Technical Report

UCAM-CL-TR-682
ISSN 1476-2986

Number 682

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Translating HOL functions to hardware

Juliano Iyoda

April 2007

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

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

This technical report is based on a dissertation submitted October 2006 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Hughes Hall.

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

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

ISSN 1476-2986

Translating HOL functions to hardware

Juliano Iyoda

Abstract

Delivering error-free products is still a major challenge for hardware and software engineers. Due to the increasingly growing complexity of computing systems, there is a demand for higher levels of automation in formal verification.

This dissertation proposes an approach to generate formally verified circuits automatically. The main outcome of our project is a compiler implemented on top of the theorem prover HOL4 which translates a subset of higher-order logic to circuits. The subset of the logic is a first-order tail-recursive functional language. The compiler takes a function f as argument and automatically produces the theorem

$$\vdash \mathsf{C} \textit{ implements } f$$

where C is a circuit and *implements* is a correctness relation between a circuit and a function. We achieve full mechanisation of proofs by defining theorems which are composable. The correctness of a circuit can be mechanically determined by the correctness of its sub-circuits. This technology allows the designer to focus on higher levels of abstraction instead of reasoning and verifying systems at the gate level.

A pretty-printer translates netlists described in higher-order logic to structural Verilog. Our compiler is integrated with Altera tools to run our circuits in FPGAs. Thus the theorem prover is used as an environment for supporting the development process from formal specification to implementation.

Our approach has been tested with fairly substantial case studies. We describe the design and the verification of a multiplier and a simple microcomputer which has shown us that the compiler supports small and medium-sized applications. Although this approach does not scale to industrial-sized applications yet, it is a first step towards the implementation of a new technology that can raise the level of mechanisation in formal verification.

Acknowledgements

First I want to thank my supervisor Mike Gordon. He was always available to listen and to help. His constant enthusiasm, inspiring advices and encouragement guided me gently towards the conclusion of this dissertation. In particular, his motivation and expertise in HOL helped me to program parts of the compiler faster than I thought possible.

The ARG members have always helped me with whatever problem I had. Special thanks go to Joe Hurd and my office mates Anthony Fox and Hasan Amjad, who have always kindly helped me in my work.

Thanks to Robert Mullins and Simon Moore for helping me in integrating FPGAs and Quartus II with Linux. David Greaves provided me insightful comments on hardware design. Ken Friis Larsen integrated my UART program in C with Moscow ML. Konrad Slind and Scott Owens have been great colleagues who supported this project with enthusiasm and contributed with significant improvements to the compiler.

The good people of the then Portuguese Society introduced me to their culture in their unique entertaining way. It was a pleasure to later join the committee presided by Eliana Lucas.

Thanks to those who changed my life in Cambridge. I couldn't have asked for better friends: Carol Anselmo, Pedro Anselmo Filho, Clint Ballinger, Ronaldo Batista, Marco Cariglia, Raquel Costa, Val Feltrim, Liliana Ferreira, Carol Gasperin, Rodrigo Gribel, Carlos Hotta, Paul Hunter, Daniel Maciel, Ricardo Mendes, Antônio Moura, Chico Prosdocimi, Isabel Ribeiro, Leda Sampson, João Pedro dos Santos, André Sartori, Paula Signorini, Yuri Sobral and Cris Viegas.

Augusto Sampaio was my supervisor during my Masters and has always been a good friend and a true mentor. His constant advice and help in my professional career influenced my decision to come to Cambridge.

My PhD has been financially supported by the CAPES Institute (a foundation attached to the Brazilian Ministry of Education).

And, finally, thanks to my parents for all the support, patience and love.

Contents

Chapter 1

Introduction

The development of reliable systems is still a challenge to hardware and software engineers. Computing systems are increasingly growing in complexity and size, which makes subtle errors more likely to appear. Uncovered bugs in the development phase can cause severe loss of money or, in safety-critical applications, even human life. According to Wired News website, among the History's Worst Bugs are the Intel Pentium floating point division and the Ariane 5 Flight 501 [25]. These failures are estimated to have cost around half a billion US dollars each. Although there is no definitive solution to this problem, *formal methods* is the most rigorous set of techniques which provides high assurance of correctness.

Formal methods refers to techniques and tools based on mathematical logic which ensure the quality and the correctness of a design by representing specifications and implementations in a particular logical system. *Formal verification* is a rigorous deduction in the logic (a theorem) which shows that an implementation meets the specification. *Theorem provers* are software tools which mechanise a logical system and automate (as much as possible) the proof of theorems.

This dissertation proposes an approach for automatic formal verification of hardware. The specification is a subset of higher-order logic and the implementation is a circuit represented in the logic. The theorem prover HOL4 [59] translates the source code into a circuit and automatically proves that it meets the specification. In order to achieve a fully mechanised verification, we develop a technology to compose proofs automatically. We show the feasibility of this method in the development of small and medium-sized applications. Although this technology does not scale to industrial-sized applications yet, this thesis is a proof-of-concept that our approach is a promising first step in improving the degree of mechanisation in hardware verification (on any scale) using theorem provers.

In what follows, we overview how theorem provers help in mechanising hardware verification followed by a description of the contributions of this dissertation and an outline of the structure of the next chapters.

1.1 Mechanised Verification

There are two main techniques for mechanising hardware verification. *Model checking* exhaustively verifies that a model of the system satisfies some property. Systems are usually modelled as a finite state machines, which allows model checking to perform fully automatic verification. Another advantage of model checking is the generation of

counterexamples whenever a property does not hold. In contrast to model checking, *theorem proving* can deal with infinite state spaces. A theorem prover mechanises a proof system. Both the system and the properties are expressed inside some logic defined by axioms and inference rules. Theorem provers are able to model complex systems due to the high expressiveness of their logic. The disadvantage of using theorem provers is the need for user guidance in constructing proofs (although in particular domains, full automation is possible).

As our work is on theorem provers applied to hardware verification, we overview previous work using this technology. For an excellent survey on the application of model checking (and theorem provers) to hardware verification, see Kern and Greenstreet [49].

In the classic paper *Why higher-order logic is a good formalism for specifying and verifying hardware* [27], Gordon shows that specialised hardware description languages and specialised deductive systems are not needed for hardware verification. Formal logic suffices. Higher-order logic is now a well established formalism for specifying and verifying hardware [8, 19, 36, 55]. An advantage of this formalism is that higher-order functions naturally model signals as functions from time (natural numbers) to values, like Booleans. Moreover, it is possible to define additional mathematical theories on top of the logic and use them in hardware modelling. There are different versions of higher-order logic.

The HOL4 system implements Church's simple type theory with polymorphic types. The early experiments with hardware verification described by Gordon [27] were later extended by Melham, who introduced abstraction mechanisms for functional, data and time refinement [55]. Parallel to these developments, the verification of complete processors and micro-architectures were undertaken (e.g., Viper [16] and ARM6 [23]). More recently, Blumenröhr embedded the hardware description language Gropius [8] in HOL. This work is based on pre-proved theorems, which allow the automatic generation of circuits whose sub-modules are formally verified, thus providing a higher degree of mechanisation.

In VERITAS$^+$ [36], classical non-constructive logic is extended with dependent types and subtypes, thus enhancing its expressiveness and allowing polymorphism. The disadvantage of this extension is the loss of decidable type-checking. Dependent types and subtypes are also features of the PVS [64] prover. Its particular specification language is based on higher-order logic. Although PVS is a general purpose theorem prover, it has been applied to several projects in hardware verification, like the verification of microprocessors (AAMP5), arithmetic circuits and dynamic hardware reconfiguration algorithms [78, 79].

Hardware correctness has also been verified in first-order logic. The Boyer-Moore logic is a first-order, untyped, quantifier free logic of total recursive functions. The Nqthm theorem prover [13], which mechanises the Boyer-Moore logic, has been extensively used in hardware verification. For instance, Hunt et al. applied the prover to verify the correctness of the microprocessors FM8501 and FM9001 [40, 41]. The user has to be familiar with the heuristics employed in the prover in order to be able to guide Nqthm effectively [13]. The theorem prover ACL2 (A Computational Logic for Applicative Common Lisp) is a recoded version of Nqthm which supports a subset of applicative Common Lisp [48]. Like Nqthm, ACL2 has also being used in several projects related to verification of hardware. For instance, ACL2 was used to mechanically check the verification of a complex pipelined microprocessor [70] and to formalise a finite state machine language applied to the verification of a microprocessor [42].

There are several other theorem provers which have been applied to hardware verifica-

tion. However it is not our intention to cover in detail all the previous work in hardware verification here, but to illustrate how the efforts to tackle industrial-sized applications and to achieve a greater degree of mechanisation using theorem provers evolved. The level of mechanisation, abstraction and the scale of later work [13, 23, 55, 78] have shown a significant improvement in comparison to early verification efforts [27, 36, 40]. Despite the tremendous advances in the mechanisation of hardware verification which produced several impressive cases of industrial-sized verifications, in general the process still requires human input. Hardware proofs in a particular application domain may follow particular patterns of verification and, consequently, benefit from a specific automatic proof technology, but in general theorem provers need user guidance.

In our work we experiment with an approach which minimises (or eliminates) any user guidance in the verification process. As mentioned in the previous section, one of the aims of this project is to evaluate the feasibility of this method to hardware specification and verification. We have chosen to work with the HOL4 system. Although in principle any other higher-order logic theorem prover could be used, hardware verification is intrinsically related to the HOL4 system. It is the application which motivated its creation [29] and which has been vastly explored and has contributed significantly to the system's development.

1.2 Contributions

This section lists the main contributions of this dissertation.

- We develop a compiler which takes a function f as argument and *automatically* returns the theorem

$$\vdash \mathsf{C} \textit{ implements } f$$

  where C is a generated circuit and *implements* is a correctness relation between implementation and specification (*implements* is formally defined in Section 2.4). Both the circuit C and the function f are represented in HOL. In particular, C is a HOL representation of a netlist. The function f lies in a subset of HOL which constitutes a first-order tail-recursive functional language. The fully automatic verification is based on the principle of compositionality. The correctness of a complex circuit can be mechanically determined by the correctness of its sub-circuits. This is achieved by carefully designing the structure of our theorems.

- As a direct consequence of the automatic verification of circuits with respect to a functional program, the designers do not have to reason and to interactively verify systems at lower levels of abstraction like architectural, register-transfer or gate level. The development process can start from a specification in higher-order logic which is subsequently proved (interactively) to be implemented by a first-order tail-recursive function. This is the lowest level of abstraction the designer has to reason about during verification.

- We tested our approach to evaluate its feasibility. Our experiments and case studies have shown that our technology supports small and medium-sized applications. Typically the compiler takes few hours to verify the correctness of circuits with

approximately 700 components. Chapter 5 presents the verification of a simple microcomputer specified as functional program. We also developed a Booth multiplier specified originally as part of the ARM6 verification project (Section 5.1). Cryptographic algorithms like TEA [82] were verified by Konrad Slind at the University of Utah [76].

- A fully functional prototype that links HOL4 to an FPGA has been implemented. The complete development process can be carried out from the theorem prover. The HOL netlist is informally translated to structural Verilog, which is compiled and downloaded to an Altera FPGA by the Quartus II tool set.

1.3 The structure of this dissertation

In this section we outline the structure of the subsequent chapters.

Chapter 2 introduces the main concepts used in our approach. Initially we illustrate our approach informally using fictitious source and target languages. This is followed by a brief description of HOL and of how hardware is modelled and verified in the logic. We conclude by describing how the concepts introduced informally at the beginning of the chapter are formalised in HOL.

Chapter 3 presents several optimisation techniques and compilation steps for synthesis. The optimisations are basically theorems which state the equivalence between a circuit and its optimised version. We also describe a compilation step that introduces a global clock to the system and the technology used to download the netlist in an FPGA. The last section illustrates the complete verification process step-by-step.

Chapter 4 reports on the limitations of our approach and describes issues and problems we faced during the development of this project.

Chapter 5 shows the development of two case studies: an implementation of the Booth multiplier and a simple microcomputer called the DIY microcomputer. We developed two different designs for the DIY. In the first one, only the CPU is modelled in HOL. The second design includes a tiny memory connected to the CPU. We analyse the strengths and weaknesses of our compilation method revealed by the case studies.

In **Chapter 6**, we describe several closely related work. We review projects based on theorem provers and work on hardware synthesis of functional languages. We present a comparative analysis of these approaches based on the abstraction level of the source language and the level of automation of the verification.

Chapter 7 summarises our work and discusses the limitations and contributions of our approach.

Part of Chapter 2 is also described by Gordon et al. [30] and Slind et al. [77]. Chapter 3 and Section 5.1 were also introduced by Slind et al. [76].

Chapter 2

Compilation

This chapter describes the compilation-by-proof approach. Section 2.1 informally describes the automatic verification using simple fictitious languages. Sections 2.2 and 2.3 overview higher-order logic and show how circuits are modelled and verified in the logic. Finally, Section 2.4 presents the source and the target languages and formalises the method introduced informally in Section 2.1.

2.1 Introduction

This section introduces the compilation and the verification method based on composable entities. For simplicity, we use fictitious languages in our examples.

Our aim is to develop a compiler which takes a program f and *automatically* produces a theorem

$$\vdash \mathsf{C} \textit{ implements } f$$

where C is a generated circuit and *implements* is a correctness relation between implementation and specification (*implements* is formally defined in Section 2.4).

First we classify the commands or constructors of a language in two kinds: atomic and composite. Atomic commands are those which do not depend on any sub-command to exist. For example, assignments are atomic commands in imperative programming languages. Composite commands are those which are built from sub-commands. Sequential composition and `if-then-else` are examples of composite commands.

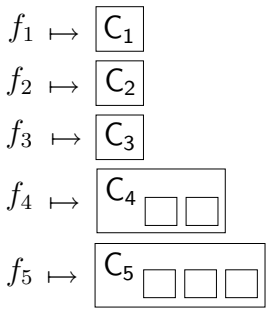
A simple source language is used in this section to illustrate the compilation and the verification method. Its BNF is shown below.

$$p \ ::= \ f_1 \mid f_2 \mid f_3 \mid f_4(p_1, p_2) \mid f_5(p_1, p_2, p_3)$$

The commands f_1, f_2 and f_3 are atomic. The composite commands f_4 and f_5 take two and three sub-commands as arguments, respectively. For example, $f_5(f_4(f_3, f_2), f_1, f_2)$ is a program written in this language.

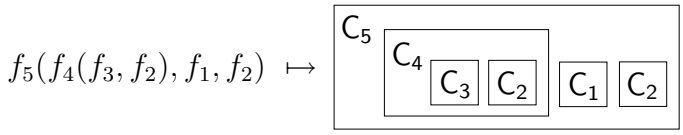
In order to translate this language into hardware (without verifying it), we define a

corresponding circuit implementation for each language constructor.



Circuits are represented by boxes, reflecting their physical structure.

The compilation is carried out by simply replacing every occurrence of a command in the source code by its corresponding circuit implementation. For example, the compilation of $f_5(f_4(f_3, f_2), f_1, f_2)$ produces



For this method to work, the circuit constructors must have the same interface. This allows every circuit to connect to every other circuit. In the example above, the circuit $\mathsf{C_5}$ is connected to both primitive and composite sub-circuits. This is basically how SAFL [71] translates a functional program to hardware.The key idea is to deal with *composable* circuits.

This method inspired us to develop a similar approach concerning verification. The main challenge is to prove theorems which state the correctness of the circuits and also satisfy the composability property.

First we prove that the atomic circuits are correct.

$$\vdash \boxed{\mathsf{C_1}} \textit{ implements } f_1$$

$$\vdash \boxed{\mathsf{C_2}} \textit{ implements } f_2$$

$$\vdash \boxed{\mathsf{C_3}} \textit{ implements } f_3$$

The next step is to prove that the correctness of composite circuits depends on the correctness of its sub-circuits.

$$\vdash (\boxed{\mathsf{C}_f} \textit{ implements } f) \ \wedge \ (\boxed{\mathsf{C}_g} \textit{ implements } g)$$

$$\Rightarrow \boxed{\mathsf{C_4}\ \boxed{\mathsf{C}_f}\ \boxed{\mathsf{C}_g}} \textit{ implements } f_4(f, g)$$

$$\vdash (\boxed{\mathsf{C}_e} \textit{ implements } e) \ \wedge \ (\boxed{\mathsf{C}_f} \textit{ implements } f) \ \wedge \ (\boxed{\mathsf{C}_g} \textit{ implements } g)$$

$$\Rightarrow \boxed{\mathsf{C_5}\ \boxed{\mathsf{C}_e}\ \boxed{\mathsf{C}_f}\ \boxed{\mathsf{C}_g}} \textit{ implements } f_5(e, f, g)$$

The circuits C_e, C_f and C_g are implementations of the functions e, f and g, respectively. Note that they are not necessarily atomic. The composability of these theorems comes from the uniform structure of the antecedents and the consequents of the implication "(C *implements* f)". In addition to that, circuits and commands are also composable. This property allows a proof assistant to build proofs of correctness automatically.

