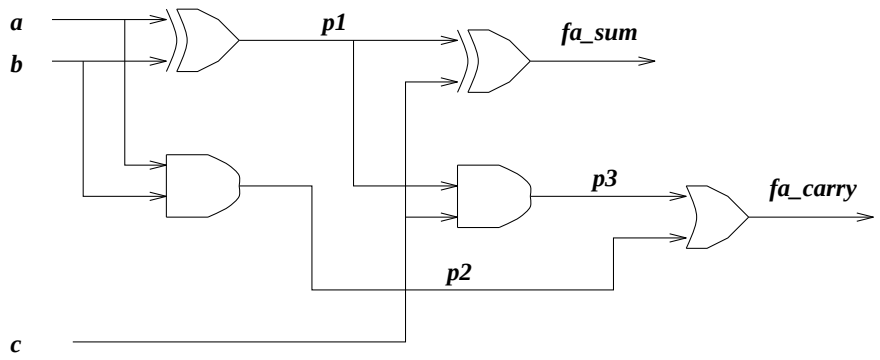

Figure 4–1: 1-bit Full adder

is obtained by two XOR gates connected by an internal wire p_1. The carry output fa_carry is obtained from two AND gates, one OR gate, one XOR gate, and three internal wires p_1, p_2, p_3. These circuits are formalised as follows:

$$\begin{aligned} fa\_sum(a, b, c) &= xor(a, xor(b, c)) \\ fa\_carry(a, b, c) &= or(and(xor(a, b), c), and(b, c)) \\ full\_adder(a, b, c) &= fa\_sum(a, b, c) :: fa\_carry(a, b, c) :: nil \end{aligned}$$

The verification of the full adder is stated by the following conjecture:

$$\vdash \forall a, b, c : bool\ word2nat(fa\_adder(a, b, c)) = full\_adder\_spec(a, b, c)$$

4.2.2 Verification

The verification is done in two stages: proof planning and execution of the composite tactic (proof plan) resulting from the proof planning.

Proof planning

The conjecture is proof planned automatically by *Clam* using the depth-first planner. The method `sym_eval` solves the conjecture by doing rewriting, Boolean case analysis on the variables a, b and c, and propositional reasoning. Figure 4–2 displays the structure of the proof plan. There are 22 method and sub-method application shown by the numbers in the nodes of the tree.

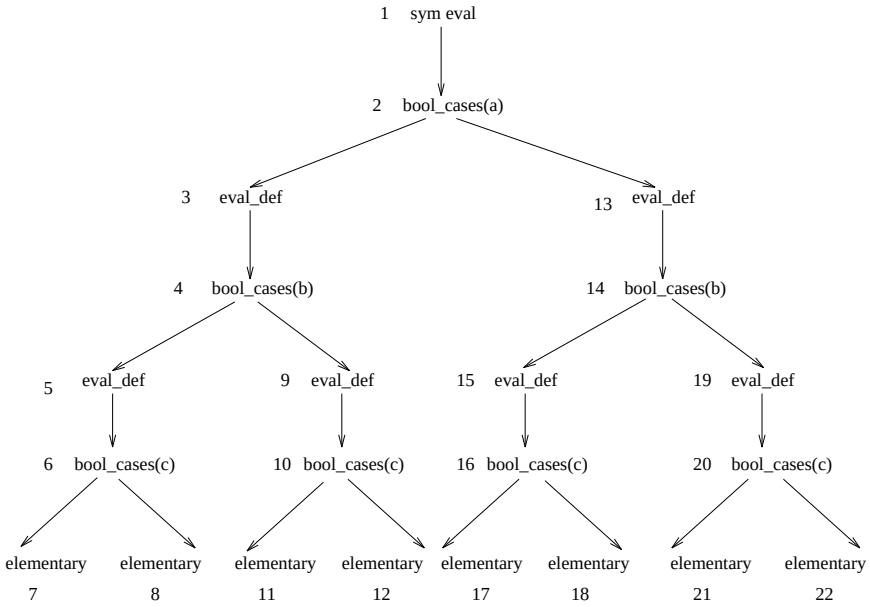

Figure 4–2: Structure of proof plan for verifying the full adder

- the method `sym_eval` applies symbolic evaluation to simplify the goal by calling the sub-methods `eval_def`, `bool_cases` and `elementary`.

- `eval_def` rewrites using the rules derived from the definitions of $word2nat$, $times$, $plus$, fa_sum, and fa_carry.

- `bool_cases` does a case split on variable a to give two subgoals, one for the $false$ case and another for the $true$ case (this sub-method is described in chapter 6).

- `eval_def` simplifies again the resulting subgoals using the rewrites of *and*, *or*, *xor*, *bool2nat* and *plus*. Similar simplifications are done for the boolean variables b and c giving 8 cases which are solved by propositional reasoning using `elementary`.

Plan execution

The proof plan is then executed by the *Oyster* system which proves the conjecture executing the tactic associated to each method. The formalisation of this proof is displayed in appendix B.

Statistics

The proof planning is done by *Clam* in 4 seconds. The execution of the proof plan is done by *Oyster* in 1:45 minutes. In general, plan execution times are higher than proof planning times. This has to do with the use of a prover based on type theory with time consuming type checking sub-goals.

4.3 An inductive proof

We now describe the verification of a recursive combinational circuit using an inductive proof plan. An n-bit incrementer takes as input a word x of length n and a Boolean carry input c, and produces as output a word of length $n+1$ which is the result of adding c to x. If c is $true$ then x in incremented by 1 otherwise x is left unchanged. In both cases there is a carry output c_o.

4.3.1 Formalisation

The formalisation of the n-bit incrementer is given by formulae that describe the specification, the implementation and the verification conjecture.

The specification of an n-bit incrementer is described by the following equation:

$$inc\_spec(x, c) = word2nat(x) + bool2nat(c)$$

An implementation of the n-bit incrementer is obtained by propagating the effect of adding the carry input to the least significant bit of the word to be incremented. This effect is achieved by cascading n half-adders as shown in figure 4–3. A half-adder is a full-adder with two inputs instead of three and is defined by:

$$ha\_sum(a, c) = xor(a, c)$$

$$ha\_carry(a, c) = and(a, c)$$

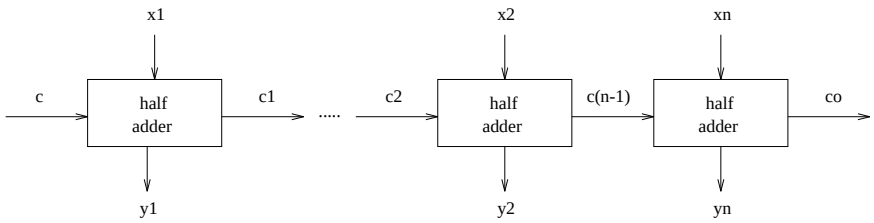

Figure 4–3: Implementation of n-bit incrementer

This design is formalised by the following recursive function:

$$\begin{aligned} inc(nil, c) &= c :: nil \\ inc(a :: x, c) &= ha\_sum(a, c) :: inc(x, ha\_carry(a, c)) \end{aligned} \tag{4.3}$$

The conjecture that establishes the equivalence between the specification and the implementation is the following:

$$\vdash \forall c : bool,\ \forall x : word,\ word2nat(inc(x, c)) = word2nat(x) + bool2nat(c)$$

The specification is given unfolded, because this way, we can have a wave-rule from the conjecture, once we prove it.

4.3.2 Verification

The verification is done again in two stages: proof planning and execution of the composite tactic (proof plan) resulting from the proof planning.

Proof planning

The proof plan is generated automatically by *Clam* using the depth-first planner. The proof plan for this theorem is graphically displayed in figure 4–4.

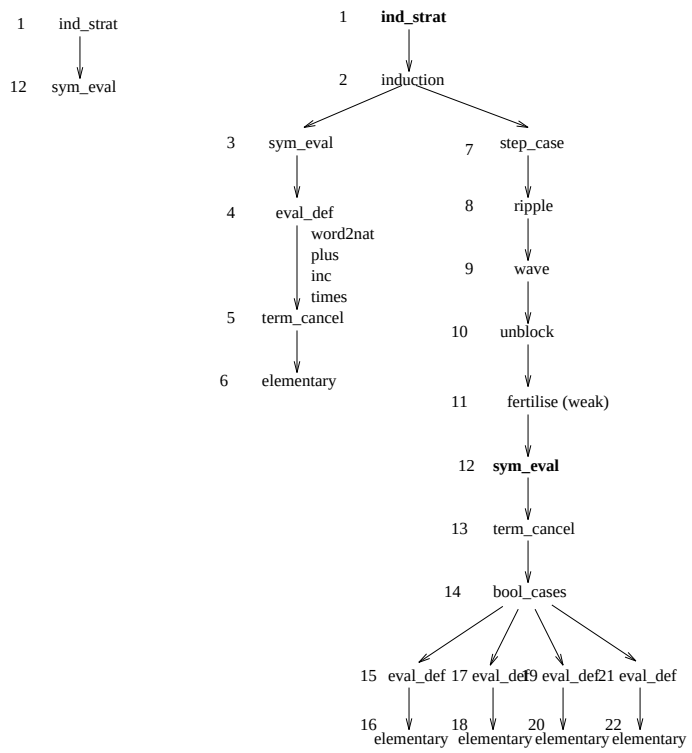

Figure 4–4: Structure of the proof plan for n-bit incrementer

The left-hand side shows method application and the tree in the right-hand side shows both method and (the main) sub-method application. There are 22 method/sub-method application steps indicated by the numbers on the nodes:

- the verification conjecture matches the input slot $H ==> G$ of the method `ind_strat`, with H set to the empty list and G set to the verification conjecture (step 1);

- the sub-method `induction` is tried: rippling analysis and the recursive definition of *inc* suggest an induction on *word* using variable x (step 2). This induction scheme is a special case of induction on lists where the elements of the list are of type *bool*. Thus, we have the following new goals:

  the base case:

  $$word2nat(inc(nil, c)) = word2nat(nil) + bool2nat(c)$$

  and the step case:

  $$word2nat(inc(\boxed{v_0 :: \underline{v_1}}^{\uparrow}, \lfloor c \rfloor)) = word2nat(\boxed{v_0 :: \underline{v_1}}^{\uparrow}) + bool2nat(\lfloor c \rfloor)$$

- The base case is solved by symbolic evaluation (steps 3-6) as follows: rewriting is applied to the goal using rules from the equations for *word2nat*, *plus*, *inc* and *times*. Then the sub-method `term_cancel` (which is explained in chapter 6) applies arithmetic reasoning by cancelling out equal additive terms on both sides of an equation. Finally, the sub-method `elementary` finishes the remaining subgoal.

- The step case is simplified by the sub-method `step_case` as follows (steps 7-11): The annotations of the sub-term $word2nat(v_0 :: v_1)$ on the right-hand side of the equation match the annotations of the left-hand side of the following outwards wave-rule obtained from equation 4.2, the recursive definition of *word2nat*:

  $$word2nat(\boxed{H :: \underline{U}}^{\uparrow}) \Rightarrow \boxed{bool2nat(H) + 2 * \underline{word2nat(U)}}^{\uparrow} \qquad (4.4)$$

  so the `wave` method is applied to give:

  $$\ldots = \boxed{bool2nat(v_0) + 2 * \underline{word2nat(x)}}^{\uparrow} + bool2nat(\lfloor c \rfloor)$$

  The annotations of the sub-term $inc(v_1, c)$ on the left-hand side of the equation match the annotations of the left-hand side of the following outwards wave-rule obtained from equation 4.3, the recursive definition of *inc*:

  $$inc(\boxed{H :: \underline{U}}^{\uparrow}, V) \Rightarrow \boxed{ha\_sum(H, V) :: \underline{inc(U, ha\_carry(H, V))}}^{\uparrow} \qquad (4.5)$$

so the `wave` method is applied again to yield:

$$word2nat(\boxed{ha\_sum(v_0, c) :: \underline{inc(x, ha\_carry(v_0, c))}}^{\uparrow}) = \ldots$$

The annotations on the right-hand side of the equation match the annotations of the left-hand side of wave-rule 4.4, so that the `wave` method applies once more to give:

$$\boxed{bool2nat(ha\_sum(v_0, c)) + (2 * \underline{word2nat(inc(x, ha\_carry(v_0, c)))})}^{\uparrow} = \ldots$$

No more rippling or unblocking is possible, thus rippling finishes its role. But now the *fertilise* method applies, as the sub-term

$$word2nat(inc(x, ha\_carry(v_0, c)))$$

on the left-hand side of the equation matches the left-hand side of the induction hypothesis

$$word2nat(inc(U, V)) = word2nat(U) + bool2nat(V)$$

with the substitution $ha\_carry(v_0, c)/V$. Thus, the output of the `fertilise` method is obtained by rewriting from left to right with the induction hypothesis, giving:

$$bool2nat(ha\_sum(v_0, c)) + (2 * (word2nat(v_1) + bool2nat(ha\_carry(v_0, c))))$$

$$=$$

$$(bool2nat(v_0) + 2 * word2nat(v_1)) + bool2nat(c)$$

- The resulting subgoal is simplified by symbolic evaluation (steps 12-22) by applying arithmetic cancellation (this is a new method which will be explained in chapter 6), Boolean case analysis (also a new sub-method to be explained in chapter 6) on variables v_0 and c, rewriting and propositional reasoning on the four Boolean cases, to finish the proof.

Plan Execution

The resulting composite tactic (or proof plan) customised for proving this particular conjecture is then ran in *Oyster* which executes the tactic associated to each of the methods which compose the proof plan, obtaining in this way a verification proof.

Statistics

Clam plans the proof in 9 seconds. *Oyster* executes the plan in 41 seconds. The proof planning of the n-bit incrementer required no lemmas. Another feature of *Clam* is that, in general, because of its automation facilities, it requires very few lemmas in order to find proof plans, compared to other systems. However, proof plan generation can be made faster if we add certain lemmas. For instance if we add the lemma

$$x + 0 = x$$

Clam generates another proof plan in only 5 seconds.

4.4 Summary

We have presented two examples to introduce the basic ideas about the use of proof planning for hardware verification. The first example was a full-adder, a non-recursive, combinational circuit verified by case analysis. The second example was an n-bit incrementer, a recursive combinational circuit verified by inductive proof planning, using no lemmas in its verification. These two proofs are simple, but illustrate the way in which proof planning can be applied to verify circuits built by replicating a basic component. For this type of circuit we exploit the automation facilities of inductive proof planning to produce a proof. Tasks required involved formalising the problem, adding two new sub-methods (e.g. Boolean case analysis and arithmetic reasoning), planning the conjecture and executing the plan.

Chapter 5

A Methodology

The hardware domain presents many examples of circuits with replicated components and feedback loops. Formal methods provide techniques to reason about such circuits and remain an active area of research. Among formal methods are meta-level reasoning techniques, of which proof planning is a type. In chapter 4 we saw how poof planning is applied to verify simple circuits. Proof planning can be scaled up and applied systematically to verify hardware with replicated components and feedback loops. Our work shows that proof planning can be successfully applied at least in the following cases: verification of combinational circuits and verification of synchronous sequential circuits. In the former, if the circuit is built by replicating a set of basic hardware components, then it can be formalised using recursive functions. In the latter, if the circuit operation is controlled by a global clock, then we can formalise it using a finite-state machine to represent time transitions. Given appropriate extensions, both types of verification proofs can be done using inductive proof planning by reasoning about the number of components and the time parameter.

In this chapter we describe a simple methodology for hardware verification using proof planning for verifying both types of hardware. Section 5.1 describes a methodology based on the concept of *proof engineering*. Sections 5.2 and 5.3, apply the methodology to verify a set of combinational and sequential circuits of increasing complexity: an adder, an arithmetic logic unit, a parallel array multiplier, and

a simple microprocessor. Section 5.4 discusses the features of extendability and scalability of proof planning. Finally, section 5.5 summarises the chapter.

5.1 A Proof Engineering based Methodology

We describe a methodology based on the idea of *proof engineering* for the verification of hardware using proof planning. Proof engineering refers to the development of formal proof for systems (product) design and verification.

The methodology consists of partitioning automation of formal proof into tasks classified at three levels: user, proof and systems (tool) tasks. User level tasks have to do with formalising a particular verification problem and using a formal tool to obtain a proof. Tasks typically will include the following:

- formalise the problem by providing definitions of the specification and the implementation, and a statement of the verification conjecture;
- instruct a proof planner to use one of the planners and a predefined set of methods, to generate a proof plan; and
- execute the proof plan to obtain a proof
- if a proof is not obtained, then revise the formalisation, which may include bugs, do appropriate corrections, and try again.

In case of failure, then proof tasks come into action. Proof level tasks have to do with tuning proof techniques without modifying the tool system, and typically will include the following:

- apply a different method configuration;
- provide a possible missing lemma;
- modify an existing method and tactic;

- provide a new method and tactic;
- formalise a new theory.

If the problem persists (we expect this to be rarely the case), then systems level tasks, which refer to the modification of the proof system tool, may be necessary. This may include:

- correct a bug or inefficiency in either the planner or the prover;
- provide a new predicate or connective for the method language;
- provide a new planner or extend an existing one;
- extend the library mechanism with a new type of logical object;
- extend the wave-rule parser or provide a new one; or
- develop a new interface for a different tactic-based prover, a model checker, or a decision procedure.

The line between the tasks is sometimes rather fine, e.g. providing a different method ordering could be done at the user level as well, especially for users with a certain experience with the use of proof planning. Also, providing a new method or a new predicate are not that different, although, conceptually, they are different, because the latter involves a modification to the planner and the former is only input to the planner.

We believe that this methodology could be supported organisationally, by having different individuals with different levels of competence carrying out these tasks. For instance, the user level tasks could be performed by design engineers, with expertise in product design in a particular domain. The proof level tasks would be performed by an individual with experience in the use of formal methods for system design, and who we call a *proof engineer*. The systems level tasks will be done by software developers who build, maintain and commercialise formal methods tools. In this case, we have played these three roles.

The methodology proposed is not exclusive for hardware, it could equally be applied to applications in other domains e.g. software development, product configuration, product design, production scheduling, etc.

To formalise a verification problem, the user provides equations, both recursive and non-recursive for the specification and the implementation, a verification conjecture, and dependencies among logical objects. The conjecture is typically an equation stating that the specification is equal to an abstract form of the implementation. For combinational hardware it takes the form:

$$\forall x_1, \ldots, x_n \; conditions \rightarrow spec(x_1, \ldots, x_n) = abs(imp(x_1, \ldots, x_n))$$

In the case of synchronous sequential hardware, there will be a time parameter t, and if the specification and the implementation involve different time scales, then, we must provide a mapping function f that converts times from one scale into the other:

$$\forall t : time \; \forall x_1, \ldots, x_n$$

$$conditions \rightarrow spec(t, x_1, \ldots, x_n) = abs(imp(f(t), x_1, \ldots, x_n))$$

In either case, the same set of methods is applied to plan the conjecture.

In the following sections we illustrate the methodology by describing several examples of combinational and sequential circuits. In presenting these examples, we also emphasise two features of proof planning which support the methodology: *extendability* and *scalability*. Extendability means that the proof and systems tasks which involve extensions to proof planning to verify a particular circuit will also work to verify other circuits of the same type. Scalability means that the proof and systems tasks which involve extensions to proof planning to verify a particular circuit will also work to verify more complex circuits. We conjecture that these two features of extendability and scalability will hold for circuits of industrial-strength, if we apply to them a proof engineering methodology based on proof planning.

5.2 Combinational circuits

In this section we describe the verification of some combinational circuits using the methodology. The circuits presented in this section are: an *adder*, a *multiplier*, and an *arithmetic logic unit*. We focus on the aspects which are relevant to the verification methodology: user, proof, and systems level tasks.

5.2.1 An n-bit adder

An n-bit adder takes as input two words x and y of the same length n, and a carry input bit c and produces a word $adder$ of length $n+1$ which is the sum of x and y.

User tasks

Tasks involved here include formalising the problem and running *Clam-Oyster* to obtain a verification proof. The formalisation of the adder included experimenting with relational and functional representations. A relational formalism was tried first, just to find out that *Clam* (especially the rippling technique) was designed to deal with functional representations, and we preferred the functional formalism thereafter. Another representation decision was determining whether to use functions or lists to represent words. The first attempt was to use functions, but this revealed several inefficiencies of *Clam*'s wave-rule parser in handling large terms. The decision to use a functional formalism to represent circuits and lists to represent words was supported by reading Hunt's work with *Nqthm* [Hunt 86].

The specification is given by the equation:

$$adder\_spec(x, y, c) = word2nat(x) + word2nat(y) + bitval(c)$$

A common implementation of the n-bit adder is the ripple-carry adder obtained by cascading n full adders. This implementation is formalised by the following

recursive function, where the first two arguments are assumed to have the same length:

$$\begin{aligned} adder(nil, nil, c) &= c \\ adder(a :: x, b :: y, c) &= fa\_sum(a, b, c) :: adder(x, y, fa\_carry(a, b, c)) \end{aligned}$$

The verification conjecture has the form of a combinational circuit:

$$length(x) = length(y)$$

$$\rightarrow$$

$$word2nat(adder(x, y, c)) = word2nat(x) + word2nat(y) + bitval(c)$$

The next task is to run *Clam-Oyster* to get a proof. The proof plan that proves this conjecture is generated automatically by *Clam* and is displayed graphically in figure 5–1. Notice that it has the same structure as the proof plan for the incrementer circuit described in chapter 4. The left-hand side shows the method application and the right-hand side shows both, method and sub-method application. The planning takes 31 method and sub-method application steps, indicated by the numbers on the nodes of the tree, we briefly explain the main steps of the planning:

- rippling analysis, the recursive definition of the adder, and the condition $length(x) = length(y)$ suggest a double induction on the variables x and y (steps 1-2). An induction scheme with these characteristics was added to *Clam*'s database of induction schemes.

- the base case:

  $$word2nat(adder(nil, nil, c)) = word2nat(nil) + word2nat(nil) + bitval(c)$$

  is solved by symbolic evaluation, rewriting, term cancellation, and propositional reasoning (steps 3-6)

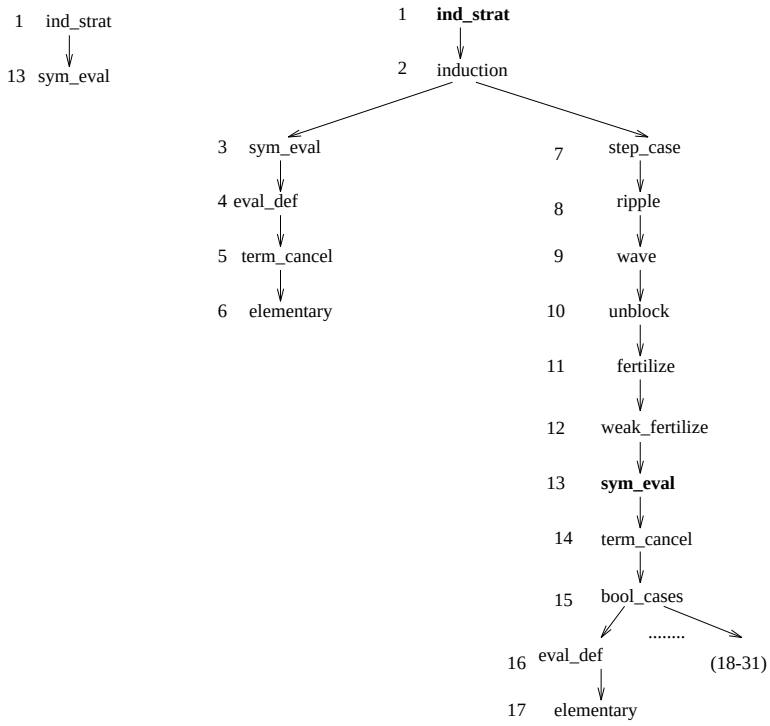

Figure 5–1: Proof plan for the n-bit Adder

- the step case:

$$word2nat(adder(\boxed{v_0 :: \underline{x}}^{\uparrow}, \boxed{v_2 :: \underline{y}}^{\uparrow}, c)) =$$

$$word2nat(\boxed{v_0 :: \underline{x}}^{\uparrow}) + word2nat(\boxed{v_0 :: \underline{x}}^{\uparrow}) + bitval(c)$$

  is simplified by rippling, by applying the `wave` sub-method with the wave rule 4.4, and the outwards wave rule derived from equation 5.1:

$$adder(\boxed{H :: \underline{U}}^{\uparrow}, \boxed{I :: \underline{V}}^{\uparrow}, C) \Rightarrow$$

$$\boxed{fa\_sum(H, I, C) :: \underline{adder(U, V, fa\_carry(H, I, C))}}^{\uparrow}$$

  Unblocking is applied next, and then weak-fertilisation (steps 7-13).

- the remaining subgoal is solved by symbolic evaluation applying term cancellation, Boolean case analysis on variables v_0, v_2, and c, which results in 8 cases, each of which is solved by rewriting and propositional reasoning (steps 13-31).

Plan execution is then carried by *Oyster* by executing the tactic associated to each of the methods which appear in the plan.

Proof tasks

The proof tasks consisted of: writing an extra clause for the Boolean case analysis method, writing a method for doing arithmetic reasoning; adding a new clause for weak fertilisation; modifying the `generalise` method to handle Boolean case analysis; adding a new scheme for induction on two words of the same length; experimenting with method orderings; and writing or updating tactics associated to the new or the updated methods. All this extensions are described in chapter 6 and their statistics are analysed in chapter 7.

- Boolean case analysis was first used for verifying the full-adder. The method contained two clauses to cover a situation in which a Boolean variable is among the variables of the goal and another situation in which a Boolean variable is already in the hypothesis list. A third clause was added for the adder to recognise Boolean terms like the head of a word and do case analysis on them. The tactic for this method was adapted from the case-split tactic.

- Cancellation of top-level additive terms on both sides of an equality was carried out by a term cancellation sub-method. The sub-method required some new predicates and was easy to implement.

- A key step for the successful planning of the adder verification conjecture was the addition of a clause in the weak fertilise method to handle wave-fronts that contain several sinks, each of which stores an inwards wave-front. With this addition, the step case was completed and the proof was finished by term cancellation and Boolean case analysis.

- The generalisation method was slightly modified to add a precondition for generalising Boolean terms. The lack of this condition would cause *Clam* to loop, reflecting a bug in the type guessing predicate.

- A special induction scheme was added to *Clam*'s data base to verify hardware devices like the adder, in which there are replicated components and two input words of the same length.

- For this particular proof we had to apply induction first to avoid applying Boolean case analysis and unfolding of definitions which made proof by induction more difficult. This was done by a change in method ordering and required experimentation with various orderings. Otherwise, backtracking when a method fails will generate a different ordering, which in this case, will find the desired proof plan.

- The modification to the tactics were in most cases straightforward except the one for term cancellation which was very difficult to implement as explained in 6.

Systems tasks

The systems level extensions were minor, although in some cases they were time consuming (e.g. parsing well-annotated terms), and consisted of: writing a new method predicate for finding terms of type Boolean, updating the type information of the Prolog program which does type checking, and correcting inefficiencies (e.g. the predicate for well-annotated terms was implemented in such a way that made un-necessary, time consuming calculations).

5.2.2 Arithmetic Logic Unit

An Arithmetic Logic Unit (ALU) takes as input two words x and y of the same length n, a carry input c, and three selection boolean variables s_0, s_1, s_2 which determine the kind of micro-operation to execute, and produces a word of length $n + 1$. It is interesting to note that although this circuit is more complex that the adder, *its proof did not require any new extensions* to *Clam-Oyster*. The ALU performs the 12 operations displayed in the following table:

Selection				
s_2	s_1	s_0	c	Operation
0	0	0	0	Move x
0	0	0	1	Increment x
0	0	1	0	Add x and y
0	0	1	1	Add x and y with carry
0	1	0	0	Subtract y from x with borrow
0	1	0	1	Subtract y from x
0	1	1	0	Decrement x
0	1	1	1	Move x with carry
1	0	0	X	Bitwise Or of x and y
1	0	1	X	Bitwise Xor of x and y
1	1	0	X	Bitwise And of x and y
1	1	1	X	Not of x (complement)

When s_2 equals zero, we have an arithmetic operation. The carry input c is used to further determine the operation. When s_2 equals 1 we have a logic operation and the value of c is a *don't care*, which is indicated by X.

User tasks

The formalisation of this circuit proceeded as follows: the implementation has the same form as the incrementer and the adder, thus, implementing the ALU was straightforward; the specification gave more trouble and most of the time was spent debugging each of its 12 components.

The specification of the ALU is formalised in terms of the adder and bit-wise Boolean operations on words. When $c = false$ in

$$adder(x, nat2word(length(x), 0), c)$$

we have the effect of transferring x, when $c = true$, x is incremented by 1. When $c = false$ in $adder(x, y, c)$, we get the addition of x and y, when $c = true$ we have addition with carry. Subtraction of two words x and y is defined in terms of

addition by adding to x the 1-complement representation of y:

$$subtract(x, y, c) = adder(x, not\_w(y), c)$$

When $c = false$ we have subtraction with borrow, when $c = true$, y is subtracted from x. Bitwise boolean operations on two words are specified by the functions or_w, xor_w, and and_w. Negation of a word is specified by not_w.

There are several possible hardware designs to implement the micro-operations of the ALU. One approach is to construct and verify a 1-bit ALU that implements the 12 micro-operations and then cascade n of them to obtain an n-bit ALU. Figure 5–2 shows a hardware implementation for a 1-bit ALU that implements the 12 micro-operations [Mano 79].

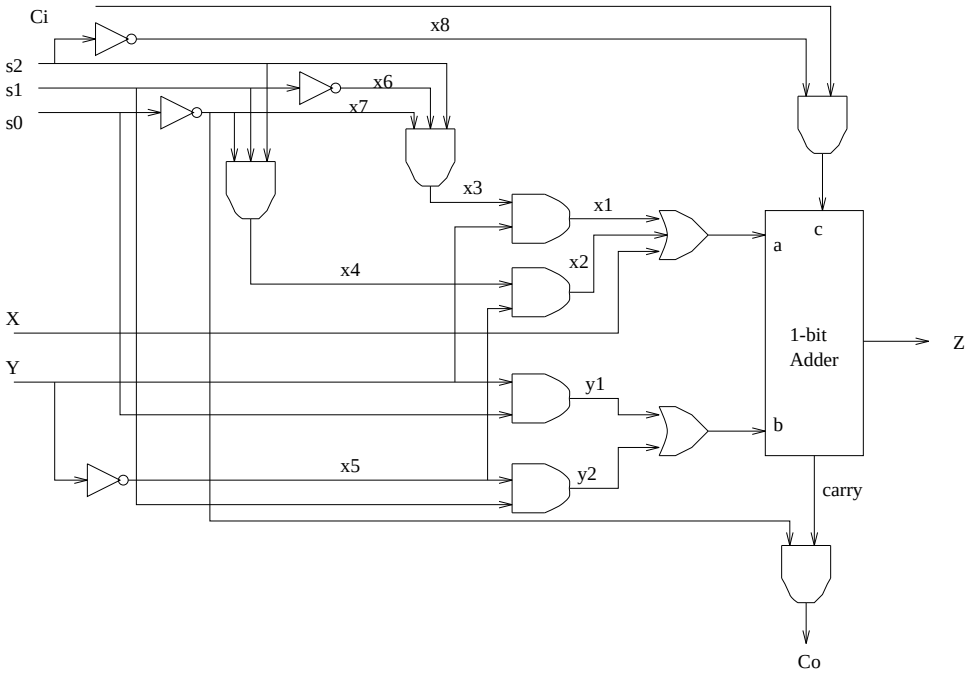

Figure 5–2: Hardware Design of a 1-bit ALU

The verification of this circuit is done in *Clam-Oyster* by Boolean case analysis on the six variables giving 64 cases.