Figure 2.1 shows the compilation-by-proof of the program $f_5(f_4(f_3, f_2), f_1, f_2)$. Steps

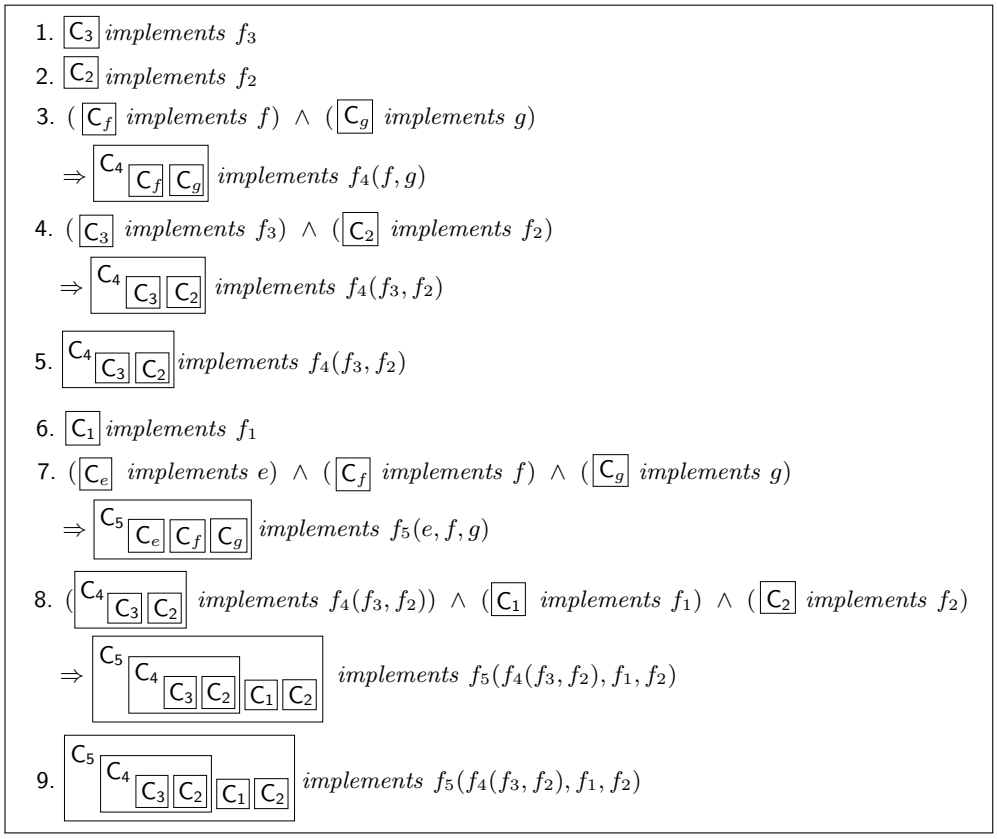


Figure 2.1: Verification of $f_5(f_4(f_3, f_2), f_1, f_2)$.

1, 2 and 3 are the correctness theorems for circuits $\mathsf{C_3}$, $\mathsf{C_2}$ and $\mathsf{C_4}$, respectively. On step 4, the variables C_f and C_g of step 3 are specialised with the circuits $\mathsf{C_3}$ and $\mathsf{C_2}$. Step 5 eliminates the antecedent of step 4 by applying $\wedge$-Introduction and *modus ponens* with respect to steps 1 and 2. A similar proof is carried out from steps 6 to 9.

This example shows that we depend solely on $\wedge$-Introduction, *modus ponens* and specialisation of variables in order to verify the correctness of a circuit. The compilation process incrementally constructs the source code on the right-hand side of *implements*. This is easily mechanisable by a theorem proving system like HOL4 and requires no user guidance. Notice that the proof of correctness of each circuit constructor presented above is carried out in a interactive way, but the proof of any particular circuit is done automatically. This idea is not new. Composable theorems have already been used to implement algebraic compilers based on a refinement calculus [12, 69].

This section provided an informal description on how the automatic verification works. Before formalising the concepts presented here, higher-order logic is introduced, followed by an overview of hardware verification in this logic.

2.2 A Brief Introduction to HOL

This section briefly introduces the HOL logic [28]. We intend to explain the logical system from the hardware designer point of view. The terms *HOL* and *higher-order logic* are used interchangeably to mean the particular formulation developed by Mike Gordon at the University of Cambridge [26].

HOL is basically a predicate calculus with typed λ-calculus terms.

The predicate calculus of HOL allows variables to range over functions and predicates. For example, Peano's *Mathematical Induction* postulate is naturally formalised in HOL.

$$\vdash\ \forall P.\ P\ 0\ \wedge\ (\forall n.\ P\ n \Rightarrow\ P\ (\mathsf{SUC}\ n))\ \Rightarrow\ (\forall n.\ P\ n)$$

The variable P ranges over predicates. If P holds for 0 and if whenever it holds for a number n, it also holds for its successor ($\mathsf{SUC}\ n$), then P holds for all natural numbers. Table 2.1 summarises the predicate logic notation.

Term	Description
T	*true*
F	*false*
$\neg t$	*not* t
$t_1\ \vee\ t_2$	t_1 *or* t_2
$t_1\ \wedge\ t_2$	t_1 *and* t_2
$t_1\ \Rightarrow\ t_2$	t_1 *implies* t_2
$t_1\ =\ t_2$	t_1 *equals* t_2
$\forall x.\ t$	*for all* $x : t$
$\exists x.\ t$	*for some* $x : t$
$\varepsilon x.\ t$	*an* x *such that* $: t$
if t *then* t_1 *else* t_2	*conditional*

Table 2.1: Terms of the HOL logic.

The BNF for the untyped λ-terms is shown below.

$$M\ ::=\ c \mid v \mid (M\ N) \mid \lambda v.\ M$$

The syntactical variables c and v range over constants and variables, respectively. Function applications have the form $(M\ N)$ and λ-abstractions are of the form $\lambda v.\ M$. For example, $(\lambda x.\ x{+}1)$ denotes the function that takes a number and returns its successor. The term $((\lambda x.\ x{+}1)\ 5)$ evaluates to 6. Functions can take functions as arguments and return functions as results. For example, the function $(\lambda n.\ \lambda m.\ n{+}m)$ takes an argument, say 3, and returns a function which takes a number and adds 3 to it: $(\lambda m.\ 3{+}m)$. Actually, by defining new constants, it is possible to manipulate higher-order functions in a more user-friendly way. For example, it is easy to define the constant *add* as $(add\ n\ m = n{+}m)$. In Section 2.3 we illustrate how higher-order functions are used to model circuits.

HOL is a typed logic. The version of higher-order logic presented here extends Church's simple type theory [15] with polymorphic types. For example, the equality operator $=$ is a higher-order function of type $\alpha \rightarrow (\alpha \rightarrow bool)$. The type of its arguments is not defined *a priori*. Type variables are represented by the Greek letters α, β, γ, etc. The type $\sigma_1 \rightarrow \sigma_2$

denotes the set of all total functions from values of σ_1 to values of σ_2. In the Mathematical Induction postulate presented above, the predicate P is of type $num \rightarrow bool$, where num is the type of natural numbers. We can write $P : num \rightarrow bool$ to explicitly declare its type.

The HOL logic is actually built from a very small set of primitive definitions. The primitive terms are those of the λ-expressions. The primitive constants are equality ($=: \alpha \rightarrow \alpha \rightarrow bool$), implication ($\Rightarrow: bool \rightarrow bool \rightarrow bool$) and the choice operator ($\varepsilon : (\alpha \rightarrow bool) \rightarrow \alpha$). The primitive types are *bool*, *ind* (set of *individuals*) and the type operator *fun*, which is abbreviated to the infix notation $\rightarrow$. All other types and constants are introduced in terms of these primitives by rules of definition, which guarantee to preserve the consistency of the system. From the primitive definitions, it is possible to develop a large library of theories like those for natural numbers, sets, lists, groups, etc. One of the first attempts to derive portions of mathematics from logic is described in *Principia Mathematica* by Russell and Whitehead [83] (but using a different logic).

There are quite a few theorem provers which mechanise HOL.

Isabelle is a generic theorem prover which provides a mechanism (a simple metalogic) to allow different object logics to be represented in it. The Isabelle system already includes several logics like Logic for Computable Functions, classical and intuitionistic first-order logic and Zermelo-Fraenkel set theory. Its specialisation for higher-order logic is called Isabelle/HOL [58].

HOL Light is an implementation of HOL built on top of a smaller and simpler logical core in comparison to previous implementations [37, 38]. Although its logical kernel is implemented in just about 400 lines of OCaml [60], HOL light is powerful enough to be used in large verification projects.

ProofPower [65] is another implementation of HOL. It was originally implemented by International Computers Ltd. (ICL) in collaboration with Program Validation Ltd. and the universities of Kent and Cambridge. Their aim was to develop a tool to support both HOL and the Z notation. In 1993 the first version of ProofPower was released. It was applied to high assurance security systems and safety-critical software. Since 1997, it has being developed by Lemma 1 Ltd.

The system we use in this work is the HOL4 [59]. The HOL4 system is latest version of a series of implementations first released in 1988. The HOL system was the first mechanisation of higher-order logic and was originally developed for hardware verification [28]. All of the systems mentioned above are descendants of the LCF system [56]. They follow many of the design concepts and remarkable ideas created by Robin Milner.

The next section illustrates how higher-order logic can be used to naturally model circuits and how hardware verification benefits from a proof assistant.

2.3 Hardware Verification in HOL

There are different ways of specifying the behaviour of a hardware component in higher-order logic. They vary in the level of abstraction and style. This section describes the model developed by Mike Gordon and Tom Melham for sequential circuits [27, 55].

The behaviour of a hardware component is described by a predicate which restricts the observable values on its external wires. The component is regarded as a black box (see Figure 2.2).

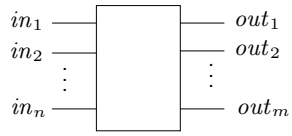
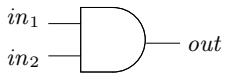


Figure 2.2: Hardware component as a black box.

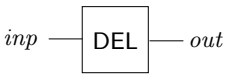
A predicate C, which specifies the behaviour of a component, is defined such that $\mathsf{C}(in_1, in_2, \ldots, in_n, out_1, out_2, \ldots, out_m)$ is true if and only if the wires in_1, in_2, ..., in_n, out_1, out_2 and out_m drive observable values.

Wires are modelled as functions from time to some type. They represent the sequence of values that appear on wires. Time is represented by natural numbers (type *num*) and values are either Booleans (representing a single bit) or words of a given size. A theory of n-bit words was formalised in HOL to support the verification of the ARM6 micro-architecture [23].

Figure 2.3 shows the specifications of an AND gate and a delay component.



$$\vdash \mathsf{AND}(in_1, in_2, out) \;=\; \forall t.\; out\; t = (in_1\; t) \;\wedge\; (in_2\; t)$$



$$\vdash \mathsf{DEL}(inp, out) \;=\; \forall t.\; out\; (t{+}1) = inp\; t$$

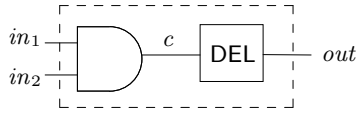
Figure 2.3: Primitive components in HOL.

Note that the predicates are higher-order. The wires in_1, in_2, *inp* and *out* are functions from time to Boolean ($num \rightarrow bool$). The definition of AND states that for all times we observe its external wires, the value of the output is the conjunction of the values of the inputs at that time. The delay component specifies the value of the output in terms of the value of the input at the *previous* time. At time zero, the value of the output is undefined.

The values of the external wires of a single component are restricted by its predicate. In order to specify the behaviour of two components connected together, we simply conjoin their predicates. The new specification restricts all wires of the composite circuit to satisfy the constraints imposed by both sub-circuits. In order to hide internal wires we can simply use the existential quantifier. Figure 2.4 shows the definition of a device which comprises an AND gate connected to a delay component by the internal wire c.

2.3.1 Verification

This section describes a simple example that illustrates how we can formally verify properties of a given circuit. The example shows how to verify that a particular circuit built from AND and NOT gates correctly implements an OR-gate.



$$\vdash \ \mathsf{AND\_DEL}(in_1, in_2, out) \ = \ \exists c.\ \mathsf{AND}(in_1, in_2, c) \ \wedge \ \mathsf{DEL}(c, out)$$

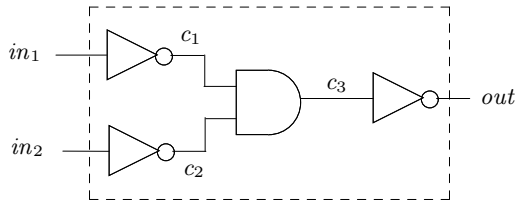
Figure 2.4: Circuit composition in HOL.

First we define the formal *specification* of the system, i.e. a predicate which specifies the required behaviour of the circuit.

$$\vdash \ \mathsf{OR\_SPEC}(in_1, in_2, out) \ = \ \forall t.\ out\ t = (in_1\ t) \ \vee \ (in_2\ t)$$

The specification states that the value of the output is always the disjunction of the inputs.

After specifying the required behaviour of the system, an *implementation* must be formally defined (see Figure 2.5).



$$\vdash \ \mathsf{OR\_IMP}(in_1, in_2, out) \ = \exists c_1\ c_2\ c_3.\ \mathsf{NOT}(in_1, c_1) \ \wedge \ \mathsf{NOT}(in_2, c_2) \ \wedge \ \mathsf{AND}(c_1, c_2, c_3) \ \wedge \ \mathsf{NOT}(c_3, out)$$

Figure 2.5: Is the output the disjunction of the inputs?

The specification of an inverter is similar to the definition of an AND gate showed in the previous section.

$$\vdash \ \mathsf{NOT}(inp, out) \ = \ \forall t.\ out\ t = \neg(inp\ t)$$

In order to prove that the circuit OR_IMP correctly implements the specification OR_SPEC, we have to prove that if the values on the external wires in_1, in_2 and out satisfy the constraints imposed by OR_IMP, then they must also satisfy OR_SPEC. This notion of correctness is formalised by a logical implication.

$$\forall in_1\ in_2\ out.\ \mathsf{OR\_IMP}(in_1, in_2, out) \ \Rightarrow \ \mathsf{OR\_SPEC}(in_1, in_2, out)$$

Proof: The proof starts by assuming that:

$$\mathsf{OR\_IMP}(in_1,in_2,out)$$

Replacing OR_IMP, AND and NOT by their definitions gives:

$$\exists c_1\ c_2\ c_3.\ (\forall t.\ c_1\ t = \neg(in_1\ t)) \ \wedge \ (\forall t.\ c_2\ t = \neg(in_2\ t)) \ \wedge \ (\forall t.\ c_3\ t = (c_1\ t) \ \wedge \ (c_2\ t)) \ \wedge \ (\forall t.\ out\ t = \neg(c_3\ t))$$

Now we can move all the equations under the scope of a single $\forall$ quantifier.

$$\exists c_1\ c_2\ c_3.\ \forall t.\ (c_1\ t = \neg(in_1\ t))\ \wedge\ (c_2\ t = \neg(in_2\ t))\ \wedge \\ (c_3\ t = (c_1\ t)\ \wedge\ (c_2\ t))\ \wedge\ (out\ t = \neg(c_3\ t))$$

Replacing the equation $(out\ t = \neg(c_3\ t))$ with the right-hand sides of $(c_1\ t = \ldots)$, $(c_2\ t = \ldots)$ and $(c_3\ t = \ldots)$ gives:

$$\exists c_1\ c_2\ c_3.\ \forall t.\ out\ t = \neg(\neg(in_1\ t)\ \wedge\ \neg(in_2\ t))$$

De Morgan's Law can now be applied to simplify to:

$$\exists c_1\ c_2\ c_3.\ \forall t.\ out\ t = (in_1\ t)\ \vee\ (in_2\ t)$$

Eliminating the existential quantifier and using the definition of OR_IMP yield:

$$\mathsf{OR\_SPEC}(in_1, in_2, out)$$

Based on the assumption in the first step, we can conclude that

$$\mathsf{OR\_IMP}(in_1, in_2, out)\ \Rightarrow\ \mathsf{OR\_SPEC}(in_1, in_2, out)$$

By generalising the free variables, we prove that

$$\forall in_1\ in_2\ out.\ \mathsf{OR\_IMP}(in_1, in_2, out)\ \Rightarrow\ \mathsf{OR\_SPEC}(in_1, in_2, out)$$

□

The theorem above is actually proved in fewer steps using the HOL4 system. For this simple example, very little user-guidance is needed and even a pencil-and-paper proof is easy to prove and check. However this is not the case for more elaborate implementations. For instance, the case studies presented in Chapter 5 manipulate circuits with several hundreds primitive components. This clearly makes manual or even interactive proof long and tedious.

2.4 Compilation-by-proof in HOL

This section presents the formalisation in HOL of the concepts introduced in Section 2.1. We use the same principles for hardware modelling and verification presented in the previous section.

2.4.1 The Source Language

Our source language is a subset of HOL which constitutes a tail-recursive first-order functional language.

A function is tail-recursive if its recursive calls are the last operation executed in the body of the function, i.e. there are no operations to be carried out on the results of the recursive calls. For example, the function mult below is tail-recursive.

$$\vdash\ \mathsf{mult}(m, n, acc) = if\ (m = 0w)\ then\ acc\ else\ \mathsf{mult}(m{-}1w, n, acc{+}n)$$

This function manipulates only 32-bit words (type *word32* in HOL). The terms $0w$ and $1w$ represent the numbers 0 and 1, respectively. The function call $\mathsf{mult}(m, n, 0w)$ returns $m \times n$.

A standard definition of mult which is not tail-recursive is shown below.

$$\vdash \mathsf{mult\_standard}(m, n) = \mathit{if}\ (m = 0w)\ \mathit{then}\ 0w\ \mathit{else}\ n + \mathsf{mult\_standard}(m - 1w, n)$$

Tail-recursive functions are of particular interest in hardware compilation because they eliminate the problem of saving the state of the function before the recursive call. In the example above, as $\mathsf{mult}(m-1w, n, acc+n)$ is the last operation to be executed, the compiler will simply connect the arguments $(m-1w, n, acc+n)$ back to the hardware component that implements mult.

The abstract syntax of our source language is shown below.

$$\begin{array}{l} e ::= c \mid x \mid (\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3) \mid f(e_1, \ldots, e_{(\#f)}) \\ p ::= (f_1(x_1, \ldots, x_{(\#f_1)}) = e_1),\ \ldots,\ (f_n(x_1, \ldots, x_{(\#f_n)}) = e_n) \end{array}$$

An expression is either a constant, a variable, a conditional or a function call. The term $\#f$ denotes the arity of f. A program is simply a list of function definitions. A recursive function must have the form:

$$f(x_1, \ldots, x_n) = \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ f(e_3)$$

where e_1, e_2 and e_3 do not contain a call to f. However, these expressions can contain calls to other (recursive) functions provided that they are *not* mutually recursive.

For example, the function fact below uses the multiplier defined above to compute the factorial of a number (whenever the accumulator *acc* is initialised to $1w$).

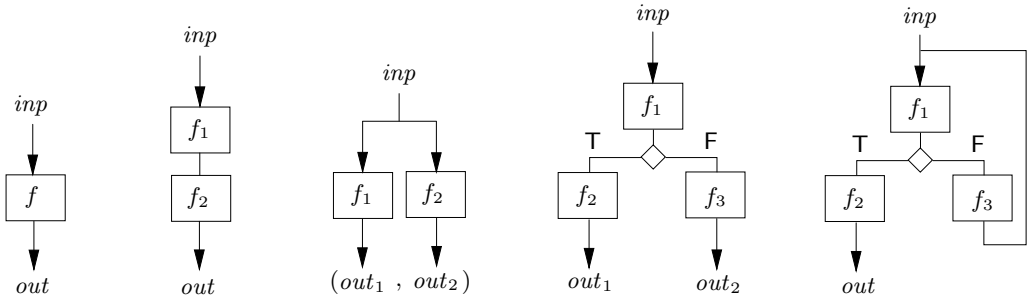
$$\vdash \mathsf{fact}(n, acc) = \mathit{if}\ (n = 0w)\ \mathit{then}\ acc\ \mathit{else}\ \mathsf{fact}(n - 1w, \mathsf{mult}(n, acc, 0w))$$

A list containing the function definitions of mult and fact is an example of a typical program in our source language.

$$\begin{array}{l} (\mathsf{mult}(m, n, acc) = \mathit{if}\ (m = 0w)\ \mathit{then}\ acc\ \mathit{else}\ \mathsf{mult}(m - 1w, n, acc + n)), \\ (\mathsf{fact}(n, acc) = \mathit{if}\ (n = 0w)\ \mathit{then}\ acc\ \mathit{else}\ \mathsf{fact}(n - 1w, \mathsf{mult}(n, acc, 0w))) \end{array}$$

The Intermediate Language

We need to translate the source language into an intermediate form. The compiler transforms the source code to one which contains only *atomic* operators, sequential and parallel compositions, conditional constructors and tail-recursive calls. Atomic operators are the primitive operators of the language, like addition or subtraction. These operators are part of a library and are not built from sub-components (recall the notion of atomic commands given in Section 2.1). The intermediate code is still a functional program, but it reflects the structure of the circuit to be built. The constructors are called Atm (atomic), Seq (sequential), Par (parallel), Ite (`if-then-else`) and Rec (see Figure 2.6).



(a) Atm f (b) Seq f_1 f_2 (c) Par f_1 f_2 (d) Ite f_1 f_2 f_3 (e) Rec f_1 f_2 f_3

Figure 2.6: The intermediate language.

Each box depicted in Figure 2.6 represents a function which is eventually compiled into a hardware block. The constructors are higher-order functions defined as:

$$\begin{array}{lcl}
\vdash \mathsf{Atm}\ f & = & \lambda inp.\ f\ inp \\
\vdash \mathsf{Seq}\ f_1\ f_2 & = & \lambda inp.\ f_2(f_1\ inp) \\
\vdash \mathsf{Par}\ f_1\ f_2 & = & \lambda inp.\ (f_1\ inp,\ f_2\ inp) \\
\vdash \mathsf{Ite}\ f_1\ f_2\ f_3 & = & \lambda inp.\ if\ (f_1\ inp)\ then\ (f_2\ inp)\ else\ (f_3\ inp) \\
\vdash \mathsf{Rec}\ f_1\ f_2\ f_3 & = & \lambda inp.\ if\ (f_1\ inp)\ then\ (f_2\ inp)\ else\ (\mathsf{Rec}\ f_1\ f_2\ f_3\ (f_3\ inp))
\end{array}$$

The atomic constructor acts as the identity function. It is used to identify the primitive operators. The Seq constructor is a simple function composition. The term (Par f_1 f_2) is a function that takes an argument, say v, and produces the pair $(f_1\ v,\ f_2\ v)$. The conditional constructor represents the usual `if-then-else` and the constructor Rec implements a tail-recursive function.

HOL4 automatically translates a source program into its intermediate form. For example, the function word2bool transforms a 32-bit word into a Boolean.

$$\vdash \mathsf{word2bool}\ n = (if\ (n = 0w)\ then\ \mathsf{F}\ else\ \mathsf{T})$$

Its intermediate code is shown below.

$$\begin{array}{l}
\vdash \mathsf{word2bool} = \mathsf{Ite}\ (\mathsf{Seq}\ (\mathsf{Par}\ (\mathsf{Atm}\ \lambda n.\ n) \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{Atm}\ \lambda n.\ 0w)) \\
\qquad\qquad\qquad\qquad\qquad (\mathsf{Atm}\ \lambda(x, y).\ x = y)) \\
\qquad\qquad\qquad\qquad (\mathsf{Atm}\ \lambda n.\ \mathsf{F}) \\
\qquad\qquad\qquad\qquad (\mathsf{Atm}\ \lambda n.\ \mathsf{T})
\end{array}$$

The atomic operators which occur in word2bool are the identity function, the equality and constant generators for zero, true and false. The constructor Ite takes three arguments. The first is the test of the conditional. The parallel constructor produces the pair $(n, 0w)$, which is subsequently sent to the comparator via the sequential constructor. The remaining arguments of Ite are the conditional branches. In the example above, they both generate a Boolean constant. For a slightly more elaborate example, Figure 2.7 shows the intermediate code for the function mult.

```
⊢ mult(m, n, acc) = if (m = 0w) then acc else mult(m−1w, n, acc+n)

⊢ mult = Rec (Seq (Par (Atm λ(m, n, acc). m)
                       (Atm λ(m, n, acc). 0w))
                  (Atm λ(x, y). x = y))
             (Atm λ(m, n, acc).acc)
             (Par (Seq (Par (Atm λ(m, n, acc). m)
                            (Atm λ(m, n, acc). 1w))
                       (Atm λ(x, y). x−y))
                  (Par (Atm λ(m, n, acc).n)
                       (Seq (Par (Atm λ(m, n, acc). acc)
                                 (Atm λ(m, n, acc). n))
                            (Atm λ(x, y). x+y))))
```

Figure 2.7: Intermediate code for mult.

2.4.2 The Specification

Section 2.1 presented a correctness relation between a circuit C and a function f.

$$\vdash \mathsf{C}\ \mathit{implements}\ f$$

In this section we formalise the notion of a circuit implementing a function.

A *device* is a black box which computes some function f via a four-phase handshaking protocol (see Figure 2.8). Its external wires are *load*, *inp*, *done* and *out*. The wires *load* and *done* are control signals; *done* indicates when the device is available and *load* is used by the environment to trigger the device. Data are received and sent over the *inp* and *out* buses, respectively.

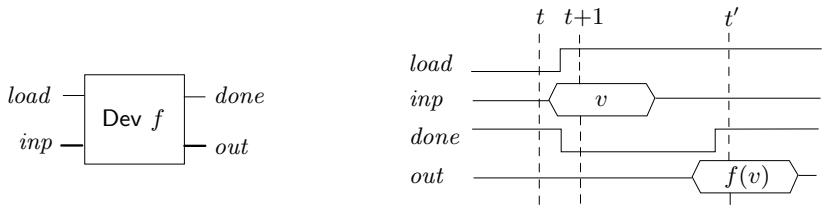


Figure 2.8: Handshaking device.

Figure 2.8 shows a timing diagram of the handshake protocol. At the start of a transaction (say at time t) the device must be outputting T on *done* (to indicate it is ready) and the environment must be asserting F on *load*, i.e. in a state such that a positive edge on *load* can be generated. A transaction is initiated by asserting (at time t+1) the value T on *load*, i.e. *load* has a positive edge at time t+1. This causes the device to read the value, v say, input on *inp* (at time t+1) and to set *done* to F. The device then becomes insensitive to inputs until T is next asserted on *done*, when the computed value $f(v)$ will be output on *out*.

The formal specification of the four-phase handshake protocol is defined by the predicate Dev below, which uses the auxiliary predicates Posedge and HoldF.

A positive edge of a signal is defined as the transition of its value from low to high, i.e. from F to T. There is no positive edge at time zero.

$$\vdash \mathsf{Posedge}\ s\ t\ =\ \mathit{if}\ (t{=}0)\ \mathit{then}\ \mathsf{F}\ \mathit{else}\ (\neg(s\ (t{-}1))\ \wedge\ (s\ t))$$

The formula (HoldF (t_1, t_2) s) says that a signal s holds a low value F during a half-open interval starting at t_1 to just before t_2.

$$\vdash \ \mathsf{HoldF}\ (t_1, t_2)\ s \ = \ \forall t.\ t_1 \leq t < t_2 \ \Rightarrow \ \neg(s\ t)$$

The formula Dev f $(load, inp, done, out)$ specifies the behaviour of the handshaking device computing a function f.

$$\begin{array}{l}
\vdash \ \mathsf{Dev}\ f\ (load, inp, done, out) = \\
\quad (\forall t.\ done\ t\ \wedge\ \mathsf{Posedge}\ load\ (t{+}1) \\
\qquad \Rightarrow \\
\qquad \exists t'.\ t' > t{+}1\ \wedge\ \mathsf{HoldF}\ (t{+}1, t')\ done\ \wedge \\
\qquad\quad done\ t'\ \wedge\ (out\ t' = f(inp\ (t{+}1))))\ \wedge \\
\quad (\forall t.\ done\ t\ \wedge\ \neg(\mathsf{Posedge}\ load\ (t{+}1))\ \Rightarrow\ done\ (t{+}1))\ \wedge \\
\quad (\forall t.\ \neg(done\ t)\ \Rightarrow\ \exists t'.\ t' > t\ \wedge done\ t')
\end{array}$$

The first conjunct in the right-hand side specifies that if the device is available and a positive edge occurs on *load*, then there exists a time t' in future when *done* signals its termination and the output is produced. The value of the output at time t' is the result of applying f to the value of the input at time $t{+}1$. The signal *done* holds the value F during the computation. The second conjunct specifies the situation where no call is made on *load* and the device simply remains idle. Finally, the last conjunct states that if the device is busy, it will eventually finish its computation and become idle. This liveness condition prevents a circuit which constantly outputs F on *done* from satisfying any (Dev f).

Now we can formally state the notion of correctness for a circuit implementing a function. A circuit C implements a device which computes the function f if, whenever the values on the external wires *load*, *inp*, *done* and *out* satisfy the constraints imposed by the circuit, then they also satisfy the constraints imposed by (Dev f).

$$\begin{array}{l}
\vdash \ \mathsf{C}\ implements\ f\ = \\
\quad \forall load\ inp\ done\ out.\ \mathsf{C}(load, inp, done, out)\ \Rightarrow\ \mathsf{Dev}\ f\ (load, inp, done, out)
\end{array}$$

2.4.3 Automatic Verification

As shown in Section 2.1, for each language constructor we have to develop a corresponding circuit constructor and prove its correctness. Let us assume that the circuit constructors ATM, SEQ, PAR, ITE and REC have already been defined (their formal definitions are presented in Section 2.4.4). This section presents the theorems that establish they are correct implementations of the functions Atm, Seq, Par, Ite and Rec.

In order to verify circuits automatically, our theorems must have the same composable structure of the theorems shown in Section 2.1. We prove the following theorems in the HOL4 system.

$$\vdash\ \forall f.\ (\mathsf{ATM}\ f)\ \mathit{implements}\ (\mathsf{Atm}\ f)$$

$$\vdash\ \forall \mathsf{G}\ \mathsf{H}\ g\ h.\ (\mathsf{G}\ \mathit{implements}\ g)\ \wedge\ (\mathsf{H}\ \mathit{implements}\ h)\ \Rightarrow\ (\mathsf{SEQ}\ \mathsf{G}\ \mathsf{H})\ \mathit{implements}\ (\mathsf{Seq}\ g\ h)$$

$$\vdash\ \forall \mathsf{G}\ \mathsf{H}\ g\ h.\ (\mathsf{G}\ \mathit{implements}\ g)\ \wedge\ (\mathsf{H}\ \mathit{implements}\ h)\ \Rightarrow\ (\mathsf{PAR}\ \mathsf{G}\ \mathsf{H})\ \mathit{implements}\ (\mathsf{Par}\ g\ h)$$

$$\vdash\ \forall \mathsf{E}\ \mathsf{G}\ \mathsf{H}\ e\ g\ h.\ (\mathsf{E}\ \mathit{implements}\ e)\ \wedge\ (\mathsf{G}\ \mathit{implements}\ g)\ \wedge\ (\mathsf{H}\ \mathit{implements}\ h)\ \Rightarrow\ (\mathsf{ITE}\ \mathsf{E}\ \mathsf{G}\ \mathsf{H})\ \mathit{implements}\ (\mathsf{Ite}\ e\ g\ h)$$

$$\vdash\ \forall \mathsf{E}\ \mathsf{G}\ \mathsf{H}\ e\ g\ h.\ \mathsf{Total}(e,g,h)\ \Rightarrow\ (\mathsf{E}\ \mathit{implements}\ e)\ \wedge\ (\mathsf{G}\ \mathit{implements}\ g)\ \wedge\ (\mathsf{H}\ \mathit{implements}\ h)\ \Rightarrow\ (\mathsf{REC}\ \mathsf{E}\ \mathsf{G}\ \mathsf{H})\ \mathit{implements}\ (\mathsf{Rec}\ e\ g\ h)$$

Notice that ATM is a parameterised circuit which takes a *function* as argument. For example, the circuit (ATM $(\lambda(x,y).\ x{+}y)$) implements the primitive operator $(\lambda(x,y).\ x{+}y)$. The circuits SEQ, PAR, ITE and REC take *sub-circuits* as arguments. We do not show the proofs here. They are straightforward and are similar to the proof of correctness of the circuit OR_IMP (Section 2.3.1), but they are much longer and tedious to read. These proofs are available in the `examples` directory of the HOL4 CVS repository [81]. The first theorem is the base case of the compilation. The ATM circuit implements handshaking devices that compute the primitive operator f. The theorems for SEQ, PAR and ITE state that if their sub-circuits are correct, then the composite circuit is correct.

The theorem for REC has a pre-condition: the correctness of the circuit (REC E G H) can only be established if the function (Rec $e\ g\ h$) terminates. This is necessary because functions in HOL are total. Termination is characterised by the predicate Total.

$$\vdash\ \mathsf{Total}(e,g,h) = \exists(\mathit{variant} : \alpha \rightarrow \mathit{num}).\ (\forall \mathit{inp}.\ \neg(e\ \mathit{inp})\ \Rightarrow\ \mathit{variant}(h\ \mathit{inp}) < \mathit{variant}(\mathit{inp}))$$

Intuitively, this predicate is based on the fact that if the arguments of a function can be related by a well-founded relation, then the function terminates. The predicate states that if the input is not in the base case, then the recursive call will compute over a new input (h *inp*) which is smaller in some sense than the current input *inp*.

In principle, the restriction of proving termination eliminates the possibility of an automatic verification. The theorem for REC has lost the pure structure of a composable theorem. However, one of the facilities provided by the HOL4 system is the `TFL` package [74], which mechanises the proof of termination with little user guidance — the user must provide only a proper variant.

For example, the compilation of the recursive function mult is done by calling the function `hwDefine`.

```
hwDefine ‘(mult(m:word32,n:word32,acc:word32) =
   if m=0w then acc else mult(m-1w,n,acc+n)) measuring (w2n o FST)‘
```

The function mult is defined for 32-bit words. The variant is given by the function composition (w2n ∘ FST), which takes the first element of the tuple (m, n, acc) and transforms a word into a natural number (the totality is defined in terms of the $<$ relation for natural numbers)[1]. This variant isolates the variable m, which is a key variable for TFL to prove the termination of mult. The function hwDefine automatically returns the theorem below.

$$
\begin{array}{l}
\vdash\ \forall load\ inp\ done\ out.\\
\quad \mathsf{REC}\ (\mathsf{SEQ}\ (\mathsf{PAR}\ (\mathsf{ATM}\ \lambda(m,n,acc).\ m)\ (\mathsf{ATM}\ \lambda(m,n,acc).\ 0w))\\
\qquad\qquad (\mathsf{ATM}\ \lambda(x,y).\ x = y))\\
\qquad (\mathsf{ATM}\ \lambda(m,n,acc).\ acc)\\
\qquad (\mathsf{PAR}\ (\mathsf{SEQ}\ (\mathsf{PAR}\ (\mathsf{ATM}\ \lambda(m,n,acc).\ m)\ (\mathsf{ATM}\ \lambda(m,n,acc).\ 1w))\\
\qquad\qquad\qquad (\mathsf{ATM}\ \lambda(x,y).\ x{-}y))\\
\qquad\qquad (\mathsf{PAR}\ (\mathsf{ATM}\ \lambda(m,n,acc).\ n)\\
\qquad\qquad\qquad (\mathsf{SEQ}\ (\mathsf{PAR}\ (\mathsf{ATM}\ \lambda(m,n,acc).\ acc)\ (\mathsf{ATM}\ \lambda(m,n,acc).\ n))\\
\qquad\qquad\qquad\qquad (\mathsf{ATM}\ \lambda(x,y).\ x{+}y))))\\
\qquad (load, inp, done, out)\ \Rightarrow\ \mathsf{Dev\ mult}\ (load, inp, done, out)
\end{array}
$$

This theorem states that the circuit (REC (SEQ ...) (ATM ...) (PAR ...)) implements a handshaking device which computes the function mult. The proof is done essentially in the same way of the proof shown in Figure 2.1.