The n-bit ALU can then be implemented by cascading n 1-bit ALUs. This implementation is formalised by the following recursive function:

$$alu(s_0, s_1, s_2, nil, nil, c) = and(not(s_2), c) :: nil$$

$$alu(s_0, s_1, s_2, a :: x, b :: y, c) \tag{5.1}$$

$$=$$

$$alu\_sum(s_0, s_1, s_2, a, b, c) :: alu(s_0, s_1, s_2, x, y, alu\_carry(s_0, s_1, s_2, a, b, c)) \tag{5.2}$$

The verification conjecture has the form of a combinational circuit:

$$length(x) = length(y)$$

$$\rightarrow$$

$$alu(s_0, s_1, s_2, x, y, c) = alu\_spec(s_0, s_1, s_2, x, y, c)$$

The next task is to run *Clam-Oyster*. The proof plan is generated automatically by *Clam* and is displayed graphically in figure 5–3. Again, notice that it has the same structure as the structure of the proof plans for the incrementer and the adder. The left-hand side shows the method application and the right-hand side shows both, method and sub-method application. The planning takes 134 method

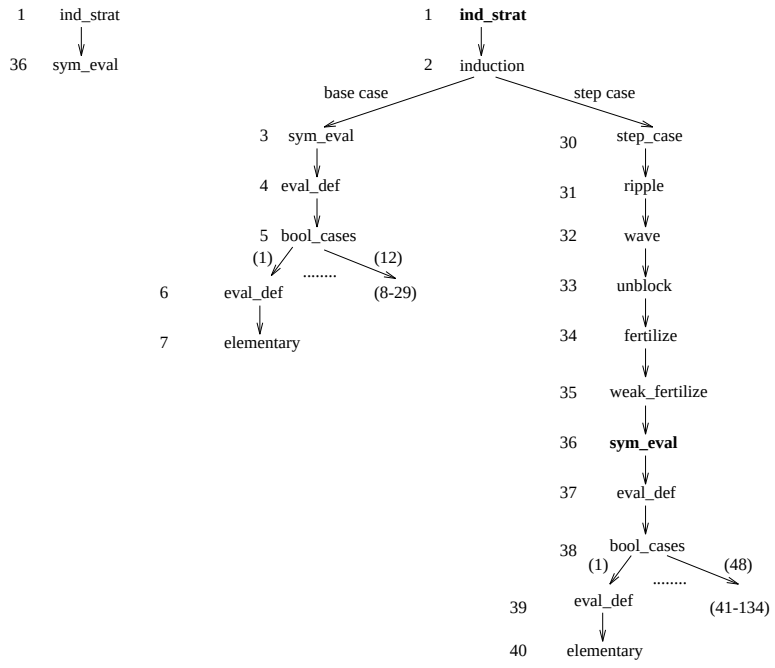

Figure 5–3: Proof plan for the n-bit ALU

and sub-method application steps, indicated by the numbers on the nodes of the tree:

- rippling analysis, the recursive definition of the ALU, and the condition $length(x) = length(y)$ suggest a double induction on the variables x and y (steps 1-2).

- the base case:

  $$alu(s_0, s_1, s_2, nil, nil, c) = alu\_spec(s_0, s_1, s_2, nil, nil, c)$$

  is solved by symbolic evaluation, rewriting, Boolean case analysis on s_2, s_1, s_0, and c (12 cases, one for each micro-operation), and propositional reasoning (steps 8-29)

- the step case:

  $$alu(s_0, s_1, s_2, \boxed{v_0 :: \underline{x}}^{\uparrow}, \boxed{v_2 :: \underline{y}}^{\uparrow}, c) = alu\_spec(s_0, s_1, s_2, \boxed{v_0 :: \underline{x}}^{\uparrow}, \boxed{v_2 :: \underline{y}}^{\uparrow}, c)$$

  is simplified by rippling, by applying the wave sub-method with the outwards wave rule derived from equation 5.1:

  $$alu(s_0, s_1, s_2, \boxed{H :: \underline{U}}^{\uparrow}, \boxed{I :: \underline{V}}^{\uparrow}, C)$$

  $$\Rightarrow$$

  $$\boxed{alu\_sum(s_0, s_1, s_2, U, V, C) :: \underline{alu(s_0, s_1, s_2, x, y, alu\_carry(s_0, s_1, s_2, a, b, c))}}^{\uparrow}$$

  unblocking is applied next, and then weak-fertilisation (steps 30-35).

- the remaining subgoal is solved by symbolic evaluation applying rewriting, Boolean case analysis on variables $v_0, v_2, s_2, s_1, s_0, c$ which results in 48 cases, each of which is solved by rewriting and propositional reasoning (steps 39-134).

Next, plan execution is carried by *Oyster* by executing again the tactic associated to each of the methods which appear in the plan.

Proof tasks

We were surprised to see that there were no proof tasks involved. The extensions made for the adder were exactly the ones needed for the ALU. This example illustrates the extendability and scalability features of proof planning.

Systems tasks

The systems tasks were again minor although time consuming, and consisted mainly of debugging the code and finding a way around the inefficiencies of *Clam* in parsing large terms. For instance, the wave-rule parser of version 2.2 could not load the specification of the ALU in a reasonable time (e.g. it would run for two hours without finishing). The reason is that it was looking for well-annotated terms in a very inefficient way. A new implementation of the wave rule parser in version 2.3 solved this problem. For the time being, we had to patch the library mechanism program to avoid un-necessary calculations.

5.2.3 Multiplier

This circuit takes as input two words of lengths n and m and outputs a word of length $n + m$ corresponding to the product of x and y. Multiplication of binary numbers can be implemented by a simple parallel array multiplier using binary additions. Consider, for example, multiplying a 3-bit word by a 2-bit word. This is represented by:

$$\begin{array}{ccccc} & & x_2 & x_1 & x_0 \\ & & \mathrm{X} & y_1 & y_0 \\ \hline & & x_2y_0 & x_1y_0 & x_0y_0 \\ & x_2y_1 & x_1y_1 & x_0y_1 & \\ \hline z_4 & z_3 & z_2 & z_1 & z_0 \end{array}$$

to multiply an n bit word by an m bit word the array multiplier uses $n * m$ AND gates to compute each of these intermediate terms in parallel, and then m binary additions are used to sum together the rows. This requires a total of $n * m$ one bit adders.

User tasks

Formalisation of this circuit was easier using a little endian representation of words (i.e. more significant bits to the left).

The specification of the multiplier is expressed by the following formula:

$$mult\_spec(x, y) = word2nat(x) * word2nat(y)$$

The implementation is formalised by the following recursive function:

$$mult(x, nil) = zeroes(length(x))$$

$$mult(x, h :: y) =$$

$$adder\_l(mult\_one(x, h) <> zeroes(length(y)), mult(x, y), false) \tag{5.3}$$

where $<>$ is the list append operation, $zeroes(n)$ yields n zeroes, $mult\_one$ multiplies the word x times the boolean h; $adder\_l$ is an n bit adder which sums words in little endian notation.

The conjecture which states the equivalence of the specification and the implementation is as follows:

$$\forall x, y : word,\ word2nat(x) \times word2nat(y) = word2nat(mult(x, y))$$

The proof plan is generated automatically by *Clam* and is graphically displayed in figure 5–4. Once again, notice that the structure of this proof plan is similar to the ones for the *incrementer*, the *adder*, and the ALU, except that the induction strategy is applied 7 times instead of 1. The planning takes 9 method application

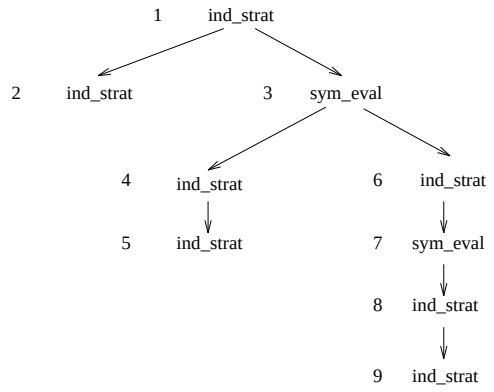

Figure 5–4: Proof plan for the nm-bit multiplier

steps, indicated by the numbers on the nodes of the tree. In this tree we show just the method application and omit the sub-methods, to make the tree more readable:

- rippling analysis, and the recursive definition of $mult$, suggest an induction on the variables y (step 1).

- the base case:

$$word2nat(x) * word2nat(nil) = word2nat(mult(x, nil))$$

  is simplified by symbolic evaluation applying rewriting, giving:

$$0 = word2nat(zeroes(len(x)))$$

  which is solved by another application of the induction strategy by an induction on x (step 2)

- the step case:

$$word2nat(\lfloor x \rfloor) * word2nat(\boxed{v_0 :: \underline{v_1}}^{\uparrow}) = word2nat(mult(\lfloor x \rfloor, \boxed{v_0 :: \underline{v_1}}^{\uparrow}))$$

  is simplified by rippling, by applying the wave sub-method with the outwards wave-rule derived from equation 5.3:

$$mult(U, \boxed{H :: \underline{V}}^{\uparrow}) \Rightarrow$$

$$\boxed{adder\_l(mult\_one(U, H) <> zeroes(length(V)), \underline{mult(U, V)}, false)}^{\uparrow}$$

  the wave rule obtained from the verification of the n-bit adder, $word2nat$, distributivity of times over plus, unblocking, and weak fertilisation, to yield:

$$word2nat(multOne(x, v_0) <> zeroes(length(v_1))) + word2nat(mult(x, v_1))$$

$$=$$

$$word2nat(x) * (bitval(v_0) * exptwo(length(v_1))) + word2nat(mult(x, v_1))$$

  This equation involves mostly arithmetic reasoning, and is solved by symbolic evaluation and the induction strategy in steps 3-9.

Plan execution is then carried by *Oyster* by executing the tactic associated to each of the methods which appear in the plan.

Proof tasks

The main proof task was finding the various lemmas required for the proof. The extensions made for the adder worked for the multiplier as well. The lemmas needed correspond to the verification of the n-bit adder, distributivity of times over plus, associativity of times, and reduction rules like $x + 0 = x$, $x * 0 = 0$. Reduction rule like this can be solved by induction, although they occur in a context in which the equation in quite complicated and the planning will take more time.

Systems tasks

There were no systems tasks involved for this proof.

5.3 A sequential circuit

We now present the verification of a sequential circuit: a simple microprocessor, called the *Gordon computer*. This is a 16-bit microprocessor, with 8 programming instructions, no interrupts, and a synchronous communication interface with memory, designed by Mike Gordon and his group at Cambridge University. Its architecture has been described elsewhere [Joyce *et al* 86,Camilleri 88,Joyce 90]. There are three sets of lights: 13 PC display lights which show the contents of the program counter, 16 ACC display lights which show the contents of the accumulator, and the idle light which is *on* when the computer is idling (i.e. not executing a program). There are also 16 two-position switches which are used for inputing data. There are also 3 boolean switches and a knob in the front panel. Pushing the *button* switch when the computer is running (i.e. executing a program) interrupts the execution of the program and the computer idles. The effect of pushing the *button* switch when the computer is idling is determined by the position of the *knob*. When the knob is in position 0 the effect of pushing the button is that the word determined by the state of the thirteen rightmost switches is loaded into the

program counter. Pushing the button when the knob is in position 1 loads the word determined by the sixteen switches into the accumulator. When the knob is in position 2 the contents of the accumulator is stored in the memory at the location held in the program counter. Finally, knob position 3 is used to start the execution of the program stored in memory beginning at the location in the program counter. When execution of a program begins the *idle* light goes off and stays off until execution stops. Execution can only stop if a $HALT$ instruction is encountered or an interrupt is generated by pushing the button.

The specification, given at the instruction level, is defined in terms of the semantics of the 8 programming instructions. Each instruction consists of the set of operations that determines a new computer state, where a state is determined by the contents of the memory, the program counter, the accumulator and the idle/running status of the computer. The execution of an instruction defines a transition from a state into a new state and this transition determines the time-unit of an instruction-level time-scale. Thus, for each instruction we must specify the way in which each of the four components of a computer state are calculated. The implementation is at the register-transfer level. It consists of a data-path and a microprogrammed control unit. A computer state at the register-transfer level is determined by the contents of 11 components: the memory, the program counter, the accumulator, the idle/running flag, the memory address register, the instruction register, the argument register, the buffer register, the bus, the micro-code program counter and the ready flag. The control unit generates the necessary control flags to update the computer registers. Communication between the bus and the registers is regulated by a set of gates. The implementation uses a micro-instruction time-scale. The number of micro-instructions required to compute a given instruction is calculated automatically by using the ready flag in the microcode, and the associated time at the micro-instruction-level time-scale mapped onto the respective time at the instruction-level time-scale. The correctness theorem asserts that the state of the computer at the specification level is equal to an abstract state of the implementation level each time an instruction is executed.

The computer has the following programming instructions, each consisting of a 3-bit operation code and a 13-bit address:

opcode	address	mnemonic	description
000	13 bits	HALT	stops execution
001	13 bits	JMP L	jump to address
010	13 bits	JZR L	jump to address if ACC=0
011	13 bits	ADD L	add contents of address to ACC
100	13 bits	SUB L	subtract contents of address from ACC
101	13 bits	LD L	load contents of address into ACC
110	13 bits	ST L	store contents of ACC in address
111	13 bits	SKIP	skip to next instruction

5.3.1 User tasks

The formalisation involved activities like: translating from the relational specification given in [Joyce *et al* 86] into a functional representation, and debugging the definitions. The translation of representation was time-consuming and it would have been easier to start from scratch. Other user tasks included: defining the types and converts for signals and states described in chapter 4, providing a time-scale mapping function, and characterising the stability of signals.

There are two time-scales, the instruction time-scale and the micro-instruction time-scale. Each time an instruction is executed the instruction time-scale is incremented by 1. Each instruction requires a variable number of micro-operations to execute. This variability is controlled by the *ready* flag which becomes true at the beginning and at the end of each instruction execution. Each time a micro-operation is executed the micro-instruction time-scale is incremented by 1. The function *microtime* takes an instruction level time and other arguments and converts it to a micro-instruction level time. There are also abstraction functions for the signals *swithches*, the *knob*, the *button* and the *idle* flag to map them between the two time-scales.

$$microtime(s(u), swt, knob, button) \;=\; iterate\_time(u, swt, knob, button)$$

$$
\begin{aligned}
iterate\_time(u, swt, knob, button) &= next\_time(microtime(\\
&\quad u, swt, knob, button), swt, knob, button) \\
next\_time(t, swt, knob, button) &= if\ ready(s(t), swt, knob, button) = false \\
&\quad then\ next\_time(s(t), swt, knob, button) \\
&\quad else\ s(t)
\end{aligned}
$$

The predicate $stable(sig, t_1, t_2)$ means that the signal sig is stable between times t_1 and t_2. It has the following property

$$\forall t : pnat,\ t_1 \leq t \leq t_2 \rightarrow sig(t) = sig(t_1)$$

From here we can prove the lemma:

$$stable(sig, t_1, t_2) \rightarrow \forall t,\ t_1 \leq t < t_2 \rightarrow sig(s(t)) = sig(t)$$

Specification

The specification of the Gordon computer is graphically displayed in figure 5–5. The computer can be in any of two states, idling or running. When idling the computer can execute any of 6 operations depending on the values of *button* and *knob*. When running the computer can execute any of 9 instructions depending upon the value of the operation code. The jump on zero instruction is split in two cases depending on the contents of the accumulator. These 15 cases specify the instruction level behaviour of Gordon computer. The following conditional equation illustrates the case corresponding to the specification of the *add* instruction:

$$\forall u : pnat,\ swt : signaln(16),\ knob : signaln(2),\ button : flag$$

$$idle(microtime(u, swt, knob, button), swt, knob, button) = false \wedge$$

$$button(microtime(u, swt, knob, button)) = false \wedge$$

$$opcode = (false :: true :: true :: nil)$$

$$\rightarrow$$

$$computer(s(u), swt, knob, button) = \tag{5.4}$$

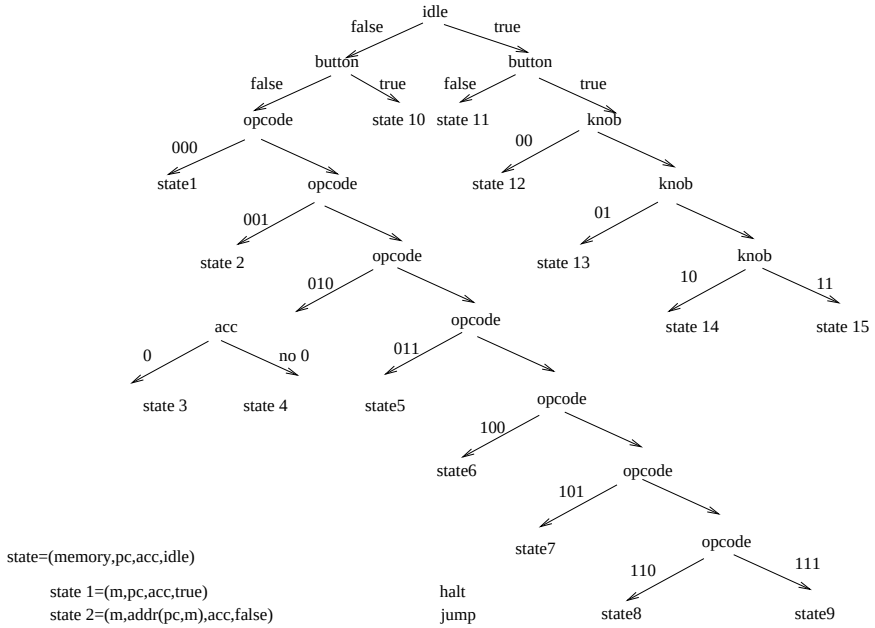

Figure 5–5: Specification of the Gordon computer

$$execute\_add(computer(u, swt, knob, button))$$

where

$$\forall x : state\ execute\_add(x) = \tag{5.5}$$

$$fst(x)\&inc\_g(snd(x))\&$$

$$add\_g(trd(x), fetch(cut(fetch(snd(x), fst(x)))), false)\&true$$

where fst, snd and trd access the first, second and third elements of a specification state respectively. The other 14 cases are defined in the same way. These cases use the following uninterpreted functions: $add\_g(x, y, c) : wordn(16)$, $subtract\_g(x, y, c) : wordn(16)$, $inc\_g(x, y, c) : wordn(16)$, $cut(x) : wordn(13)$, $pad(x) : wordn(16)$, $fetch(x) : wordn(16)$, $store(x) : memn(16)$.

Implementation

The implementation description of the *Gordon* computer is given at the register-transfer level as shown in figure 5–6 [Joyce *et al* 86]. The implementation has

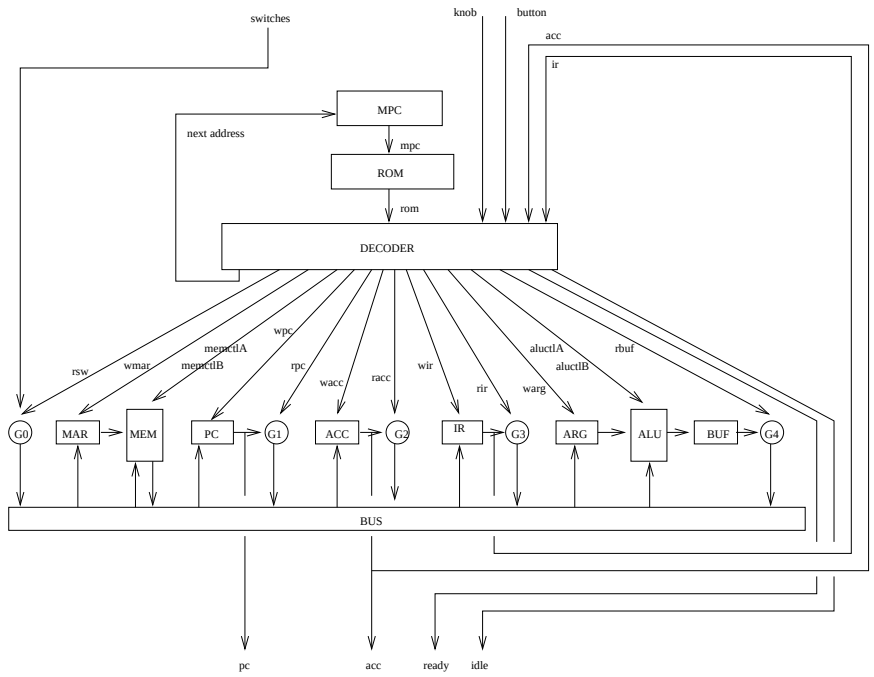

Figure 5–6: Register-transfer level implementation of the Gordon computer

a number of registers in addition to the program counter and accumulator of the instruction level. The instruction currently under execution is held in the instruction register IR, addresses of memory locations to be read or written to are held in the memory address register MAR, arguments to the arithmetic and logic unit ALU are held in the ARG register and the results of the ALU are held in the buffer register $BUFF$. The implementation also used five gate devices $G0$, $G1$, $G2$, $G3$ and $G4$ to control the reading of data onto the 16-bit bus BUS.

The fetch-decode-execute cycle is driven by a microcoded control unit. The microcode is stored in a read-only ROM which can hold 32 microcode instructions, each 30 bits wide. On every clock cycle, the microcode instruction addressed by the microcode program counter MPC is read from the ROM and decoded by the decode unit $DECODE$. Output from the decode unit consists of signals which

control the operation of the data part of the implementation. These signals correspond to control lines which are labelled in figure 5–6 as rsw, $wmar$, $memctlA$, $memctlB$, wpc, rpc, $wacc$, $racc$, wir, rir, $warg$, $aluctlA$, $aluctlB$ and $rbuf$. For instance, when the boolean signal rpc has the value $true$ the low 13 bits of the bus are read into the program counter register PC. All of the control signals have $flag$ type (function from time to Bool). The decode unit also produces the address of the next microcode instruction which is loaded in the microcode program counter.

The register-transfer implementation of the *Gordon computer* is divided into two parts, the data-path and the control unit. The components of the data-path are the registers, the memory, the arithmetic and logic unit, and the bus. Types of registers include 16-bit registers or 13-bit registers; selectively loadable registers or directly loadable registers; bi-stable registers or tri-stable registers. Memory words and registers that store their contents are 16-bits long. Addresses and the registers that store them are 13-bits long. Selectively loadable registers use a boolean flag to determine when they can be loaded. These boolean flags are generated by the Control Unit. Among these registers are the $memory$, defined by the type $memn$ which consists of a list of words in memory. The operation performed on memory is determined by the values of the flags $memctlA$ and $memctlB$. The *buffer* is a 16-bit, directly loadable, bi-stable register which stores the output of the Arithmetic Logic Unit. The *bus* is used for the transfer of both 16-bit data and 13-bit addresses. It merges signal from the various devices and ensures that just one device has access to it. Each of these and the other registers are defined by a recursive function. For instance, the following recursive function defines the accumulator:

$$\begin{aligned} acc(s(t), swt, knob, button) \;=\; & if\ wacc(t, swt, knob, button) = false \\ & then\ acc(t, swt, knob, button) \\ & else\ bus(t, swt, knob, button) \end{aligned}$$

The control unit generates the 16 control flags that drive the data-path. It contains s 30-bit microcode store, a 5-bits microcode program counter mpc, and a combinational logic to compute the next value of the mpc. For instance, the control flag

wacc is calculated by:

$$wacc(t, swt, knob, button) = mc\_wacc(mpc(t, swt, knob, button))$$

mc_wacc is the column of the microcode that stores the 32 values of the write accumulator flag. The other 15 control flags are defined in a similar way. The microcode program counter is a 5-bit register that loads every clock cycle and calculates the next value of the *mpc*.

The implementation of the *Gordon computer* is established by the function *computer_imp* that defines the computer implementation state at any time of the microinstruction time-scale.

$$computer\_imp(p)$$

$$=$$

$$memory(p)\&pc(p)\&acc(p)\&idle(p)\&$$

$$buffer(p)\&mar(p)\&ir(p)\&arg(p)\&mpc(p)\&ready(p)$$

where $p = (t, swt, knob, button)$.

Conjecture

The first statement of correctness is given by the following verification conjecture which establishes that if the specification and the implementation match at a given point in time t of the instruction time-scale then they will also match at time $t+1$.

$$\forall u : pnat,\ swt : signaln(16), knob : signaln(2), button : flag$$
$$stable(swt, microtime(u), microtime(s(u))) \rightarrow$$
$$stable(knob, microtime(u), microtime(s(u))) \rightarrow$$
$$mpc(microtime(u, swt, knob, button), swt, knob, button) =$$
$$false :: false :: false :: false :: false :: nil\ in\ wordn(5) \rightarrow$$

$$abs\_imp(computer\_imp(microtime(u, swt, knob, button), swt, knob, button))$$

$$=$$

$$computer(u, swt, knob, button)$$

$$\rightarrow$$

$$abs\_imp(computer\_imp(microtime(s(u), swt, knob, button), swt, knob, button))$$

$$=$$

$$computer(s(u), swt, knob, button)$$

This version of the conjecture assumes that the computer is stable after power up. It can be shown that there exists a time at which the microcode program counter is reset, and this value is either the address 00000 (0) or the address 00101 (5), corresponding to the initial states when the computer is idling or executing an instruction, respectively. These values of the *mpc* can be reached by an initialisation procedure for the computer after power up. The first fabrication of Gordon computer did not include a reset button for the microcode program counter. Formal verification did not catch this fact because the output values of the *mpc* were assumed bistable without regard to the type of input values of the *mpc*. Although the deduction was correct, the register transfer model was not accurate enough to detect this fact [Joyce *et al* 86]. Thus, we assume that the computer powers up and reaches a stable state where the *mpc* is set to zeroes when the computer is idling. When the computer is running the *mpc* is set to 5. Also, in this version of the theorem we assume that the bus carries bistable signals, i.e. we do not merge/split signals with floating values.

A second statement of correctness is given by the following verification conjecture which establishes the equivalence of the specification and an abstract representation of the implementation at any point of the instruction-level time-scale:

$$\forall u : pnat,\ swt : signaln(16), knob : signaln(2), button : flag$$
$$(\forall t : pnat, stable(swt, microtime(t), microtime(s(t)))) \rightarrow$$
$$(\forall t : pnat, stable(knob, microtime(t), microtime(s(t)))) \rightarrow$$
$$mpc(microtime(u, swt, knob, button), swt, knob, button) =$$
$$false :: false :: false :: false :: false :: nil\ in\ wordn(5) \rightarrow$$

$$abs\_imp(computer\_imp(microtime(u, swt, knob, button), swt, knob, button))$$

$$=$$

$$computer(u, swt, knob, button)$$

The proof of this conjecture is done by induction on time. The base case corresponds to an unrealistic situation in which the computer powers up and establishes at time 0. The step case coincides with the previous version of the conjecture.

A third version of the conjecture is as follows:

$$\begin{aligned}
&\forall u : pnat,\ swt : signaln(16), knob : signaln(2), button : flag \\
&\forall swt_abs : signaln(16), knob_abs : signaln(2), button_abs : flag \\
&(\forall t : pnat, stable(swt, microtime(t), microtime(s(t)))) \rightarrow \\
&(\forall t : pnat, stable(knob, microtime(t), microtime(s(t)))) \rightarrow \\
&(\forall t : pnat, swt_abs\ of\ t = swt\ of\ microtime(t, swt, knob, button)) \rightarrow \\
&(\forall t : pnat, knob_abs\ of\ t = knob\ of\ microtime(t, swt, knob, button)) \rightarrow \\
&(\forall t : pnat, button_abs\ of\ t = knob\ of\ microtime(t, swt, knob, button)) \rightarrow \\
&mpc(microtime(u, swt, knob, button), swt, knob, button) = \\
&false :: false :: false :: false :: false :: nil\ in\ wordn(5) \rightarrow
\end{aligned}$$

$$abs\_imp(computer\_imp(microtime(u, swt, knob, button), swt, knob, button))$$

$$=$$

$$computer(u, swt\_abs, knob\_abs, button\_abs)$$

This version considers abstract views of the variables *swt*, *knob* and *button* which are used in the specification of the computer and are unfolded by the abstraction equations included in the conditions of the conjecture. The proof also assumes the the computer powers up and establishes at time 0.

In the following version of the conjecture, we allow for an arbitrary period of time for initialisation after power up:

$$\begin{aligned}
&\forall u, v : pnat,\ swt : signaln(16), knob : signaln(2), button : flag \\
&\forall swt_abs : signaln(16), knob_abs : signaln(2), button_abs : flag \\
&(\forall t : pnat, stable(swt, microtime(t), microtime(s(t)))) \rightarrow \\
&(\forall t : pnat, stable(knob, microtime(t), microtime(s(t)))) \rightarrow \\
&(\forall t : pnat, swt_abs\ of\ t = swt\ of\ microtime(t, swt, knob, button)) \rightarrow \\
&(\forall t : pnat, knob_abs\ of\ t = knob\ of\ microtime(t, swt, knob, button)) \rightarrow \\
&(\forall t : pnat, button_abs\ of\ t = knob\ of\ microtime(t, swt, knob, button)) \rightarrow \\
&mpc(microtime(u, swt, knob, button), swt, knob, button) = \\
&false :: false :: false :: false :: false :: nil\ in\ wordn(5) \rightarrow
\end{aligned}$$

$$\begin{aligned}
&abs_imp(computer_imp(microtime(u, swt, knob, button), swt, knob, button)) \\
&= \\
&computer(u, swt_abs, knob_abs, button_abs) \\
&\rightarrow \\
&u \leq v \rightarrow \\
&abs_imp(computer_imp(microtime(v, swt, knob, button), swt, knob, button)) \\
&= \\
&computer(v, swt_abs, knob_abs, button_abs)
\end{aligned}$$

A scheme like the following would be used to prove this conjecture by induction on time:

$$\forall \phi : pnat \rightarrow u(2)\ (\frac{H \vdash \forall x : pnat,\ Conditions \rightarrow \phi(x) \rightarrow \phi(s(x))}{H \vdash \forall u, t : pnat,\ Conditions \rightarrow u \leq t \rightarrow \phi(u) \rightarrow \phi(t)})$$

Verification

The proof plan corresponding to the second version of the conjecture is generated automatically by *Clam* and is graphically displayed in figure 5–7. Once again,

notice that the structure of this proof plan is similar to the structure of the previous plans. The planning consists of the method application steps, indicated by the

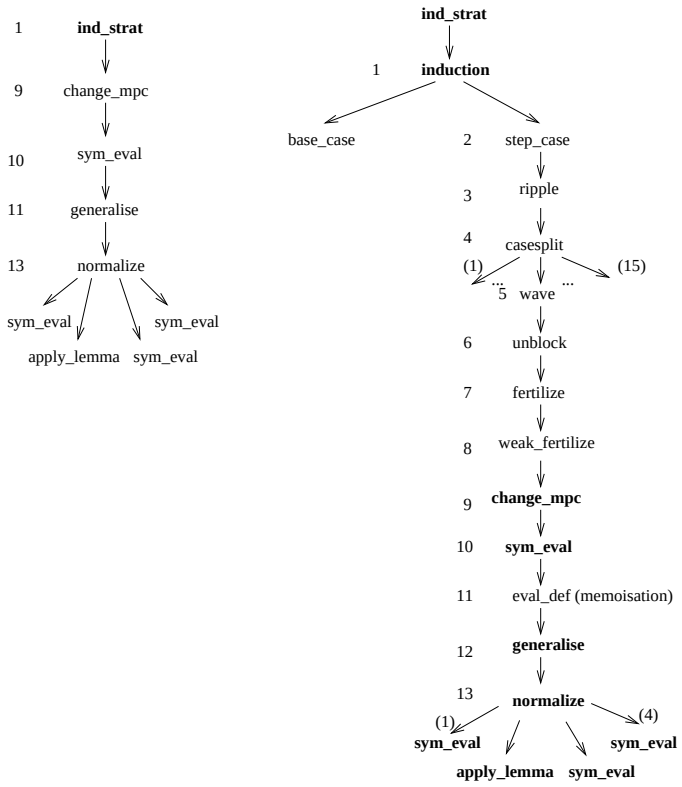

Figure 5–7: Proof plan for the Gordon computer

numbers on the nodes of the tree:

- the proof is done by induction on the time variable u, which is suggested by the recursive definition of *computer*. The base case, is solved by symbolic evaluations using initial values for the specification and the implementation at time 0. For the step case, the induction hypothesis establishes the equality of the specification and an abstract view of the implementation at time u of the instruction-level time-scale, and the induction conclusion establishes the same equality at time $s(u)$ of the same time-scale. Mapping this time unit into the microinstruction time-scale, results in the execution of a variable number of micro-operations, each requiring a time unit of the

microinstruction time-scale. The sequence of micro-operations associated to each instruction is recorded in the microcode.