2.4.4 The Implementation

This section describes our target language. Following the approach described in Section 2.1, we define circuits which implement each constructor of the intermediate language.

The circuits introduced in this section are defined in terms of the following primitive components.

$$
\begin{array}{l}
\vdash\ \mathsf{AND}\ (in_1, in_2, out)\ = \forall t.\ out\ t = (in_1\ t\ \wedge\ in_2\ t)\\
\vdash\ \mathsf{OR}\ (in_1, in_2, out)\ = \forall t.\ out\ t = (in_1\ t\ \vee\ in_2\ t)\\
\vdash\ \mathsf{NOT}\ (inp, out)\ = \forall t.\ out\ t = \neg(inp\ t)\\
\vdash\ \mathsf{MUX}(sw, in_1, in_2, out)\ =\ \forall t.\ out\ t = if\ (sw\ t)\ then\ (in_1\ t)\ else\ (in_2\ t)\\
\vdash\ \mathsf{DEL}\ (inp, out)\ = \forall t.\ out\ (t{+}1) = inp\ t\\
\vdash\ \mathsf{DELT}\ (inp, out)\ = (out\ 0 = \mathsf{T})\ \wedge\ \forall t.\ out\ (t{+}1) = inp\ t\\
\vdash\ \mathsf{COMB}\ f\ (inp, out)\ = \forall t.\ out\ t = f(inp\ t)
\end{array}
$$

The components AND, OR, NOT and MUX are defined in a standard way. We introduce two sequential components: DEL and DELT. The delays DEL and DELT output the value of the current input one time-unit later. The only difference between them is that DELT outputs T at time zero. The component (COMB f) is a combinational circuit which

[1] In the most recent version of the compiler, we can omit the measuring function in some cases. For instance, mult can be defined by:
`hwDefine ‘mult(m:num,n:num,acc:num) = if m=0 then acc else mult(m-1,n,acc+n)’`
This facility is available to variables of type :*num*, but we still have to extend it to words.

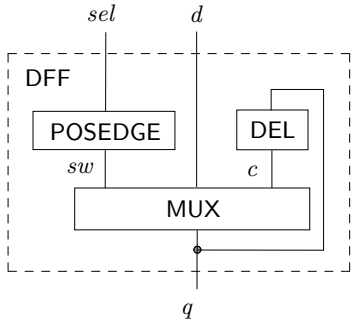


Figure 2.9: The implementation of DFF_SPEC.

applies the function f to the input. This component is used to implement the primitive operators of the language. Note that COMB is a parameterised circuit. A real circuit is only synthesisable after applying COMB to some function, e.g. COMB $(\lambda(x, y).\ (x = y))$. This indicates that $(\lambda(x, y).\ x = y)$ is a primitive operator of the language.

Before presenting the implementations of the intermediate language constructors, we introduce two auxiliary circuits.

The circuit POSEDGE detects a positive edge on the input (see Figure 2.10(a) on page 26).

$$\vdash\ \mathsf{POSEDGE}(inp, out) = \exists c_0\ c_1.\ \mathsf{DELT}(inp, c_0) \wedge \mathsf{NOT}(c_0, c_1) \wedge \mathsf{AND}(c_1, inp, out)$$

The specification DFF_SPEC describes a device which outputs the value of the input whenever there is a positive edge on the signal *sel*.

$$\begin{array}{l}\vdash\ \mathsf{DFF\_SPEC}(d, sel, q)\ = \\ \quad \forall t.\ q\ (t{+}1) = \mathit{if}\ (\mathsf{Posedge}\ sel\ (t{+}1))\ \mathit{then}\ (d\ (t{+}1))\ \mathit{else}\ (q\ t)\end{array}$$

One possible implementation of DFF_SPEC is defined below (see Figure 2.9).

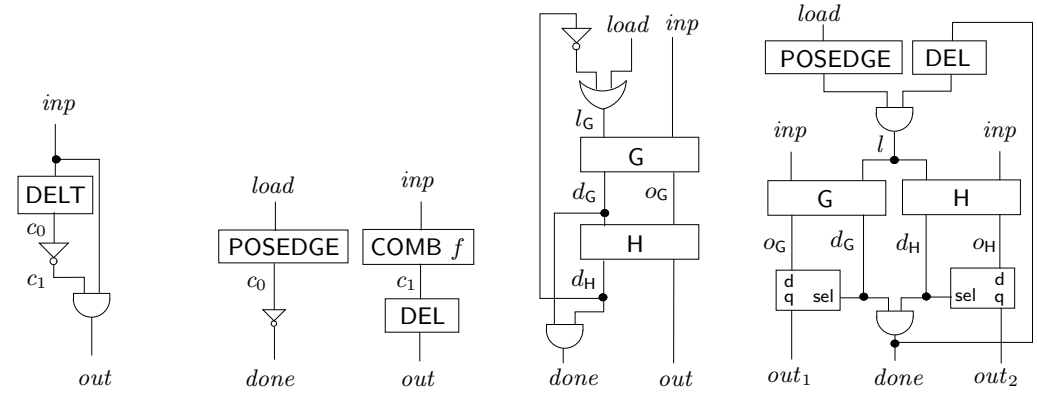
$$\vdash\ \mathsf{DFF}(d, sel, q)\ =\ \exists c\ sw.\ \mathsf{POSEDGE}(sel, sw)\ \wedge\ \mathsf{DEL}(q, c)\ \wedge\ \mathsf{MUX}(sw, d, c, q)$$

It is easy to prove that the circuit DFF is an implementation of the DFF_SPEC.

$$\vdash\ \mathsf{DFF}(d, sel, q)\ \Rightarrow\ \mathsf{DFF\_SPEC}(d, sel, q)$$

Recall that the language constructors Atm, Seq, Par, Ite and Rec presented in Section 2.4.1 are *functions* which perform general computations like sequential execution or recursion. As shown in Section 2.4.2, a circuit implements a function f if it behaves like a handshaking device that computes f. Therefore, we develop, for each language constructor, a corresponding circuit which implements the functionality of the constructor over a handshaking interface. Figures 2.10 and 2.11 show these handshaking circuits, where the local wires l, i, d and o represent the external wires *load*, *inp*, *done* and *out* of a sub-circuit. The *circuits* ATM, SEQ, PAR, ITE and REC implement the *functions* Atm, Seq, Par, Ite and Rec, respectively.

In what follows, we describe the behaviour of these five circuits. All of them satisfy the properties shown in Section 2.4.3.



(a) POSEDGE (b) ATM f (c) SEQ G H (d) PAR G H

Figure 2.10: Implementation of composite devices.

ATM

$$\vdash \mathsf{ATM}\ f\ (load, inp, done, out) = \\ \exists c_0\ c_1.\ \mathsf{POSEDGE}(load, c_0)\ \wedge\ \mathsf{NOT}(c_0, done)\ \wedge \\ \mathsf{COMB}\ f\ (inp, c_1)\ \wedge\ \mathsf{DEL}(c_1, out)$$

We assume that the primitive operators are combinational. The atomic constructor encapsulates the combinational circuit (COMB f) inside a handshaking interface, where f is a primitive operator of the language. The computation of an atomic circuit takes exactly one time-unit. As soon as a positive edge on *load* is detected, *done* outputs F and DEL stores the value of f applied to the current value of *inp*. At the next observable time, the circuit is ready again and *out* returns valid data.

SEQ

$$\vdash \mathsf{SEQ\ G\ H}\ (load, inp, done, out) = \\ \exists c_0\ l_\mathsf{G}\ d_\mathsf{G}\ o_\mathsf{G}\ d_\mathsf{H}. \\ \mathsf{NOT}(d_\mathsf{H}, c_0)\ \wedge\ \mathsf{OR}(c_0, load, l_\mathsf{G})\ \wedge\ \mathsf{G}(l_\mathsf{G}, inp, d_\mathsf{G}, o_\mathsf{G})\ \wedge \\ \mathsf{H}(d_\mathsf{G}, o_\mathsf{G}, d_\mathsf{H}, out)\ \wedge\ \mathsf{AND}(d_\mathsf{G}, d_\mathsf{H}, done)$$

The circuit SEQ takes two sub-circuits G and H and connect them in sequence. For the handshaking protocol of the entire circuit to work, the sub-circuits must also be handshaking circuits. If the sequential computation is not finished, the circuit ignores any calls on *load* (see the OR-gate in Figure 2.10(c)). Actually, all handshaking circuits have this behaviour.

PAR

$$\vdash \mathsf{PAR\ G\ H}\ (load, inp, done, out) = \\ \exists c_0\ c_1\ l\ d_\mathsf{G}\ o_\mathsf{G}\ d_\mathsf{H}\ o_\mathsf{H}. \\ \mathsf{POSEDGE}(load, c_0)\ \wedge\ \mathsf{DEL}(done, c_1)\ \wedge\ \mathsf{AND}(c_0, c_1, l)\ \wedge \\ \mathsf{G}(l, inp, d_\mathsf{G}, o_\mathsf{G})\ \wedge\ \mathsf{H}(l, inp, d_\mathsf{H}, o_\mathsf{H})\ \wedge \\ \mathsf{DFF}(o_\mathsf{G}, d_\mathsf{G}, out_1)\ \wedge\ \mathsf{DFF}(o_\mathsf{H}, d_\mathsf{H}, out_2)\ \wedge \\ \mathsf{AND}(d_\mathsf{G}, d_\mathsf{H}, done)\ \wedge\ (out = \lambda t.\ (out_1\ t, out_2\ t))$$

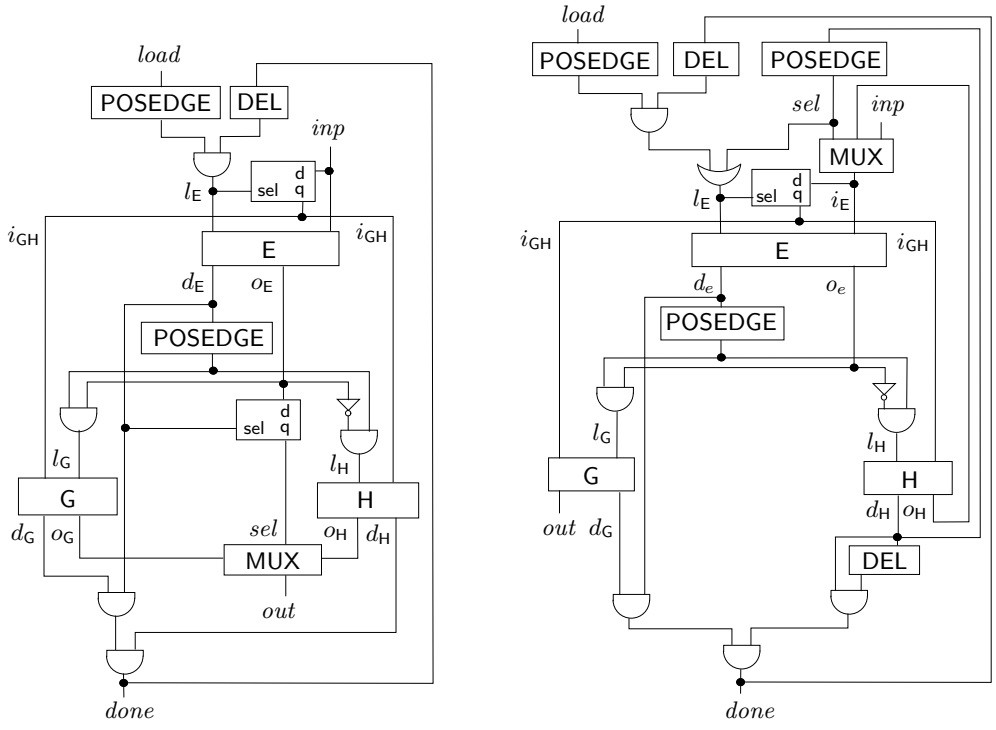
The parallel circuit triggers both of its sub-circuits G and H at the same time (see the wire

l in Figure 2.10(d)). The computation finishes when both sub-circuits finish. In order to output valid results from both sub-circuits, their outputs are stored in DFFs, which are triggered by the positive edge that occurs on d_{G} and d_{H} when the circuits G and H finish their computation.

ITE

$$
\begin{aligned}
\vdash\ & \mathsf{ITE\ E\ F\ G}\ (load, inp, done, out) = \\
& \exists c_0\ c_1\ c_2\ c_3\ c_4\ l_{\mathsf{E}}\ i_{\mathsf{GH}}\ d_{\mathsf{E}}\ o_{\mathsf{E}}\ l_{\mathsf{G}}\ l_{\mathsf{H}}\ d_{\mathsf{G}}\ o_{\mathsf{G}}\ o_{\mathsf{H}}\ d_{\mathsf{H}}\ sel. \\
& \quad \mathsf{POSEDGE}(load, c_0)\ \wedge\ \mathsf{DEL}(done, c_1)\ \wedge\ \mathsf{AND}(c_0, c_1, l_{\mathsf{E}})\ \wedge \\
& \quad \mathsf{E}(l_{\mathsf{E}}, inp, d_{\mathsf{E}}, o_{\mathsf{E}})\ \wedge\ \mathsf{POSEDGE}(d_{\mathsf{E}}, c_2)\ \wedge\ \mathsf{DFF}(o_{\mathsf{E}}, d_{\mathsf{E}}, sel)\ \wedge \\
& \quad \mathsf{DFF}(inp, l_{\mathsf{E}}, i_{\mathsf{GH}})\ \wedge\ \mathsf{AND}(c_2, o_{\mathsf{E}}, l_{\mathsf{G}})\ \wedge\ \mathsf{NOT}(o_{\mathsf{E}}, c_3)\ \wedge \\
& \quad \mathsf{AND}(c_2, c_3, l_{\mathsf{H}})\ \wedge\ \mathsf{G}(l_{\mathsf{G}}, i_{\mathsf{GH}}, d_{\mathsf{G}}, o_{\mathsf{G}})\ \wedge\ \mathsf{H}(l_{\mathsf{H}}, i_{\mathsf{GH}}, d_{\mathsf{H}}, o_{\mathsf{H}})\ \wedge \\
& \quad \mathsf{MUX}(sel, o_{\mathsf{G}}, o_{\mathsf{H}}, out)\ \wedge\ \mathsf{AND}(d_{\mathsf{G}}, d_{\mathsf{E}}, c_4)\ \wedge\ \mathsf{AND}(c_4, d_{\mathsf{H}}, done)
\end{aligned}
$$

The *if-then-else* circuit ITE takes three sub-circuits E, G and H as arguments. The circuit E tests the condition and the circuits G and H compute the conditional branches. The result of the test is stored in a DFF connected to a multiplexer (Figure 2.11(a)).



(a) ITE E G H (b) REC E G H

Figure 2.11: The conditional and the recursive constructors.

REC

$$\begin{aligned}\vdash\ & \mathsf{REC\ E\ F\ G}\ (load, inp, done, out) = \\ & \exists c_0\ c_1\ c_2\ c_3\ c_4\ c_5\ c_6\ c_7\ sel\ l_{\mathsf{E}}\ i_{\mathsf{E}}\ i_{\mathsf{GH}}\ d_{\mathsf{E}}\ o_{\mathsf{E}}\ l_{\mathsf{G}}\ l_{\mathsf{H}}\ d_{\mathsf{G}}\ d_{\mathsf{H}}\ o_{\mathsf{H}}. \\ & \quad \mathsf{POSEDGE}(load, c_0)\ \wedge\ \mathsf{DEL}(done, c_1)\ \wedge\ \mathsf{AND}(c_0, c_1, c_2)\ \wedge \\ & \quad \mathsf{OR}(c_2, sel, l_{\mathsf{E}})\ \wedge\ \mathsf{POSEDGE}(d_{\mathsf{H}}, sel)\ \wedge\ \mathsf{MUX}(sel, o_{\mathsf{H}}, inp, i_{\mathsf{E}})\ \wedge \\ & \quad \mathsf{DFF}(i_{\mathsf{E}}, l_{\mathsf{E}}, i_{\mathsf{GH}})\ \wedge\ \mathsf{E}(l_{\mathsf{E}}, i_{\mathsf{E}}, d_{\mathsf{E}}, o_{\mathsf{E}})\ \wedge\ \mathsf{POSEDGE}(d_{\mathsf{E}}, c_3)\ \wedge \\ & \quad \mathsf{AND}(c_3, o_{\mathsf{E}}, l_{\mathsf{G}})\ \wedge\ \mathsf{NOT}(o_{\mathsf{E}}, c_4)\ \wedge\ \mathsf{AND}(c_4, c_3, l_{\mathsf{H}})\ \wedge\ \mathsf{G}(l_{\mathsf{G}}, i_{\mathsf{GH}}, d_{\mathsf{G}}, out)\ \wedge \\ & \quad \mathsf{H}(l_{\mathsf{H}}, i_{\mathsf{GH}}, d_{\mathsf{H}}, o_{\mathsf{H}})\ \wedge\ \mathsf{DEL}(d_{\mathsf{H}}, c_5)\ \wedge\ \mathsf{AND}(d_{\mathsf{H}}, c_5, c_7)\ \wedge\ \mathsf{AND}(d_{\mathsf{G}}, d_{\mathsf{E}}, c_6)\ \wedge \\ & \quad \mathsf{AND}(c_6, c_7, done)\end{aligned}$$

The recursive function is very similar to the conditional one. It also takes three circuits E, G and H as arguments. The only difference is that it connects the output of H to the input of E to implement the tail-recursion. A delay is connected to the *done* signal of H in order to distinguish a recursive call from the complete termination of the computation (Figure 2.11(b)).

2.5 Summary

This chapter presented the main ideas related to the compilation-by-proof method. We started by introducing the notion of automatic verification via composable theorems, followed by the introduction of higher-order logic and hardware verification.

Then, the source language was presented together with an intermediate language, which is the one manipulated by the compiler. This language structures the source code in terms of the composable functions Atm, Seq, Par, Ite and Rec.

For each of these constructors, we defined five circuit constructors which implement them: ATM, SEQ, PAR, ITE and REC.

The notion of correctness for circuits implementing functions is given via the concept of a handshaking device which computes a function. We showed that each circuit constructor is correct, provided that their sub-circuits are correct. As usual in verification tasks, proofs of correctness are not mathematically challenging, but rather long and tedious. Our proofs take nearly 3,000 lines of ML [81] and took several months of work.

As the main aim of this chapter is to introduce the principles and concepts involved in our compilation, we have not addressed issues like clocked circuits generation or optimisations. These steps are presented in the next chapter.

Chapter 3

Optimisations and Synthesis

Chapter 2 focused on the fundamental concepts underlying our compiler. However in order to deal with more realistic designs, we implement optimisations and integrate HOL4 with external tools for FPGAs. This chapter describes these features.

The optimisations aim to reduce the size of a circuit and its execution time. Section 3.1 presents four optimisations introduced to the compiler.

In order to run our circuit, we transform it into a clocked synchronous circuit, translate it to Verilog and download it to an FPGA. Section 3.2 describes how to introduce a clock signal using time refinement. The translation of clocked circuits to Verilog and the integration of HOL4 to an FPGA are presented in Section 3.3. The last section shows a simple example which illustrates all the steps of the compilation.

3.1 Optimisations

In what follows we introduce four optimisations: the circuits PRECEDE and FOLLOW, the extension of the primitive constructors, *whole program compaction* and let-expressions.

PRECEDE and FOLLOW

In the last chapter we defined handshaking circuits for every constructor of the language. In particular, the atomic constructors implement a handshaking interface to encapsulate a combinational circuit[1]. We can eliminate some of these internal handshakes by directly connecting a combinational part to another circuit.

We create two circuit constructors which connects a combinational component in sequence to another circuit without introducing a handshake. These constructors are introduced during the compilation of (Seq g h), where g or h is combinational.

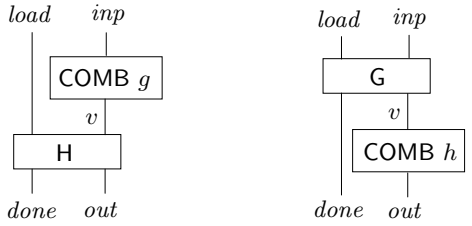
$$\begin{aligned} \vdash\ & \mathsf{PRECEDE}\ g\ \mathsf{H}\ (load, inp, done, out)\ = \\ & \quad \exists v.\ \mathsf{COMB}\ g\ (inp, v)\ \wedge\ \mathsf{H}(load, v, done, out) \end{aligned}$$

$$\begin{aligned} \vdash\ & \mathsf{FOLLOW}\ \mathsf{G}\ h(load, inp, done, out)\ = \\ & \quad \exists v.\ \mathsf{G}(load, inp, done, v)\ \wedge\ \mathsf{COMB}\ h\ (v, out) \end{aligned}$$

The circuit (PRECEDE g H) connects the combinational circuit (COMB g) to the *input*

[1]Combinational circuits are those whose output is a function of the present input only.

of circuit H, while (FOLLOW G h) connects (COMB h) to the *output* of circuit G (see Figure 3.1).



(a) PRECEDE g H (b) FOLLOW G h

Figure 3.1: PRECEDE and FOLLOW.

In order to include these constructors in the compilation, we have to prove their correctness:

$$\vdash \ (\mathsf{H}\ \mathit{implements}\ h) \ \Rightarrow \ ((\mathsf{PRECEDE}\ g\ \mathsf{H})\ \mathit{implements}\ (\mathsf{Seq}\ g\ h))$$
$$\vdash \ (\mathsf{G}\ \mathit{implements}\ g) \ \Rightarrow \ ((\mathsf{FOLLOW\ G}\ h)\ \mathit{implements}\ (\mathsf{Seq}\ g\ h))$$

This technique is applied when either g or h in (Seq g h) is combinational. If both are combinational, then the following optimisation is more effective.

Extending the primitive operators

The same principle of eliminating handshakes which motivated the creation of PRECEDE and FOLLOW can be applied to the constructors Seq, Par and Ite. If we include these constructors in the library of the *primitive operators* of the language, the compiler eliminates their handshakes provided that their arguments are primitive operators (or are built from primitive operators only). If they are not combinational, the compiler generates the usual handshake circuits SEQ, PAR and ITE. See below the transformations performed by the compiler assuming that e, g and h are primitive operators of the language or can be recursively reduced to a netlist of primitive operators only.

The sequential composition is transformed into two combinational circuits in series.

$$\begin{array}{l}\vdash\ \mathsf{COMB}\ (\mathsf{Seq}\ g\ h)\ (\mathit{inp}, \mathit{out}) \ = \\ \quad \exists c.\ \mathsf{COMB}\ g\ (\mathit{inp}, c)\ \wedge\ \mathsf{COMB}\ h\ (c, \mathit{out})\end{array}$$

The parallel composition is broken into two different combinational components.

$$\begin{array}{l}\vdash\ \mathsf{COMB}\ (\mathsf{Par}\ g\ h)\ (\mathit{inp}, \mathit{out}_1 \diamond \mathit{out}_2) \ = \\ \quad \mathsf{COMB}\ g\ (\mathit{inp}, \mathit{out}_1)\ \wedge\ \mathsf{COMB}\ h\ (\mathit{inp}, \mathit{out}_2)\end{array}$$

Note that the output of (COMB (Par g h)) is a concatenation of two signals, namely out_1 and out_2. We concatenate signals by using the operator $\diamond$.

$$\vdash\ s_1 \diamond s_2 = (\lambda t.\ (s_1\ t,\ s_2\ t))$$

The conditional operator is implemented by a multiplexer.

$$\begin{array}{l}\vdash\ \mathsf{COMB}\ (\mathsf{Ite}\ e\ g\ h)\ (\mathit{inp}, \mathit{out}) \ = \\ \quad \exists c_0\ c_1\ c_2.\ \mathsf{COMB}\ e\ (\mathit{inp}, c_0)\ \wedge\ \mathsf{COMB}\ g\ (\mathit{inp}, c_1)\ \wedge \\ \qquad\qquad \mathsf{COMB}\ h\ (\mathit{inp}, c_2)\ \wedge\ \mathsf{MUX}(c_0, c_1, c_2, \mathit{out})\end{array}$$

These optimisations connect as many combinational components as possible in order to reduce the amount of handshakes. In addition to the transformations presented above, the process of reducing a circuit to a netlist of primitive operators makes use of auxiliary transformations.

For example, assuming that $(\lambda n.\ n)$ and $(\lambda n.\ 1w)$ are defined as primitive operators, the compiler generates the following simplifications.

$$\vdash \mathsf{COMB}\ (\lambda n.\ n)\ (inp, out) = (inp = out)$$
$$\vdash \mathsf{COMB}\ (\lambda n.\ 1w)\ (inp, out) = \mathsf{CONSTANT}\ 1w\ out$$

where CONSTANT is a primitive hardware component defined as

$$\vdash \mathsf{CONSTANT}\ v\ out = \forall t.\ out\ t = v$$

Whole program compaction

A source program is a list of functions. Any optimisation like the ones performed above is restricted to the scope of a function.

Whole program compaction or *inline expansion* replaces every function call by the body of the function. This produces a monolithic source program defined by a single function, namely the function main. This pre-compilation is performed before any optimisation step.

The optimisations can now have a global view of the program. For example, we are now able to introduce PRECEDE and FOLLOW between circuits that were initially allocated to different functions. Without whole program compaction these circuits had to necessarily communicate via a handshake.

Let-expressions

This is an idea by Scott Owens from the University of Utah. In our compilation, whenever we have the same function being called more than once, several instances of its corresponding circuit are generated for each call. Although this approach produces a fast circuit (as no hardware block is shared and no arbiter is needed), it also produces a large circuit.

One solution that can give more flexibility to the compiler is the introduction of let-expressions. A let-expression has the form (let $v = e_1$ in e_2) where v is a variable structure, i.e. it is either a single variable or, recursively, a non-empty tuple of "varstructs" (e.g. $(x, (m, n), y)$). How a let-expression is implemented as a circuit is shown below.

$$\vdash (\lambda inp.\ \mathsf{let}\ v = (f_1\ inp)\ \mathsf{in}\ (f_2\ (inp, v))) = \mathsf{Seq}\ (\mathsf{Par}\ (\lambda x.\ x)\ f_1)\ f_2$$

The designer can replace several calls to the same function by a single call. For example, the program below shows how three calls to the function inc are replaced by a single call using the variable y, thus preventing three circuit implementations of inc from being generated.

$$\vdash \mathsf{inc}\ n = n{+}1w$$
$$\vdash \mathsf{main}\ n = \mathsf{let}\ y = (\mathsf{inc}\ n)\ \mathsf{in}\ (y{+}y{+}y)$$

We can still optimise the compilation of let-expressions. In the expression (let $v = e_1$ in e_2), if e_1 is combinational, then the expression is synthesised into a circuit consisting of e_1 driving wires corresponding to v that are inputs to the circuit corresponding to e_2. If e_1 is

not combinational, the usual compilation in terms of Seq and Par takes place. In the example above, if inc is not declared combinational, then main is compiled to a circuit with 30 components (10 of which are registers) and 28 variables, but if inc is declared to be combinational, then main compiles to a circuit with 7 variables and 9 components (2 of which are registers).

Results
The optimisations presented above address two issues: the elimination of handshakes and the elimination of circuit duplication. The handshake elimination can sometimes produce aggressive optimisations in the sense that almost all handshakes are eliminated. The only exception is for handshakes that appear in the circuit REC. The optimisations reduce the size of the circuit and, by eliminating handshakes, minimise the clock ticks per computation. However there is a trade-off here as they also generate long combinational paths and eventually force the clock to slow down (the next section shows how the clock signal is introduced).

The user has control over most of the optimisations. Whole program compaction can be applied by using an alternative compiler called `inlineCompile`. A let-expression is just an extra language feature available to the user, and the primitive constructors of the language can be defined by the user, although the compiler initialises a default library which includes Seq, Par and Ite.

We tried our optimised compiler with arithmetic and cryptographic hardware [76]. The arithmetic example is a version of the Booth Multiplier (see Section 5.1). The optimisations reduce the circuit size of the multiplier in 32%. The cryptographic hardware implements an encryption algorithm called TEA [82]. In this case, the circuit had a 50% reduction. It is not possible to tell precisely how much reduction is expected because the optimisations depend on the structure of the source program.

3.2 Clock Introduction

This section presents the concepts and technology developed by Tom Melham on time abstractions [55] and shows how we use them to generate clocked circuits.

First let us recall the definitions of the sequential components DEL and DELT introduced in Chapter 2.

$$\vdash \mathsf{DEL}\ (inp, out) \ = \ \forall t.\ out\ (t{+}1) = inp\ t$$
$$\vdash \mathsf{DELT}\ (inp, out) \ = \ (out\ 0 = \mathsf{T}) \ \wedge \ \forall t.\ out\ (t{+}1) = inp\ t$$

These definitions describe components which output the value of the current input one time unit later. Note that these sequential circuits are abstract components which can be implemented, for instance, by rising edge-triggered Dtypes. In this case, DEL and DELT are regarded as specifications in the 'cycle level' timescale which abstract away points of time which have no rising edge of a clock triggering the Dtypes.

In order to create an implementation of DEL and DELT using edge-triggered Dtypes we need to refine their behaviour to a finer-grained timescale which allows us to represent clock edges. The Dtypes are modelled at a concrete or finer-grained timescale (see Figure 3.2).

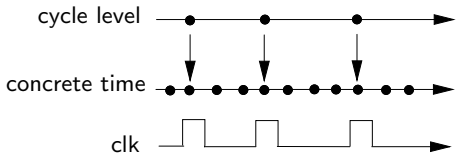


Figure 3.2: Time projection.

Before introducing the specification of a Dtype, we define the predicate Rise, which captures the notion of a rising edge of a clock.

$$\vdash \ \mathsf{Rise}\ s\ t = \neg s(t)\ \wedge\ s(t{+}1)$$

Note that this is similar to Posedge, except that (Posedge s 0) is always F and (Rise s) is defined to be true just before a rising edge, whereas (Posedge s) is true just after the edge. Also Posedge is used at the 'cycle level' timescale, but Rise is used at the concrete timescale. In this section, the variable t always represents time in the fine-grained concrete timescale.

The component DTYPE is similar to the DFF, but operates in the concrete timescale.

$$\begin{aligned}\vdash \ \mathsf{DTYPE}\ v\ (clk, d, q)\ =\ & (q\ 0 = v)\ \wedge \\ & (\forall t.\ q\ (t{+}1) = \mathit{if}\ (\mathsf{Rise}\ clk\ t)\ \mathit{then}\ (d\ t)\ \mathit{else}\ (q\ t))\end{aligned}$$

The predicate DTYPE represents a parameterised circuit which takes as argument the initial value of the output and stores the value of the input at every rising edge of the clock.

Now we can refine our sequential components and implement them using DTYPEs. The following theorems are proved following Melham's book [55].

$$\vdash\ (\mathsf{InfRise}\ clk)\ \Rightarrow\ \forall d\ q.\ (\exists v.\ \mathsf{DTYPE}\ v\ (clk, d, q))\ \Rightarrow\ \mathsf{DEL}(d\,@\,clk,\ q\,@\,clk)$$
$$\vdash\ (\mathsf{InfRise}\ clk)\ \Rightarrow\ \forall d\ q.\ \mathsf{DTYPE}\ \mathsf{T}\ (clk, d, q)\ \Rightarrow\ \mathsf{DELT}(d\,@\,clk,\ q\,@\,clk)$$

The term $(s\,@\,clk)$ is a signal whose sequence of values is the result of sampling s at the rising edges of clk. The theorems state that a DTYPE (running in the concrete time) is an implementation of DEL (or DELT) if we observe it at the rising edges of clk. The term (InfRise clk) asserts that clk has infinitely many rising edges. The formal definitions of the @ operator and InfRise can be seen in Melham [55].

Combinational components are still the same in the concrete timescale. Although the concrete time is in a finer-grained timescale, a model in which we abstract away delays can still be considered accurate.

$$\vdash\ \mathsf{AND}(in_1, in_2, out)\ \Rightarrow\ \mathsf{AND}(in_1\,@\,clk,\ in_2\,@\,clk,\ out\,@\,clk)$$
$$\vdash\ \mathsf{OR}(in_1, in_2, out)\ \Rightarrow\ \mathsf{OR}(in_1\,@\,clk,\ in_2\,@\,clk,\ out\,@\,clk)$$
$$\vdash\ \mathsf{NOT}(inp, out)\ \Rightarrow\ \mathsf{NOT}(inp\,@\,clk,\ out\,@\,clk)$$
$$\vdash\ \mathsf{MUX}(sw, in_1, in_2, out)\ \Rightarrow\ \mathsf{MUX}(sw\,@\,clk,\ in_1\,@\,clk,\ in_2\,@\,clk,\ out\,@\,clk)$$
$$\vdash\ \mathsf{COMB}\ f\ (inp, out)\ \Rightarrow\ \mathsf{COMB}\ f\ (inp\,@\,clk,\ out\,@\,clk)$$

In what follows we show how these theorems are introduced in the compilation process. In the last chapter we saw that the compiler returns a circuit C whose components are defined in the cycle level timescale.

$$\vdash\ \forall load\ inp\ done\ out.\ \mathsf{C}(load, inp, done, out)\ \Rightarrow\ \mathsf{Dev}\ f\ (load, inp, done, out)$$

The compilation continues by instantiating *load*, *inp*, *done* and *out* to $(load \mathbin{@} clk)$, $(inp \mathbin{@} clk)$, $(done \mathbin{@} clk)$ and $(out \mathbin{@} clk)$, respectively, and then by performing some deductions using the above theorems and the monotonicity of existential quantification and conjunction with respect to implication. The final theorem is shown below.

$$\begin{array}{l}\vdash\ \forall load\ inp\ done\ out.\\ \quad (\mathsf{InfRise}\ clk)\\ \quad \Rightarrow\ \mathsf{C_{clk}}(load, inp, done, out)\\ \quad\quad \Rightarrow\ \mathsf{Dev}\ f\ (load \mathbin{@} clk,\ inp \mathbin{@} clk,\ done \mathbin{@} clk,\ out \mathbin{@} clk)\end{array}$$

The term $\mathsf{C_{clk}}$ represents the circuit with combinational components and clocked DTYPEs only. The final theorem shows that if we observe the clocked circuit $\mathsf{C_{clk}}$ at the rising edges of *clk*, then it behaves like a handshaking device which computes the function f.