- the step case is (step 1):

$$\ldots abs\_imp(computer\_imp(mt(u, swt, knob, button), swt, knob, button))$$

$$=$$

$$computer(u, swt, knob, button)$$

$$\vdash$$

$$\ldots abs\_imp(computer\_imp(mt(\boxed{s(\underline{u})}^{\uparrow}, swt, knob, button), swt, knob, button))$$

$$=$$

$$computer(\boxed{s(\underline{u}}^{\uparrow}), swt, knob, button)$$

which is then simplified by the `step_case` method (step 2). mt is an abbreviation for $microtime$. Rippling tries to reduce the differences between the goal and the hypothesis using the 15 wave-rules that define the specification (step 3). This induces a case-split generating 15 cases, one for each of the possible computer instructions that result when the computer is idling (6) or when the computer is running (9) (step 4). For each of these cases the `wave` method applies the outwards wave rule associated with each of the 15 recursive equations of the specification (step 5), does unblocking (step 6), and applies weak fertilisation (steps 7-8). For instance, consider the case when $idle$ is $false$, $button$ is $false$ and the operation code is 3. This correspond to the add operation. After case-split the goal is:

$$\forall u : pnat,\ swt : signaln(16),\ knob : signaln(2),\ button : flag$$

$$idle(mt(u, swt, knob, button), swt, knob, button) = false \wedge$$

$$button(mt(u, swt, knob, button)) = false \wedge$$

$$opcode = (false :: true :: true :: nil)$$

$$\rightarrow$$

$$abs\_imp(computer\_imp(mt(\boxed{s(\underline{u})}^{\uparrow}, swt, knob, button), swt, knob, button))$$

$$=$$

$$computer(\boxed{s(\underline{u})}^{\uparrow}, swt, knob, button)$$

the `wave` sub-method applies the following outwards wave-rule associated with its recursive definition given by equation 5.4:

$$\forall u : pnat,\ swt : signaln(16),\ knob : signaln(2),\ button : flag$$

$$idle(microtime(U, swt, knob, button), swt, knob, button) = false \wedge$$

$$button(microtime(U, swt, knob, button)) = false \wedge$$

$$opcode = (false :: true :: true :: nil) \rightarrow$$

$$computer(\boxed{s(\underline{U})}^{\uparrow}, swt, knob, button) \Rightarrow$$

$$\boxed{execute\_add(\underline{computer(U, swt, knob, button)})}^{\uparrow}$$

The resulting subgoal:

$$abs\_imp(computer\_imp(mt(\boxed{s(\underline{u})}^{\uparrow}, swt, knob, button), swt, knob, button))$$

$$=$$

$$\boxed{execute\_add(\underline{computer(u, swt, knob, button)})}^{\uparrow}$$

is simplified by unblocking using equation 5.5 and weak fertilisation to give:

$$v = fst(v) \& inc\_g(snd(v)) \&$$

$$add\_g(trd(v), fetch(tl(tl(tl(fetch(snd(v), fst(v))))), fst(v)), false) \& false$$

where

$$v = abs\_imp(computer\_imp(mt(u, swt, knob, button), swt, knob, button))$$

- since *idle* is *true*, the *mpc* should be set to 5. This is done by the method `change_mpc` (step 9)

- The resulting subgoal is proved by symbolic evaluation using the recursive definitions of the implementation state components, namely mpc, $memory$, pc, acc, arg, mar, ir, $buffer$ (step 10). The recursion is done 10 times, which is the number of micro-operations the computer takes to execute the *add* instruction. The *memoisation* procedure applied by the `eval_def` sub-method, plays a critical role here, for storing and reusing values previously computed, otherwise the memory is exhausted after two or three iterations (step 11). The memoisation procedure is described in section 5.3.2.

- the `generalise` method generalises a term which results when fetching the contents of the program counter from memory (step 12)

- in the next step the `normalise` method generates four subgoals, one for each element of a specification state. Three of these subgoals are tautologies and are solved by the `sym_eval` method, and the fourth:

  $$cut(inc\_g(pad(pc(w, swt, knob, button)))) = inc\_g(pc(w, swt, knob, button))$$

  is solved by the `apply_lemma` method which using the lemma

  $$cut(inc\_g(pad(x))) = inc\_g(x)$$

- the other 14 cases are solved in a similar way

5.3.2 Proof tasks

The proof tasks consisted of: writing a new sub-method for implementing a memoisation algorithm for computing recursive functions, writing a new method for experimenting with difference matching, extending the normalisation method to normalise equations with product terms, experimenting with various representations (e.g. time abstraction, state-transition graphs, conditional rewrites, vectors vs matrices, etc), finding an effective way of representing and traversing the microcode table, experimenting with method orderings.

Memoisation

The evaluation of a recursive function $f(x)$ at a point $x = u_n$ requires the calculation of all the previous values of f at $x = u_0, \ldots, u_{n-1}$. If later we need to evaluate $f(x)$ again at another point $x = v_m$, where $m > n$, normally we would calculate all the previous values of f at $x = v_0, \ldots, v_{m-1}$. But the values of f at $x = u_0, \ldots, u_{n-1}$ and at $x = v_0, \ldots, v_{n-1}$ are the same, so strictly speaking, we do not need to re-compute them, we just need to calculate the values of f at $x = v_n, \ldots, v_m$. Memoisation allows us to store values which have been already computed and re-use them when needed again. This simple procedure yields enormous savings in calculation time. Camilleri implemented a memoisation algorithm for the symbolic execution of the Gordon computer in *HOL*. As an example of the computational savings that memoisation provides, he calculated that in order to compute the value of $bus(t + 10)$ the equation that defines *bus* is executed over 60.5 million times [Camilleri 88]. We had a similar experience. The evaluation of the recursive functions for the memory, the program counter, the accumulator, the memory address register, the instruction register, the buffer, and the microcode program counter for values of time between t and $t + 10$ shares the same problem. The memoisation we implemented for proof planning stores computed values of a function identified as 'memoisable' in the hypothesis list, and each time a new value of the function is needed, we first check whether the value already exist in the hypothesis list. If it is there, we use it, otherwise, it is computed and stored for future use.

Difference match

The formalisation of the Gordon computer was first made using a deterministic finite-state machine to represent state transitions both at the instruction and register-transfer levels. The conjecture stated that if the implementation and the specification are equal at time t in the instruction-level time-scale, then they must be also equal at time $s(t)$, in the same scale, provided t and $s(t)$ are mapped to microinstruction-level time-scale within the implementation. This situation is

represented by:

$$spec(t,\ldots) = imp(mt(t),\ldots) \rightarrow spec(s(t),\ldots) = imp(mt(s(t)),\ldots)$$

To prove this conjecture we assume the antecedent of the implication and try to prove the consequent. To prove the consequent we used difference matching [Basin & Walsh 93], a techniques that annotates the differences between two expressions and then tries to reduce the differences. In our case we difference matched the consequent and the antecedent and wrote a method (`diff_match`) which annotates the consequent based on these differences. Then we used rippling to eliminate the differences and fertilisation to use the antecedent as a rewrite rule to simplify the consequent. We obtained a proof plan for the Gordon computer following this procedure. The proof plan is displayed in appendix E. But then we realised that this procedure was just the step case of an induction on time and that it was more general to see it in this way. In any case, learning about difference matching was a useful experience.

Normalisation

The `normalise` method was extended to normalise a formula which consists of an equality of product terms. If we have the goal:

$$U_1 \ldots \& U_n = V_1 \ldots \& V_n$$

we want to split it into a set of n subgoals:

$$U_1 = V_1, \ldots U_n = V_n$$

and prove each of these subgoals individually. This situation arises in the proof planning of the Gordon computer after doing memoisation where we get a goal which is an equality between two terms, each of which is a product of four types (the instruction-level state).

5.3.3 Systems tasks

The systems tasks consisted of: finding an effective way to parse conditional terms (e.g. predicate expression_at), solving inefficiencies of *Clam* in parsing large terms.

For instance, the microcode table was implemented by a set of 32 lists, where each list has 30 elements. Access to the elements of the microcode table was implemented by a recursive function which traverses the list to locate an element. But the execution of this function was very inefficient and time consuming, so we changed the representation of the table and used vectors to represent the columns of the table, and were able to access faster the elements by indexing instead of by list traversal. Thus, we did not solve a systems problem, but did a change of representation to avoid a inefficiency of the tool, which still exists. Therefore, this task can be classified as a formalisation task, but we include it here because it addressed a systems issue. The same kind of inefficiency arose earlier when we tried to represent words by functions, so we preferred lists to represent words rather than functions.

5.4 Extendability and Scalability

We have shown by experimentation that proof planning posses the features of extendibility and scalability, which are important characteristics which contribute to the generalisation of the methodology we have proposed. We do a brief analysis of extendability and scalability.

5.4.1 Extendability

Extendability means that the same set of methods that were used to verify a circuit of a certain type will work in the verification of a circuit of the same type. For instance, the same methods used in the verification of the *adder*, worked in verifying the *ALU* and the *incrementer*. Although the *ALU* is more complex that the *adder* and the incrementer is simpler than the *adder*, in the three cases, the implementation is obtained by replicating a basic component: n half adders make an *incrementer*, n full adders make an *adder*, and n 1-bit ALU make an n-bit *ALU*. This pattern should also apply to other circuits implemented by cascading a basic component.

5.4.2 Scalability

Scalability means that the same set of methods that were used to verify a circuit of a certain type will work in the verification of a circuit of higher complexity. For instance, the same methods used in the verification of the *adder*, worked in verifying the *multiplier* and the *Gordon computer*, which are much more complex than the *adder*.

We can summarise graphically these two features in the chart displayed in figure 5–8. The horizontal axis represents extendability and the vertical axis rep-

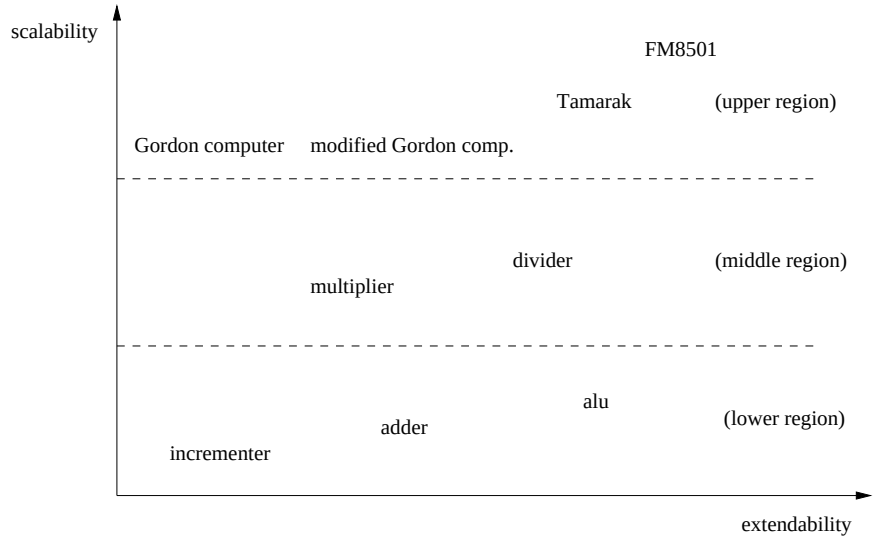

Figure 5–8: Extendability and scalability of proof planning

resents scalability. In the lower region and along the extendability axis, we find circuits such as the *incrementer*, the *adder* and the *ALU*. Following the scalability axis and in the middle region, we find parallel-array circuits, such as families of multipliers and dividers. Further up the scalability axis and in the upper region, we find microprocessor systems of various types along the extendability axis.

As an extendability test, we have done a slight modification to the instruction set of the *Gordon computer* to see how proof planning methods behave. The test consisted in deleting one (any) of the eight programming instructions and

replacing the gap by a no-operation (NOP) instruction [1]. The instruction deleted was the *add* operation. This involved a modification of the *next-address-A* field of address 13 in the microcode table (implementation) to change the sequence of the *add* micro-operations so that the new computer state associated with the no-operation, consists only of an increment in the program counter *pc*, and a modification of the specification branch to reflect a no-operation computer state in place of the state produced by the *add* instruction. We were happy to see that these changes did not require any modification of the methods or the planner. The planner was able to automatically generate a slightly modified tactic to prove the new conjecture.

Thus, these experiments and tests show the *robustness* of proof planning, and provide support to our belief that the features of scalability and extendability posed by proof planning can be the basis for the adoption of our proof engineering-based methodology on a wider basis.

5.5 Summary

In this chapter we have presented a methodology based on the concept of proof engineering, using proof planning, to verify combinational and sequential circuits, and have illustrated its use in verifying circuits of increasing complexity. Formal proof was divided into three conceptually different kind of activities: user, proof, and systems tasks. We have seen how the features of *extendability* and *scalability* of proof planning allow us to transport the same methods to verify circuits of similar classes and higher complexities. Also, we have seen how slight modifications in the implementation and specification of the proof of a circuit, are transparent to proof planning, and do not require user intervention to obtain a modified proof, which illustrates the *robustness* of proof planning. In general, little user interaction was

[1] This experiment was suggested by Mike Fourman

required to get the proofs through. These features of proof planning make the methodology look promising for systematic hardware verification.

Chapter 6

Extensions to Proof Planning

In this chapter we explain the extensions to proof planning which were required for verifying hardware. Most of these extensions are not particular to hardware and will apply to other domains as well. In general, the extensions were fairly minor, as the existing implementation of proof planning contained most of the elements necessary for the verification task. The most important extension was the tuning of the existing methods, which, in most cases, consisted of addition or modification of the method's preconditions, or the addition of new clauses; in a few cases, new methods were written. The declarative nature of the meta-logic in which the method language is based makes it easy to write or modify methods. Sometimes a method may require a new predicate to be added to the method language, but usually, these predicates are easy to implement. Along with each method added or modified, the corresponding tactic specified by the method must be supplied.

We group the extensions according to the kind of tasks introduced in chapter 5: user, proof and systems level tasks. The user level extensions refer to the use of proof planning to verify new kinds of circuits and are summarised in chapter 7. Proof level extensions are described in section 6.1, which explains the extensions to methods, tactics, inductive schemes, and equations; and systems level extensions are explained in section 6.2, which describes new predicates of the method language and the evolution of the *Clam* system. Section 6.3 summarises the chapter.

6.1 Proof level

The knowledge required at this level is the one held by a typical user of tactic-based theorem provers like *HOL*, *LAMBDA*, *NUPRL* and *PVS*, and consists of the ability to write tactics, and specifications of their behaviour using methods, except we require only knowledge of general purpose methods and tactics. The verification of the circuits described in chapter 5, was obtained using slight variations of the set of methods for inductive theorem proving presented in chapter 3. The new set of methods is displayed in figure 6–1: we call it *verify*. *Verify* has exactly the same

Methods:

1. sym_eval	simplifies symbolic expression
a. elementary	applies simple propositional reasoning
b. equal	applies an equality in the hypothesis
c. eval_def	applies rewriting and memoisation
d. term_cancel	cancels out common term on both sides of an equation
e. bool_cases	does Boolean case analysis
2. generalise	generalises a common term on both sides of a formula
3. normalise	normalises a formula
4. ind_strat	applies induction strategy
a. induction	selects induction scheme and induction variables
b. sym_eval	applies symbolic evaluation to the base case
c. step_case	applies ripple and fertilise to the step case
i. ripple	applies rippling heuristic
- wave	applies wave rules
- casesplit	does a case split
- unblock	unblocks rippling
ii. fertilize	simplifies induction conclusion with induction hypothesis
- strong fertil.	applies induction hypothesis directly
- weak fertil.	applies induction hypothesis as rewrite rule

Figure 6–1: Verify: set of methods for hardware verification

abstract procedural interpretation as the set of methods given in chapter 3:

- try to solve the conjecture by symbolic evaluation. If successful, finish, if there are remaining subgoals, try the set of methods again;

- try to solve the conjecture by induction, generalising or normalising any terms, if necessary. Solve the base case by applying symbolic evaluation and solve the step case by applying rippling and fertilisation. In either case, if there are remaining subgoals, try the set of methods again.

In theory, the ordering of the methods should not matter, because in backtracking, the planner can find a sequence of method applications that would prove a conjecture. However, in practice this depends on the type of planner used. The depth-first planner is very efficient, but in some cases it may fail to find a plan when there is one, if it follows an infinite branch which does not contain a plan. A remedy to this situation, which again is just for practical considerations, is to re-order the methods. This will produce a different search sequence in looking for a plan, and may find a plan without backtracking or avoid the possibility of following an infinite branch. A more informed search may be more useful (e.g. a best-first planner). A best-first planner will do an analysis of the current conjecture and utilise domain knowledge encoded as heuristics of the planner, to determine which of the methods which are applicable at a certain point in the proof, should be tried next. In our case, we have used the depth-first planner and generated different orderings of *verify* when required. For instance, the order `ind_strat`, `generalise`, `normalise`, `sym_eval` (verify1) was used in the verification of the *adder*, and the order `sym_eval`, `ind_strat`, `generalise`, `normalise`, (verify2) was used in the verification of the *ALU*. However, the original order (verify) can also find these plans in backtracking. In general, a good order for a given conjecture was found by experimentation.

For instance, suppose that we have the method set *verify* and a goal $\ldots f(Imp) = Spec$. Suppose also that methods `sym_eval` and `ind_strat` are both applicable, with `sym_eval` unfolding the definition of $Spec$ (assume it is given by a large term) and then applying a Boolean case analysis. In this situation, we would like `ind_strat` to be applied before `sym_eval`, because otherwise, the induction would become very complicated. As we have seen, a remedy to this situation is experiment with other ordering of *verify*. This sort of dynamic ordering could be calculated by a heuristic planner as we have pointed out. However, we have not

investigated this alternative any further, and have used the un-informed depth-first search with off-line changes in the orderings when required instead, but in principle, it could be implemented to make the planning more automatic. Now, we give a brief description of the extensions to methods and tactics.

6.1.1 Methods and tactics

There are two new methods and tactics: `term_cancel` and `bool_cases`, which play an important role in planning hardware verification conjectures but they are not specific for the hardware domain and can be applied to other domains as well.

Term cancellation

This sub-method was added to strengthen *Clam*'s arithmetic reasoning capabilities [Boyer & Moore 81]. It reads an input sequent $H \vdash G$, the preconditions of the method check that G is an equation of the form

$$term_1 + term_2 + \ldots + term_n = term'_1 + term'_2 + \ldots + term'_m$$

and the method repeatedly cancels out equal pairs on both sides of the equation. The output is the equation that results after cancelling out equal terms. The method is displayed in figure 6–2.

The output of the method is a simplified expression which is given as input to the tactic with the same name. This tactic was a bit difficult to implement because reasoning about the associativity and commutativity of addition yields permutations of an expression which evaluate to the same value and the inference rules that justify the simplification steps should consider all of these permutations, which yield a large number of cases to consider. We want a tactic that keeps the system *fully expansive*, in the sense that all the proof steps are justified at the inference rule level. In a tactic-based fully expansive system, the execution of a tactic results in the execution of the inference rules that support that tactic [Boulton 94].

```
submethod(term_cancel(NG),
          H==>G,
          [matrix(V,LS = RS in pnat,G),
           \+ (LS=0 v RS=0),
           only_sum(LS,LstPlusL),
           only_sum(RS,LstPlusR),
           cancel_eq(LstPlusL,LstPlusR,OtherL,OtherR),
           length(LstPlusL,X),
           length(OtherL,Y),
           X>Y],
          [put_plus(OtherL,Left),
           putPlus(OtherR,Right),
           matrix(V,Left=Right in pnat,NG)],
          [H==>NG],
          term_cancel(NG)).
```

Figure 6–2: Method for term cancellation

The tactic is implemented by defining a type for *multi-set* of natural numbers (also called a bag), defining the addition operation on bags, and relating type theory arithmetic terms to appropriate multi-sets. We relate type theory arithmetic terms and multi-sets by defining another tactic that maps addition of natural numbers into addition of members of a bag. Given an arithmetic expression we build a bag such that the sum of its elements equals the value of the arithmetic expression. To do this, we define the *union* operation on bags and prove the bag lemma:

$$\forall x, y, z : bag,\ bag\_sum(x) = bag\_sum(y) \rightarrow$$

$$bag\_sum(bag\_union(x, z)) = bag\_sum(bag\_union(x, z))$$

We then define one more tactic that applies the bag lemma to the sequent whose hypothesis is the final expression after cancellation and whose goal is the expression before cancellation.

Boolean case analysis

This sub-method reads as input a sequent $H \vdash G$, and its preconditions check for a Boolean variable or a term of type Boolean on either H or G. If there is one, the

method outputs two subgoals, replacing the Boolean variable or term by the values $true$ and $false$ respectively. One of the clauses of this sub-method is displayed in figure 6–3. The tactic was implemented as a special case of the case_split tactic.

```
submethod(bool_cases(Term),
          H==>G,
          [matrix(Vars,Matrix,G),
           exp_at(Matrix,Pos,Term),
           append(Vars,H,NewH),
           find_type(NewH,Term,{bool})],
          [replace_all(Term,{false},Matrix,MG1),
           replace_all(Term,{true},Matrix,MG2),
           matrix(Vars,MG1,G1),
           matrix(Vars,MG2,G2),
           hfree([Id],H)],
          [Id:Term={false} in {bool}|H]==>G1,
           Id:Term={true} in {bool}|H]==>G2],
          bool_cases(Term)).
```

Figure 6–3: Method for Boolean case analysis

Modifications

The following methods and tactics were extended as follows:

`sym_eval`: this method (and sub-method) was modified to admit two new sub-methods: `term_cancel` and `bool_cases`. It was also extended to handle branching outputs, as generated by the `bool_cases` method.

`equal`: this sub-method was extended to apply equations from the hypotheses list of the form $term1 = term2$. Before, $term1$ had to be just a variable.

`eval_def`: this sub-method was extended with a memoisation algorithm to compute values of recursive functions. This sub-method was very useful in the verification of the Gordon computer for the calculation of values for the recursive functions of the memory, the program counter, the accumulator, the memory address register, the buffer, and the microcode program counter, during the execution of a computer instruction. Memoisation save a lot of computer time. Figure 6–4 shows this method.

```
submethod(eval_def(Pos,[Rule,Dir]),
          H==>G,
          [matrix(Vars,Matrix,G),,
           wave_fronts(_,[],Matrix),
           new_exp_at(Matrix,Pos,Exp),
           not metavar(Exp),
           %expression is memoisable
           ((Exp=mpc(s(X1),_,_,_) v
             Exp=memory(s(X2),_,_,_) v
             Exp=pc(s(X3),_,_,_) v
             Exp(acc(s(X4),_,_,_) v
             Exp(buffer(s(X5),_,_,_) v
             Exp(mar(s(X6),_,_,_) v
             Exp(arg(s(X7),_,_,_) v
             Exp(ir(s(X8),_,_,_)),
           % value of function has been calculated already
           ((member(A:Exp=NewExp in word(5),H),
             NewH=H) v
           % value hasn't  been calculated, do it and add it to hypothesis list
             func_defeqn(Exp,Dir,Rule:C=>Exp:=>NewExp2),
             matrix(Vars,NewExp2,Goalmemo),
             applicable_submethod(H==>Goalmemo,sym_eval(_),_,[Hmemo==>NewExp]),!,
             hfree([HVar],Hmemo),
             NewH=[HVar:Exp=NewExp in word(5)|Hmemo])),
           func_defeqn(Exp,Dir,Rule:C=>Exp:=>NewExp2),
           polarity_compatible(Matrix,Pos,Dir),
           elementary(NewH==>C,_)],
          [replace(Pos,NewExp,Matrix,NewMatrix),
           matrix(Vars,NewMatrix,NewG)],
          [NewH==>NewG],
          eval_def(Pos,[Rule,Dir]))
```

Figure 6–4: Method for memoisation

`generalise`: this method (sub-method) was extended to accept variables and terms of type Boolean. A precondition was added to recognise terms of type Boolean on a formula and apply the generalisation criteria to them.

`normalise`: this method (sub-method) was extended to normalise terms of type product into its individual components. It was used in the verification of the Gordon computer to split the components of the specification and implementation states which are represented by a product type.

`weak_fertilise`: this method was extended with a clause to handle wave-fronts that contain several sinks, each of which stores an inwards wave-front. This situation had not appeared previously in inductive theorem proving, but it has been found in most of the hardware verification proofs.

6.1.2 Induction Schemes

Clam provides a data base of induction schemes for inductive theorem proving. Some of them were used or adapted to meet the requirements in verifying hardware.

`induction on natural numbers`: the 1-step induction on natural numbers was used in the verification of various circuits in which the representation is done by using an explicit parameter of the length of a word as well as in circuits in circuits with a time parameter. In the first case we have circuits like the *adder*, the *ALU*, and the *shifter*. In the second case we have circuits like the *counter* and the *Gordon computer*. We did not modify this scheme, we just used it.

$$\forall \phi : natural \rightarrow u(2)\ (\frac{\phi(0),\ \forall x : natural\ \phi(x) \rightarrow \phi(s(x))}{\forall y : natural\ \phi(y)})$$

`induction on word`: the 1-step induction on words was adapted as a special case of 1-step induction on lists with elements of type Boolean. This induction scheme was used in the verification of the *incrementer*, the *multiplier*, and in other combinational circuits.

$$\forall \phi : word \rightarrow u(2)\ (\frac{\phi(nil),\ \forall a : bool\ \forall x : word\ \phi(x) \rightarrow \phi(a :: x)}{\forall y : word\ \phi(y)})$$

`induction on word increment`: this is a 1-step induction on words where the induction constructor is the *increment* operation on words. It was used in the verification of arithmetic operations on words.

$$\forall \phi : word \rightarrow u(2)\ (\frac{\phi(nil),\ \forall x : word\ \phi(x) \rightarrow \phi(inc(x))}{\forall y : word\ \phi(y)})$$

`induction on two words`: this is a specialised induction scheme on two words that have the same length. It was used in the verification of the big and little endian versions of the *adder* and the *ALU*.

$$\forall \phi : word \rightarrow word \rightarrow u(2)$$

$$\phi(nil, nil),$$

$$\forall a : bool\ \forall b : bool\ \forall x : word\ \forall y : word$$

$$\frac{length(x) = length(y) \rightarrow \phi(x, y) \rightarrow \phi(a :: x, b :: y)}{\forall u : word\ \forall v : word\ length(u) = length(v) \rightarrow \phi(u, v)}$$

Each of these schemes is justified by an *Oyster* theorem.

6.1.3 Equations

Clam derives rewrite rules from equations, including wave-rules. These equations must be justified as theorems in *Oyster* from the object-level definitions. For instance, the inductive definition of $word2nat$ is:

$$word2nat(x) = list\_ind(x, 0, [h, t, r, plus(bitval(h), times(s(s(0)), r))])$$

where $list\_ind$ is the inductive term constructor, x is the induction variable, h represents the head of the list, t represents the tail of the list, r is the recursive value, and the last argument is a formula that calculates next recursive term. From here we obtain the recursive function 4.2 which consists of a base and a recursive equation. These are given as the following conjectures:

$$\vdash word2nat(nil) = 0\ in\ pnat$$

$$\vdash \forall a : bool, x : word,$$

$$word2nat(a :: x) = plus(times(s(s(0)), word2nat(x)), bitval(a))\ in\ pnat$$

Each of these conjectures are proved by simple *Oyster* tactics. Other equations are justified in a similar way. Proving this sort of conjectures can be easily automated by writing a generic tactic that would prove them.

6.2 Systems level

Extensions at this level include the writing of new predicates for the method language, debugging, and the addition of new functionality to the *Clam* system.

6.2.1 Predicates

The meta-logic was extended with a few predicates for the new methods. Among these are the following:

`find_type` which determines the type of terms like natural numbers, integers, and Booleans and `exp_at` which finds a sub-expression within an expression in a more efficient way, specially when searching the arguments of a conditional expression such as *if a=b then x then y*, since it is more appropriate to first evaluate the arguments a and b, and depending on whether the condition is true of false, evaluate x or y, rather than evaluate y, x, b and a. This is important when x and y contain other nested if expressions, which happens frequently when the data-path components of a microprocessor are formalised by recursive functions. The term cancellation method uses other auxiliary predicates.

6.2.2 Debugging and versions of *Clam*

As a research vehicle, *Clam* is constantly evolving, to reflect new developments and debugging of the code, for which the *MRG* research and technical staff are responsible. We started our experiments using version 1.5, and in the following four years new versions were released, to reach the current version 2.5. We reported both inefficiencies in the current code and the impossibility of obtaining certain plans. The most significant change was the replacement of the wave-rule parser. A wave-rule parser generates automatically wave rules from recursive formulae (e.g recursive equations) identifying and annotating well-annotated terms [vanHarmelen & group 96]. The parser was changed from a static style, in which all the rewrite rules are generated when a theorem is loaded, to a dynamic style, as described in [Basin & Walsh 96a], in which rewrite rules are created dynamically, when needed. With this new parser, in was possible to do verifications that could not be obtained with the static one, such as the *shifter* and the *Gordon computer* proofs as explained in section 7.1, besides making execution faster. Other improvements that were helpful were the method for handling case splits and the updating

of the preconditions of the sub-methods: `step case`, `ripple`, and `induction`. In a production version of *Clam* this sort of developments would rarely be needed.

6.3 Summary

We have presented extensions to proof planning for hardware verification. We described proof and systems level extensions. The required extensions were fairly minor and worked in verifying a variety of circuits. We expect proof level extensions to be done un-frequently, and systems level extensions to be done rarely. The experience with proof and systems tasks provides evidence to our belief that proof planning can be used for hardware verification in such a way that most of the time will be spent by the users of the planner formalising particular conjectures and using the automation facilities of the planner and the prover to obtain proofs of the conjectures.

Chapter 7

Results

The verification methodology described in chapter 5 was applied to verify other circuits, which allowed us to obtain timing statistics like: planning time, execution time, user tasks time, proof tasks time, systems tasks time, and number of lemmas. In this chapter we present a recapitulation of all the experiments done in this research: section 7.1 summarises the experiments and statistics of the experiments, which show evidence that a proof planning based methodology can be of use in verifying hardware; section 7.2 analyses the statistics of the experiments; and section 7.3 summarises the chapter.