Clock introduction is carried out fully automatically by proof in HOL4.

3.3 Translating HOL Circuits to Verilog

The clocked synchronous hardware generated in HOL4 is 'pretty-printed' to Verilog. There are no formal methods involved in this step. Our main motivation for doing this translation was to be able to download and run our circuits in an FPGA.

As circuits in HOL are already in a register transfer level, the translation is straightforward. A DTYPE is defined in Verilog as:

```
module dtype (clk,d,q);
   parameter size = 31;
   parameter value = 1;
   input clk;
   input  [size:0] d;
   output [size:0] q;
   reg    [size:0] q = value;
   always @(posedge clk) q <= d;
endmodule
```

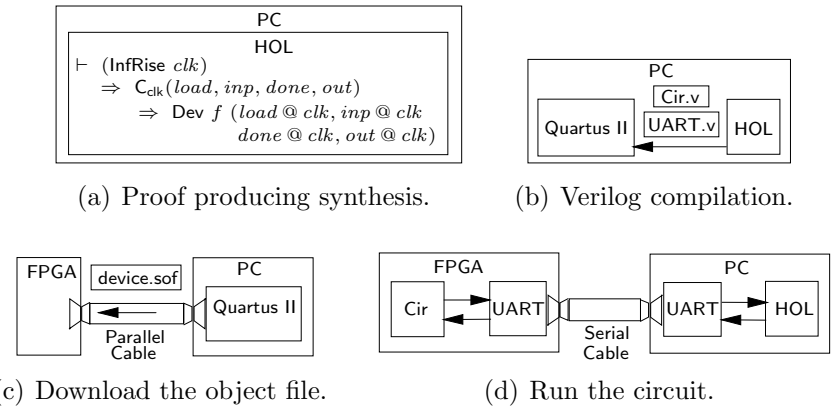
The `dtype` module allows us to change the size of the data and its initial value during its instantiation. The parameters `size` and `value` above are defined with the default values 31 and 1, respectively. Notice that in order to implement DEL and DELT, the DTYPEs occur in our circuits in two forms: $(\exists v.\ \mathsf{DTYPE}\ v\ (clk, d, q))$ and $(\mathsf{DTYPE}\ \mathsf{T}\ (clk, d, q))$. The implementation in Verilog above initialises the D-type registers with 1, thus satisfying both kinds of DTYPEs (as $(\exists v.\ \mathsf{DTYPE}\ v\ (clk, d, q))$ can be initialised with any value).

The remaining components are implemented using the primitive operators of Verilog. For example, the circuit $(\mathsf{AND}(in_1, in_2, c)\ \wedge\ \mathsf{NOT}(c, out))$ is implemented as:

```
wire in1; wire in2, wire c; wire out;
assign c = in1 && in2;
assign out = ~ c;
```

Once HOL4 generates a Verilog file, it is possible to simulate our designs or run them in an FPGA. We have used an Altera FPGA (Excalibur) and the software Quartus II to compile, download and run the circuits. For simulations, we used GPL-Cver [31]

(an open-source Verilog simulator) and GTKWave [33] (a wave viewer). We developed a package of functions in HOL4 to interface with Quartus II and the FPGA. The four steps for programming the FPGA and running our circuits are shown diagrammatically in Figure 3.3.



(a) Proof producing synthesis. (b) Verilog compilation.

(c) Download the object file. (d) Run the circuit.

Figure 3.3: The design process.

(a) Proof Producing Synthesis. The first step is the automatic proof of the certifying theorem

$$\begin{aligned}\vdash\ & \forall load\ inp\ done\ out.\\ & (\mathsf{InfRise}\ clk)\\ & \Rightarrow\ \mathsf{C_{clk}}(load, inp, done, out)\\ & \quad\Rightarrow\ \mathsf{Dev}\ f\ (load\,@\,clk,\ inp\,@\,clk,\ done\,@\,clk,\ out\,@\,clk)\end{aligned}$$

This step has been covered mostly in the previous chapter.

(b) Verilog compilation. The pretty-printer translates the circuit $\mathsf{C_{clk}}$ to Verilog and saves it in a file (called Cir.v in Figure 3.3(b)). In order to communicate with the FPGA, we developed the Verilog program UART.v which connects our devices to a UART interface available on the FPGA board.[2] This program is *not* formally verified. Both files are sent to Quartus II for compilation (by "compilation" we mean timing analysis, routing, fitting and generation of an object file used to program the FPGA).

(c) Download the object file. Quartus II generates the object file `device.sof` and uses it to program the FPGA via a parallel cable.

(d) Run the circuit. After programming the FPGA, we can now run the circuit. We developed a C program that communicates with the serial cable connected to the FPGA. This program can actually be accessed from Moscow ML (i.e., from HOL4), thanks to an ML code provided by Ken Friis Larsen. After the FPGA is programmed, we generate a file which contains an ML function which calls the circuit. The user has to load this file (using the ML command "use") in order to be able to run the circuit interactively.

[2]This code was adapted from the micro-UART program developed by Jeung Joon Lee. `http://www.cmosexod.com/micro_uart.htm`

3.4 A Simple Example

This section presents a simple example to illustrate the entire design process step-by-step. Our source program is a simple function that increments a 32-bit word by one.

$$\vdash \mathsf{Inc}\ n = n{+}1w$$

First the compiler transforms the source code to the intermediate language.

$$\vdash \mathsf{Inc} = \mathsf{Seq}\ (\mathsf{Par}\ (\lambda n.\ n)\ (\lambda n.\ 1w))\ (\lambda(x,y).\ x{+}y)$$

Let us now assume that $(\lambda n.\ n)$, $(\lambda n.\ 1w)$ and $(\lambda(x,y).\ x{+}y)$ are the only primitive operators of the language. In this case, the compilation produces the theorem:

$$\begin{array}{l}\vdash \mathsf{FOLLOW}\ (\mathsf{PAR}\ (\mathsf{ATM}\ \lambda n.\ n)\ (\mathsf{ATM}\ \lambda n.\ 1w))\\ \qquad\qquad (\lambda(x,y).\ x{+}y)\ (load, inp, done, out)\\ \quad \Rightarrow\ \mathsf{Dev\ Inc}\ (load, inp, done, out)\end{array}$$

This circuit contains three handshakes: one for the PAR and two for the ATMs.

We can still reduce the number of handshakes by considering Seq and Par primitive operators of the language. In this case, the compiler implements all functions as combinational.

$$\begin{array}{l}\vdash \mathsf{ATM}\ (\mathsf{Seq}\ (\mathsf{Par}\ (\lambda n.\ n)\ (\lambda n.\ 1w))\ (\lambda(x,y).\ x{+}y))\ (load, inp, done, out)\\ \quad \Rightarrow\ \mathsf{Dev\ Inc}\ (load, inp, done, out)\end{array}$$

Recall that it is not always possible to implement a Seq or Par as combinational hardware. If their arguments are non-primitive operators (like Rec), then the handshaking circuits SEQ or PAR are generated. However, in the example above all the elements are reducible to a netlist of primitive operators.

Replacing the definition of ATM gives

$$\begin{array}{l}\vdash\ (\exists v_0\ v_1.\ (\exists v_2\ v_3.\ \mathsf{DELT}(load, v_2) \wedge\ \mathsf{NOT}(v_2, v_3) \wedge\ \mathsf{AND}(v_3, load, v_0)) \wedge\\ \qquad\qquad \mathsf{NOT}(v_0, done) \wedge\\ \qquad\qquad \mathsf{COMB}\ (\mathsf{Seq}\ (\mathsf{Par}\ (\lambda n.\ n)\ (\lambda n.\ 1w))\ (\lambda(x,y).\ x{+}y))\ (inp, v_1) \wedge\\ \qquad\qquad \mathsf{DEL}(v_1, out))\\ \quad \Rightarrow\ \mathsf{Dev\ Inc}\ (load, inp, done, out)\end{array}$$

The sub-term (COMB ...) is reduced to a netlist of primitive operators by applying the theorems shown in Section 3.1 (see the sub-section *extending the primitive operators*).

$$\begin{array}{ll} & \mathsf{COMB}\ (\mathsf{Seq}\ (\mathsf{Par}\ (\lambda n.\ n)\ (\lambda n.\ 1w))\ (\lambda(x,y).\ x{+}y))\ (inp, v_1)\\ = & \exists v_3\ v_4.\ \mathsf{COMB}\ (\mathsf{Par}\ (\lambda n.\ n)\ (\lambda n.\ 1w))\ (inp, v_3 \diamond v_4) \wedge\\ & \qquad \mathsf{COMB}\ (\lambda(x,y).\ x{+}y)\ (v_3 \diamond v_4, v_1)\\ = & \exists v_3\ v_4.\ \mathsf{COMB}\ (\lambda n.\ n)\ (inp, v_3)\ \wedge\ \mathsf{COMB}\ (\lambda n.\ 1w)\ (inp, v_4) \wedge\\ & \qquad \mathsf{COMB}\ (\lambda(x,y).\ x{+}y)\ (v_3 \diamond v_4, v_1)\\ = & \exists v_3\ v_4.\ (inp = v_3)\ \wedge\ \mathsf{CONSTANT}\ 1w\ (inp, v_4) \wedge\\ & \qquad \mathsf{COMB}\ (\lambda(x,y).\ x{+}y)\ (v_3 \diamond v_4, v_1)\\ = & \exists v_4.\ \mathsf{CONSTANT}\ 1w\ (inp, v_4)\ \wedge\ \mathsf{COMB}\ (\lambda(x,y).\ x{+}y)\ (inp \diamond v_4, v_1)\end{array}$$

The first step serialises the arguments of Seq. Note that as the parallel constructor takes a concatenated signal as argument, the variables v_3 and v_4 are introduced. The second step simplifies the parallel composition, followed by the elimination of the primitive operator $(\lambda n.\ n)$ and the introduction of the primitive circuit CONSTANT. The final step eliminates the variable v_3.

After the rewriting, the compiler proves

$$\begin{array}{l}\vdash \exists v_0\ v_1\ v_2\ v_3\ v_4.\\ \qquad \mathsf{DELT}(load, v_2)\ \wedge\ \mathsf{NOT}(v_2, v_3)\ \wedge\ \mathsf{AND}(v_3, load, v_0)\ \wedge\\ \qquad \mathsf{NOT}(v_0, done)\ \wedge\ \mathsf{CONSTANT}\ 1w\ v_4\ \wedge\\ \qquad \mathsf{COMB}\ (\lambda(x,y).\ x{+}y)\ (inp \diamond v_4, v_1)\ \wedge\ \mathsf{DEL}(v_1, out))\\ \quad \Rightarrow\ \mathsf{Dev\ Inc}\ (load, inp, done, out)\end{array}$$

The next step prepares the circuit for the introduction of the clock signal. The compiler eliminates the existential quantifier and specialises the variables with $(@\ clk)$.

$$\begin{array}{l}\vdash \mathsf{DELT}(load\ @\ clk, v_2\ @\ clk)\ \wedge\ \mathsf{NOT}(v_2\ @\ clk, v_3\ @\ clk)\ \wedge\\ \mathsf{AND}(v_3\ @\ clk, load\ @\ clk, v_0\ @\ clk)\ \wedge\ \mathsf{NOT}(v_0\ @\ clk, done\ @\ clk)\ \wedge\\ \mathsf{CONSTANT}\ 1w\ (v_4\ @\ clk)\ \wedge\ \mathsf{COMB}\ (\lambda(x,y).\ x{+}y)\ ((inp \diamond v_4)\ @\ clk, v_1\ @\ clk)\ \wedge\\ \mathsf{DEL}(v_1\ @\ clk, out\ @\ clk))\\ \quad \Rightarrow\ \mathsf{Dev\ Inc}\ (load\ @\ clk, inp\ @\ clk, done\ @\ clk, out\ @\ clk)\end{array}$$

By using the theorems presented in Section 3.2, we replace all sequential components operating at cycle level timescale by their implementations based on DTYPEs. The final theorem is shown below.

$$\begin{array}{l}\vdash (\mathsf{InfRise}\ clk)\\ \quad \Rightarrow\ (\exists v_0\ v_1\ v_2\ v_3\ v_4.\\ \qquad \mathsf{DTYPE\ T}\ (clk, load, v_2)\ \wedge\ \mathsf{NOT}(v_2, v_3)\ \wedge\\ \qquad \mathsf{AND}(v_3, load, v_0)\ \wedge\ \mathsf{NOT}(v_0, done)\ \wedge\\ \qquad \mathsf{CONSTANT}\ 1w\ v_4\ \wedge\ \mathsf{COMB}\ (\lambda(x,y).\ x{+}y)\ (inp \diamond v_4, v_1)\ \wedge\\ \qquad (\exists v.\ \mathsf{DTYPE}\ v\ (clk, v_1, out)))\\ \quad\ \Rightarrow\ \mathsf{DEV\ Inc}\ (load\ @\ clk, inp\ @\ clk, done\ @\ clk, out\ @\ clk)\end{array}$$

We can now generate a Verilog code for Inc (Figure 3.4) and program the FPGA. The ML function `programFPGA` takes the theorem above (called `inc_cir`), generates a Verilog code of it and runs Quartus II to compile and download it to the FPGA. Quartus II actually compiles the circuit Inc integrated with the UART module. See below a snapshot of this step.

```
- programFPGA inc_cir;
Info: *******************************************************
Info: Running Quartus II Shell
```

... lots of messages from Quartus II appear.
At the end, an ML file is created in the directory ./quartus:

```
*************************************************************
New function created: inc_fn
at directory ./quartus
```

```
module dtype (clk,d,q);
 parameter size = 31;
 parameter value = 1;
 input clk;
 input [size:0] d;
 output [size:0] q;
 reg [size:0] q = value;

 always @(posedge clk) q <= d;
endmodule

module inc (clk,load,inp,done,out);
 input clk,load;
 input [31:0] inp;
 output done;
 output [31:0] out;
 wire clk,done;
 wire [31:0] v0;
 wire [0:0] v1;
 wire [0:0] v2;
 wire [0:0] v3;
 wire [31:0] v4;

 dtype dtype_2 (clk,load,v3); defparam dtype_2.size = 0;
 assign v2 = ~ v3;
 assign v1 = v2 && load;
 assign done = ~ v1;
 assign v4 = 1;
 assign v0 = inp+v4;
 dtype dtype_3 (clk,v0,out); defparam dtype_3.size = 31;
endmodule
```

Figure 3.4: The circuit Inc in Verilog.

```
In order to use it, type in:
FileSys.chDir "quartus"; use "main.sml"; FileSys.chDir "..";
************************************************************
```

If we load this file, the function `inc_fn` becomes available.

```
- FileSys.chDir "quartus"; use "main.sml"; FileSys.chDir "..";
> val ERR = fn : string -> string -> exn
> val 'a inc_inp2bits = fn : 'a list -> 'a list
> val 'a inc_bits2out = fn : 'a list -> 'a list
> val inc_fn = fn : bool list -> bool list
```

The ML function generated always manipulates Boolean lists. The file loaded also contains auxiliary functions which help in converting more standard inputs and outputs into Boolean lists. We are not going to use them in this example.

In order to run the FPGA, we can simply call `inc_fn` over a 32-bit word, which in this case is represented by a Boolean list with 32 elements.

```
- inc_fn  [false,false,false,false,false,false,false,false,false,
           false,false,false,false,false,false,false,false,false,
           false,false,false,false,false,false,false,false,false,
           false,false,false,false,true];
> val it =
 [false,false,false,false,false,false,false,false,false,false,false,
  false,false,false,false,false,false,false,false,false,false,false,
  false,false,false,false,false,false,false,false,true, false]
  :bool list
```

The circuit received 32 bits representing the number $1w$ and returned 32 bits representing $2w$.

All the steps described in this section are done automatically in HOL4.

3.5 Summary

This chapter introduced two kinds of optimisations: one based on elimination of handshakes (PRECEDE, FOLLOW and the extension of primitive operators) and another based on circuit duplication avoidance (let-expressions). Whole program compaction allows the elimination of handshakes to operate on a global scope. These techniques reduce the size of our case studies considerably (from 30% to 50% reduction). The drawback for handshaking elimination is the generation of long combinational paths, which slow down the clock.

We also addressed the problem of refining our circuit descriptions into a fine-grained timescale. The model presented in the previous chapter deals with observations of a circuit made at each clock cycle. In order to produce a clocked synchronous system, we applied the technology on time abstraction developed by Melham [55].

Finally, we described how our circuits are converted to Verilog and how we integrate HOL4 with Quartus II and an Altera FPGA.

Chapter 4

Limitations and Problems

This chapter describes problems, issues and limitations of our approach. Some of the problems were uncovered by our case studies (Chapter 5) while others were found during the development of the compiler.

4.1 Atomic Circuits Not Verified

We assume that primitive operators of the language are available in a library of previously verified combinational circuits. For example, the circuit (COMB ($\lambda(x, y).\ x+y$)) implements a combinational adder. We have neither implemented nor proved the correctness of such operators. Ideally these circuits should had been implemented by primitive gates and proved correct. We omit these proofs mainly because our work is focused on synthesis of systems specified at a higher level of abstraction. Moreover, there is previous work on verification of basic circuits which we could re-use [3, 14]. And, in any case, the verification task has to stop at some point. We have chosen to rely on the synthesis of basic circuits provided by Quartus II.