7.1 Experiments

An experiment consists of: taking a circuit design, applying the proof planning methodology to verify it, and obtaining the statistics. Table 7–1 summarises the experiments and displays the numbers. Some circuits are from the *IFIP WG10.2 Hardware Verification Benchmark Circuit Set* (n-bit adder, parallel multiplier, Gordon computer), some are from other sources [Mano 79]. The first column lists the circuits, which can be either combinational or sequential. Combinational circuits can be non-recursive or recursive. "Explicit parameter" means that the circuit is parameterised and the parameter is made explicit rather than being calculated (e.g. the length of a word). "increment" means that an induction

Circuit	*computer time*		*human time*			*lemmas*
	planning	proof	user	proof	systems	
	(min:sec)		(hours)			
COMBINATIONAL						
non-recursive						
half adder	0:02	0:51	1	0	0	1
full adder	0:04	1:45	2	16	0	1
1-bit ALU	0:18	10:13	4	0	0	1
4-1 multiplexer	0:10	4:23	3	0	0	1
n-bit incrementer						
big endian	0:09	0:41	2	0	0	2
n-bit adder						
explicit parameter	0:53	3:31	8	88	20	2
big endian	0:47	1:41	4	16	0	2
adder (increment)	0:02	0:05	4	8	0	3
subtracter (increment)	0:07	0:06	2	0	0	3
n-bit ALU						
explicit parameter	4:01	234:19	32	0	48	2
big endian	11:04	274:00	8	0	0	2
n-bit shifter						
big endian	:54	8:57	24	0	16	2
n-bit processing unit						
big endian	0:001	0:01	2	0	2	2
parallel array						
nm-bit multiplier	0:36	2:12	12	36	0	6
multiplier (increment)	0:04	0:08	4	0	0	3
exponentiator (increment)	0:10	0:27	4	0	0	3
factorial (increment)	0:13	0:26	4	4	0	3
divider-quotient (plus)	0:03	0:18	4	12	0	5
divider-remainder (plus)	0:04	0:16	4	0	0	5
SEQUENTIAL						
counter	0:05	0:17	4	4	0	3
Gordon computer	27:50	274:15	120	160	280	4
modified *Gordon* computer	23:35	232:26	1	0	0	4

Table 7–1: Circuits verified using proof planning

scheme with constructor *increment on words* was used in the proof. Similarly, "plus" means that an induction scheme with constructor *addition on words* was used in the proof.

We distinguish two kinds of timings: computer and human. Computer timings refer to the time necessary to obtain a proof plan and the time required to execute the plan in the computer. Human timings include times at three levels: user, proof and systems tasks. The second column lists computer timings, broken down two ways: *proof planning* time which is the time *Clam* takes to find a plan, and *proof* time, which is the time *Oyster* takes to execute a proof plan[1]. The third column lists human timings at three levels:

- first, the user tasks timing, which is the time spent understanding the problem, finding the right representation for the specification and the implementation, writing the conjecture, and debugging them, until a proof was obtained, assuming that the required proof and systems level extensions were in place;

- second, the proof tasks timing, which is the time spent tuning methods and tactics, and finding missing lemmas;

- and third, systems tasks timing, which is the time spent in tuning *Clam*: debugging and improving the planner, debugging and extending the method language, debugging and improving the wave-rule parser, extending the library mechanism, developing an interface to a theorem prover, and the like.

Finally, the last column indicates the number of lemmas used in the proof beyond the definitional equations.

---

[1]Experiments were done in a SUN Ultra 1 with a 167 Mhz UltraSPARC CPU and 128Mb of RAM running Solaris 5.5.1

7.2 Analysis

In this section we present an analysis of the statistics: the human timings, the computer timings, and the number of lemmas used by the proofs.

7.2.1 Analysis of human timings

To calculate the human timings we recorded the starting and ending dates for each circuit verification, obtained the number of weeks, and multiplied the number of weeks times 40, assuming 40 hours per week.

Some of the user, proof and systems level times may appear excessive, and one may ask how often do extensions of this kind have to be made. To give an accurate picture, we are accounting for everything, including many *one-time costs*. The time to obtain a proof tended to be high for the first circuit of a certain kind, but dramatically decreased for subsequent circuits. For instance, the n-bit adder which was the first circuit that we verified, took about 116 man-hours, of which 8 hours correspond to user tasks, 88 hours correspond to proof level tasks and 20 correspond to systems level tasks. However, the rest of the circuits in the table utilised these extensions and their proofs were obtained in shorter times because these extensions were already there. For example, the big-endian version of the adder, which took just 4 hours, used all of the previous extensions. The 16 hours reported under proof level tasks were spent defining a new induction scheme which was also used by other big endian proofs. When we tried the *explicit parameter* version of the n-bit ALU, it turned out that no proof level extensions were required because all of them were already there. The 32 hours reported under user tasks were mainly spent debugging the specification and the 48 hours reported under systems level tasks were spent debugging the *Clam* code. The big endian version of the ALU took advantage of these system level extensions, thus, the user level tasks took just 8 hours and no proof nor system level extensions were necessary. The *shifter* required 24 hours of user tasks which were spent

mainly in debugging the specification. No proof level extensions were required, but the proof planning of this circuit revealed the need for the new wave-rule parser which had already been designed but not implemented. Thus, we solicited the implementation of the parser which was done by the *Clam* development team in about 160 hours. The verification of the *processing unit* required 2 hours of user tasks to formalise the problem. No proof level tasks were needed, and one system level task which took 2 hours was necessary: a modification of the reduction rules code in order to apply the *ALU* and *shifter* verification theorems as lemmas. The *multiplier* did not require any extensions; most of the time was spent in finding the lemmas required by the proof. For the remaining arithmetic circuits, the extensions required included deriving new induction schemes such as induction with increment of a word and addition of two words, which made the proofs very easy to find.

The Gordon computer required a larger effort to scale up proof planning capabilities at both proof and systems level. The very scale of the specification required that we make a number of extensions such as memoisation (proof level), so that the system would more gracefully handle large terms, and the definition of new predicates to handle terms efficiently (systems level). Also significant is that the theorem involves two different time-scales with automatic calculation of the number of cycles for each instruction. Again, all these extensions will be used in the verification of similar circuits, so the 440 hours of proof and systems level time (11 man-weeks) is a one-time effort and the verification effort should significantly decrease for new circuits of the same kind. We conjecture that the 120 hours of formalisation time (3 man-weeks) may also be reduced in verifying other microprocessors of the same kind (e.g. the FM8501 microprocessor).

7.2.2 Analysis of experiments

We now present a brief description of the experiments and a more detailed analysis of the computer and human timing statistics.

Non-recursive circuits

The non-recursive circuits are basic hardware elements to build replicated hardware devices like adders, ALUs and shifters. The *half adder* is used by the incrementer and is the simplest of all the circuits. The verification of the *full adder* was explained in chapter 4. This was the very first circuit verified. The computer timings are as follows: the planning time is 4 seconds and the plan execution time is 1:45 minutes. The human timings are 2, 16 and 0 hours for the user, proof level and systems level tasks respectively. Proof level tasks involved the development of two clauses of the method for Boolean case analysis.

The verification of the 1-bit ALU was done by case analysis on six Boolean variables: three are selection variables, and the other three are input variables. The *4-1 multiplexer* receives four input lines and selects one of them as output, depending on the values of two Boolean selection variables. The verification is done by Boolean case analysis on the 6 variables. No additional proof/system level tasks were required.

Incrementer

Clam plans the *n-bit incrementer* conjecture in 9 seconds using the depth-first planner and *Oyster* executes the plan in *41* seconds. The formalisation time was 2 hours and no additional proof/systems level tasks were required because all the required extensions were already there. This circuit was verified after the n-bit adder and was the simplest of the circuits which are built by replicating a basic component.

Adder

The n-bit *adder* was verified in five different versions. The very first inductive verification that we attempted was the *explicit parameter* version of the adder. *Clam* plans this version in *53* seconds using the depth-first planner and *Oyster* executes the plan in about 3:31 minutes. Its formalisation time was about 8 hours.

The time for the proof level task was about 88 hours split as follows: adding a clause for the `bool_cases` method (16 hours), writing the `term_cancel` method for doing arithmetic reasoning (40 hours), adding a new clause for the `weak_fertilize` method (20 hours); modifying the `generalise` method to handle boolean terms (4 hours); and experimenting with method orderings (8 hours). The time for the systems level tasks was 20 hours split as follows: writing a new predicate for finding terms of type Boolean (12 hours), correcting small bugs in the code, and avoiding inefficiencies (e.g. the predicate for well-annotated terms) (8 hours). The version described in chapter 5 used a big endian notation.

The other four versions of the adder used the proof and system extensions done for the first version and the statistics are displayed in the table. The big endian version needed an induction scheme for doing induction on two words of the same length. The 16 hours reported under proof level tasks correspond to the formalisation of the induction rule and the corresponding tactic. The increment induction scheme versions of the adder and the subtracter use an induction scheme where the induction term is the increment operation on words. The 8 hours reported under proof level tasks correspond to the formalisation of increment induction scheme and the tactic. We noticed that there is a direct correspondence between arithmetic operations on the natural numbers and their counterparts on the type *word*. The equivalent term to the constructor $s(\ldots)$ in the natural numbers is $i(\ldots)$ which is defined on words by $i(x) = inc(x, true)$. Then we obtain the equivalence:

$$word2nat(i(x)) = s(word2nat(x))$$

which is used as a lemma in establishing the correspondence between arithmetic operations on words (circuits implementations) and the equivalent operation on the natural numbers (circuit specifications) such as addition and subtraction (and also multiplication, division, factorial and exponentiation). The increment induction scheme used by the adder circuit was also used in the verification of the subtracter, multiplication, exponentiation and factorial circuits. For instance, the conjecture for the adder is:

$$word2nat(plus\_word(x, y)) = word2nat(x) + word2nat(y) \qquad (7.1)$$

where *plus_word* is defined by:

$$\begin{aligned} plus\_word(nil, y) &= y \\ plus\_word(i(x), y) &= i(plus\_word(x, y)) \end{aligned}$$

ALU

The n-bit ALU was verified in two different versions. After verifying the adder, we attempted the *explicit parameter* version of the *ALU*. *Clam* plans the proof in 4:01 minutes using the depth-first planner and *Oyster* executes the plan in 234:19 minutes. The user tasks time was about 32 hours split as follows: the implementation was straight forward, it has the same form as the incrementer and the adder (4 hours); the specification gave more trouble and most of the time was spent debugging each of its 12 components (28 hours). *There were no proof level extensions*; the ones made for the adder worked for the *ALU*. The systems level time was spent in debugging the code and finding a way around the inefficiencies of *Clam* in parsing large terms. The time for this activity was about 48 hours. The proof planning time of the big-endian representation was 11:04 minutes and the proof time time was 274:00 minutes. This version used all of the extensions made for the *explicit parameter* version of the *ALU* as well as the extensions made for the big endian version of the n-bit adder, such as the induction scheme and the induction tactic.

Shifter

The next verification attempted after the *ALU* was the n-bit *shifter unit*. The basic *shifter unit* performs four operations: no-shift, shift-right with serial input ir, shift-left with serial input il, and output zeroes. The operation is determined by two selection variables h_1 and h_2. *Clam* plans the proof in 54 seconds using the depth-first planner and *Oyster* executes the plan in 8:57 minutes. The time of the user tasks was about 24 hours split as follows: the implementation was straightforward as it has the same form as the incrementer, the *adder* and the *ALU* (4 hours); the specification was more problematic and most of the time

was also spent debugging each of its four components (20 hours). *There were no proof level extensions*; the ones made for the *ALU* worked for the *shifter*. The systems level time was significant as it involved developing a new wave-rule parser. The old parser could not deal with a situation in which a variable argument of a recursive function changed to a term not containing that variable in the same argument position in the recursive call. The new wave-rule parser can deal with these cases. This situation was a general requirement that had been anticipated by Alan Bundy but was not implemented when we started this proof. The new parser was developed by the *Clam* development team and we had to wait about three months until it became implemented. The estimated development time for the new parser is about 16 hours.

Processing unit

A processing unit is obtained by composing an *ALU* and a *shifter*. The shifter operates on the output of the *ALU* and the two combined implement 16 operation. The proof planning and execution of this circuit was done very easily by providing the *ALU* and shifter verification theorems as lemmas. *Clam* plans the proof in 0.1 seconds using the depth-first planner and *Oyster* executes the plan in 1 second. The formalisation was straightforward and took about 2 hours. There were no proof level tasks for this proof. Regarding systems level tasks, there was a slight modification of the reduction rules code to apply lemmas which correspond to theorems given by non-wave equations (2 hours).

Multiplier

The next circuit was the *nm*-bit parallel array *multiplier*. This verification was suggested by Toby Walsh[2] who was using this description of a multiplier implementation to encode a factorisation algorithm into *SAT*. The verification of the

[2]Personal communication

multiplier was explained in chapter 5. *Clam* plans the proof in 36 seconds using the depth-first planner and *Oyster* executes the plan in 2:12 minutes. The formalisation was straightforward and took about 12 hours. The proof level extensions mainly consisted of looking for the required lemmas and took about 36 hours. There were no systems level tasks for this proof.

The conjecture for the increment induction scheme version of the multiplier is:

$$word2nat(times\_word(x, y)) = word2nat(x) * word2nat(y)$$

where *times_word* defines multiplication of words in terms of *plus_word*. *Clam* plans the proof in 4 seconds using the depth-first planner and *Oyster* executes the plan in 5 seconds. There were no proof level nor systems level tasks for this proof.

Exponentiation

Having verified the *multiplier* we used it in the verification of more complex circuits like the *exponentiation* and *factorial* circuits. The verification conjecture for the exponentiation circuit is given by:

$$word2nat(exp\_word(x)) = exp(word2nat(x))$$

where *exp_word* is defined in terms of *times_word*. *Clam* plans the proof in 10 seconds using the depth-first planner and *Oyster* executes the plan in 22 seconds. The formalisation was straightforward and took about 4 hours. There were no proof level nor systems level tasks for this proof.

Factorial

The verification conjecture for the factorial circuit is given by:

$$word2nat(fact\_word(x)) = fact(word2nat(x))$$

where *fact_word* is defined in terms of *times_word*. *Clam* plans the proof in 13 seconds using the depth-first planner and *Oyster* executes the plan in 21 seconds. The formalisation was straightforward and took about 4 hours. The proof level tasks took about 4 hours and consisted mainly of giving the required lemmas. There were no systems level tasks for this proof.

Divider

The conjectures for the division operation are these:

$$word2nat(div\_word(x, y)) = word2nat(x)/word2nat(y)$$

$$word2nat(rem\_word(x, y)) = word2nat(x)\ rem\ word2nat(y)$$

where *div_word* and *rem_word* are defined using the constructor *plus_word*. *Clam* plans the proofs in 3 and 4 seconds using the depth-first planner and *Oyster* executes the plan in 15 and 17 seconds respectively. The formalisations were straightforward and took about 4 hours. The proof level tasks took about 12 hours and consisted mainly of giving the required lemmas and finding a new induction scheme where the inductive term is *plus_word* and writing the tactic. There were no systems level tasks for this proof.

Binary counter

The first sequential circuit verification attempted was a binary counter [Gordon 86]. We translated the representation from a relational to a functional representation. We used second order functions to represent the specification and the implementation at the object level and did the proof planning using first-order rewrites. The proof planning time for the counter is *0:05* seconds. The proof time is *0:13* seconds. Signals were defined function types from the natural numbers to words.

The Gordon computer

The next circuit we attempted was the classical *Gordon computer* microprocessor system. As we have already pointed out, translating from a relational into a functional representation was time consuming and it would have been easier to derive the functional representation from scratch. We also had to refresh our computer architecture concepts from undergraduate days and understand the internal workings of the *Gordon computer*. After *14* weeks of calendar time (around 560 hours) *Clam* was able to produce a proof plan which was then executed by *Oyster*. The

planning takes 35:38 minutes using the depth-first planner and *Oyster* executes the plan in 274:15 minutes. The formalisation time was about 120 hours and involved tasks like: re-learning computer architecture and understanding the *Gordon computer* (40 hrs), translating from the relational specification given in [Joyce *et al* 86] into a functional representation (20 hrs), experimenting with various representations (e.g. time abstraction, state-transition graphs, conditional rewrites, vectors vs matrices, stability of signals, etc.) (20 hrs), finding an effective way of representing and traversing the microcode table (20 hrs), and debugging the definitions (20 hrs). The proof level extensions took around 160 hours and consisted of: writing a new sub-method for implementing a memoisation algorithm for computing recursive functions (50 hrs), writing a new method for experimenting with difference matching (40 hrs), experimenting with method orderings (10), and writing the required lemmas and tactics (60). The systems level extensions took around 280 hours. Most of this time was spent in debugging tasks, and consisted of: finding an effective way to parse conditional terms (80 hrs), finding a way around the inefficiencies of *Clam* in parsing large terms (120 hrs), and testing changes to the code (80 hrs).

Modified Gordon computer

The extendability test of the Gordon computer which consisted of replacing the *add* programming instruction with a no-operation instruction was straightforward and took one hour to figure out and perform. The computer timings are slightly lower than the timings for the *Gordon* computer as the no-operation instruction represents less work than the *add* instruction.

7.2.3 Analysis of object-level times

In general, proof timings tend to be higher that proof planning timings. This has to do with the use of type theory specification language at the object level (i.e. *Oyster* level) with time consuming well-formedness goals. Table 7–2 shows a comparison of proof timing where the type checking procedure has been set on

and off. On average, about 70 % of the time is spent in solving type checking subgoals.

7.2.4 Analysis of hierarchical proofs

Proof planning is able to deal with hierarchical verification. For instance, the verification theorems for the *full adder* (fa_sum_ver and fa_carry_ver) were used as lemmas in a verification experiment of the n-bit adder. The verification of the *full adder* presented in appendix B cannot be used in the verification of the n-bit adder because the *sum* and the *carry* are appended together in a list and are not available separately as needed by the adder proof. Despite this, proof planning finds its way without the *full adder* lemmas by expanding the definitions of the *sum* and the *carry* when needed, to prove the n-bit adder conjecture.

Another example of hierarchical verification using proof planning is the sequence of proofs that involve the *incrementer*, the *adder*, the *multiplier*, and the *exponentiator* and *factorial*. The adder proof uses the incrementer theorem as a lemma, the multiplier proof uses the adder theorem as a lemma, and the exponentiator and the factorial proofs use the multiplier theorem as a lemma. As another experiment, for the exponentiator and factorial proofs we deleted the multiplier and adder lemmas, and proof planning was still able to find verification proofs for the exponentiator and factorial circuits without those lemmas.

The *Gordon* computer proof used uninterpreted definitions of the *ALU*, the *adder*, the *subtracter*, and other operations (memory fetch and store), which means that proof planning is also able to deal with *generic specifications* as described by Joyce [Joyce 90].

7.2.5 Analysis of lemmas

In this section we do an analysis of the automation features of proof planning and of the lemmas used by the verification proofs.

Circuit	type checking on (min:sec)	type checking off (min:sec)	% type checking
COMBINATIONAL			
non-recursive			
half adder	0:51	0:12	76
full adder	1:45	0:23	78
1-bit ALU	10:13	2:10	79
4-1 multiplexer	4:23	1:18	70
n-bit incrementer			
big endian	0:41	0:12	71
n-bit adder			
explicit parameter	3:31	0:40	81
big endian	1:41	0:21	79
adder (increment)	0:05	0:02	60
subtracter (increment)	0:06	0:02	67
n-bit alu			
explicit parameter	234:19	7:04	97
big endian	274:00	4:59	98
n-bit shifter			
big endian	8:57	1:40	81
n-bit processing unit			
big endian	0:01	0:004	60
parallel array			
nm-bit multiplier	2:12	0:23	83
multiplier (increment)	0:08	0:02	75
exponentiator (increment)	0:27	0:08	70
factorial (increment)	0:26	0:07	73
divider-quotient (plus)	0:18	0:04	77
divider-remainder (plus)	0:16	0:03	81
SEQUENTIAL			
counter	0:17	0:04	76
Gordon computer	274:15	42:38	84
modified *Gordon* computer	232:26	35:52	85

Table 7–2: Percentage of time spent in type checking

Proof automation

The *Clam-Oyster* system does not give fully automatic proofs in all the cases. We have seen that for some circuits user intervention is required to supply, say, missing lemmas, as is indicated in the last column of the table. However, we have found that the number of lemmas utilised is lower compared to other systems and the type of lemmas are not specialised ones, but rather, general purpose lemmas shared by other proofs, like associativity, commutativity, distributivity, stability of signals, induction schemes, and non-definitional equations of recursive functions (e.g addition by recursion on the second argument). Reasons why *Clam* can find proofs in many cases without supplying extra lemmas include:

- the possibility of establishing new proof strategies by defining general-purpose (or domain-specific) methods and tactics which integrate heuristics;
- the flexibility in the application of the heuristics embedded in the methods by keeping an explicit proof tree of the conjecture and using Prolog's backtracking mechanism, which allows *Clam* to look for alternative method applications when a method fails. This feature makes method ordering unnecessary when using depth-first search, provided the search does not follow a infinite path.
- a look ahead algorithm implemented in rippling analysis which permits the application of predefined induction schemes which do not have to be derivable directly from the recursion functions present in the conjecture.

These features provide a higher degree of automation, although in some cases, as in the case of the *multiplier*, user intervention will be necessary.

Lemmas

The hardware verification proofs require the definitional equations of the implementation, the specification, the definitions of their components, and a few lemmas of various classes. We distinguish five classes of lemmas that we have used in the

verification proofs reported in this thesis. The first three classes refer to the generality of the lemmas: how general or how specific the lemmas are; the fourth class is about lemmas for induction rules, and the fifth class is about lemmas used in hierarchical verification:

general lemmas: these correspond to properties of theories which appear across application domains. For instance, the property of distributivity of multiplication over addition on the natural numbers, and the property of associativity of append over lists, belong to this class.

domain lemmas: these correspond to properties in a given application domain. For instance, the stability of signals in the domain of time-sequential circuits.

problem type lemmas: these are properties specific to a problem or type of problems within a given domain. For instance, converting 13-bit words stored by a program counter to 16-bit words broadcast by a bus, and vice-versa.

induction lemmas: these correspond to the justification of induction rules of inference, and are found across the three classes of lemmas described above. Thus, *general*, *domain* and *problem* lemmas about induction are categorised as induction lemmas and we can find: (1) general induction lemmas in theories, like Peano induction on the natural numbers and structural induction on lists, (2) domain induction lemmas like induction on computer words (lists of booleans), and (3) problem induction lemmas like induction on two words of the same length which model devices such as adders and ALUs.

hierarchical lemmas: these correspond to the verification of systems which are used in building more complex systems. For instance, the verification theorem of a hardware device (i.e. an adder) can be used as a lemma in the verification of a compound device (i.e. a multiplier).

We now describe the lemmas we used to verify hardware:

- *general lemmas:* the non-recursive circuits of table 7–1 are verified by Boolean case analysis. However, the Boolean case analysis tactic, needs a general

lemma which asserts that any Boolean variable is either true or false, and nothing else, so that the values of a Boolean variable form a partition. Thus, proofs by Boolean case analysis require this lemma.

The multiplier required general lemmas which correspond to distributivity of times over plus, a commuted version of distributivity of times over plus, associativity of times, and reduction rules like $x + 0 = x$, $x * 0 = 0$.

- *domain lemmas:* the *Gordon computer* used a lemma about stability of signals: the *switches* to enter data into the computer and the *knob* to select the operation when the computer is idling,

- *problem type lemmas:* the *Gordon computer* used a lemma for doing a case split from the conditions of the specification which are determined by the variables *idle*, *knob*, and *opcode*; and a lemma to ignore three leading zeroes in a 16 bits word to convert it to a 13-bit word.

- *induction lemmas:* The explicit parameter versions of the n-bit adder and the ALU are done by induction on the natural numbers, so we report a induction lemma (general) for these proofs. However, the lemma that justifies this induction scheme is built-into *Oyster*, so the user does not provide it. The n-bit incrementer, the shifter, and the multiplier proofs are done by induction on words, Also, the lemma that justifies this induction scheme is built-into *Oyster* too, so the user does not provide it. The increment versions of the adder, the subtracter the multiplier, the exponentiator, and the factorial circuit are done by induction on words with constructor *word increment*. Similarly the divider (quotient and remainer) proofs are done by induction on words with constructor *word addition*, so we report an induction lemma (domain) for all these proofs. The big endian versions of the adder and the ALU as well as the look-ahead carry version of the adder are done by induction on two words of the same length, so we report an induction lemma (problem) for these proofs.

Circuit	Lemmas					Total
	general	domain	problem	induction	hierarchical	
half adder	1	0	0	0	0	1
full adder	1	0	0	0	0	1
1-bit ALU	1	0	0	0	0	1
4-1 multiplexer	1	0	0	0	0	1
incrementer	1	0	0	1	0	2
adder (explicit parameter)	1	0	0	1	0	2
adder (big endian)	1	0	0	1	0	2
adder (look-ahead carry)	1	0	0	1	0	2
adder (increment)	1	0	0	1	1	3
subtracter (increment)	0	0	0	1	2	3
alu (explicit parameter)	1	0	0	1	0	2
alu (big endian)	1	0	0	1	0	2
shifter (big endian)	1	0	0	1	0	2
processing unit (big endian)	0	0	0	0	2	2
multiplier (little endian)	4	0	0	1	1	6
multiplier (increment)	0	0	0	1	2	3
exponentiator (increment)	0	0	0	1	2	3
factorial (increment)	1	0	0	1	1	3
divider-quotient (plus)	3	0	0	1	1	5
divider-remainder (plus)	3	0	0	1	1	5
Gordon computer	0	1	2	1	0	4

Table 7–3: Types of lemmas used in hardware verification

- *hierarchical lemmas:* The multiplier used the verification theorem of the n-bit adder as a lemma. The increment versions of the adder, subtracter, multiplier, exponentiator, factorial and divider circuits used the verification theorem of the increment operation on words given by equation 7.1, as a lemma. Also, the factorial and exponentiation circuits used the verification theorem of the multiplier as a lemma. Similarly, the processing unit used the shifter and ALU verification theorems as lemmas.

The last column of table 7–1 summarises the number of lemmas needed by each circuit. Table 7–3 summarises the type of lemmas for each of these circuits. The appendices show all of the definitions and lemmas used by the various proofs.

7.3 Summary

We have presented a summary of the experiments done using a proof planning based methodology. We have given statistics that show that, proof planning scales up well and that the initial proof and systems level extensions required by the first circuit of a certain kind served to prove more complex circuits of the same kind. Discounting various one-time development efforts, proof planning may allow users to formalise circuit verification in times which are lower than what they may appear at first sight and provide a methodological way of investigating proofs that fail. User interaction is limited, although in some circumstances human (e.g. the user, proof engineer, or system designer) intervention may be called for in supplying extra lemmas or particular heuristics. The examples show the scalability feature of proof planning, and the test on the modified version of the *Gordon* computer, illustrate the extendability claim. Also, the full adder - adder - multiplier proofs as well as the the adder - multiplier - exponentiation/factorial proofs illustrate the hierarchical verification capability of proof planning. We conjecture that the approach presented can be scaled up once more, to verify more complex circuits of practical interest.

Chapter 8

Related and future work

In this chapter we describe related work on theorem proving for hardware verification and outline future directions of research. In chapter 2 we surveyed various verification environments based on theorem proving. Now, section 8.1 describes experiments similar to the ones we have presented in chapter 7 using some of these well known theorem provers; section 8.2 explores ways in which proof planning can be extended both, to increase its functionality and to verify more complex circuits; and section 8.3 summarises the chapter.

8.1 Related work

The application of automated deduction to hardware verification has been and continues to be an active area of research. Many efforts have been made and currently are going on to apply formal methods to verify all sorts of hardware systems. In this section we examine some of the systems that have been used in verifying circuits similar to the ones we have presented. The theorem provers we examine are *NQTHM*, *HOL*, *PVS*, and *VERIFY*, as well as the *MONA* decision procedure. As we have pointed out, *Clam/Oyster* is a *fully-expansive* system, and so are theorem provers like *HOL* and *NUPRL*. This feature provides the user with the security characteristic of these sort of systems, as opposed to provers like *NQTHM* and *PVS* which don't incorporate this feature [Boulton 94].

8.1.1 NQTHM

Most of the circuits in the table (and many others) have been verified elsewhere using *NQTHM* [CLINC 96]. As an experiment, some of the circuits we have reported were re-implemented in *NQTHM* by a newcomer to formal verification [Rangel 96]. Most of the proof-development time was spent determining the lemmas required by the proof. For instance the proof of the n-bit adder uses the following lemmas: commutativity of plus, commutativity of times, and addition, and multiplication by recursion on the second argument. *Clam* required none of these lemmas for the same proof. We have explained in section 7.2 reasons for which *Clam* uses fewer lemmas compared to systems like *NQTHM*, in which, proof techniques are pre-established and the application ordering of these techniques is fixed with no backtracking. The induction schemes in *NQTHM* are derived automatically from the recursive functions present in the conjectures but is not able to derive schemes which are not directly related to these recursive functions. Whereas in systems like *Clam*, although the induction schemes are not yet generated automatically[1], because of rippling analysis, it can apply induction schemes suggested by the recursive functions present in the conjecture, as well as other type of induction schemes whose derivation is not direct [Kraan *et al* 96]. In general, *Clam* required fewer lemmas than *NQTHM* to verify these circuits. On the other hand, *NQTHM* provided a very stable implementation, and was much easier to learn, since the *Clam* system, being an evolving research tool, is not properly engineered yet.

In other efforts using *NQTHM*, the n-bit adder was verified (big-endian) in half a man-day, where the discovery of the required lemmas was the most difficult part [Stavridou *et al* 88]. A combinational processing unit (ALU and shifter) was verified by Warren Hunt as part of the verification of the FM8501 microprocessor. This processing unit is verified in three theorems corresponding to the word, natural number, and two's complement interpretations. It took about two months effort,

[1] although in principle they can. Currently, the available induction schemes are stored in a data base

runs in a few seconds[2], and used 53 lemmas [Hunt 86]. Although the processor unit reported here is less complicated than FM8501's because we do not include the two's complement interpretation, we use just 2 lemmas in its proof planning. A parallel array multiplier was verified in *NQTHM* by L. Pierre. The proof used five lemmas and took 99.2 seconds in a SUN Sparc2 machine [Pierre 94].

8.1.2 HOL

One of the earliest examples using *HOL* was the verification of the *Gordon computer* by Mike Gordon and his group [Joyce *et al* 86]. This design was later implemented and verified as the Tamarack-3 [3] microprocessor by Jeffrey Joyce [Joyce 90]. The verification took about 5 weeks of proof-development effort and required the derivation of at least 200 lemmas including general lemmas for arithmetic reasoning and temporal logic operators which are now built into *HOL*. It did not require to tune *HOL* and runs in about 30 minutes in a modern machine [4]. Although we assume a synchronous communication with memory whereas Joyce uses an asynchronous handshaking protocol as well as interrupts, we use almost no lemmas in its proof planning (e.g. stability of signals, convert from 16 to 13 bit words, change of *mpc* initial value). Viper's ALU was verified by Wong using *HOL*. The ALU implements a look-ahead carry facility. The proof took one year, involved 488,760 inference steps, and took 53:52 minutes to execute [Wong 93].

Clam could enhance the functionality of *HOL* by automating: (1) the derivation of customised tactics to prove particular conjectures; (2) the selection of

---

[2]Personal communication

[3]A refined implementation of the Gordon computer. Its verification in HOL and PVS is also more abstract, as tri-state values and gates to access the bus, and the input of manual information through the switches and the knob, are not considered. However, Tamarack-3 includes an option for asynchronous communication with memory.

[4]Personal communication

induction parameters (scheme and variables), case splits, and generalisations; (3) the speculation of missing lemmas.

8.1.3 PVS

After we had verified the *ALU*, Shankar took the specification and the implementation we used to reproduce a verification of the same *ALU* in *PVS* [5] [PVS 96]. The *ALU* as well as the n-bit adder, and the Tamarack-3 microprocessor have been verified in *PVS* [Cyrluk *et al* 94,Owre *et al* 94]. The run time for verifying each of these circuits was 2:07, 1:27 and 9:05 minutes respectively in a Sun Sparc-Station 10. These low run-times are explained by the built-in decision procedures available to *PVS*. In these experiments, the user must provide the induction parameters and use a predefined proof strategy. A proof strategy is packaged as a set of *PVS* commands for certain kind of circuits. The *adder* and the *ALU* use the same proof strategy, the Tamarack and the pipelined Saxe microprocessor share another proof strategy. *Clam* could also enhance the automation facilities of *PVS* in a similar way as in the *HOL* case.

8.1.4 VERIFY

The *Gordon computer* was verified by Harry Barrow using the *VERIFY* system. The verification is split into two parts e.g. the microinstruction level and the instruction level and does not use induction, rather, it uses enumeration techniques and symbolic evaluation. At the microinstruction level the proof uses enumeration techniques and requires user interaction to assist the proof when state equations which describe a finite-state machine representation of this level are encountered. At the instruction level the proof is done by symbolic simulation of the microcode and by proving the equivalence between sequences of microinstructions and

[5] Personal communication

behavioural descriptions of each programming instruction. The second part is completed in just 6 minutes of execution time [Barrow 84b].

8.1.5 MONA

The n-bit adder, the n-bit ALU, a commercial implementation of the binary counter and other parameterised circuits which show regularities in their structures have been verified using the *MONA* decision procedure system [MONA 96]. If a circuit which exhibits regularity in its structure can be encoded in the Monadic second-order logic, then MONA gets verifications in times which are orders of magnitude faster that the times taken by theorem proving based tools. If a circuit design is faulty, MONA finds a counter-example. However, many circuits of interest like multipliers and other devices, cannot be encoded in the logic, thus the applicability of the approach for practical use remains limited [Basin & Klarlund 95].

8.1.6 VOSS

The AMD2901 4-bit ALU from Advanced Micro Devices Inc, is a high-speed cascadable microprocessor slice for use in CPUs, peripheral controllers and programmable microprocessors. This circuit was verified using the VOSS system [Seger 93]. The VOSS system does formal verification using symbolic simulation and symbolic trajectory evaluation. The meta-language FL, is a general functional language in which Ordered Binary Decision Diagrams (OBDDs) are built-in so that Boolean functions can be represented, manipulated and compared very efficiently. User interaction is required to create the FSM model which is used in the symbolic trajectory evaluation, and for structuring the specification-verification file including variable ordering.

System	Circuits			
	adder	ALU	multiplier	microprocessor
NQTHM	0.1	10	2	12
HOL	4	52	*	8
VOSS	-	0.2	-	-
REVE	3	-	*	*
CLAM-OYSTER	0.1	2.4	1	12

Table 8–1: Human timings in number of weeks for circuit verification using various tools

8.1.7 Comparison

Table 8–1 displays the human timings for the verification of four standard circuits using the tools described in the previous subsections. We believe that human timings are highly important and more relevant that their computer timings counterparts. The reason for this is that, in general, more that 90% of the total timings (computer and human) are spent on the human side, and any methodology or automated technique that reduces human timings has a high impact in the total development effort. Another reason for the importance of displaying human timings is the fact that often, a great deal of effort is invested in developing efficient programs to verify a given circuit, attaining very low computer timings, without mentioning the amount of labour spent in making the proof look 'automatic'. However, it is not easy to find these timings in the literature nor by asking the developers, because these are not always reported, or the developers did not record them and have forgotten how much time was spent.

The time unit in table 8–1 is number of weeks. The *Clam-Oyster* times were converted to weeks assuming 8 hours per day and 5 days a week. The symbol '-' means that the circuit was verified using that tool but the timing was not reported or was not known by the developers; The symbol '*' means that we did not find in the literature any reports on the use of the system to verify the circuit. The circuits verified using each of these tools are not exactly the same, but have many similarities. For instance, the VIPERS's ALU verified using *HOL* has a look-

ahead carry feature, thus, this circuits is more complex than the other ALUs in the table. Also, the developers do not have the same background and experience, so the actual numbers may be difficult to compare on a one-to-one basis, however, we think that they give an idea of the importance of human timings in the verification effort.

The timings for verifying the adder using NQTHM, HOL and REVE were reported in [Stavridou *et al* 88]; the timings for verifying the ALU in HOL were obtained from [Wong 93], and by personal communication with the developers (W. Hunt and C. Seger) for NQTHM and VOSS. The verification of the multiplier using NQTHM is reported in [Pierre 94] and the timing was obtained from Laurence Pierre by personal communication. The verification of the FM8501 using NQTHM was obtained by personal communication from Warren Hunt, and the timing for the verification of the Gordon computer using HOL is reported in [Joyce *et al* 86]. The verification of the adder, the ALU, and the SAXE microprocessor is reported in [Cyrluk *et al* 94], but the human timings are not listed.

8.2 Future Work

There are several directions for future work. These include the following:

- interface of *Clam* with other tactic-based provers;
- temporal logic reasoning;
- informed search procedures;
- interface of *Clam* with hardware description languages;
- propositional reasoning;
- automatic generation of induction schemes;
- higher-order rippling;

- relational verification; and
- verification of other microprocessor systems.

In the following subsections we describe each of these directions.

8.2.1 Interface with other provers

As noted, much of our time was spent entering and debugging specifications. Part of this is due to our use of a somewhat complicated type-theory as a specification language at the object level. However, our planning approach should work with any tactic based theorem prover; one need only port the tactics associated with the current methods from one system to another so that they produce the same effects. Interfacing *Clam* with a prover like *HOL*, *LAMBDA* or *PVS* could significantly reduce the verification times. It would also provide immediate access to the large collection of tactics and theorems already available for these systems.

LAMBDA

Mike Fourman used the *LAMBDA* system to prove several inductive theorems from the *NQTHM* corpus that had been proved in *Clam* [Cantu 93]. We studied these examples to outline an interface between *Clam* and *LAMBDA* and use *LAMBDA* as our object level system, but we decided to use the existing *Clam-Oyster* interface instead. However, *Clam* proof plans can be translated into *LAMBDA*'s tactics. As a simple example, consider the transcript generated by *LAMBDA* to prove the associativity of addition displayed in figure 8.2.1. This transcript was produced by a tactic written by Mike Fourman to prove a set of theorems by induction from the *NQTHM* corpus [Cantu 93]. Lines 1-4 display the goal. Then an inference rule for induction on the natural numbers is applied. Lines 6-16 discharge the base case. Lines 18-20 apply rewriting on the left-hand side using the recursive equation of addition. Lines 22-24 apply weak fertilisation on the left-hand side. Lines 26-32 apply rewriting twice on the right-hand side using the same equation. The proof is completed by applying the equation for the cancellation of successor

```
1 *******   LEVEL 1   *******
2 1: G // H |- plus (x,plus (y,z)) == plus (plus (x,y),z)
3    --------------------------------------------------
4    G // H |- plus (x,plus (y,z)) == plus (plus (x,y),z)
5
6 plus (0,plus (y,z))
7  ==>> plus (y,z)
8  using G // H |- plus (0,y) == y
9
10 plus (0,y)
11  ==>> y
12  using G // H |- plus (0,y) == y
13
14 plus (y,z) == plus (y,z)
15  ==>> TRUE
16  using G // H |- x == x == TRUE
17
18 plus (1'x',plus (y,z))
19  ==>> 1'(plus (x',plus (y,z)))
20  using G // H |- plus (1'x,y) == 1'(plus (x,y))
21
22 plus (x',plus (y,z))
23  ==>> plus (plus (x',y),z)
24  using G // x == y $ H |- x == y
25
26 plus (1'x',y)
27  ==>> 1'(plus (x',y))
28  using G // H |- plus (1'x,y) == 1'(plus (x,y))
29
30 plus (1'(plus (x',y)),z)
31  ==>> 1'(plus (plus (x',y),z))
32  using G // H |- plus (1'x,y) == 1'(plus (x,y))
33
34 1'(plus (plus (x',y),z)) == 1'(plus (plus (x',y),z))
35  ==>> plus (plus (x',y),z) == plus (plus (x',y),z)
36  using built-in conversion: {n}'x == {n}'y --> x == y
37
38 plus (plus (x',y),z) == plus (plus (x',y),z)
39  ==>> TRUE
40  using G // H |- x == x == TRUE
41
42 *******   LEVEL 2   *******
43    --------------------------------------------------
44    G // H |- plus (x,plus (y,z)) == plus (plus (x,y),z)
```

Figure 8–1: LAMBDA proof for the associativity of addition

```
induction([(x:pnat)-s(v0)]) then
 [base_case(
   [sym_eval(
    [normalize_term(
     [reduction([1,1],[plus1,equ(left)]),
      reduction([1,2,1],[plus1,equ(left)])])]),
    elementary(intro(new[y]) then
     [intro(new[z]) then
      [identity,wfftacs],wfftacs])]),
  step_case(
   ripple(
    wave(direction_out,[1,1],[plus2,equ(left)],[]) then
     [wave(direction_out,[1,2,1],[plus2,equ(left)],[]) then
      [wave(direction_out,[2,1],[plus2,equ(left)],[]) then
       [wave(direction_out,[],[cnc_s,imp(right)],[])]]] then
    [fertilize(strong,v1)])]
```

Figure 8–2: Proof plan for the associativity of addition

in lines 34-36 and the identity rule in lines 38-40. The goal with the hypotheses discharged is shown in lines 42-44.

We notice two facts: the selection of the induction scheme and the induction variable are determined by the user, and the generation of the proof is obtained by the exhaustive application of rewriting. By using *Clam* to guide *LAMBDA*, a customised tactic could be automatically generated to prove a conjecture, the selection of the induction parameters can be automated, and the application of rewriting can be selective rather than exhaustive, thus reducing the search, which can be big if there is a large numbers of inference rules in the data base of rules, which is typically the case in real applications. The proof plan generated by *Clam* to prove this conjecture is displayed in figure 8.2.1 which shows many similarities with the *LAMBDA* proof. It is possible to map this proof plan into a sequence of *LAMBDA* commands that prove the theorem, automating the selection of the

induction parameters and applying just the rewriting steps indicated in the proof plan. For instance, the `wave` method determines the position of the wave-front to be rippled-out, (e.g. [1,1], [1,2,1] and [2,1]) which can be done with *LAMBDA*'s `guide` commands which locate sub-expressions within a expression, and then apply rewriting at the indicated positions [Francis *et al* 92].

HOL

Clam could also be interfaced to the *HOL* system. In fact, a 3-year project between Edinburgh and Cambridge has already started to build this interface [Bundy & Gordon 95] which will make it possible to provide the automation facilities described in section 8.1.2. We have joined this project to develop a 3-way effort between Edinburgh, Cambridge and Monterrey to design hardware verification cases and develop proof engineering activities to test and tune the interface [Cantu 96].

8.2.2 Temporal logic

Clam reasoning capabilities could be strengthened by defining a kind of object-level temporal logic, e.g. *linear time* propositional logic and then proving rewrite equations and implications from which wave-rules can be derived to prove properties of sequential circuits like flip-flops and controllers, as well as verify microprocessor systems with asynchronous communication to external devices. We need to define the modal operators *henceforth*, *eventually*, *next*, and (weak) *until* as follows [Joyce 90]:

$$\begin{aligned}
\Box P &= \lambda t.\ \forall n\ P(t+n) \\
\Diamond P &= \lambda t.\ \exists n\ P(t+n) \\
\bigcirc P &= \lambda t.\ P(t+1) \\
P \cup Q &= \lambda t.\ \forall n\ (\forall m,\ m < n \rightarrow \neg(Q(t+m))) \rightarrow P(t+n)
\end{aligned}$$

and the following temporal propositional operators:

$$\begin{aligned} \neg_t P &= \lambda t.\ \neg P(t) \\ P \wedge_t Q &= \lambda t.\ P(t) \wedge Q(t) \\ P \vee_t Q &= \lambda t.\ P(t) \vee Q(t) \\ P \rightarrow_t Q &= \lambda t.\ P(t) \rightarrow Q(t) \end{aligned}$$

We can then define the following temporal propositional logic ($LTTL$):

1. every atomic formula in higher-order logic is a formula in $LTTL$
2. if P is a formula in $LTTL$, then so are $\neg_t P$, $\Box P$, $\Diamond P$, and $\bigcirc P$
3. if P and Q are formulae in higher-order logic, then so are $P \wedge_t Q$, $P \vee_t Q$, $P \rightarrow_t Q$, $P \cup Q$

A formula in $LTTL$ is valid if and only if it is true at all times:

$$Valid\ P = \forall t.\ P(t)$$

We could then prove equations like:

$$\Box(P \rightarrow_t Q) = \Box P \rightarrow_t \Box Q$$

$$Valid((P \wedge_t (\neg_t Q)) \rightarrow_t (\bigcirc P)) \rightarrow Valid(P \rightarrow_t (P \cup Q))$$

from which wave rules like the following can be derived:

$$\Box(\boxed{\underline{P} \rightarrow_t \underline{Q}}^{\uparrow}) \Rightarrow \boxed{\underline{\Box P} \rightarrow_t \underline{\Box Q}}^{\uparrow}$$

$$Valid(P \rightarrow_t (\boxed{\underline{P} \cup \underline{Q}}^{\uparrow})) \Rightarrow Valid(\boxed{(\underline{P} \wedge_t (\neg_t Q)) \rightarrow_t (\underline{\bigcirc P})}^{\uparrow})$$

and used to prove conjectures in the logic. For instance, we may have the following handshaking protocol: "A data request at time t_1 by a sender is acknowledged at time t_2 by a receiver and a request to end the interaction is signalled by the sender at time t_3 and eventually acknowledged by the receiver at time t_4". The behaviour

of the sender and the receiver can be formalised in the temporal logic, and used to prove conjectures that show that the constraints imposed by the protocol are observed. The logic can also be used to specify memory and interfacing memory with a processor in an asynchronous communication mode [Joyce 90]. Other formalisations can be derived to reason about stability and correctness properties of devices like flip-flops and controllers [Basin & Klarlund 95]. Currently, we do not foresee extensions to methods or tactics to prove theorems in this type of logic. Formalising the logic at the object level as well as the conjectures, and using the proof planning facilities to plan the conjectures seem to be the type of tasks required.

8.2.3 Heuristic search

A more informed search procedure that uses heuristic information and analyses the structure of the current goal to determine dynamically in which order to apply methods would make proof planning more automatic and less user/proof engineer dependent. This heuristic knowledge could be encoded in a search procedure like *best-first* search [Manning *et al* 93] which has been implemented but is not part of the standard version of *Clam*.

8.2.4 Interface to a HDL

An interface to a standard hardware description language would also ease specification and make it possible to integrate proof planning better into the hardware design cycle. Many efforts have been made to provide interfaces between hardware description languages used by design engineers (e.g. IEEE's VHDL) and formal methods tools [Delgado & Breuer 95]. However, the semantics of HDLs is often not well defined and the interface is not easy to develop. HDLs that meet the required properties are then developed, but these lack the required generality, or alternatively, subsets of standard HDLs are defined and interfaced with provers [Hunt & Brock 92]. We think that the latter approach could be explored in the *Clam* context [Gordon 95].

8.2.5 Propositional reasoning

Another area for further work concerns improving the power or efficiency of some of our methods. For example, reasoning about boolean circuits by case evaluation, which has exponential growth, could be replaced by a more efficient routine based on BDDs and be implemented as a sub-method of the method `sym_eval`.

8.2.6 Lemma speculation

Although sometimes *Clam* is able to get a proof through without a missing lemma, in other cases the user has to supply lemmas that cannot be guessed by *Clam*. However, proof planning provides additional features that permit the speculation of lemmas before appealing to the user. The *Critics mechanism* is such a feature for finding missing lemmas, induction schemes, and generalisations during the rippling process and is described in [Ireland & Bundy 96]. We have done a preliminary investigation to speculate lemmas for some of the circuits verified using a prototype version of *Clam* which implements critics. For instance, if we delete the sub-method `term_cancel` in the experiment about the *incrementer*, the planning fails because it requires the wave rule $U + s(V) \Rightarrow s(U + V)$ which is missing. With this lemma the planning succeeds. This lemma can be speculated easily by the current critics in *Clam* which are available only in an experimental version of *Clam* which we did not used. Similarly, the lemmas that we have supplied for the multiplier proof (e.g. associativity, commutativity, and distributivity of addition and multiplication), can be speculated by similar critics. The work needed consists of translating methods and sub-methods from the syntax of the official version of *Clam* to the one used by the critics prototype, and try the experiments.

8.2.7 Automatic generation of induction schemes

Clam keeps a data base of predefined induction schemes. *Clam* searches this data base to select a scheme for proof by induction. This provides flexibility in use of induction schemes. In systems like *NQTHM* although the type of schemes used

are those derivable from the recursive functions in the conjecture, the generation of such schemes is automatic. The generation of the type of induction schemes used by *Clam* for a particular conjecture can be automated and avoid keeping an explicit data base of induction schemes. This extension of *Clam* applies to proof by induction in general, and is not exclusive of hardware verification. This has been a research goal of the *MRG* but has not been implemented.

8.2.8 Higher-order rippling

The kind of rippling we have described and used to verify circuits applies to first-order logic. Rippling in higher-logic requires higher-order wave annotations. Another version of *Clam* implemented in Lambda-Prolog has been developed by the *MRG* staff to address this kind of problems [Smaill & Green 96]. Reasoning about circuits with higher-order functions like signals is commonplace in sequential circuits, thus, the Lambda-Prolog version of *Clam* could be used to prove properties of this kind of circuits.

8.2.9 Relational verification

The style of meta-level inference implemented in *Clam* is very effective for reasoning about functional representations, but cannot deal with their relational counterparts. We did the verification of the full adder in a relational style, and have outlined the correctness of the n-bit adder, but extending it to circuits with replicated structures and feedback loops such as the ones we have presented requires adaptation of most of the proof planning techniques (e.g. rippling, fertilisation, rippling analysis, etc). Efforts have been done by various researchers to address these issues [Bundy & Lombart 95,Lombart & Deville 94,Åhs & Wiggins 94].

8.2.10 Microprocessor verification

With some of these improvements (e.g. temporal reasoning), *Clam* could be used to verify other microprocessor systems with extended functionality. For instance,

the verification of *Tamarack-3* which includes interrupts and a handshaking asynchronous communication protocol with memory would be a natural extension to the *Gordon computer*. Another 16-bit microprocessor that provides asynchronous communication with memory and that can be verified in *Clam* is the FM8501. At the 32-bit level, the FM9001 could be attempted next as well as microprocessors with pipe-line architectures verified in HOL [Windley & Coe 94].

8.3 Summary

We have presented a summary of work in hardware verification using theorem provers which have verified circuits similar to the ones we have reported. Human and computer timings in many cases are comparable, but *Clam* provides more automation facilities, reflected in the automatic generation of customised tactics, the automation of induction decisions, the reduced number of lemmas utilised in the proofs, and the the selective application of rewriting. We also have described future work that would enhance *Clam* capabilities for hardware verification and with appropriate integration would allow us to attempt the verification of more complex circuits of the type typically found in commercial and safety-critical applications.

Chapter 9

Conclusions

We have investigated the application of proof planning for automating hardware verification tasks. Our starting point was the current implementation of proof planning in the *Clam* system in October 1992 and its application in domains like inductive theorem proving. During the following four years we conducted a series of experiments and drew lessons which can be summarised in the following contributions:

- show by experimentation the applicability, scalability and extendability of proof planning for automating hardware verification tasks;
- a methodology for hardware verification;
- profiling of the *Clam* system.

First, our working hypothesis was that proof planning could be used to verify hardware. Experiments show that the *Clam* system and the proof planning idea carry over well to automate proofs in this new domain, although a number of extensions in the details (as opposed to the spirit) of *Clam* and the development of new general tactics and methods were required. The kind of automation provided by proof planning comprises:

- the automatic generation of a customised tactic, given a conjecture, from a set of general-purpose methods and tactics, and a search strategy;

- the automatic selection of induction schemes and variables from a predefined set of induction schemes;

- the automatic generation of case splits and generalisations in the inductive step;

- and the use of fewer lemmas compared to other provers.

One may ask how important these features are. Critics of interactive theorem proving have claimed that full proof automation is required in order to have formal methods used by industry in a regular basis [Saiedian 96]. We think that the type of automation provided by proof planning is a step in this direction.

Most importantly, we have seen the robustness of proof planning and the features of extendability and scalability of proof planning. Robustness means that proof planning adapts to modifications on the implementation and the specification of a circuit and finds a tactic automatically without user intervention. Extendability and scalability means that similar circuits of a certain class or variants of a particular circuit can be verified using the same methods and planner (extendability), and that circuits of higher complexity can also be verified using the same methods and planner (scalability). This is interesting, because allows us to conjecture that the proof planning technique can be applied to industrial-strength circuits, which are of higher complexity, and are found in the upper region of figure 9–1. The verification of this sort of systems will be further facilitated by integrating *Clam* with tactic-based hardware verification oriented theorem provers such as HOL and LAMBDA. *Clam* will enhance the automating capabilities of these provers as well as providing them with the extendability and scalability features of proof planning, and will benefit from all of the tactics that have been developed for these provers in verifying real-world hardware systems.

Second, we have presented a verification methodology that decomposes formal proof into three conceptually different kind of tasks: user, proof, and systems tasks. In this investigation these activities were performed by one and the same person, but if we think about an organisational context, this distinction can facil-

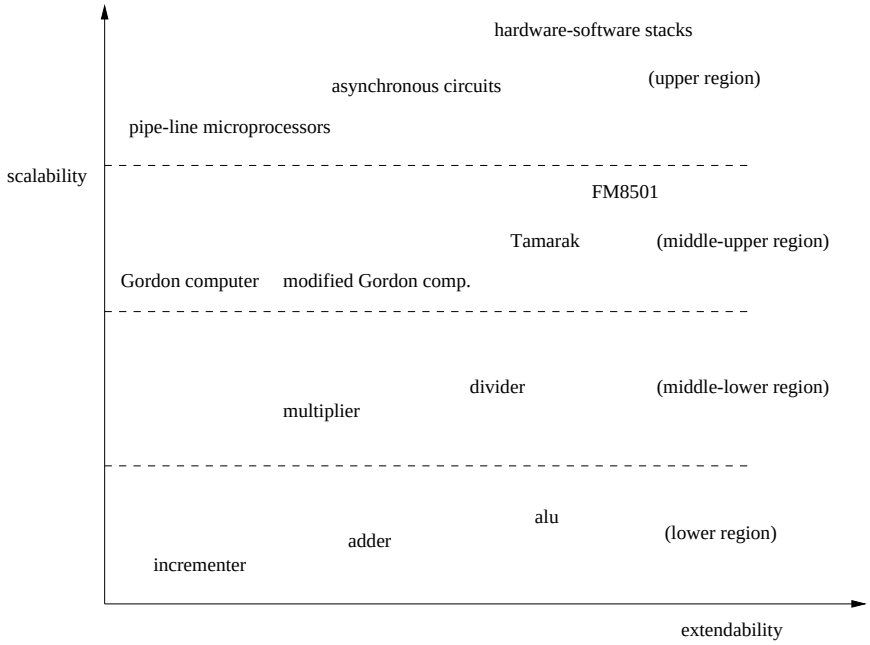

Figure 9–1: Extendability and scalability of proof planning

itate the adoption of formal methods in a firm and provide a smoother transition towards its use. Typically, the formalisation task will be done by a hardware design engineer with a background in digital logic, who will write the specification, the implementation and a verification conjecture of a hardware design in a logic formalism, and will run *Clam* and a prover to get a composite tactic specific for that conjecture. There may be many users like this within the organisation. Then the proof engineer who will belong to a formal methods department/unit within the same organisation, will provide assistance to all of the users in the company if something goes wrong with some proof. And finally, a systems engineer, who will provide support to the verification environment and will work for a consulting company outside the organisation. It may be the case that the proof engineering activities would be offered by an outside consulting company as well, but the important point is making the conceptual distinction among the three types of activities.

Proof planning mechanisms apply both to combinational and sequential hardware, the difference being the handling of time. We investigated several kinds of

parameterised circuits and were able to develop methods which captured heuristics suitable for reasoning about families of such designs. We reported on our timings for formalising particular circuits and doing proof/systems engineering activities to tune the methods and to tune *Clam*. Although the times are sometimes high for initially verifying new kinds of circuits, subsequent verifications times were respectable and would gracefully decrease once we discount one-time activities, with the majority of time being spent on simply entering and debugging the formalisation. This provided support for our belief that a system like *Clam* might be usable by hardware engineers, provided that there is a type of proof engineering support in the background to tune the proof techniques (e.g. methods, tactics, lemmas, induction schemes, etc.) when required.

Third, we contributed to the profiling of the *Clam* system. We obtained the largest proofs that had been done in *Clam-Oyster* so far. The experiments forced situations that had not arisen in previous proofs, which leaded to improvements in the code, and suggested extensions that were implemented either by us, or by the *MRG* staff, improving in this way the functionality and the robustness of *Clam*.

And finally, we think that the research reported in this thesis contributes to our knowledge about the use of meta-level reasoning based on proof planning to verify hardware in tactic-based settings. With additional enhancements and integration with other proving systems, our approach could become a viable way of automating industrial-strength verification tasks in the hardware domain.

Bibliography

[Åhs & Wiggins 94] T. Åhs and G. A. Wiggins. Relational rippling for logic program synthesis and transformation, 1994. Presented at the Fourth International Workshop on Logic Program Synthesis and Transformation.

[Akers 78] S.B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C27:509–516, 1978.

[Barrow 84a] Harry G. Barrow. *Proving the Correctness of Digital Hardware Designs*, pages 64–77. CMP Publications, Inc., 600 Community Drive, Manhasset, NY 11030, 1984.

[Barrow 84b] Harry G. Barrow. Verify: A program for proving correctness of digital hardware designs. *AI Journal*, 24:437–491, 1984.

[Basin & DelVecchio 89] David A. Basin and Peter DelVecchio. Verification of combinational logic in Nuprl. In *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, Ithaca, New York, 1989. Springer-Verlag.

[Basin & Friedrich 96] David Basin and Stefan Friedrich. Modeling a hardware synthesis methodology in isabel. In *Proceedings of the Theorem Proving in Higher Order Logics, LNCS 1125*, pages 33–50. Springer, 1996.

[Basin & Klarlund 95] David Basin and Nils Klarlund. Hardware verification using monadic second-order logic. In *Proceedings of the Conputer Aided Verification Conference, LNCS 939*, pages 31–41. Springer-Verlag, 1995.

[Basin & Walsh 93] David Basin and Toby Walsh. Difference unification. In R. Bajcsy, editor, *Proc. 13th Intern. Joint Conference on Artificial Intelligence (IJCAI '93)*, volume 1, pages 116–122, San Mateo, CA, 1993. Morgan Kaufmann.

[Basin & Walsh 96a] David Basin and Toby Walsh. Annotated rewriting in inductive theorem proving. *Journal of Automated Reasoning*, 16(1–2):147–180, 1996.

[Basin & Walsh 96b] David Basin and Toby Walsh. A calculus for and termination of rippling. *Journal of Automated Reasoning*, 16(1–2):147–180, 1996.

[Basin 91] David A. Basin. Extracting circuits from constructive proofs. Research Paper 533, Dept. of Artificial Intelligence, University of Edinburgh, 1991. Also appeared in Proceedings of the IFIP-IEEE International Workshop on Formal Methods in VLSI Design, Miami USA, 1991.

[Bishop *et al* 95] B. Bishop, W. Hunt, and M. Kaufmann. The fm9001 microprocessor proof. Technical report 86, Computational Logic, Inc, 1995.

[Boulton 94] Richard J. Boulton. Efficiency in a fully-expansive theorem prover. Technical Report 337, University of Cambridge Computer Laboratory, 1994.

[Boyer & Moore 79] R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, 1979. ACM monograph series.

[Boyer & Moore 81] R. S. Boyer and J S. Moore. Metafunctions. In R. S. Boyer and J S. Moore, editors, *The Correctness Problem in Computer Science*, pages 103–184. Academic Press, 1981.

[Boyer & Moore 88] R. S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, 1988. Perspectives in Computing, Vol 23.

[Brayton 96] R.K. et al Brayton. Vis. In M. Srivas and A. Camilleri, editors, *Proceedings of the Formal Methods for Computer-Aided Design Conference*, number 1166 in Lecture Notes in Computer Science, pages 248–256. Springer-Verlag, 1996.

[Bryant 86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C35:677–691, 1986.

[Bryant *et al* 87] R.E. Bryant, D. Beatty, K. Brace, K. Chao, and T. Sheffer. A compiler simulator for mos circuits. In *Proceedings of the 24th ACM/IEEE Design Automation Conference*, pages 9–16. ACM, 1987.

[Bundy & Gordon 95] A. Bundy and M. Gordon. Automatic guidance of mechanically generated proofs. Epsrc research proposal, Edinburgh and Cambridge Universities, 1995.

[Bundy & Lombart 95] Alan Bundy and V. Lombart. Relational rippling: a general approach. In C. Mellish, editor, *Proceedings of IJCAI-95*, pages 175–181. IJCAI, 1995.

[Bundy 88] Alan Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988. Longer version available from Edinburgh as DAI Research Paper No. 349.

[Bundy *et al* 89] Alan Bundy, F. van Harmelen, J. Hesketh, A. Smaill, and A. Stevens. A rational reconstruction and extension of recursion analysis. In N. S. Sridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 359–365. Morgan Kaufmann, 1989. Also available from Edinburgh as DAI Research Paper 419.

[Bundy *et al* 90] Alan Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.

[Bundy *et al* 91] Alan Bundy, Frank van Harmelen, Jane Hesketh, and Alan Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324, 1991. Earlier version available from Edinburgh as DAI Research Paper No 413.

[Bundy *et al* 93] Alan Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993. Also available from Edinburgh as DAI Research Paper No. 567.

[Camilleri 88] Albert J. Camilleri. Executing behavioural definitions in higher order logic. Technical Report 140, University of Cambridge Computer Laboratory, 1988.

[Cantu 93] Francisco J. Cantu. Inductive proofs in LAMBDA. Blue Book Note 819, MRG, University of Edinburgh, 1993.

[Cantu 96] Francisco J. Cantu. Design, experimentation, and analysis of hardware verification cases in a proof-planning/higher-order logic framework. Research proposal submitted to CONACYT, Mexico, Monterrey Institute of Technology, 1996.

[Cantu *et al* 96] Francisco Cantu, Alan Bundy, Alan Smaill, and David Basin. Experiments in automating hardware verification using inductive proof planning. In M. Srivas and A. Camilleri, editors, *Proceedings of the Formal Methods for Computer-Aided Design Conference*, number 1166 in Lecture Notes in Computer Science, pages 94–108. Springer-Verlag, 1996.

[CHECKOFF-E 96] CHECKOFF-E. Abstract Hardware Limited. http://www.ahl.co.uk/chkoff-e.html, 1996.

[CHECKOFF-M 96] CHECKOFF-M. Abstract Hardware Limited. http://www.ahl.co.uk/chkoff-m.html, 1996.