4.2 Combinational Loops

We found out that our circuit constructors produce combinational feedback loops[1]. The loops appear through the circuit POSEDGE. Figure 4.1 shows how it happens when we compile the function ($neg(b) = \neg b$) with no optimisations.

Similar loops occur inside PAR, ITE and REC. The combinational loops passed unnoticed almost until the completion of the project. We did not notice any unstable behaviour when running the FPGA or during the simulations. Although the Quartus II compiler warns about the possible occurrence of a combinational loop involving the variable *inp*, we did not spot it – possibly because the loop happens through the variable *done* (not *inp*). The process of proving circuit correctness also did not reveal the problem. As our formal model assumes zero combinational delay, we are not able to detect combinational loops from the proofs.

As both the FPGA and the simulations work as expected, the combinational loops do not appear to be a problem, despite the fact that they are generally regarded as a "bad

[1] This was recently pointed out by Martin Ellis from Newcastle University as well.

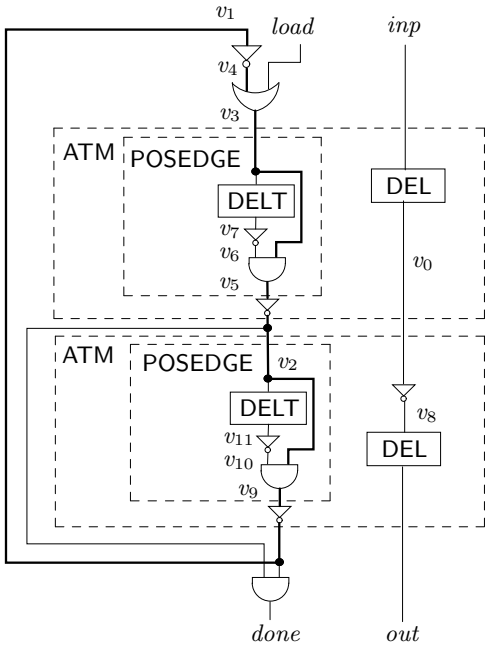


Figure 4.1: Combinational loop in (SEQ (ATM ($\lambda x.\ x$)) (ATM ($\lambda x.\ \neg x$))).

practice"[2].

In what follows we discuss this problem and suggest some solutions.

The false-implies-anything problem

The notion of correctness used in this work is formalised in terms of a logical implication.

$$\vdash\ \forall load, inp, done, out.\ \mathsf{C}(load, inp, done, out)\ \Rightarrow\ \mathsf{Dev}(load, inp, done, out)$$

A well known problem associated with correctness expressed in terms of logical implication is the false-implies-anything problem [6, 55].

Whenever $\mathsf{C}(load, inp, done, out)$ is false so is the implementation, thus one needs to show that for those signals *load* and *inp* that *arise in practice* the circuit model is always consistent.

One solution to this problem is to prove that for every input value assigned to *load* and *inp*, there exists output values which are consistent.

$$\forall load, inp.\ \exists done, out.\ \mathsf{C}(load, inp, done, out)$$

A circuit with a harmful combinational loop does not satisfy this condition. The proof of correctness of such circuits usually reduces to the proof of implications like

$$\ldots\ \wedge\ (some\_wire\ t = \neg(some\_wire\ t))\ \wedge\ \ldots\ \Rightarrow\ \mathsf{Dev}(load, inp, done, out)$$

The left hand side reduces to false and the theorem can be proved.

[2]Not only have all examples in this dissertation worked, but so have substantial cryptographic examples supplied by our colleagues at the University of Utah.

Empirical investigation

This section informally describes why the circuit shown in Figure 4.1 is consistent. Figure 4.2 shows the simulation of the circuit of Figure 4.1. We can see that no inconsistent

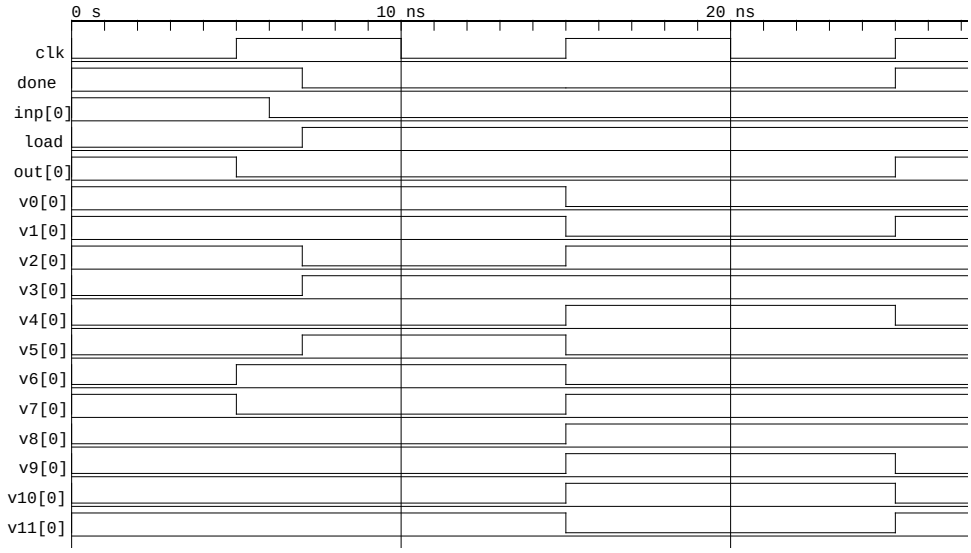


Figure 4.2: Well-behaved simulation with combinational loop.

values are generated for this circuit. The absence of inconsistent values for this particular circuit can be explained by analysing the variables v_7, v_{11} and *load*. The current state of this circuit is defined by the state of the DELTs (wires v_7 and v_{11}) and the value of *load*. These variables can be assigned to 8 different values. We manually checked them and discovered that inconsistent values appear only when *load*, v_7 and v_{11} are false. None of the 7 remaining states generates inconsistent values. Moreover, it is easy to check that they also define a *next* state which is also well-behaved. As the circuit is initialised in a well-behaved state (DELTs initialise with true), the circuit never reaches the bad state. We believe this to be the explanation for the correct behaviour of the other circuits we generated.

The analysis described above suggests that one could use a model checker to show the absence of bad states on synthesised circuits, although the ideal solution is the proof of consistency for all circuits generated.

Possible solutions

A general and complete solution to this problem is to define the predicate OK which states that a circuit is consistent (i.e. the circuit can be satisfied by some values assigned to the free variables).

$$\vdash \ \mathsf{OK}\ \mathsf{C} = \forall \mathit{load}, \mathit{inp}.\ \exists \mathit{done}, \mathit{out}.\ \mathsf{C}(\mathit{load}, \mathit{inp}, \mathit{done}, \mathit{out})$$

This theorem states that for every input values assigned to *load* and *inp*, there exists output values which are consistent.

In order to implement this solution one has to prove that

$$\vdash \ \forall f.\ \mathsf{OK}\ (\mathsf{ATM}\ \mathsf{f})$$

For each composite circuit, we have to prove that it is consistent provided that its sub-circuits are consistent. For instance, in order to check the consistency of SEQ we have to prove that

$$\vdash \forall \mathsf{E}\ \mathsf{G}.\ (\mathsf{OK}\ \mathsf{E}) \wedge (\mathsf{OK}\ \mathsf{G}) \Rightarrow (\mathsf{OK}\ (\mathsf{SEQ}\ \mathsf{E}\ \mathsf{G}))$$

Another possible solution to this problem is to change the implementation of POSEDGE (see Figure 4.3). By adding an extra delay component we eliminate the possibility of any combinational path to occur through POSEDGE. This solution has a significant impact on the definition of Dev and on the correctness theorems. The extra delay added to POSEDGE will require the reverification of the correctness of each circuit constructor and will generate larger and slower circuits.

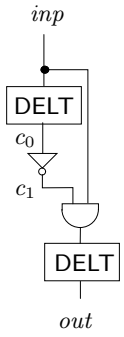


Figure 4.3: Another implementation for POSEDGE.

We have not proved the consistency of our circuits yet. Our main priority in this project was to establish a proof-of-concept of our approach, especially with respect to its scalability. However we do recognise the importance and the necessity of formally verifying the absence of problems caused by the combinational loops.

4.3 Proof Effort

The formalisation in HOL of the definitions and proofs takes 3,130 lines of ML code (2,971 of which are proofs). These figures do not include comments or the compiler code. The proofs took about 4-5 months of work. It takes 24 minutes to prove all theorems needed in our compilation method[3]. Note that these proofs need to be done just once. The HOL4 system allows us to quickly load previously proved theorems. Although 2,971 lines of code sounds like a rather long proof, usually proofs carried out by theorem provers are much longer than pencil and paper proofs. This happens mainly because it requires fine-grained proof steps in comparison with proofs described as a mix of formal mathematical language and say, the English language.

It is hard to tell if verifying a compiler would be easier than verifying language constructs. Although it seems to be easier to prove the correctness of the *result* of the compiler (in our cases, the circuits SEQ, PAR, etc.) instead of proving the correctness of the compiler itself, we have no firm evidence to support our claim. There are data available on the effort to verify compilers (for instance, Blazy et al. [7] and Leroy [50]

[3]We used an Intel® Pentium® 4, 3.2GHz, 1GB RAM.

report 41,000 lines of Coq statements used in the verification of a C compiler), but a fair analysis should compare compilations of the same source language and using the same theorem prover.

4.4 Undefined Values

Our circuits deal only with the Boolean values T and F or words of bits. We did not consider the four valued logic usually employed to model signal values in digital circuits. As undefined (unknown) values and high impedance (open circuit) are abstracted away in our models, our simulations were showing several occurrences of undefined values. In general, our circuits are not able to establish a Boolean value to the signal *done* if some wire has an undefined value. Figure 4.4 shows the simulation of a device which computes factorial. Once undefined values are carried into the circuit by *load*, it never asserts *done* in future.

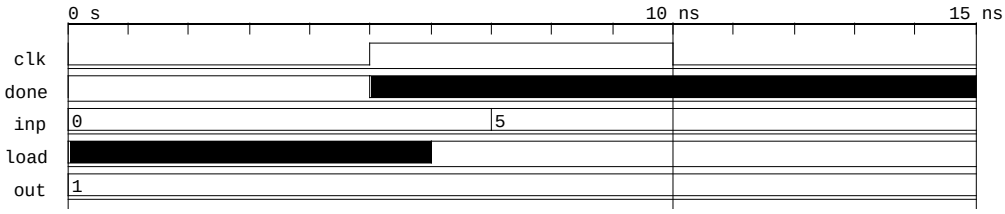


Figure 4.4: Undefined values on the simulation of factorial.

We overcome this problem by configuring the simulation according to our model. We initialise all the DFFs and the environment (i.e. *load* and *inp*) to well-defined values. Notice that this problem did not happen to FPGAs. As the board initialises all wires with well-defined values, we have never seen any initialisation delay on the signal *done*. If we assign well-defined values to *load* and *inp* from time zero, the simulation of the factorial circuit works as expected. Figure 4.5 shows the computation of factorial of 5.

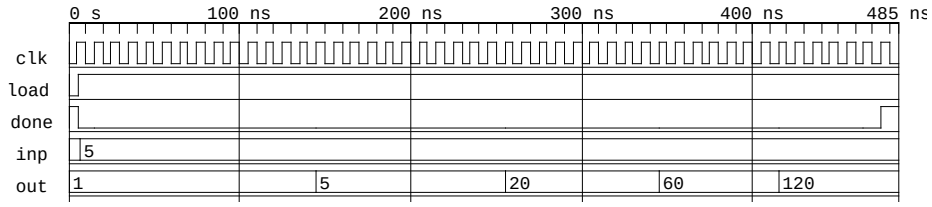


Figure 4.5: Well-behaved simulation of factorial of 5.

4.5 Industrial-scale Specifications

As we show in Chapter 5, our approach does not scale to an industrial-level application yet. The main emphasis in this work was to produce a proof-of-concept for the automatic verification of circuits. We should now put more effort into optimising the compiler.

4.6 Summary

This chapter described problems and limitations of our approach.

The fact that atomic circuits are not verified may reduce one's confidence in the reliability of our circuits. However, this was a design decision we took as our main aim was to develop a proof-producing technology. Moreover, there are several previous work which developed verified circuits which we can use to implement our atomic operators (although we have not tried to integrate them with our compiler yet).

Combinational loops passed unnoticed almost until the end of the project. The Verilog compiler, the proofs of correctness, the simulations and the tests with the FPGA did not help us to spot this problem earlier.

The main theorems took a reasonable amount of time to prove, mostly because of their size. Usually formal verification of hardware and software is not an intellectually challenging task (in comparison to the field of pure mathematics), but demands proof of theorems of substantial size. Our experiments have not produced yet conclusive results that our approach requires less work in comparison to other techniques like compiler verification.

It took a lot of time for us to fix the undefined values of the simulations. Although our final solution was simple (we adjusted the simulator to adhere to our mathematical model), we spent a long time trying to change our circuits in order to have a well-behaved simulation.

Finally, a problem uncovered by one of our case studies was the lack of scalability of our compiler. Industrial-sized projects are still too large for the compiler to handle. See more details in Chapter 5.

Chapter 5

Case Studies

This chapter illustrates our approach by describing the development of two case studies. The first one is an implementation of the Booth multiplier used in the verification of the ARM6 micro-architecture. The second case study is a subset of the DIY microcomputer. We develop two different designs for this machine. The first one models only the CPU and assumes that it interacts with a memory running outside the FPGA. The second design specifies the CPU and the memory, i.e. both are implemented in the FPGA. Section 5.3 summarises the performance of the compiler for both case studies and reports the difficulties faced during their development.

5.1 Booth Multiplier

The specification of our Booth multiplier was developed by Fox [22] as part of the verification of the ARM6 micro-architecture. Although the ARM6 implementation of multiplication is a variant of the standard Booth's algorithm, its main idea is still preserved.

In what follows we give the intuition behind the standard Booth's multiplication algorithm before describing the ARM6 version.

First we introduce an extended notation for binary numbers. The *Booth code* [39] allows *negative* bits to be represented: a $\overline{1}$ means -1. For example,

$$\begin{aligned} 0110 &= 0{\cdot}2^3 + 1{\cdot}2^2 + 1{\cdot}2^1 + 0{\cdot}2^0 = 6 \\ 01\overline{1}0 &= 0{\cdot}2^3 + 1{\cdot}2^2 + (-1){\cdot}2^1 + 0{\cdot}2^0 = 2 \end{aligned}$$

Booth's multiplier algorithm speeds up the multiplication by using subtraction as well as addition. It is based on the observation that a number, say, 7 can be coded as $8-1$: $0111 = 1000-0001 = 100\overline{1}$. Booth performs fewer additions than the standard multiplication algorithm. For example, the multiplication of 0101×0111 (5×7) is shown in Figure 5.1.

In this example, Booth's algorithm requires only 2 additions in comparison to the 3 additions of the standard method. Note that the first addition is actually the subtraction of 5 — we add 11111011 (-5) to the result.

Booth's algorithm infers the Booth code of the multiplier by analysing its blocks of consecutive 1's. For instance, note that $0\mathbf{1111}0 = 1000\overline{1}0$. The multiplicand is subtracted from the result whenever a 10 occurs in the multiplier (scanning from right to left). This

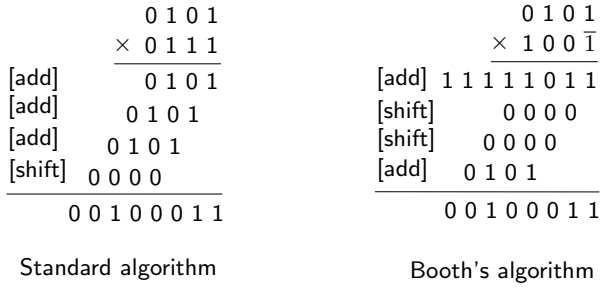


Figure 5.1: Differences between the standard and Booth's algorithms.

pattern defines the beginning of a 1's block, which is equivalent of having a $\overline{1}$ in the Booth code. Similarly, the multiplicand is added to the result whenever a 01 is detected, which indicates the end of a 1's block and, consequently, the occurrence of a 1 in the Booth code. If the bits 00 or 11 are detected, no arithmetic operation is needed — these states represent the middle of a 1's block or a 0's block. In both cases, they are converted to 0's blocks in the Booth code.

5.1.1 The Compilation

The complete code is shown in Figure 5.2. We changed the syntax of the original specification to the one that our compiler accepts.

The multiplication is implemented by the processor's control logic, which uses the ALU and a barrel shifter. The program analyses two bits of the multiplier in each iteration (starting from the least significant ones). The state of the system is defined by the tuple `(mul,mul2,borrow2,mshift,rm,rd)`: `mul` stores the current two bits under evaluation; `mul2` contains the remaining bits; `borrow2` is the most significant bit of `mul`; `mshift` stores the number of shifts to be applied; `rm` is the multiplicand and `rd` accumulates the final result. The only recursive function is `APPLY_NEXTd`. `TFL` [73] can prove its termination automatically if the function `FST` is given as measure function (`FST` returns the first element of a pair). A complete description of this specification is presented by Fox [22].