[Clarke & Emerson 81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchonization scheletons using branching time temporal logic. In *The Proceedings of Workshop on Logics of Programs, LNCS 131*, pages 52–71. Springer-Verlag, 1981.

[CLINC 96] CLINC. Computational Logic Inc., USA. http://www.cli.com, 1996.

[Cohn 88] Avra Cohn. A proof of correctness of the viper microprocessor. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*, pages 27–71. Kluwer Academic Publishers, 1988.

[Constable *et al* 86] R. L. Constable, S. F. Allen, H. M. Bromley, *et al. Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[Cyrluk *et al* 94] D. Cyrluk, N. Rajan, N. Shankar, and M.K. Srivas. Effective theorem proving for hardware verification. In *2nd Conference on Theorem Provers in Circuit Design.* Springer-Verlag, 1994.

[Delgado & Breuer 95] C. Delgado and P. Breuer. *Formal Semantics for VHDL.* Kluwer Academic, 1995.

[DelVecchio 90] Peter E. DelVecchio. The design and formal verification of an integrated circuit for use in a floating-point systolic array fast fourier transform processor. Unpublished M.Sc. thesis, Cornell University, 1990.

[Devadas *et al* 87] S. Devadas, H.T. Ma, and A.R. Newton. On the verification of sequential machines at differing levels of abstraction. In *The Proceedings of the 24th ACM/IEEE Design Automation Conference*, pages 271–276. IEEE Computer Society Press, 1987.

[Dill 89] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.*

Unpublished PhD thesis, Carnegie-Mellon University, 1989.

[Emerson & Lei 86] E.A. Emerson and C.L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *The Proceedings of the Annual Symposium on Logic in Computer Science*, pages 267–278. IEEE Computer Society Press, 1986.

[Fourman & Hexsel 91] M. P. Fourman and R. Hexsel. Formal synthesis. In Graham Birtwistle, editor, *IV Higher Order Workshop*. Springer-Verlag, 1991.

[Francis *et al* 92] M. Francis, S. Finn, E. Mayger, and R.B. Hughes. *Reference Manuel for the Lambda System 4.2 Vol I.* Abstract Hardware Ltd, 1992.

[Frank *et al* 92] I. Frank, D. Basin, and Alan Bundy. An adaptation of proof-planning to declarer play in bridge. In *Proceedings of ECAI-92*, pages 72–76, Vienna, Austria, 1992. Longer Version available from Edinburgh as DAI Research Paper No. 575.

[Gallier 86] J. Gallier. *Logic for Computer Science.* Harper & Row, New York, 1986.

[Galton 87] Antony Galton, editor. *Temporal Logics and their Applications.* Academic Press, 1987.

[Genesereth & Nilsson 87] M. R. Genesereth and N. J. Nilsson. *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann, Palo Alto, CA., 1987.

[Gordon 83] Michael Gordon. Lcf-lsm. Technical Report 41, University of Cambridge Computer Laboratory, 1983.

[Gordon 85] Michael Gordon. HOL: A machine oriented formulation of higher-order logic. Technical Report 68, University of Cambridge Computer Laboratory, 1985.

[Gordon 86] Michael Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. Milne and P.A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 153–177. Elsevier Science Publishers, 1986.

[Gordon 88] Michael Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*, pages 73–128. Kluwer Academic Publishers, 1988.

[Gordon 95] Michael Gordon. The semantic challenge of Verilog HDL. In *The Proceedings of Tenth Annual IEEE Symposium on Logic in Computer Science,*. IEEE, Inc., 1995.

[Gordon *et al* 79] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF - A mechanised logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer Verlag, 1979.

[Gupta 91] A. Gupta. Formal hardware verification methods: A survey. Technical Report CMU-CS-91-193, Carnegie-Mellon University, 1991.

[Hanna & Daeche 86] F.K. Hanna and N. Daeche. Specification and verification of digital systems using higher-order predicate logic. In *IEE Proceedings*, pages 242–254. IEE, 1986.

[Hanna *et al* 89a] F.K. Hanna, N. Daeche, and M. Longley. Veritas+: A specification language based on type theory. In *Workshop on Hardware, Specification, Verification and Synthesis: Mathematical Aspects*. Springer-Verlag LNCS, 1989.

[Hanna *et al* 89b] F.K. Hanna, M. Longley, and N. Daeche. Formal synthesis of digital systems. In Luc Claesen, editor, *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pages 532–548. Elsevier Sciences Publishers, 1989.

[Havelund & Shankar 96] Klaus Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe FME '96*, number 1051 in Lecture Notes in Computer Science, pages 662–681, Oxford, UK, March 1996. Springer-Verlag.

[Hesketh 91] J. T. Hesketh. *Using Middle-Out Reasoning to Guide Inductive Theorem Proving*. Unpublished PhD thesis, University of Edinburgh, 1991.

[Hoare 78] C. A.R. Hoare. Communicating Sequential Processes. *Communications of the Association for Computing Machinery*, 21(8):666–677, 1978.

[HOL 96] HOL.
University of Cambridge Computer Laboratory, UK.
http://www.cl.cam.ac.uk/Research/HVG/HOL/HOL.html, 1996.

[Hopcroft & Ullman 79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.

[Horn 95] Christian Horn. The Oyster proof development system. Technical report, Department of Artificial Intelligence, U. of Edinburgh, 1995.

[Hughes & Cresswell 90] G. E. Hughes and M. J. Cresswell. *An Introduction to Modal Logic*. Routledge, 1990.

[Hunt & Brock 92] Warren Jr. Hunt and Bishop C. Brock. A formal hdl and its use in the fm9001 verification. In *Mechanized Reasoning and Hardware Design*, pages 35–47. Prentice Hall, 1992.

[Hunt 86] Warren Hunt. FM8501: A verified microprocessor. Technical Report 47, Institute for Computing Science, University of Texas at Austin, 1986.

[IEEE 88] IEEE. *Standard VHDL Language Reference Manual*. IEEE Press, 1988.

[Ireland & Bundy 96] A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111, 1996. Also available as DAI Research Paper No 716, Dept. of Artificial Intelligence, Edinburgh.

[ISABELLE 96] ISABELLE. University of Cambridge, Computer Laboratory. http://www.cl.cam.ac.uk/Research/HVG/isabelle.html, 1996.

[Joyce 90] Jeffrey J. Joyce. Multi-level verification of microprocessor-based systems. Technical Report 195, University of Cambridge Computer Laboratory, 1990.

[Joyce *et al* 86] Jeff Joyce, G. Graham Birtwistle, and M. Gordon. Proving a computer correct in higher order logic. Technical Report 100, University of Cambridge Computer Laboratory, 1986.

[Kraan *et al* 93] I. Kraan, D. Basin, and A. Bundy. Logic program synthesis via proof planning. In K. K. Lau and T. Clement, editors, *Logic Program Synthesis and Transformation*, pages 1–14. Springer-Verlag, 1993. Also available as Max-Planck-Institut für Informatik Report MPI-I-92-244 and Edinburgh DAI Research Report 603.

[Kraan *et al* 96] I. Kraan, D. Basin, and A. Bundy. Middle-out reasoning for synthesis and induction. *Journal of Automated Reasoning*, 16(1–2):113–145, 1996. Also available from Edinburgh as DAI Research Paper 729.

[Kurshan 87] R.P. Kurshan. Reducibility in analysis of coordination. In *The Proceedings of Discrete Event Systems: Models and Applications , LNCS 103*, pages 19–39. Springer-Verlag, 1987.

[LAMBDA 96] LAMBDA. Abstract Hardware Limited. http://www.ahl.co.uk/lambda.html, 1996.

[Lombart & Deville 94] V. Lombart and Y. Deville. Rippling on relational structures. Research report, November 1994. Available as research report RR94-16, Département d'ingénierie informatique, Université catholique de Louvain, Belgium.

[Lowe 91] Helen Lowe. Extending the proof plan methodology to computer configuration problems. *Artificial Intelli-*

gence Applications Journal, 5(3), 1991. Also available from Edinburgh as DAI Research Paper 537.

[Madre & Billon 88] J.C. Madre and J.P. Billon. Proving circuit correctness using formal comparison between expected and extracted behaviour. In *The Proceedings of the 25th ACM/IEEE Design Automation Conference*, pages 205–210. IEEE, Inc., 1988.

[Manna & Pnuelli 81] Z. Manna and A. Pnuelli. Verification of concurrent programs: Temporal proof principles. In Springer-Verlag, editor, *Proceedings of the Workshop on Logics of Programs, LLNCS 131*, New York, 1981.

[Manning *et al* 93] A. Manning, A. Ireland, and Alan Bundy. Increasing the versatility of heuristic based theorem provers. In A. Voronkov, editor, *International Conference on Logic Programming and Automated Reasoning – LPAR 93, St. Petersburg*, number 698 in Lecture Notes in Artificial Intelligence, pages pp 194–204. Springer-Verlag, 1993.

[Mano 79] M. Morris Mano. *Digital Logic and Computer Design.* Prentice Hall, Inc, 1979.

[Mayger & Harris 91] E. Mayger and R.L. Harris. *User Guide for the Lambda System 4.1.* Abstract Hardware Ltd, 1991.

[Mayger 91] Eleanor Mayger. *Dialog Tutorial.* Abstract Hardware Ltd, 1991.

[McMillan 92] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1992.

[Melham 88] Thomas Melham. Abstraction mechanisms for hardware verification. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*, pages 127–157. Kluwer Academic Publishers, 1988.

[Milne 85] George J. Milne. Simulation and verification: Related techniques for hardware analysis. In C.J. Koomen and T. Moto-oka, editors, *The Proceedings of the Seventh Conference on Computer Hardware Description Languages and their Applications*. Elsevier Science Publishers, 1985.

[MONA 96] MONA. University of Aarhus, Department of Computer Science, Denmark. http://www.daimi.aau.dk/ mbiehl/Mona/main.html, 1996.

[Moore 89] J Moore. System verification. *Journal of Automated Reasoning*, 5(4):409–410, 1989.

[Moore *et al* 96] J S. Moore, T. Lynch, and M. Kaufmann. A mechanically checked proof of the correctness of the kernel of the AMD5k86 floating-point division algorithm. Technical report 112, Computational Logic, Inc, 1996.

[Moszkowski 85] B. Moszkowski. A temporal logic for multi-level reasoning about hardware. *Computer*, pages 10–19, 1985.

[OTTER: 96] OTTER:. An automated Deduction System. http://www.mcs.anl.gov/home/mccune/ar/otter/index.html, 1996.

[Owre *et al* 94] S. Owre, J.M. Rushby, N. Shankar, and M.K. Srivas. A tutorial on using pvs for hardware verification. In *2nd*

Conference on Theorem Provers in Circuit Design. Springer-Verlag, 1994.

[Pierre 94] Laurence Pierre. An automatic generalization method for the inductive proof of replicated and parallel architectures. In *2nd Conference on Theorem Provers in Circuit Design*, pages 72–91. Springer-Verlag, 1994.

[Pnuelli 77] A. Pnuelli. A temporal logic of programs. In IEEE, editor, *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, New York, 1977.

[PVS 96] PVS. Stanford Research Intritute Computer Science Lab., usa. http://www.csl.sri.com, 1996.

[Rangel 96] Victor Rangel. Metodos formales para verificacion de hardware: Un estudio comparativo. Unpublished M.Sc. thesis, Instituto Tecnologico de Monterrey, Mexico, 1996.

[Richardson 95] J. D. C. Richardson. *Proof planning data type changes in pure functional programs.* Unpublished PhD thesis, Department of Artificial Intelligence, University of Edinburgh, September 1995.

[Robinson 65] J. A. Robinson. A machine oriented logic based on the resolution principle. *J Assoc. Comput. Mach.*, 12:23–41, 1965.

[Rueß *et al* 96] H. Rueß, N. Shankar, and M. K. Srivas. Modular verification of SRT division. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in Lecture Notes in Computer Science, pages 123–134, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[Rushby 96] John Rushby. Automated deduction and formal methods. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in Lecture Notes in Computer Science, pages 169–183, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[Saiedian 96] Hossein Saiedian. An invitation to formal methods. *IEEE Computer*, 29(4):16–30, April 1996. A "roundtable" of short articles by several authors.

[Seger 93] Carl-Johan H. Seger. Voss - a formal hardware verification system user's guide. Technical Report 45, Department of Computer Science, University of British Columbia, Canada, 1993.

[Silver 83] B. Silver. Learning equation solving methods from examples. In Alan Bundy, editor, *Proceedings of the Eighth IJCAI*, pages 429–431. International Joint Conference on Artificial Intelligence, 1983. Also available from Edinburgh as DAI Research Paper 184.

[Silver 85] B. Silver. *Meta-level inference: Representing and Learning Control Information in Artificial Intelligence*. North Holland, 1985. Revised version of the author's PhD thesis, Department of Artificial Intelligence, U. of Edinburgh, 1984.

[Smaill & Green 96] A. Smaill and I. Green. Higher-order annotated terms for proof search. volume 1125 of *Lecture Notes in Computer Science*, pages 399–414. Springer, 1996. Also available as DAI Research Paper 799.

[Srivas & Miller 95] Mandayam Srivas and Steven P. Miller. Applying formal verification to a commercial microprocessor. In *Proceedings of the IFIP International Conference of Computer Hardware Description Languages*, pages 493–502. IFIP, 1995.

[Stavridou *et al* 88] V. Stavridou, H. Barringer, and D.A. Edwards. Formal specification and verification of hardware: A comparative case study. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pages 89–96. IEEE, 1988.

[Sterling *et al* 82] L. Sterling, Alan Bundy, L. Byrd, R. O'Keefe, and B. Silver. Solving symbolic equations with PRESS. In J. Calmet, editor, *Computer Algebra, Lecture Notes in Computer Science No. 144.*, pages 109–116. Springer Verlag, 1982. Also available from Edinburgh as DAI Research Paper 171.

[vanHarmelen & group 96] van Harmelen and MRG group. The Clam proof planner, user manual and programmer manual (version 2.5). Technical report, Department of Artificial Intelligence, University of Edinburgh, 1996.

[Walsh *et al* 92] T. Walsh, A. Nunes, and Alan Bundy. The use of proof plans to sum series. In D. Kapur, editor, *11th Conference on Automated Deduction*, pages 325–339. Springer Verlag, 1992. Lecture Notes in Computer Science No. 607. Also available from Edinburgh as DAI Research Paper 563.

[Windley & Coe 94] P.J. Windley and M.L. Coe. A correctness model for pipelined microprocessors. In *2nd Conference on Theorem Provers in Circuit Design*, Lecture Notes

in Computer Science, pages 35–54. Springer-Verlag, 1994.

[Wong 93] Wai Wong. Formal verification of VIPER's ALU. Technical Report 300, University of Cambridge Computer Laboratory, 1993.

[Yoeli 90] Michael Yoeli. *Formal Verification of Hardware Design*. IEEE Computer Society Press, 1990.

Appendix A

Object level definitions

This appendix describes basic definitions on types, as well as rewrite equations for operations and conversion functions on types. An *Oyster* definition has the form: `term1<==>term2`. A rewrite equation has the form of an *Oyster* theorem: `[hyp1,...,hypn]==>formula in Type`.

A.1 Types

```
{bool}<==>unary\unary
{true}<==>inl(unit)
{false}<==>inr(unit)
{word}<==>{bool} list
wordn(n)<==>{w:{word}\length(w)=n in pnat}
memn(n)<==>wordn(n) list
signal<==>(pnat=>{word})
signaln(n)<==>(pnat=>wordn(n))
state<==>(memn(16)#wordn(13)#wordn(16)#{bool})
imp_state<==>(memn(16)#wordn(13)#wordn(16)#{bool}#
              wordn(16)#wordn(13)#wordn(16)#
              wordn(16)#wordn(5)#{bool})
```

A.2 Operations on types

A.2.1 Booleans

```
NOT
[]==>(not{false})={true}in{bool}
[]==>(not{true})={false}in{bool}

AND
[]==>y:{bool}=>(y and{false})={false}in{bool}
[]==>y:{bool}=>({false}and y)={false}in{bool}
[]==>({true}and{true})={true}
```

```
OR
[]==>y:{bool}=>(y or{true})={true}in{bool}
[]==>y:{bool}=>({true}or y)={true}in{bool}
[]==>({false}or{false})={false}in{bool}

XOR
[]==>y:{bool}=>xor(y,y)={false}in{bool}
[]==>xor({true},{false})={true}in{bool}
[]==>xor({false},{true})={true}in{bool}
```

A.2.2 Natural numbers

```
PLUS
[]==>y:pnat=>plus(0,y)=y in pnat
[]==>x:pnat=>y:pnat=>plus(s(x),y)=s(plus(x,y))in pnat

TIMES
[]==>y:pnat=>times(0,y)=0 in pnat
[]==>x:pnat=>y:pnat=>times(s(x),y)=plus(times(x,y),y)in pnat

EXPONENTIATION
[]==>n:pnat=>exp(n,0)=s(0)in pnat
[]==>n:pnat=>i:pnat=>exp(n,s(i))=times(n,exp(n,i))in pnat

FACTORIAL
[]==>fac(0)=s(0)in pnat
[]==>n:pnat=>fac(s(n))=times(s(n),fac(n))in pnat

PREDECESSOR
[]==>pred(0)=0 in pnat
[]==>x:pnat=>pred(s(x))=x in pnat

MINUS
[]==>x:pnat=>minus(x,0)=x in pnat
[]==>y:pnat=>minus(0,y)=0 in pnat
[]==>x:pnat=>y:pnat=>minus(s(x),s(y))=minus(x,y) in pnat

LESS
[]==>x:pnat=>less(x,0)=void in u(1)
[]==>x:pnat=>less(0,x)= (x=0 in pnat=>void)in u(1)
[]==>x:pnat=>y:pnat=>less(s(x),s(y))=less(x,y)in u(1)

QUOTIENT
[]==>x:pnat=>quot(x,0)=0 in pnat
[]==>x:pnat=>y:pnat=>less(x,y)=>quot(x,y)=0 in pnat
[]==>x:pnat=>y:pnat=>quot(plus(x,y),y)=s(quot(x,y)) in pnat

REMAINER
[]==>x:pnat=>rem(x,0)=0 in pnat
[]==>x:pnat=>y:pnat=>less(x,y)=>rem(x,y)=x in pnat
[]==>x:pnat=>y:pnat=>rem(plus(x,y),y)=rem(x,y) in pnat

HALF
[]==>half(0)=0 in pnat
[]==>half(s(0))=0 in pnat
[]==>n:pnat=>half(s(s(n)))=s(half(n))in pnat
```

A.2.3 Lists

```
APPEND
[]==>l:{word}=>app(nil,l)=l in{word}
[]==>h:{bool}=>l1:{word}=>l2:{word}=>
 app(h::l1,l2)=h::app(l1,l2)in{word}

LENGTH
[]==>length(nil)=0 in pnat
[]==>h:{bool}=>l:{word}=>length(h::l)=s(length(l))in pnat

REVERSE
[]==>rev(nil)=nil in {word}
[]==>h:{bool}=>l:{word}=>rev(h::l)=app(rev(l),h::nil)in {word}
```

A.2.4 Words

```
NOT
[]==>not_word(nil)=nil in {word}
[]==>a:{bool}=>x:{word}=>
 not_word(a::x)= (not a)::not_word(x) in {word}

AND
[]==>and_word(nil,nil)=nil in {word}
[]==>a:{bool}=>b:{bool}=>x:{word}=>y:{word}=>
 and_word(a::x,b::y)= and(a,b)::and_word(x,y) in {word}

OR
[]==>or_word(nil,nil)=nil in {word}
[]==>a:{bool}=>b:{bool}=>x:{word}=>y:{word}=>
 or_word(a::x,b::y)= or(a,b)::or_word(x,y) in {word}

XOR
[]==>xor_word(nil,nil)=nil in {word}
[]==>a:{bool}=>b:{bool}=>x:{word}=>y:{word}=>
 xor_word(a::x,b::y)=xor(a,b)::xor_word(x,y) in {word}
```

A.3 Conversion functions

```
BOOL TO NAT
[]==>bool2nat({false})=0 in pnat
[]==>bool2nat({true})=s(0)in pnat

NAT TO BOOL
[]==>nat2bool(0)={false}in{bool}
[]==>n:pnat=>nat2bool(s(n))= not(nat2bool(n)) in {bool}

WORD TO NAT
[]==>word2nat(nil)=0 in pnat
[]==>a:{bool}=>x:{word}=>
 word2nat(a::x)=
 plus(bool2nat(a),times(s(s(0)),word2nat(x)))in pnat

WORD TO NAT (LITTLE ENDIAN)
```

```
[]==>word2nat_le(nil)=0 in pnat
[]==>h:{bool}=>t:{word}=> word2nat_le(h::t)=
 plus(times(bool2nat(h),exp(s(s(0)),length(t))),
      word2nat_le(t))in pnat

WORD TO NAT (EXPLICIT PARAMETER)
[]==>w:{word}=>word2nat_ep(0,w)=0 in pnat
[]==>n:pnat=>w:{word}=> word2nat_ep(s(n),w)=
 plus(bool2nat(hd(w)),times(s(s(0)),
      word2nat_ep(n,tl(w)))) in pnat

NAT TO WORD
[]==>n:pnat=>nat2word(0,n)=nil in{word}
[]==>n:pnat=>m:pnat=>nat2word(s(n),m)=
 nat2bool(m)::nat2word(n,half(m)) in {word}
```

A.4 Conditional functions

```
IF (BOOL-WORD)
[]==>a:{word}=>b:{word}=>if({false},{false},a,b)=a in{word}
[]==>a:{word}=>b:{word}=>if({false},{true},a,b)=b in{word}
[]==>a:{word}=>b:{word}=>if({true},{false},a,b)=b in{word}
[]==>a:{word}=>b:{word}=>if({true},{true},a,b)=a in{word}

IF (BOOL-NAT)
[]==>a:pnat=>b:pnat=>ifbn({false},{false},a,b)=a in pnat
[]==>a:pnat=>b:pnat=>ifbn({false},{true},a,b)=b in pnat
[]==>a:pnat=>b:pnat=>ifbn({true},{false},a,b)=b in pnat
[]==>a:pnat=>b:pnat=>ifbn({true},{true},a,b)=a in pnat

IF (NAT-BOOL)
[]==>x:pnat=>y:pnat=>a:{bool}=>b:{bool}=>x=y in pnat=>
 if_nb(x,y,a,b)=a in{bool}
[]==>x:pnat=>y:pnat=>a:{bool}=>b:{bool}=> (x=y in pnat=>void)=>
 if_nb(x,y,a,b)=b in{bool}

IF (NAT-WORD)
[]==>x:pnat=>y:pnat=>a:{word}=>b:{word}=>
    x=y in pnat=>if_nw(x,y,a,b)=a in {word}
[]==>x:pnat=>y:pnat=>a:{word}=>b:{word}=>
    (x=y in pnat=>void)=>if_nw(x,y,a,b)=b in {word}
```

Appendix B

Non recursive circuits

This appendix describes the implementation, the specification and the conjecture of non-recursive circuits: *half adder*, *full adder*, *1-bit ALU* and *multiplexer*.

B.1 Half adder

```
IMPLEMENTATION

[]==>x:{bool}=>ci:{bool}=>ha_sum(x,ci)=xor(x,ci)in{bool}
[]==>x:{bool}=>ci:{bool}=>ha_carry(x,ci)= (x and ci)in{bool}

CONJECTURE:
[]==>a:{bool}=>b:{bool}=>
word2nat(ha_sum(a,b)::ha_carry(a,b)::nil)=
plus(bool2nat(a),bool2nat(b))in pnat
```

B.2 Full adder

```
IMPLEMENTATION

[]==>a:{bool}=>b:{bool}=>ci:{bool}=>
 fa_sum(a,b,ci)=xor(xor(a,b),ci) in {bool}
[]==>a:{bool}=>b:{bool}=>ci:{bool}=>
 fa_carry(a,b,ci)= (xor(a,b)and ci or a and b) in {bool}

CONJECTURE:
[]==>a:{bool}=>b:{bool}=>ci:{bool}=>
word2nat(fa_sum(a,b,ci)::fa_carry(a,b,ci)::nil)=
plus(bool2nat(a),plus(bool2nat(b),bool2nat(ci)))in pnat
```

B.3 1-bit ALU

```
IMPLEMENTATION

[]==>s0:{bool}=>s1:{bool}=>s2:{bool}=>
 a:{bool}=>b:{bool}=>ci:{bool}=>
  alu_sum(s0,s1,s2,a,b,ci)=
  fa_sum(or(and(and(s2,and(not(s1),
                           not(s0))),
                    b),
                or(and(and(s2,
                           and(s1,
                               not(s0))),
                       not(b)),
                   a)),
              or(and(b,s0),
                 and(not(b),s1)),
              and(ci,not(s2))) in bool}


[]==>s0:{bool}=>s1:{bool}=>s2:{bool}=>
 a:{bool}=>b:{bool}=>ci:{bool}=>
  alu_carry(s0,s1,s2,a,b,ci)=
  and(not(s2),
    fa_carry(or(and(and(s2,and(not(s1),
                               not(s0))),
                        b),
                    or(and(and(s2,
                               and(s1,
                                   not(s0))),
                           not(b)),
                       a)),
                or(and(b,s0),
                  and(not(b),s1)),
                and(ci,not(s2)))) in {bool}

CONJECTURE:
[]==>s0:{bool}=>s1:{bool}=>s2:{bool}=>
 a:{bool}=>b:{bool}=>ci:{bool}=>
word2nat(alu_sum(s0,s1,s2,a,b,ci)::
         alu_carry(s0,s1,s2,a,b,ci)::nil)=
alu_spec(s0,s1,s2,a::nil,b::nil,ci)in pnat
```

B.4 4-1 Multiplexer

```
IMPLEMENTATION

[]==>
 h0:{bool}=>h1:{bool}=>x0:{bool}=>
  x1:{bool}=>x2:{bool}=>x3:{bool}=>
mux(h0,h1,x0,x1,x2,x3)=
      or(and(x0,
             and(not(h1),
                 not(h0))),
```

```
          or(and(x1,
                 and(not(h1),
                     h0)),
             or(and(and(x2,
                        and(h1,
                            not(h0))),
                    and(x3,and(h1,h0))
                   )
               )
             )
          ) ) in {bool}

SPECIFICATION:

[]==>x:{word}=>muxSpec(0) of x=hd(x) in{bool}
[]==>n:pnat=>x:{word}=>
 mux_spec(s(n),x)=mux_spec(n,tl(x))in{bool}

CONJECTURE
[]==>h0:{bool}=>h1:{bool}=>
 x0:{bool}=>x1:{bool}=>x2:{bool}=>x3:{bool}=>
mux_spec(word2nat(h0::h1::nil),x0::x1::x2::x3::nil)=
mux(h0,h1,x0,x1,x2,x3)in{bool}
```

Appendix C

Incrementer

This appendix describes the implementation, the specification, the conjecture, the proof planning script, the proof script, and lists the methods used.

C.1 Formalisation

```
IMPLEMENTATION

[]==>ci:{bool}=>inc(nil,ci)=ci::nil in{word}
[]==>ci:{bool}=>a:{bool}=>x:{word}=>
 inc(a::x,ci)=ha_sum(a,ci)::inc(x,ha_carry(a,ci))in{word}

CONJECTURE
[]==>x:{word}=>ci:{bool}=>
word2nat(inc(x,ci))=plus(word2nat(x),bool2nat(ci))in pnat
```

C.2 Proof plan

Script of the proof planning generated by *Clam* using the depth-first planner:

```
runtime(dplan,T).
induction([v0::v1],[x:{word}])then
[sym_eval(
eval_def([1,2,1],[word2nat1,equ(left)])then
[eval_def([2,1],[plus1,equ(left)])then
[eval_def([1,1,1],[inc1,equ(left)])then
[eval_def([1,1],[word2nat2,equ(left)])then
[eval_def([2,1,1],[times2,equ(left)])then
[eval_def([2,2,1,1],[word2nat1,equ(left)])then
[eval_def([1,2,1,1],[times2,equ(left)])then
[eval_def([2,1,2,1,1],[word2nat1,equ(left)])then
[eval_def([1,1,2,1,1],[times1,equ(left)])then
[eval_def([1,2,1,1],[plus1,equ(left)])then
[eval_def([2,1,1],[plus1,equ(left)])then
```

```
[term_cancel(ci:{bool}=>0=0 in pnat)then
[elementary(intro(new[ci])then
[identity,wfftacs])]]]]]]]]]]]]),
step_case(ripple(wave([1,2,1],[word2nat2,equ(left)],[])then
[wave([1,1,1],[inc2,equ(left)],[])then
[wave([1,1],[word2nat2,equ(left)],[])then
[unblock(eval_def,[2,1,1],[times2,equ(left)])then
[unblock(eval_def,[1,2,1,1],[times2,equ(left)])then
[unblock(eval_def,[1,1,2,1,1],[times1,equ(left)])then
[unblock(eval_def,[1,2,1,1],[plus1,equ(left)])then
[unblock(eval_def,[1,1,1,1],[ha_sum1,equ(left)])then
[unblock(eval_def,[2,1,1,2,1,1],[ha_carry1,equ(left)])then
[unblock(eval_def,[2,1,2,2,1,1],[ha_carry1,equ(left)])then
[unblock(eval_def,[2,1,2,1],[times2,equ(left)])then
[unblock(eval_def,[1,2,1,2,1],[times2,equ(left)])then
[unblock(eval_def,[1,1,2,1,2,1],[times1,equ(left)])then
[unblock(eval_def,[1,2,1,2,1],[plus1,equ(left)])]]]]]]]]]]]]]])then
[fertilize(weak,fertilize([weak_fertilize(left,in,[1,2],v2),
weak_fertilize(left,in,[2,2],v2)]))])] then
sym_eval(
term_cancel(ci:{bool}=>
plus(bool2nat(xor(v0,ci)),
     plus(bool2nat(v0 and ci),bool2nat(v0 and ci)))=
plus(bool2nat(v0),bool2nat(ci))in pnat)then
[bool_cases(v0)then
[eval_def([1,2,1],[bool2nat1,equ(left)])then
[eval_def([2,1],[plus1,equ(left)])then
[eval_def([1,2,2,1,1],[and2,equ(left)])then
[eval_def([2,2,1,1],[bool2nat1,equ(left)])then
[eval_def([1,1,2,1,1],[and2,equ(left)])then
[eval_def([1,2,1,1],[bool2nat1,equ(left)])then
[eval_def([2,1,1],[plus1,equ(left)])then
[bool_cases(ci)then[eval_def([2,1],[bool2nat1,equ(left)])then
[eval_def([1,1,1,1],[xor1,equ(left)])then
[eval_def([1,1,1],[bool2nat1,equ(left)])then
[eval_def([1,1],[plus1,equ(left)])then
[elementary(identity)]]]],
eval_def([2,1],[bool2nat2,equ(left)])then
[eval_def([1,1,1,1],[xor3,equ(left)])then
[eval_def([1,1,1],[bool2nat2,equ(left)])then
[eval_def([1,1],[plus2,equ(left)])then
[reduction([],cnc_s)then
[eval_def([1,1],[plus1,equ(left)])then
[elementary(identity)]]]]]]]]]]]]],
eval_def([1,2,1],[bool2nat2,equ(left)])then
[eval_def([2,1],[plus2,equ(left)])then
[eval_def([1,2,1],[plus1,equ(left)])then
[bool_cases(ci)then
[eval_def([1,2,1],[bool2nat1,equ(left)])then
[eval_def([1,2,2,1,1],[and1,equ(left)])then
[eval_def([2,2,1,1],[bool2nat1,equ(left)])then
[eval_def([1,1,2,1,1],[and1,equ(left)])then
[eval_def([1,2,1,1],[bool2nat1,equ(left)])then
[eval_def([2,1,1],[plus1,equ(left)])then
[eval_def([1,1,1,1],[xor2,equ(left)])then
[eval_def([1,1,1],[bool2nat2,equ(left)])then
[eval_def([1,1],[plus2,equ(left)])then
```

```
[reduction([],cnc_s)then
[eval_def([1,1],[plus1,equ(left)])then
[elementary(identity)]]]]]]]]]]],
eval_def([1,2,1],[bool2nat2,equ(left)])then
[eval_def([1,2,2,1,1],[and3,equu(left)])then
[eval_def([2,2,1,1],[bool2nat2,equ(left)])then
[eval_def([1,1,2,1,1],[and3,equ(left)])then
[eval_def([1,2,1,1],[bool2nat2,equ(left)])then
[eval_def([2,1,1],[plus2,equ(left)])then
[eval_def([1,2,1,1],[plus1,equ(left)])then
[eval_def([1,1,1,1],[xor1,equ(left)])then
[eval_def([1,1,1],[bool2nat1,equ(left)])then
[eval_def([1,1],[plus1,equ(left)])then
[elementary(identity)]]]]]]]]]]]]]]])
T = 21316                                    (milliseconds)
```

C.3 Proof

Script of the proof generated by *Oyster*

```
runtime(apply_plan,T).
applying tactic at depth 0: ind_strat([v0::v1],[x:{word}])
applying tactic at depth 1: sym_eval(...)
T = 105750                                   (milliseconds)
```

C.4 Methods

Methods used in the proof planning

```
list_methods.
    sym_eval/1
    generalise/2
    normalize/1
    ind_strat/1
```

Appendix D

Multiplier

This appendix describes the verification of the multiplier: implementation, specification, conjecture, lemmas, proof plan script, proof script, and methods.

D.1 Formalisation

```
IMPLEMENTATION

[]==>x:{word}=>mult(x,nil)=zeroes(length(x))in{word}
[]==>h:{bool}=>x:{word}=>y:{word}=>
mult(x,h::y)=
adder_le(appnd(mult_one(x,h),zeroes(length(y))),mult(x,y),{false})
in{word}

CONJECTURE
[]==>x:{word}=>y:{word}=>
word2nat_le(mult(x,y))=times(word2nat_le(x),word2nat_le(y))
in pnat
```

D.2 Lemmas

```
VERIFICATION OF N-BIT ADDER
[]==>ci:{bool}=>x:{word}=>y:{word}=>
 word2nat_le(adder_le(x,y,ci))=
 plus(word2nat_le(x),plus(word2nat_le(y),bool2nat(ci)))in pnat

DISTRIBUTIVITY OF MULTIPLICATION OVER ADDITION
[]==>a:pnat=>b:pnat=>c:pnat=>
 times(a,plus(b,c))=plus(times(a,b),times(a,c))in pnat

a:pnat=>b:pnat=>c:pnat=>
 times(plus(b,c),a)=plus(times(b,a),times(c,a))in pnat

ASSOCIATIVITY OF MULTIPLICATION
```

```
[]==>a:pnat=>b:pnat=>c:pnat=>
 times(a,times(b,c))=times(times(a,b),c)in pnat

PLUS ZERO
[]==>x:pnat=>plus(x,0)=x in pnat

TIMES ZERO
[]==>x:pnat=>times(x,0)=0 in pnat

\section{Proof plan}
\begin{verbatim}
(only methods are displayed):

ind_strat([v0::v1],[y:{word}]) then
  [ind_strat([v0::v1],[x:{word}]),
   sym_eval(...) then
     [ind_strat([v4::v5],[x:{word}]) then
        ind_strat([v4::v5],[v1:{word}]),
      ind_strat([v4::v5],[x:{word}]) then
        [ind_strat([v4::v5],[v1:{word}]),
         sym_eval(...) then
           ind_strat([v8::v9],[v5:{word}]) then
             ind_strat([v8::v9],[v1:{word}])
        ]
     ]
  ]
```

D.3 Proof

```
runtime(apply_plan,T).
applying tactic at depth 0: ind_strat([v0::v1],[y:{word}])
applying tactic at depth 1: ind_strat([v0::v1],[x:{word}])
applying tactic at depth 1: sym_eval(...)
applying tactic at depth 2: ind_strat([v4::v5],[x:{word}])
applying tactic at depth 3: ind_strat([v4::v5],[v1:{word}])
applying tactic at depth 2: ind_strat([v4::v5],[x:{word}])
applying tactic at depth 3: ind_strat([v4::v5],[v1:{word}])
applying tactic at depth 3: sym_eval(...)
applying tactic at depth 4: ind_strat([v8::v9],[v5:{word}])
applying tactic at depth 5: ind_strat([v8::v9],[v1:{word}])
T = 334617                                    (milliseconds)
```

D.4 Methods

```
list_methods.

    sym_eval/1
    normalize/1
    ind_strat/1
    generalise/2
```

Appendix E

Gordon computer

This appendix describes the verification of the *Gordon computer*: implementation, specification, conjecture, lemmas, script of proof planning, script of proof, and methods used.

E.1 Formalisation

```
=================================================
IMPLEMENTATION

[]==>
 t:pnat=>
  swt:signaln(16)=>
   knob:signaln(2)=>
    button:flag=>
computerImp(t,swt,knob,button)
=
 (memory(t,swt,knob,button)&
  pc(t,swt,knob,button)&
  acc(t,swt,knob,button)&
  idle(t,swt,knob,button)&
  buffer(t,swt,knob,button)&
  mar(t,swt,knob,button)&
  ir(t,swt,knob,button)&
  arg(t,swt,knob,button)&
  mpc(t,swt,knob,button)&
  ready(t,swt,knob,button))
in imp_state

MEMORY
[]==>
t:pnat=>
 swt:signaln(16)=>
  knob:signaln(2)=>
   button:flag=>
memory(s(t),swt,knob,button)
=
if(memctlB(t,swt,knob,button),{true},
        store(mar(t,swt,knob,button),
```

```
              bus(t,swt,knob,button),
              memory(t,swt,knob,button)),
        memory(t,swt,knob,button))
in memn(16)

PROGRAM COUNTER
[]==>
 t:pnat=>
  swt:signaln(16)=>
   knob:signaln(2)=>
    button:flag=>
pc(s(t),swt,knob,button)=
if(wpc(t,swt,knob,button),{false},
          pc(t,swt,knob,button),
          cut(bus(t,swt,knob,button)))
in wordn(13)

ACCUMULATOR
[]==>
 t:pnat=>
  swt:signaln(16)=>
   knob:signaln(2)=>
    button:flag=>
acc(s(t),swt,knob,button)
=
if(wacc(t,swt,knob,button),{false},
        acc(t,swt,knob,button),
        bus(t,swt,knob,button))
in wordn(16)

IDLE FLAG
[]==>
 t:pnat=>
  swt:signaln(16)=>
   knob:signaln(2)=>
    button:flag=>
idle(t,swt,knob,button)=mc_idle(mpc(t,swt,knob,button)) in {bool}

MICROCODE (IDLE)
[]==>
mc_idle({false}::{false}::{false}::{false}::{false}::nil)={true}
in {bool}
.
.
.
[]==>
mc_idle({true}::{true}::{true}::{true}::{true}::nil)={false}
in {bool}

BUFFER REGISTER
[]==>
 t:pnat=>
  swt:signaln(16)=>
   knob:signaln(2)=>
    button:flag=>
     buffer(s(t),swt,knob,button)=
if(aluctlB(t,microcode(mpc(p,swt,knob,button))),{false},
 if(aluctlA(t,microcode(mpc(p,swt,knob,button))),{false},
         bus(t,swt,knob,button),
         inc_g(bus(t,swt,knob,button))),
```

```
 if(aluctlA(p,microcode(mpc(p,swt,knob,button))),{false},
         add_g(arg(t,swt,knob,button),
       bus(t,swt,knob,button),{false}),
         subt_g(arg(t,swt,knob,button),
              bus(t,swt,knob,button),{false})))
in wordn(16)

MEMORY ADDRESS REGISTER
[]==>
t:pnat=>
 swt:signaln(16)=>
  knob:signaln(2)=>
   button:flag=>
mar(s(t),swt,knob,button)
=
if(wmar(t,swt,knob,button),{false},
        mar(t,swt,knob,button),
        cut(bus(t,swt,knob,button))
       )
in wordn(13)

INSTRUCTION REGISTER
[]==>
 t:pnat=>
  swt:signaln(16)=>
   knob:signaln(2)=>
    button:flag=>
ir(s(t),swt,knob,button)=
if(wir(t,swt,knob,button),{false},
        ir(t,swt,knob,button),
        bus(t,swt,knob,button))
in wordn(16)

ARGUMENT REGISTER
[]==>
 t:pnat=>
  swt:signaln(16)=>
   knob:signaln(2)=>
    button:flag=>
arg(s(t),swt,knob,button)
=
if(warg(t,swt,knob,button),{false},
        arg(t,swt,knob,button),
        bus(t,swt,knob,button))
in wordn(16)

MICROCODE PROGRAM COUNTER
[]==>
t:pnat=>
 swt:signaln(16)=>
  knob:signaln(2)=>
   button:flag=>
mpc(s(t),swt,knob,button)=
if(testC(t,swt,knob,button),{false},
 if(testB(t,swt,knob,button),{false},
  if(testA(t,swt,knob,button),{false},
   address_a(t,swt,knob,button),
   if(button of t,{false},
    address_a(t,swt,knob,button),
    address_b(t,swt,knob,button))),
  if(testA(t,swt,knob,button),{false},
```

```
    pnatEq(word2nat_le(acc(t,swt,knob,button)),0,
     address_b(t,swt,knob,button),
     address_a(t,swt,knob,button)),
    tl(adder_le({false}::{false}::{false}::knob of t,
             address_a(t,swt,knob,button),
             {false}))
            )),
 if(testB(t,swt,knob,button),{false},
  if(testA(t,swt,knob,button),{false},
   tl(adder_le({false}::{false}::opcode,
           address_a(t,swt,knob,button),
           {false})),
   address_a(t,swt,knob,button)),
  address_a(t,swt,knob,button)))
in wordn(5)

READY FLAG
[]==>
 t:pnat=>
  swt:signaln(16)=>
   knob:signaln(2)=>
    button:flag=>
ready(t,swt,knob,button)=mc_ready(mpc(t,swt,knob,button))
in {bool}

==================================================
SPECIFICATION OF THE GORDON COMPUTER

SEMANTICS OF INSTRUCTIONS
% HALT idle=false,button=false,knob=X,opc=000
[]==>
 u:pnat=>
  swt:signaln(16)=>
   knob:signaln(2)=>
    button:flag=>
     (idle(microtime(u,swt,knob,button),swt,knob,button)={false}
        in {bool}#
      button of microtime(u,swt,knob,button)={false} in {bool}#
      opcode=({false}::{false}::{false}::nil)
      in wordn(3))=>
computer(s(u),swt,knob,button) =
 execute_halt(computer(u,swt,knob,button))
in state

% JUMP id=false,button=false,knob=X,opc=001,ac=X
[]==>
 u:pnat=>
  swt:signaln(16)=>
   knob:signaln(2)=>
    button:flag=>
     (idle(microtime(u,swt,knob,button),swt,knob,button)={false}
       in {bool}#
      button of microtime(u,swt,knob,button)={false} in {bool}#
      opcode=({false}::{false}::{true}::nil)
      in wordn(3))=>
computer(s(u),swt,knob,button) =
 execute_jump(computer(u,swt,knob,button))
in state

% JZR jump if zero
% id=false,button=false,knob=X,opc=010,ac=zeroes
```

```
[]==>
 u:pnat=>
  swt:signaln(16)=>
   knob:signaln(2)=>
    button:flag=>
     (idle(microtime(u,swt,knob,button),swt,knob,button)={false}
       in {bool}#
      button of microtime(u,swt,knob,button)={false} in {bool}#
      opcode=({false}::{true}::{false}::nil)
      in wordn(3)#
      word2nat_le(
       acc(microtime(u,swt,knob,button),swt,knob,button))=0
        in pnat)=>
computer(s(u),swt,knob,button) =
 execute_jzr(computer(u,swt,knob,button))
in state

% JZR jump if non-zero
% id=false,button=false,knob=X,opc=010,ac=no zeroes
[]==>
 u:pnat=>
  swt:signaln(16)=>
   knob:signaln(2)=>
    button:flag=>
     (idle(microtime(u,swt,knob,button),swt,knob,button)={false}
       in {bool}#
      button of microtime(u,swt,knob,button)={false} in {bool}#
      opcode=({false}::{true}::{false}::nil)
      in wordn(3)#
      word2nat(
       acc(microtime(u,swt,knob,button),swt,knob,button))=0
        in pnat=>void)=>
computer(s(u),swt,knob,button) =
 execute_jnozr(computer(u,swt,knob,button))
in state

% ADD id=false,button=false,knob=X,opc=011,ac=X
[]==>
u:pnat=>
  swt:signaln(16)=>
   knob:signaln(2)=>
    button:flag=>
     (idle(microtime(u,swt,knob,button),swt,knob,button)={false}
       in {bool}#
      button of microtime(u,swt,knob,button)={false} in {bool}#
      opcode=({false}::{true}::{true}::nil)
      in wordn(3))=>
computer(s(u),swt,knob,button) =
 execute_add(computer(u,swt,knob,button))
in state

% SUBTRACT id=false,button=false,knob=X,opc=100,ac=X
[]==>
u:pnat=>
 swt:signaln(16)=>
   knob:signaln(2)=>
    button:flag=>
     (idle(microtime(u,swt,knob,button),swt,knob,button)={false}
        in {bool}#
      button of microtime(u,swt,knob,button)={false} in {bool}#
```

```
      opcode=({true}::{false}::{false}::nil)
      in wordn(3))=>
computer(s(u),swt,knob,button) =
 execute_sub(computer(u,swt,knob,button))
in state

% LOAD id=false,button=false,knob=X,opc=101,ac=X
[]==>
u:pnat=>
  swt:signaln(16)=>
   knob:signaln(2)=>
    button:flag=>
     (idle(microtime(u,swt,knob,button),swt,knob,button)={false}
        in {bool}#
      button of microtime(u,swt,knob,button)={false} in {bool}#
      opcode=({true}::{false}::{true}::nil)
      in wordn(3))=>
computer(s(u),swt,knob,button) =
execute_load(computer(u,swt,knob,button))
in state

% STORE id=false,button=false,knob=X,opc=110,ac=X
[]==>
u:pnat=>
  swt:signaln(16)=>
   knob:signaln(2)=>
    button:flag=>
     (idle(microtime(u,swt,knob,button),swt,knob,button)={false}
        in {bool}#
      button of microtime(u,swt,knob,button)={false} in {bool}#
      opcode=({true}::{true}::{false}::nil)
      in wordn(3))=>
computer(s(u),swt,knob,button) =
 execute_store(computer(u,swt,knob,button))
in state

% SKIP id=false,button=false,knob=X,opc=111,ac=X
[]==>
u:pnat=>
  swt:signaln(16)=>
   knob:signaln(2)=>
    button:flag=>
     (idle(microtime(u,swt,knob,button),swt,knob,button)={false}
         in {bool}#
      button of microtime(u,swt,knob,button)={false} in {bool}#
      opcode=({true}::{true}::{true}::nil)
      in wordn(3))=>
computer(s(u),swt,knob,button) =
 execute_skip(computer(u,swt,knob,button))
in state

% STOP EXECUTE-1 WHEN IS RUNNING
% id=false,button=true,knob=X,opc=X,ac=X
[]==>
u:pnat=>
  swt:signaln(16)=>
   knob:signaln(2)=>
    button:flag=>
     (idle(microtime(u,swt,knob,button),swt,knob,button)={false}
        in {bool}#
```

```
      button of microtime(u,swt,knob,button)={true} in {bool})=>
computer(s(u),swt,knob,button) =
 stop_execute(computer(u,swt,knob,button))
in state

% STOP EXECUTE-2 WHEN IT'S IDLEING
% id=true,button=false,knob=X,opc=X,ac=X
[]==>
u:pnat=>
  swt:signaln(16)=>
   knob:signaln(2)=>
    button:flag=>
     (idle(microtime(u,swt,knob,button),swt,knob,button)={true}
          in {bool}#
      button of microtime(u,swt,knob,button)={false} in {bool})=>
computer(s(u),swt,knob,button) =
stop_operation(computer(u,swt,knob,button))
in state

% LOAD PC id=true,button=true,knob=00,opc=X,ac=X
[]==>
u:pnat=>
  swt:signaln(16)=>
   knob:signaln(2)=>
    button:flag=>
     (idle(microtime(u,swt,knob,button),swt,knob,button)={true}
        in {bool}#
      button of microtime(u,swt,knob,button)={true} in {bool}#
      knob of microtime(u,swt,knob,button)={false}::{false}::nil
        in wordn(2))=>
computer(s(u),swt,knob,button) =
 load_pc(
  computer(u,swt,knob,button),swt of microtime(u,swt,knob,button))
in state

% LOAD ACC id=true,button=true,knob=01,opc=X,ac=X
[]==>
u:pnat=>
  swt:signaln(16)=>
   knob:signaln(2)=>
    button:flag=>
     (idle(microtime(u,swt,knob,button),swt,knob,button)={true}
        in {bool}#
      button of microtime(u,swt,knob,button)={true} in {bool}#
      knob of microtime(u,swt,knob,button)={false}::{true}::nil
        in wordn(2))=>
computer(s(u),swt,knob,button) =
 load_acc(
  computer(u,swt,knob,button),swt of microtime(u,swt,knob,button))
in state

% LOAD MEM id=true,button=true,knob=10,opc=X,ac=X
[]==>
u:pnat=>
  swt:signaln(16)=>
   knob:signaln(2)=>
    button:flag=>
     (idle(microtime(u,swt,knob,button),swt,knob,button)={true}
       in {bool}#
      button of microtime(u,swt,knob,button)={true} in {bool}#
```

```
      knob of microtime(u,swt,knob,button)={true}::{false}::nil
       in wordn(2))=>
computer(s(u),swt,knob,button) =
 load_mem(computer(u,swt,knob,button))
in state

% START EXECUTE id=true,button=true,knob=11,opc=X,ac=X
[]==>
u:pnat=>
  swt:signaln(16)=>
   knob:signaln(2)=>
    button:flag=>
     (idle(microtime(u,swt,knob,button),swt,knob,button)={true}
        in {bool}#
      button of microtime(u,swt,knob,button)={true} in {bool}#
      knob of microtime(u,swt,knob,button)={true}::{true}::nil
        in wordn(2))=>
computer(s(u),swt,knob,button) =
 start_execute(computer(u,swt,knob,button))
in state

AUXILIARY FUNCTIONS (definition and rewrite equation):

EXECUTE HALT
[]==>
 x:state=>
execute_halt(x)=fst(x)&snd(x)&trd(x)&{true}
in state

EXECUTE JUMP
[]==>
 x:state=>
execute_jump(x)=
 fst(x)&cut(fetch(snd(x),fst(x)))&trd(x)&{false}
in state

EXECUTE JUMP ON ZERO
[]==>
 x:state=>
  execute_jzr(x)=
   (fst(x)&cut(fetch(snd(x),fst(x)))&trd(x)&{false})
in state

EXECUTE JUMP IF NO ZERO
[]==>
 x:state=>
  execute_jnozr(x)=
   (fst(x)&inc_g(snd(x))&trd(x)&{false})
in state

EXECUTE ADD
[]==>
 x:state=>
  execute_add(x)=(fst(x)&inc_g(snd(x))&
   add_g(trd(x),fetch(cut(fetch(snd(x),fst(x))),fst(x)),{false})&
   {false})
in state

EXECUTE SUBTRACT
[]==>
 x:state=>
```

```
  execute_sub(x)=(fst(x)&inc_g(snd(x))&
   subt_g(trd(x),fetch(cut(fetch(snd(x),fst(x))),fst(x)),{false})&
   {false})
in state

EXECUTE LOAD
[]==>
 x:state=>
  execute_load(x)=(fst(x)&inc_g(snd(x))&
     fetch(cut(fetch(snd(x),fst(x))),fst(x))&{false})
in state

EXECUTE STORE
[]==>
 x:state=>
  execute_store(x)=
   (store(cut(fetch(snd(x),fst(x))),trd(x),fst(x))&
     inc_g(snd(x))&trd(x)&{false})
in state

EXECUTE SKIP
[]==>
 x:state=>
  execute_skip(x)=(fst(x)&inc_g(snd(x))&trd(x)&{false})
in state

STOP EXECUTION
[]==>
  x:state=>
   stop_execute(x)=(fst(x)&snd(x)&trd(x)&{true})
in state

STOP OPERATION
[]==>
  x:state=>
   stop_operation(x)=(fst(x)&snd(x)&trd(x)&{true})
in state

LOAD PROGRAM COUNTER
[]==>
  x:state=>
   swt:wordn(16)=>
    load_pc(x,swt)=(fst(x)&cut(swt)&trd(x)&{true}) in state

LOAD ACCUMULATOR
[]==>
  x:state=>
   swt:wordn(16)=>
    load_acc(x,swt)=(fst(x)&snd(x)&swt&{true}) in state

LOAD MEMORY
[]==>
 x:state=>
   load_mem(x)=
(store(snd(x),trd(x),fst(x))& snd(x)& trd(x)& {true})in state

START EXECUTE
[]==>
  x:state=>
   start_execute(x)=(fst(x)&snd(x)&trd(x)&{false})
in state
```

```
TIME ABSTRACTION
[]==>
 u:pnat=>
  swt:signaln(16)=>
   knob:signaln(2)=>
    button:flag=>
microtime(s(u),swt,knob,button)=iterate_time(u,swt,knob,button)
in pnat

iterate_time(u,swt,knob,button)<==>
 term_of(iterate_time)of u of swt of
knob of button.

[]==>
 u:pnat=>
  swt:signaln(16)=>
   knob:signaln(2)=>
    button:flag=>
iterate_time(u,swt,knob,button)=
next_time(microtime(u,swt,knob,button),swt,knob,button)
in pnat

next_time(t,swt,knob,button)<==>
 term_of(next_time)of t of swt of knob of button.

[]==>
 t:pnat=>
  swt:signaln(16)=>
   knob:signaln(2)=>
    button:flag=>
next_time(t,swt,knob,button)=
if(ready(s(t),swt,knob,button),{false},
        next_time(s(t),swt,knob,button),
        s(t)
       )
in pnat

=====================================================
LEMMAS

STABILITY:
stable(signal,t1,t2)<==>
 p:pnat=>
    (less(t1,p)#
      less(p,t2))=>
        signal of p=signal of t1 in {word}.

[]==>
 n:pnat=>t1:pnat=>t2:pnat=>
  sig:signaln(n)=>
   stable(sig,t1,t2)=>
    t:pnat=>
     (less(t1,t)#less(t,t2))=>
      sig of s(t)=sig of t in{word}

13-16 BITS
[]==>t:pnat=>swt:signaln(16)=>knob:signaln(2)=>button:flag=>
tl(tl(tl(
inc_g({false}::{false}::{false}::pc(t,swt,knob,button)))))=
inc_g(pc(t,swt,knob,button))in wordn(13)
```

```
================================================
CONJECTURE (difference match version)
[]==>
 u:pnat=>
  swt:signaln(16)=>
   knob:signaln(2)=>
    button:flag=>
    stable(swt,microtime(u,swt,knob,button),
       microtime(s(u),swt,knob,button))=>
    stable(knob,microtime(u,swt,knob,button),
       microtime(s(u),swt,knob,button))=>
       mpc(microtime(u,swt,knob,button),swt,knob,button)=
        {false}::{false}::{false}::{false}::{false}::nil
in wordn(5)=>

abs_imp(
 computerImp(microtime(u,swt,knob,button),swt,knob,button))=
computer(u,swt,knob,button) in state
=>
abs_imp(
 computerImp(microtime(s(u),swt,knob,button),swt,knob,button))=
computer(s(u),swt,knob,button) in state
```

E.2 Proof plan

This is the proof plan for the difference match version of the conjecture. The structure of the methods applied is displayed. The method `sym_eval` and the submethod `eval_def` produce a large number of rewrite steps, thus they are shown unfolded.

```
diff_match(
 abs_imp(
  computerImp(
   microtime(``s({u})''<out>,swt,knob,button),swt,knob,button))=
computer(``s({u})''<out>,swt,knob,button))then
 [step_case(
  ripple(direction_out,strong,
   casesplit(
[idle(microtime(u,swt,knob,button),swt,knob,button)={true}
in{bool}#
button of microtime(u,swt,knob,button)={true}
in{bool}#
knob of microtime(u,swt,knob,button)={true}::{true}::nil
in wordn(2),
idle(microtime(u,swt,knob,button),swt,knob,button)={true}
in{bool}#
button of microtime(u,swt,knob,button)={true}in{bool}#
knob of microtime(u,swt,knob,button)={true}::{false}::nil
in wordn(2),
idle(microtime(u,swt,knob,button),swt,knob,button)={true}
in{bool}#
button of microtime(u,swt,knob,button)={true}in{bool}#
knob of microtime(u,swt,knob,button)={false}::{true}::nil
in wordn(2),
idle(microtime(u,swt,knob,button),swt,knob,button)={true}
in{bool}#
```

```
button of microtime(u,swt,knob,button)={true}in{bool}#
knob of microtime(u,swt,knob,button)={false}::{false}::nil
in wordn(2),
idle(microtime(u,swt,knob,button),swt,knob,button)={true}
in{bool}#
button of microtime(u,swt,knob,button)={false}in{bool},
idle(microtime(u,swt,knob,button),swt,knob,button)={false}
in{bool}#
button of microtime(u,swt,knob,button)={true}in{bool},
idle(microtime(u,swt,knob,button),swt,knob,button)={false}
in{bool}#
button of microtime(u,swt,knob,button)={false}in{bool
}#opcode={true}::{true}::{true}::nil in wordn(3),
idle(microtime(u,swt,knob,button),swt,knob,button)={false}
in{bool}#
button of microtime(u,swt,knob,button)={false}in{bool}#
opcode={true}::{true}::{false}::nil in wordn(3),
idle(microtime(u,swt,knob,button),swt,knob,button)={false}
in{bool}#
button of microtime(u,swt,knob,button)={false}in{bool}#
opcode={true}::{false}::{true}::nil in wordn(3),
idle(microtime(u,swt,knob,button),swt,knob,button)={false}
in{bool}#
button of microtime(u,swt,knob,button)={false}in{bool}#
opcode={true}::{false}::{false}::nil in wordn(3),
idle(microtime(u,swt,knob,button),swt,knob,button)={false}
in{bool}#
button of microtime(u,swt,knob,button)={false}in{bool}#
opcode={false}::{true}::{true}::nil in wordn(3),
idle(microtime(u,swt,knob,button),swt,knob,button)={false}
in{bool}#
button of microtime(u,swt,knob,button)={false}in{bool}#
opcode={false}::{true}::{false}::nil in wordn(3)#
word2nat_le(
acc(microtime(u,swt,knob,button),swt,knob,button))=0
in pnat=>void,
idle(microtime(u,swt,knob,button),swt,knob,button)={false}
in{bool}#
button of microtime(u,swt,knob,button)={false}in{bool}#
opcode={false}::{true}::{false}::nil in wordn(3)#
word2nat_le(acc(microtime(u,swt,knob,button),swt,knob,button))=0
in pnat,
idle(microtime(u,swt,knob,button),swt,knob,button)={false}
in{bool}#
button of microtime(u,swt,knob,button)={false}in{bool}#
opcode={false}::{false}::{true}::nil in wordn(3),
idle(microtime(u,swt,knob,button),swt,knob,button)={false}
in{bool}#
button of microtime(u,swt,knob,button)={false}in{bool}#
opcode={false}::{false}::{false}::nil in wordn(3)])then
        [wave(direction_out,[2,1],[computer15,equ(left)],[]) then
           unblock(eval_def,[2,1],[start_execute1,equ(left)]),
         wave(direction_out,[2,1],[computer14,equ(left)],[]) then
           unblock(eval_def,[2,1],[load_mem1,equ(left)]),
         wave(direction_out,[2,1],[computer13,equ(left)],[]) then
           unblock(eval_def,[2,1],[load_acc1,equ(left)]),
         wave(direction_out,[2,1],[computer12,equ(left)],[]) then
           unblock(eval_def,[2,1],[load_pc1,equ(left)]) then
             unblock(eval_def,[1,2,2,1],[cut1,equ(left)]),
         wave(direction_out,[2,1],[computer11,equ(left)],[]) then
```

```
          unblock(eval_def,[2,1],[stop_operation1,equ(left)]),
        wave(direction_out,[2,1],[computer10,equ(left)],[]) then
          unblock(eval_def,[2,1],[stop_execute1,equ(left)]),
        wave(direction_out,[2,1],[computer9,equ(left)],[]) then
          unblock(eval_def,[2,1],[execute_skip1,equ(left)]),
        wave(direction_out,[2,1],[computer8,equ(left)],[]) then
          unblock(eval_def,[2,1],[execute_store1,equ(left)]) then
            unblock(eval_def,[1,1,2,1],[cut1,equ(left)]),
        wave(direction_out,[2,1],[computer7,equ(left)],[]) then
          unblock(eval_def,[2,1],[execute_load1,equ(left)]) then
            unblock(eval_def,[1,1,2,2,2,1],[cut1,equ(left)]),
        wave(direction_out,[2,1],[computer6,equ(left)],[]) then
          unblock(eval_def,[2,1],[execute_sub1,equ(left)]) then
            unblock(eval_def,[1,2,1,2,2,2,1],[cut1,equ(left)]),
        wave(direction_out,[2,1],[computer5,equ(left)],[]) then
          unblock(eval_def,[2,1],[execute_add1,equ(left)]) then
            unblock(eval_def,[1,2,1,2,2,2,1],[cut1,equ(left)]),
        wave(direction_out,[2,1],[computer4,equ(left)],[]) then
          unblock(eval_def,[2,1],[execute_jnozr1,equ(left)]),
        wave(direction_out,[2,1],[computer3,equ(left)],[]) then
          unblock(eval_def,[2,1],[execute_jzr1,equ(left)]) then
            unblock(eval_def,[1,2,2,1],[cut1,equ(left)]),
        wave(direction_out,[2,1],[computer2,equ(left)],[]) then
          unblock(eval_def,[2,1],[execute_jump1,equ(left)]) then
            unblock(eval_def,[1,2,2,1],[cut1,equ(left)]),
        wave(direction_out,[2,1],[computer1,equ(left)],[]) then
          unblock(eval_def,[2,1],[execute_halt1,equ(left)])
       ]) then
      [fertilize(weak,[...]),
       fertilize(weak,[...]),
       fertilize(weak,[...]),
       fertilize(weak,[...]),
       fertilize(weak,[...]),
       fertilize(weak,[...]),
       fertilize(weak,[...]),
       fertilize(weak,[...]),
       fertilize(weak,[...]),
       fertilize(weak,[...]),
       fertilize(weak,[...]),
       fertilize(weak,[...]),
       fertilize(weak,[...]),
       fertilize(weak,[...]),
       fertilize(weak,[...])
      ]) then
[sym_eval(...),
sym_eval(...),
sym_eval(...),
sym_eval(...),
sym_eval(...),
change_mpc(v3) then
[sym_eval(...)],
change_mpc(v3)then
[sym_eval(...)then
[normalize([])then
[sym_eval(...),
apply_lemma(inc_pc),
sym_eval(...),
sym_eval(...)]]],
```

```
change_mpc(v3)then
[sym_eval(...)then
[generalise(
tl(tl(tl(
 fetch(pc(microtime(u,swt,knob,button),swt,knob,button),
       memory(microtime(u,swt,knob,button),swt,knob,button))))),
       v43:{word})then
[normalize([])then
[sym_eval(...),
apply_lemma(inc_pc),
sym_eval(...),sym_eval(...)]]]],
change_mpc(v3)then
[sym_eval(...)then
[generalise(
tl(tl(tl(
 fetch(pc(microtime(u,swt,knob,button),swt,knob,button),
       memory(microtime(u,swt,knob,button),swt,knob,button))))),
       v38:{word})then
[normalize([])then
[sym_eval(...),
apply_lemma(inc_pc),
sym_eval(...),
sym_eval(...)]]]],
change_mpc(v3)then
[sym_eval(...)then
[generalise(
tl(tl(tl(
fetch(pc(microtime(u,swt,knob,button),swt,knob,button),
      memory(microtime(u,swt,knob,button),swt,knob,button))))),
      v52:{word})then
[normalize([])then
[sym_eval(...),
apply_lemma(inc_pc),
sym_eval(...),
sym_eval(...)]]]],
change_mpc(v3)then
[sym_eval(...)then
[generalise(
tl(tl(tl(
 fetch(pc(microtime(u,swt,knob,button),swt,knob,button),
       memory(microtime(u,swt,knob,button),swt,knob,button))))),
       v52:{word})then[normalize([])then
[sym_eval(...),
apply_lemma(inc_pc),
sym_eval(...),sym_eval(...)]]]],
change_mpc(v3)then
[sym_eval(...)then
[normalize([])then
[sym_eval(...),
apply_lemma(inc_pc),
sym_eval(...),sym_eval(...)]]],
change_mpc(v3)then
[sym_eval(...)],
change_mpc(v3)then
[sym_eval(...)],
change_mpc(v3)then
[sym_eval(...)]]]
```

E.3 Proof

This script corresponds to the execution of the tactic defined by the proof plan. The parameters of the tactics are not unfolded:

```
| ?- runtime(apply_plan,T).
applying tactic at depth 0: diff_match(...)
applying tactic at depth 1: step_case([...])
applying tactic at depth 2: sym_eval(...)
applying tactic at depth 2: sym_eval(...)
applying tactic at depth 2: sym_eval(...)
applying tactic at depth 2: sym_eval(...)
applying tactic at depth 2: sym_eval(...)
applying tactic at depth 2: change_mpc(...)
applying tactic at depth 3: sym_eval(...)
applying tactic at depth 2: change_mpc(...)
applying tactic at depth 3: sym_eval(...)
applying tactic at depth 4: normalize([])
applying tactic at depth 5: elementary(identity)
applying tactic at depth 5: generalise(...)
applying tactic at depth 6: apply_lemma(inc_pc)
applying tactic at depth 5: elementary(identity)
applying tactic at depth 5: elementary(identity)
applying tactic at depth 2: change_mpc(...)
applying tactic at depth 3: sym_eval(...)
applying tactic at depth 4: normalize([])
applying tactic at depth 5: elementary(identity)
applying tactic at depth 5: generalise(...)
applying tactic at depth 6: apply_lemma(inc_pc)
applying tactic at depth 5: elementary(identity)
applying tactic at depth 5: elementary(identity)
applying tactic at depth 2: change_mpc(...)
applying tactic at depth 3: sym_eval(...)
applying tactic at depth 4: normalize([])
applying tactic at depth 5: elementary(identity)
applying tactic at depth 5: generalise(...)
applying tactic at depth 6: apply_lemma(inc_pc)
applying tactic at depth 5: elementary(identity)
applying tactic at depth 5: elementary(identity)
applying tactic at depth 2: change_mpc(...)
applying tactic at depth 3: sym_eval(...)
applying tactic at depth 4: normalize([])
applying tactic at depth 5: elementary(identity)
applying tactic at depth 5: generalise(...)
applying tactic at depth 6: apply_lemma(inc_pc)
applying tactic at depth 5: elementary(identity)
applying tactic at depth 5: elementary(identity)
applying tactic at depth 2: change_mpc(...)
applying tactic at depth 3: sym_eval(...)
applying tactic at depth 4: normalize([])
applying tactic at depth 5: elementary(identity)
applying tactic at depth 5: generalise(...)
applying tactic at depth 6: apply_lemma(inc_pc)
```

```
applying tactic at depth 5: elementary(identity)
applying tactic at depth 5: elementary(identity)
applying tactic at depth 2: change_mpc(...)
applying tactic at depth 3: sym_eval(...)
applying tactic at depth 4: normalize([])
applying tactic at depth 5: elementary(identity)
applying tactic at depth 5: generalise(...)
applying tactic at depth 6: apply_lemma(inc_pc)
applying tactic at depth 5: elementary(identity)
applying tactic at depth 5: elementary(identity)
applying tactic at depth 2: change_mpc(...)
applying tactic at depth 3: sym_eval(...)
applying tactic at depth 2: change_mpc(...)
applying tactic at depth 3: sym_eval(...)
applying tactic at depth 2: change_mpc(...)
applying tactic at depth 3: sym_eval(...)
T = 37402334                                (milliseconds)
```

E.4 Methods

```
list_methods.
```

```
    diff_match/1
    elementary/1
    step_case/1
    change_mpc/2
    sym_eval/1
    normalize/1
    generalise/2
    apply_lemma/1
    ind_strat/1
```

Appendix F

Other circuits

This appendix describes the formalisation of other circuits: adder (explicit parameter), adder (big endian), look-ahead-carry adder, ALU (explicit parameter), ALU (big endian), shifter, processing unit, and other arithmetic circuits.

F.1 Adder

F.1.1 Explicit parameter

```
IMPLEMENTATION:
[]==>x:{word}=>y:{word}=>ci:{bool}=>
 adder_ep(0,x,y,ci)=ci::nil in{word}

[]==>n:pnat=>x:{word}=>y:{word}=>ci:{bool}=>
 adder_ep(s(n),x,y,ci)=
 fa_sum(hd(x),hd(y),ci)::
  adder_ep(n,tl(x),tl(y),fa_carry(hd(x),hd(y),ci))in{word}

SPECIFICATION:
[]==>n:pnat=>x:{word}=>y:{word}=>ci:{bool}=>
 adder_ep_spec(n,x,y,ci)=
  plus(word2nat_ep(n,x),plus(word2nat_ep(n,y),bool2nat(ci)))
in pnat

CONJECTURE
[]==>n:pnat=>ci:{bool}=>x:{word}=>y:{word}=>
 word2nat_ep(s(n),(adder_ep(n,x,y,ci))) =
 adder_ep_spec(n,x,y,ci) in pnat
```

F.1.2 Big endian

```
IMPLEMENTATION
[]==>ci:{bool}=>adder(nil,nil,ci)=ci::nil in {word}
[]==>ci:{bool}=>a:{bool}=>b:{bool}=>x:{word}=>y:{word}=>
```

```
 adder(a::x,b::y,ci)=
  fa_sum(a,b,ci)::adder(x,y,fa_carry(a,b,ci)) in{word}

CONJECTURE
[]==>x:{word}=>y:{word}=>ci:{bool}=>
 length(x)=length(y) in pnat=>
  word2nat((adder(x,y,ci)))=
  plus(word2nat(x),plus(word2nat(y),bool2nat(ci))) in pnat
```

F.2 ALU

F.2.1 Explicit parameter

```
IMPLEMENTATION:
[]==>s0:{bool}=>s1:{bool}=>s2:{bool}=>
 x:{word}=>y:{word}=>ci:{bool}=>
alu_ep(0,s0,s1,s2,x,y,ci)= ((not s2)and ci)::nil in{word}

[]==>n:pnat=>s0:{bool}=>s1:{bool}=>s2:{bool}=>
x:{word}=>y:{word}=>ci:{bool}=>
alu_ep(s(n),s0,s1,s2,x,y,ci)=
alu_sum(s0,s1,s2,hd(x),hd(y),ci)::alu_ep(n,s0,s1,s2,tl(x),tl(y),
 alu_carry(s0,s1,s2,hd(x),hd(y),ci))in{word}

SPECIFICATION:
[]==>n:pnat=>s0:{bool}=>s1:{bool}=>s2:{bool}=>
 x:{word}=>y:{word}=>ci:{bool}=>
alu_ep_spec(n,s0,s1,s2,x,y,ci)=
if(s2,{false},
 if(s1,{false},
  if(s0,{false},
   adder_ep(n,x,nat2word(n,0),ci),
   adder_ep(n,x,y,ci)),
  if(s0,{false},
   adder_ep(n,x,not_ep_word(n,y),ci),
   adder_ep(n,x,not_ep_word(n,nat2word(n,0)),ci))),
 if(s1,{false},
  if(s0,{false},
   app(or_ep_word(n,x,y),{false}::nil),
   app(xor_ep_word(n,x,y),{false}::nil)),
  if(s0,{false},
   app(and_ep_word(n,x,y),{false}::nil),
   app(not_ep_word(n,x),{false}::nil))))in{word}

CONJECTURE:
[]==>n:pnat=>s0:{bool}=>s1:{bool}=>s2:{bool}=>ci:{bool}=>
 x:{word}=>y:{word}=>
alu_ep_spec(n,s0,s1,s2,x,y,ci)=alu_ep(n,s0,s1,s2,x,y,ci)in{word}
```

F.2.2 Big endian

```
IMPLEMENTATION
[]==>s0:{bool}=>s1:{bool}=>s2:{bool}=>ci:{bool}=>
alu(s0,s1,s2,nil,nil,ci)=
((not s2)and ci)::nil in{word}

[]==>s0:{bool}=>s1:{bool}=>s2:{bool}=>a:{bool}=>b:{bool}=>
x:{word}=>y:{word}=>ci:{bool}=>
alu(s0,s1,s2,a::x,b::y,ci)=
alu_sum(s0,s1,s2,a,b,ci)::
 alu(s0,s1,s2,x,y,alu_carry(s0,s1,s2,a,b,ci))in{word}

SPECIFICATION
[]==>s0:{bool}=>s1:{bool}=>s2:{bool}=>
 a:{word}=>b:{word}=>ci:{bool}=>
alu_spec(s0,s1,s2,a,b,ci)=
if(s2,{false},
 if(s1,{false},
  if(s0,{false},
   adder(a,nat2word(length(a),0),ci),
   adder(a,b,ci)),
  if(s0,{false},
   adder(a,not_word(b),ci),
   adder(a,not_word(nat2word(length(a),0)),ci))),
 if(s1,{false},
  if(s0,{false},
   app(or_word(a,b),{false}::nil),
   app(xor_word(a,b),{false}::nil)),
  if(s0,{false},
   app(and_word(a,b),{false}::nil),
   app(not_word(a),{false}::nil))))
in {word}

CONJECTURE
[]==>a:{word}=>b:{word}=>
s2:{bool}=>s1:{bool}=>s0:{bool}=>ci:{bool}=>
length(a)=length(b) in pnat=>
alu_spec(s0,s1,s2,a,b,ci)=alu(s0,s1,s2,a,b,ci) in{word}
```

F.3 Shifter

```
IMPLEMENTATION:
[]==>h0:{bool}=>h1:{bool}=>ir:{bool}=>il:{bool}=>
shifter(h0,h1,ir,il,nil)=nil in{word}

[]==>h0:{bool}=>h1:{bool}=>
 ir:{bool}=>il:{bool}=>a:{bool}=>x:{word}=>
shifter(h0,h1,ir,il,a::x)=
mux(h0,h1,a,ir,hd_or_il(x,il),{false})::shifter(h0,h1,a,il,x)
in{word}

SPECIFICATION:
[]==>h0:{bool}=>h1:{bool}=>ir:{bool}=>il:{bool}=>x:{word}=>
shifter_spec(h0,h1,ir,il,x)=
if(h1,{false},
 if(h0,{false},
  x,
  shift_right(x,ir)),
```

```
 if(h0,{false},
  shift_left(x,il),
  nat2word(length(x),0)))in{word}

shift_right(x,ir)<==>term_of(shift_right)of x of ir.
[]==>ir:{bool}=>shift_right(nil,ir)=nil in{word}
[]==>x:{word}=>a:{bool}=>ir:{bool}=>
 shift_right(a::x,ir)=ir::shift_right(x,a)in{word}

shift_left(x,il)<==>term_of(shift_left)of x of il.
[]==>il:{bool}=>shift_left(nil,il)=nil in{word}
[]==>x:{word}=>a:{bool}=>il:{bool}=>
 shift_left(a::x,il)=app(x,il::nil)in{word}

CONJECTURE:
[]==>x:{word}=>h0:{bool}=>h1:{bool}=>ir:{bool}=>il:{bool}=>
shifter_spec(h0,h1,ir,il,x)=shifter(h0,h1,ir,il,x)in{word}
```

F.4 Processing unit

```
IMPLEMENTATION:
[]==>s0:{bool}=>s1:{bool}=>s2:{bool}=>
 h0:{bool}=>h1:{bool}=>ir:{bool}=>il:{bool}=>
  x:{word}=>y:{word}=>ci:{bool}=>
processor(s0,s1,s2,h0,h1,ir,il,x,y,ci)=
shifter(h0,h1,ir,il,alu(s0,s1,s2,x,y,ci)) in {word}

SPECIFICATION:
[]==>s0:{bool}=>s1:{bool}=>s2:{bool}=>
 h0:{bool}=>h1:{bool}=>ir:{bool}=>il:{bool}=>
  x:{word}=>y:{word}=>ci:{bool}=>
processor_spec(s0,s1,s2,h0,h1,ir,il,x,y,ci)=
shifter_spec(h0,h1,ir,il,alu(s0,s1,s2,x,y,ci)) in {word}

CONJECTURE:
[]==>x:{word}=>y:{word}=>ci:{bool}=>
 s0:{bool}=>s1:{bool}=>s2:{bool}=>
  h0:{bool}=>h1:{bool}=>ir:{bool}=>il:{bool}=>
shifter_spec(h0,h1,ir,il,alu_spec(s0,s1,s2,x,y,ci))=
shifter(h0,h1,ir,il,alu(s0,s1,s2,x,y,ci))in{word
```

F.5 Other arithmetic operations

F.5.1 Adder

```
IMPLEMENTATION:
[]==>y:{word}=>plus_word(nil,y)=y in{word}
[]==>x:{word}=>y:{word}=>
 plus_word(i(x),y)=i(plus_word(x,y))in{word}

CONJECTURE:
```

```
[]==>x:{word}=>y:{word}=>
plus(word2nat(x),word2nat(y))=word2nat(plus_word(x,y))
in pnat
```

F.5.2 Multiplier

```
IMPLEMENTATION:
[]==>y:{word}=>times_word(nil,y)=nil in{word}
[]==>x:{word}=>y:{word}=>
 times_word(i(x),y)=plus_word(times_word(x,y),y)in{word}

CONJECTURE:
[]==>x:{word}=>y:{word}=>
 times(word2nat(x),word2nat(y))=word2nat(times_word(x,y))
in pnat
```

F.5.3 Exponentiator

```
IMPLEMENTATION:
[]==>x:{word}=>exp_word(x,nil)={true}::nil in{word}
[]==>x:{word}=>y:{word}=>
 exp_word(x,i(y))=times_word(x,exp_word(x,y))in{word}

CONJECTURE:
[]==>x:{word}=>y:{word}=>
 exp(word2nat(x),word2nat(y))=word2nat(exp_word(x,y))
in pnat
```

F.5.4 Factorial

```
IMPLEMENTATION:
[]==>fac_word(nil)={true}::nil in{word}
[]==>x:{word}=>y:{word}=>
 exp_word(x,i(y))=times_word(x,exp_word(x,y))in{word}

CONJECTURE:
[]==>x:{word}=>fac(word2nat(x))=word2nat(fac_word(x)) in pnat
```

F.5.5 Subtracter

```
IMPLEMENTATION:
[]==>x:{word}=>minus_word(x,nil)=x in{word}
[]==>y:{word}=>minus_word(nil,y)=nil in{word}
[]==>x:{word}=>y:{word}=>
 minus_word(x,i(y))=dec(minus_word(x,y))in{word}

CONJECTURE:
[]==>y:{word}=>x:{word}=>
```

```
 minus(word2nat(x),word2nat(y))=word2nat(minus_word(x,y))
in pnat
```

F.5.6 Divider

```
IMPLEMENTATION:
QUOTIENT
[]==>x:{word}=>div_word(x,nil)=nil in{word}
[]==>x:{word}=>y:{word}=>
 less(word2nat(x),word2nat(y))=>div_word(x,y)=nil in {word}
[]==>x:{word}=>y:{word}=>
 (less(word2nat(x),word2nat(y))=>void)=>
   div_word(plus_word(x,y),y)=i(div_word(x,y))in{word}

REMAINER:
[]==>x:{word}=>rem_word(x,nil)=nil in{word}
[]==>x:{word}=>y:{word}=>less(word2nat(x),word2nat(y))=>
 rem_word(x,y)=x in{word}
[]==>x:pnat=>y:pnat=>
 (less(word2nat(x),word2nat(y))=>void)=>
  rem_word(plus_word(x,y),y)=rem_word(x,y) in {word}

CONJECTURE:
[]==>x:{word}=>y:{word}=>
quot(word2nat(x),word2nat(y))=word2nat(div_word(x,y))in pnat

[]==>x:{word}=>y:{word}=>
rem(word2nat(x),word2nat(y))=word2nat(rem_word(x,y))in pnat
```

F.5.7 Counter

```
IMPLEMENTATION:
[]==>reset:flag=>
count_imp(reset)of 0=zeroes(length(nil))in{word}
[]==>t:pnat=>reset:flag=>
count_imp(reset)of s(t)=
del(restart(reset,incfun(count_imp(reset))))of s(t)in{word}

[]==>x:signal=>del(x)of 0=x of 0 in{word}
[]==>t:pnat=>x:signal=>del(x)of s(t)=x of t in{word}
[]==>t:pnat=>rs:flag=>x:signal=>
restart(rs,x)of t=if(rs of t,{true},nil,x of t)in{word}
[]==>t:pnat=>x:signal=>incfun(x)of t=i(x of t)in{word}

SPECIFICATION:
[]==>reset:flag=>count_spec(reset)of 0=0 in pnat
[]==>t:pnat=>reset:flag=>
count_spec(reset)of s(t)=
ifbn(reset of t,{true},0,s(count_spec(reset)of t))in pnat

CONJECTURE:
[]==>t:pnat=>reset:flag=>
word2nat(count_imp(reset)of t)=count_spec(reset)of t in pnat
```

Appendix G

Methods

In this appendix we include the main methods used in the proof planning of the circuit verification conjectures.

G.1 Symbolic evaluation

This method applies symbolic evaluation by calling the submethods `elementary`, *equal*, *reduction*, *eval_def*, *term_cancel*, *and bool_cases*.

```
method(sym_eval(SubPlan),
HG,
[repeat([HG],
Goal :=> SubGoals,
Method,
(member(Method,
                              [elementary(_),
     equal(_,_),
      reduction(_,_),
      eval_def(_,_),
term_cancel(_),
      bool_cases(_)]),
                  applicable_submethod(Goal,Method,_,SubGoals)),
                 [SubPlan],
                 SubGoals),!,
SubPlan \= idtac],
[],
SubGoals,
SubPlan).
```

G.2 Generalise

This method generalises a common term in two subexpressions.

```
method(generalise(Exp,Var:Type),
       H==>G,
       [matrix(Vs,M1,G),
        sinks(M,_,M1),      % Strip out sinks
        member(M,[(L=R in _),(L=>R),geq(L,R),leq(L,R),
                  greater(L,R),less(L,R),L#R]),
        exp_at(L,_,Exp),
        not atomic(Exp),
        not constant(Exp,_),
        object_level_term(Exp),
        exp_at(R,_,Exp),
        find_type(H,Exp,Type),
        append(Vs,H,VsH),
        hfree([Var],VsH)
       ],
       [strip_meta_annotations(G,NewG1),
        replace_all(Exp,Var,NewG1, NewG)
       ],
       [H==>Var:Type=>NewG],
        generalise(Exp,Var:Type)
      ).
```

G.3 Normalise

This method applies normalisation operations defined in the submethod `normal`.

```
method( normalize(NormTacs),
        OH==>OG,
        [ \+ member( _:[ihmarker(_,_)|_],OH),
          exp_at( OG, _, T=>_ ),
          exp_at( T, _, _=_ in _ ),
          iterate((OH==>OG) - [],
                  (Goal-Tacs):=>((H==>G)-STacs),
              (applicable_submethod(Goal,normal(A),_,
                                    [H==>G]),
                   STacs = [normal(A)|Tacs]
                   ),
                   true,
   SubGoal-RNormTacs
                 ),
          RNormTacs \= [],!,reverse(RNormTacs,NormTacs)
        ],
        [],
        [ SubGoal ],
normalize( NormTacs )
      ).
```

G.4 Induction strategy

This method applies the induction strategy.

```
method(ind_strat(induction(Scheme,VarTL) then CasesTactics),
       H==>G,
       [applicable_submethod(H==>G,induction(Scheme,VarTL))
       ],
       [scheme(Scheme,VarTL,H==>G,BSeqs,SSeqs),
        (map_list(BSeqs,BSeq:=>BSeq1-sym_eval(Ms),
                  applicable_submethod(BSeq,sym_eval(Ms),_,BSeq1),
                 BSeq1sBTs) orelse
         (BSeq1sBTs = [BSeqs-[idtac]])),
        zip(BSeq1sBTs,BSeq1s,BaseTactics1),
flatten(BaseTactics1,BaseTactics),
        flatten(BSeq1s,FBSeq1s),
        map_list(SSeqs,SSeq:=>SSeq1-step_case(Ms),
                  applicable_submethod(SSeq,step_case(Ms),_,SSeq1),
                SSeq1sSTs),
        zip(SSeq1sSTs,SSeq1s,StepTactics),
        flatten(SSeq1s,FSSeq1s),
        append(BaseTactics, StepTactics, CasesTactics),
append(FBSeq1s,FSSeq1s,AllSeqs)
       ],
       AllSeqs,
       ind_strat(induction(Scheme,VarTL) then CasesTactics)
      ).
```

G.5 Elementary

This method applies simple propositional reasoning.

```
submethod(elementary(I),
       H==>G,
       [not (G = (_:_#_)),
        elementary(H==>G,I)],
       [],
       [],
       elementary(I)
      ).
```

G.6 Use of equation in hypothesis

This method applies an equation in the hypotheses list as a rewrite rule.

```
submethod(equal(HName,Dir),
          H==>G,
          [ ((hyp(HName:Term=Var in T,H), Dir=left)
            v
            (hyp(HName:Var=Term in T,H), Dir=right)
           ),
           (not freevarinterm(Term,_)
            orelse
            (atomic(Var), not atomic(Term))
            orelse
            (atomic(Var), atomic(Term), Term @< Var)
           ),
           freevarinterm(G,Var)
          ],
          [map_list([G],In:=>Out,
                    replace_all(Var,Term,In,Out),[GG]),
           del_hyp(HName:_,H,HThin)
          ],
          [HThin==>GG],
          equal(HName,Dir)
         ).
```

G.7 Evaluate definition

This method does rewriting with equations that are not wave rules.

```
submethod(eval_def(Pos,[Rule,Dir]),
       H==>G,
       [matrix(Vars,Matrix,G),
        wave_fronts(_, [], Matrix),
        new_exp_at(Matrix,Pos,Exp),
        not metavar(Exp),
        func_defeqn(Exp,Dir,Rule:C=>Exp:=>NewExp),
        polarity_compatible(Matrix, Pos, Dir),
        elementary(H==>C,_)
        ],
       [replace(Pos,NewExp,Matrix,NewMatrix),
        matrix(Vars,NewMatrix,NewG)
       ],
       [H==>NewG],
       eval_def(Pos,[Rule,Dir])
      ).
```

G.8 Term cancellation

This method cancels out common additive terms in both sides of an equation.

```
submethod(term_cancel(NG),
      H==>G,
      [matrix(V,LS = RS in pnat,G),
       \+ (LS=0 v RS=0),
       only_sum(LS,LstPlusL),
       only_sum(RS,LstPlusR),
       cancel_eq(LstPlusL,LstPlusR,OtherL,OtherR),
       length(LstPlusL,X),
       length(OtherL,Y),
       X > Y],
      [putPlus(OtherL,Left),
       putPlus(OtherR,Right),
       matrix(V,Left=Right in pnat,NG)
       ],
      [H==>NG],
      term_cancel(NG)).
```

G.9 Boolean case analysis

This submethod applies Boolean case analysis

```
submethod(bool_cases(X),
H==>G,
member(X:{bool},H),
freevarinterm(G,X)
],
       [
replace_all(X,{false},G,G1),
replace_all(X,{true},G,G2),
        hfree([Id],H)
],
       [[Id:X={false} in {bool}|H]==>G1,
         [Id:X={true} in {bool}|H]==>G2],
       bool_cases(X)).

submethod(bool_cases(X),
       H==>X:{bool}=>G,
],
       [
replace_all(X,{false},G,G1),
replace_all(X,{true},G,G2),
        hfree([Id],H)
],
       [[Id:X={false} in {bool}|H]==>G1,
         [Id:X={true} in {bool}|H]==>G2],
        bool_cases(X)).

submethod(bool_cases(Term),
```

```
H==>G,
matrix(Vars,Matrix,G),
        exp_at(Matrix,Pos,Term),
        Term=hd(_),
        append(Vars,H,NewH),
        find_type(NewH,Term,{bool})
],
       [
        listof(V:T,P^(member(V:T,Vars),
               exp_at(Term,P,V)),BoundVars),
        subtract(Vars,BoundVars,RestVars),
        append(BoundVars,H,AllH),
  replace_all(Term,{false},Matrix,MG1),
replace_all(Term,{true},Matrix,MG2),
        hfree([Id],AllH),
        matrix(RestVars,MG1,G1),
        matrix(RestVars,MG2,G2)
],
       [[Id:Term={false} in {bool}|AllH]==>G1,
         [Id:Term={true} in {bool}|AllH]==>G2],
        bool_cases(Term)).
```

G.10 Memoise recursive function

This submethod applies the memoisation procedure on named recursive functions.

```
submethod(eval_def(Rule,[Memo,Pos,Dir,Exp,NewExp,Type]),
   H==>G,
   [matrix(Vars,Matrix,G),
    wave_fronts(_, [], Matrix),
    new_exp_at(Matrix,Pos,Exp),
    not metavar(Exp),

% expression is a memo function:
    (((
       (Exp=mpc(s(X1),_,_,_),Type=wordn(5)) v
       (Exp=memory(s(X3),_,_,_),Type=wordn(16)) v
       (Exp=pc(s(X4),_,_,_),Type=wordn(13)) v
       (Exp=acc(s(X5),_,_,_),Type=wordn(16)) v
       (Exp=buffer(s(X6),_,_,_),Type=wordn(16)) v
       (Exp=mar(s(X7),_,_,_),Type=wordn(13)) v
       (Exp=arg(s(X8),_,_,_),Type=wordn(16)) v
       (Exp=ir(s(X9),_,_,_),Type=wordn(16))
      ),

% value has been calculated already
     ((member(Rule:Exp=NewExp in Type,H),
       NewH=H
      )
                                v

% value hasn't been calculated, do it and add it to hyp list
      (func_defeqn(Exp,Dir,_:C=>Exp:=>NewExp2),
       matrix(Vars,NewExp2,Goalmpc),
       applicable_submethod(H==>Goalmpc,
                            sym_eval(_),_,[Hmemo==>NewExp]),!,
```

```
       hfree([Rule],Hmemo),
       NewH=[Rule:Exp=NewExp in Type|Hmemo]
      )
     ),
     Memo=1
    )
                                v
% expression is NOT a memo function:
     (func_defeqn(Exp,Dir,Rule:C=>Exp:=>NewExp),
      polarity_compatible(Matrix, Pos, Dir),
      elementary(H==>C,_),
      NewH=H,
      Memo=0,nl,write('7. Rule = '), write(Rule),nl
    )
   )
  ],
       [replace(Pos,NewExp,Matrix,NewMatrix),
        matrix(Vars,NewMatrix,NewG)
       ],
       [NewH==>NewG],
       eval_def(Rule,[Memo,Pos,Dir,Exp,NewExp,Type])
      ).
```

G.11 Weak fertilise

This submethod applies weak fertilisation.

```
submethod(weak_fertilize(Dir,Connective,Pos,Hyp),
       H==>G,
       [matrix(Vars,M,G),
transitive_pred( M, [LR,RL], [LRN,RLN], NewG_M ),
        exp_at(M,[0],Connective),
        ((Dir=right,GL=LR,GR=RL, GLNew=LRN, GRNew = RLN) v
          (Dir=left,GL=RL,GR=LR, GLNew=RLN, GRNew = LRN )
        ),
        (
          (wave_fronts(GR1,[[]-PosL/[Typ,out]],GR),
           select(Pos,PosL,OtherPosL),
           NewWFspec = [[]-PosL/[Typ,in]]
          )
        v
          (wave_fronts(GR1,[[]-PosL/[Typ,in]],GR),
   PosL = [_,_|_],
           select(Pos,PosL,OtherPosL),
           NewWFspec = [[]-PosL/[Typ,in]]
          )
         v
  (wave_fronts(GR1,[],GR),
           wave_fronts(_,[_-_/[_,out]|_],GL),
           PosL=[],Pos=[],OtherPosL=[[]],
           NewWFspec = [Pos-OtherPosL/_]
  )
         v
          (wave_fronts(GR1tmp,[WFPos-[WHPos]/[Typ,_]],GR),
```

```
          sinks(GR1,[WFPos],GR1tmp),
          sinks(GL1,_,GL),
   append(WHPos,WFPos,RSinkPos),
   exp_at(GR1,RSinkPos,Sink),
   exp_at(GL1,LSinkPos,Sink),
   NewWFspec = [LSinkPos-[WHPos]/[Typ,out]],
   Pos=[]
         )
        v

        (wave_fronts(GR1tmp,[[]-PosL/[Typ,out]|SunkFronts],GR),
         select(Pos,PosL,OtherPosL),
         NewWFspec = [[]-PosL/[Typ,in]],
         sinks(GR1,SinkPosns,GR1tmp),
         (forall{PP \ SunkFronts}: (PP = FrontPos-_/[_,_],
           (thereis{ SS \ SinkPosns} : append(SS,_,FrontPos))))
        )
       ),
        exp_at(GR1,Pos,GR1Sub),
% check for positive occurrence or symmetrical function symbol:
        (Connective = (in) orelse polarity(_,_,GR1,Pos,+)),
( member( _:[ihmarker(notraw(_),[])|IHyps], H ),
  nv_member(Hyp:IndHyp,IHyps);
          member( Hyp:IndHyp, H ),
  IndHyp \= [_|_]
),
        matrix(_,IndHyp_M,IndHyp),
        exp_at(IndHyp_M,[0],Connective),
        replace_all(GL,GRNewSub,M,GSub1),
        replace_all(GR,GR1Sub,GSub1,GSub),
        instantiate(IndHyp,GSub,Instan),
        ground_sinks(Instan,GL,GR,GSub),
        wave_fronts(GRNewSub,_,GRNewSub)

       ],
% postconditions
       [replace(Pos,GRNewSub,GR1,GRNew1),
        wave_fronts(GRNew1,NewWFspec,GRNew),
        wave_fronts(GLNew,_,GL),
        sinks(NNewG_M,_,NewG_M),
        matrix(Vars,NNewG_M,NewG)
       ],
       [H==>NewG],
       weak_fertilize(Dir,Connective,Pos,Hyp)
      ).
```