We instantiated the ARM6 Booth's multiplier to 32-bit numbers. The compilation generates the theorem

$$\begin{array}{l}\vdash\ \mathsf{InfRise}\ clk\\ \quad\Rightarrow (\exists v_0\ v_1\ v_2\ \ldots\ v_{249}.\\ \qquad \mathsf{CONSTANT\ F}\ v_1\ \wedge\ \mathsf{CONSTANT}\ 0w\ v_0\ \wedge\ \mathsf{COMB\ w2n}\ (inp_2, v_{13})\ \wedge\\ \qquad \ldots\ \mathsf{AND}(v_{112}, v_{123}, v_{249})\ \wedge\ \mathsf{AND}(v_{249}, v_{247}, done))\\ \qquad\Rightarrow\ \mathsf{Dev\ MULTd}\ (load\ @\ clk, (inp_1 \diamond inp_2)\ @\ clk, done\ @\ clk, out\ @\ clk)\end{array}$$

This netlist contains 252 primitive components. The compilation takes 3 minutes (running in an Intel® Pentium® 4, 3.2GHz, 3.7GB RAM).

In order to establish a connection between MULTd and the standard 32-bit multiplication, Fox also proved that

$$\vdash\ \mathsf{MULTd} = \mathsf{MULT32}$$

The function MULT32 is the standard definition of multiplication for 32-bit numbers.

```
MOD_CNTWd n = n MOD (WL DIV 2)
MSHIFTd(borrow,mul,count1) = count1 * 2 +
                             if borrow /\ (mul=1) \/
                                ~borrow /\ (mul=2) then 1 else 0
ALUd(borrow2,mul,alua,alub) =
      if ~borrow2 /\ (mul = 0) \/ borrow2 /\ (mul = 3) then alua
      else if borrow2 /\ (mul = 0) \/ (mul = 1) then  alua + alub
      else alua - alub
INITd(a,rm,rs,rn) = (BITS 1 0 (w2n rs),BITS HB 2 (w2n rs), F,
                     if (BITS 1 0 (w2n rs)) = 2 then 1 else 0,
                     rm, if a then rn else 0w)
NEXTd(mul,mul2,borrow2,mshift,rm,rd) =
      (BITS 1 0 (BITS (HB-2) 0 mul2),BITS HB 2 (BITS (HB-2) 0 mul2),
       BIT 1 mul, MSHIFTd(BIT 1 mul,BITS 1 0 (BITS (HB-2) 0 mul2),
       MOD_CNTWd (mshift DIV 2 +1)),rm,
       ALUd(borrow2,mul,rd,rm << mshift))
APPLY_NEXTd(t,inp) = if t=0 then inp
                     else APPLY_NEXTd(t-1,NEXTd inp)
STATEd(t,(a,rm,rs,rn)) = APPLY_NEXTd(t,INITd(a,rm,rs,rn))
DURd w = if BITS 31 1 (w2n w) = 0 then 1
         else if BITS 31 3 (w2n w) = 0 then 2
         else if BITS 31 5 (w2n w) = 0 then 3
         else if BITS 31 7 (w2n w) = 0 then 4
         else if BITS 31 9 (w2n w) = 0 then 5
         else if BITS 31 11 (w2n w) = 0 then 6
         else if BITS 31 13 (w2n w) = 0 then 7
         else if BITS 31 15 (w2n w) = 0 then 8
         else if BITS 31 17 (w2n w) = 0 then 9
         else if BITS 31 19 (w2n w) = 0 then 10
         else if BITS 31 21 (w2n w) = 0 then 11
         else if BITS 31 23 (w2n w) = 0 then 12
         else if BITS 31 25 (w2n w) = 0 then 13
         else if BITS 31 27 (w2n w) = 0 then 14
         else if BITS 31 29 (w2n w) = 0 then 15
         else 16
PROJ_RDd(mul,mul2,borrow2,mshift,rm,rd) = rd
BOOTHMULTIPLYd(a,rm,rs,rn) = PROJ_RDd(STATEd(DURd rs,a,rm,rs,rn))
MULTd(a,b) = BOOTHMULTIPLYd(F,a,b,0w)
```

Figure 5.2: The specification of the ARM6 Booth's algorithm.

Our pretty-printer generated a Verilog file with 524 lines of code (after deleting the comments):

```
module dtype (clk,d,q);
   parameter size = 31;
   parameter value = 1;
   input clk;
   input  [size:0] d;
   output [size:0] q;
   reg    [size:0] q = value;
   always @(posedge clk) q <= d;
endmodule

module MULTd (clk,load,inp1,inp2,done,out);
   input clk,load;
   input [31:0] inp1;
   input [31:0] inp2;
   output done;
```

```
   output [31:0] out;
   wire clk,done;
   wire [31:0] v0;
   wire [0:0] v1;
   ...
   wire [0:0] v249;

   assign v1 = 0;
   assign v0 = 0;
   assign v13 = inp2;
   ...
   assign done = v249 && v247;
endmodule
```

This design is downloaded into the Excalibur™ FPGA with 100,000 gates. It used 3,897 logical elements (93% of the total amount available). We also simulated it using GPL-Cver and and GTKWave [31, 33]. Figure 5.3 shows the multiplication of 5×7.

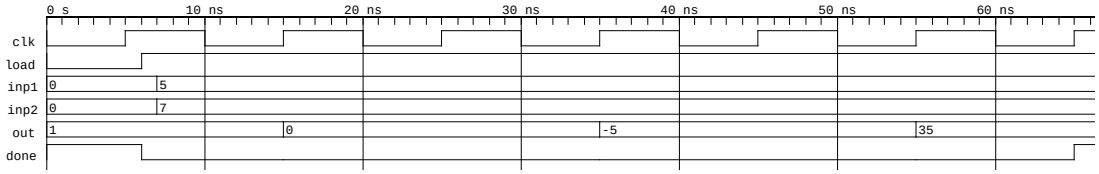


Figure 5.3: Simulation of the ARM6 Booth's algorithm computing 5×7.

The multiplier specified in Figure 5.2 implements a variant of the Booth's algorithm in terms of state functions. Although we know a Booth's multiplier could be specified by a smaller code, this case study tested the performance of our compiler with a medium-sized specification.

5.2 The DIY Microcomputer

The book *How Computers Do Math* by Maxfield and Brown [53] introduces the way in which computers work. The authors explain it by developing a calculator called the DIY Calculator. The DIY Calculator is programmed in the assembly language of a simple microcomputer, which we call here the DIY microcomputer or simply, the DIY.

We choose this microprocessor as a case study mainly because its development has been thoroughly documented. In addition to that, it has the right size for our purposes (neither too complex like a real computer, nor too simple like a toy). There is also a physical implementation of the DIY carried out by final year students at the Newcastle University[1]. This implementation was modelled in VHDL and downloaded to an Altera UP2 FPGA, thus enabling us to compare our approach with this traditional development method (although we have not produced this analysis yet). This section describes the subset of the DIY used in our case study.

[1]The project was supervised by Albert Koelmans. For more information, see:
`http://www.diycalculator.com/popup-m-phyver.shtml`

The DIY microcomputer comprises a CPU, a memory and input/output ports. The CPU has an 8-bit data bus, a 16-bit address bus and contains a small number of registers. The CPU is initialised whenever we drive a logical zero (or F in HOL) on the reset line. All control signals are active low. The CPU interfaces with the memory via the control signals read and write, and the buses data and addr (address). There are nine registers in the CPU, namely: ACC (accumulator), PC (program counter), IR (instruction register), SP (stack pointer), TMP (temporary register), OV (overflow flag), N (negative flag), Z (zero flag) and C (carry bit). Figure 5.4 shows the external interface of the DIY together with its registers. The size of each register is shown next to their names.

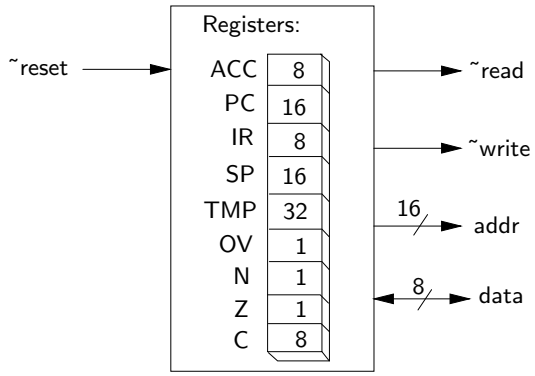


Figure 5.4: The DIY CPU.

The carry bit is originally a 1-bit register, but we decided to define it as an 8-bit register in order to manipulate it with 8-bit numbers without having to use type conversions. The symbol ~ which precedes control signals in Figure 5.4 indicates that the signal is active low.

The address bus is 16-bit wide — therefore it addresses 2^{16} or 65,536 memory locations. Each location is 8-bit wide. In order to read a byte from the memory, the DIY sets read to zero and sends the address through the addr bus. The memory returns the byte requested on the data bus. In order to write a byte in the memory, the DIY sets write to zero and sends the data and address on the buses data and addr, respectively.

Before presenting the instructions of the DIY, we briefly explain the addressing modes we use.

In the *immediate addressing* mode, the byte that follows the instruction in memory is the actual operand. For example, consider the instruction LDA (load) using immediate addressing. Suppose that the byte following its opcode in memory has value 4. The execution of this instruction loads the contents of the accumulator with the value 4.

Implied addressing mode is used by instructions that do not take any arguments. For example, the instruction INCA adds one to the contents of the accumulator. In this case, the DIY does not need to read any byte following the instruction as the argument is already known, namely the value 1.

In the *absolute addressing* mode, the instruction in memory is followed by two bytes which are the least and the most-significant bytes of a 16-bit address. This address points to the actual operand in memory (or points to the next instruction in case of a jump).

Tables 5.2 and 5.3 (see pages 58 and 59) show the instructions implemented in our case study. Note that these instructions are just a subset of all instructions provided

by the DIY. We indicate the addressing modes of each instruction by the suffixes IMM (immediate), IMP (implied) and ABS (absolute).

We develop the DIY in two different ways. The *CPU Design* models only the DIY CPU, while the *Microcomputer Design* models both the CPU and a (tiny) memory. In what follows we present these designs separately and compare them in the last section.

5.2.1 The CPU Design

This section presents the formal specification of the DIY CPU. In this design, we implement a CPU as a function that computes a single interaction with the memory. The external interface of the system is shown in Figure 5.5.

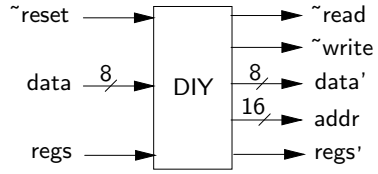


Figure 5.5: The DIY interface.

The function DIY takes as inputs the signal reset and the byte data, and returns the signals read, write, data′ and addr. However, as we are dealing with a pure functional language, we have to include all registers as part of the external interface. The variables regs and regs′ are tuples which comprise all DIY registers in addition to a variable called step: (ACC, PC, IR, step, SP, TMP, OV, N, Z, C). The variable step stores the current state of the CPU (explained below). We assume that the environment does not change the value of regs.

The function DIY models the state machine shown in Figure 5.6. The only instruction not implemented in this version of the DIY is the *load* using immediate addressing (LDA_IMM). In the state FETCH, the CPU reads an instruction in memory and stores it in the register IR. The states LD_LSB and LD_MSB read the least and the most-significant bytes of the operand address, respectively. In the state RUN the CPU executes the instruction.

We present an outline of the source code in what follows. The complete code can be seen in Section A.

The top-level function simply tests in which state the processor is and returns the next state and requests to the memory.

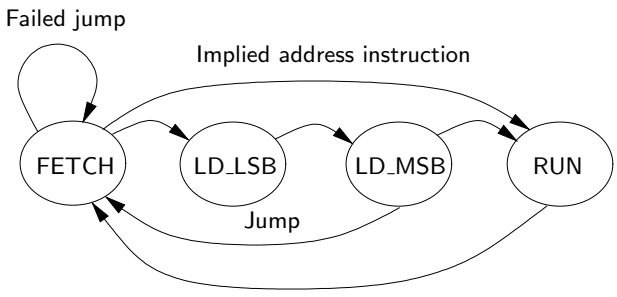


Figure 5.6: The DIY states.

```
diy (reset:bool, data:word8, ACC:word8, PC:word16, IR:word8, step:word4,
     SPTMPOVNZC: word16 # word32 # bool # bool # bool # word8)
= if (~reset) then
     ...
  else if step = FETCH then
     ...
  else if step = LD_LSB then
     ...
  else if step = LD_MSB then
     ...
  else alu(data,ACC,PC,IR,step,SPTMPOVNZC)
```

The names FETCH, LD_LSB, LD_MSB and RUN are constants whose values are 0, 1, 2 and 3, respectively. The variable step is a 4-bit word which stores values from 0 to 3 representing the current state. The first case tests if the CPU is reset. In this case, the CPU requests to read the first byte in memory and sets the next state to FETCH.

```
if (~reset) then
   ((* read *) F, (* write *) T, (* addr *) 0w, (* data *) data,
    ACC, (* PC *) 1w, (* IR *) 0w, FETCH, SPTMPOVNZC)
else if step = FETCH then ...
```

Comments in ML and HOL4 are enclosed by (* and *).

In the FETCH state there are three possible sub-cases to analyse. If the instruction uses implied addressing, then the next state is RUN. If the instruction is a jump which failed its test, the function ignores the jump and reads the next instruction. In this case, the next instruction is located at PC+3w as we have to skip the operand (and the next state is still FETCH). Otherwise, the next state is LD_LSB.

```
else if step = FETCH then
   let (SP:word16,TMP:word32,OVNZC) = SPTMPOVNZC
   in if (~(isJUMP(data)) \/ testJUMP(data,OVNZC)) then
         (F,T,PC,data,ACC,PC+1w,data,
          if implied_addr(data) then RUN else LD_LSB, SPTMPOVNZC)
      else (F,T,PC+2w,data,ACC,PC+3w,data,FETCH,SPTMPOVNZC)
else if step = LD_LSB then ...
```

In the LD_LSB state, we store the least-significant byte of the operand address at the temporary register. The primitive operator @@ implements word concatenation.

```
else if step = LD_LSB then
   let (SP:word16,TMP:word32,OVNZC) = SPTMPOVNZC
   in (F,T,PC,data,ACC,PC+1w,IR,LD_MSB,SP,(* TMP *) (0w:word24) @@ data,
       OVNZC)
else if step = LD_MSB then ...
```

The LD_MSB checks if the instruction is a *store* (STA_ABS). In this case, it writes the contents of the accumulator in the memory. If the instruction is a jump, the PC is assigned to the value of the operand and the next state is FETCH. Otherwise, we move to RUN.

```
else if step = LD_MSB then
   let (SP:word16,TMP:word32,OVNZC) = SPTMPOVNZC
   in (IR=STA_ABS, (* write *) ~(IR = STA_ABS),
       (* addr *) (data @@ ((7 >< 0) TMP):word8):word16,
       (* data *) ACC, ACC,
       if isJUMP(IR) then ((data @@ ((7><0) TMP):word8)+1w) else PC,
       IR, if isJUMP(IR) then FETCH else RUN, SP, TMP, OVNZC)
else ...
```

The primitive operator `><` takes two numbers h and l representing bit positions and a word w, and extracts the word $w[h:l]$ (shown here in a Verilog-like notation). This operator is used to extract the least-significant byte of the operand's address stored in `TMP`.

The state `RUN` simply calls the function `alu`.

```
else alu(data,ACC,PC,IR,step,SPTMPOVNZC)
```

The function `alu` executes the instruction. The code below shows the execution of *load* and *add* (`LDA_ABS` and `ADD_ABS`).

```
val (alu_def,_,alu_dev0,alu_comb,_) = hwDefine2
   `alu (data:word8,ACC:word8,PC:word16, IR:word8, step:word4, SP:word16,
         TMP:word32, OV: bool, N:bool, Z:bool, C:word8)
    = if IR = LDA_ABS then
         ((* read *) F,       (* write *) T,     (* addr *) PC,
          (* data *) data,    (* ACC   *) data,  (* PC   *) PC+1w,
          (* IR   *) IR,      (* step  *) FETCH, SP, TMP, OV,
          (* N  *) word_msb(data),(* Z  *) data=0w, C)
      else if IR = ADD_ABS then
         let result = ACC+data
         in (F,T,PC,data, (* ACC *) result,
             PC+1w, IR, FETCH, SP, TMP,
             (* OV *) add_ov(ACC,data),
             (* N  *) word_msb(result),
             (* Z  *) result=0w,
             (* C  *) b2w(w2n(ACC) > (255-w2n(data)))
            )
      ...
```

The *load* instruction simply stores the byte on `data` in the accumulator. The addition stores the result of (`ACC+data`) in the accumulator. The auxiliary function `add_ov` tests if the addition overflows. The primitive operators `w2n` and `n2w` convert a word to a number and a number to a word, respectively. The next state is `FETCH`.

After compiling (and automatically verifying) the DIY CPU, we downloaded it to an FPGA and ran a small program on it. We tested the CPU by running Euclid's Greatest Common Divisor (GCD) for two unsigned 16-bit numbers.

```
while (a != b)
   if a > b then
      a := a-b
   else
      b := b-a
   return a
```

We use the ML interface described in Chapter 3 to communicate with the CPU. We develop a memory simulator in ML to interact with the FGPA and code the GCD algorithm in the DIY machine language. The program has 129 lines of code as we had to develop sub-routines which implement a 16-bit comparator and a 16-bit subtracter (recall that the standard DIY instructions deal with 8-bit numbers). Unfortunately, the serial cable connection acts as a very slow bus. For instance, it takes about 30 seconds to compute $\mathsf{GCD}(64326, 33254)$, where most of this time is being used for communicating the PC to the FPGA. This is one of the reasons which motivated us to develop the Microcomputer Design.

The DIY CPU connects to input/output ports via memory. Our memory simulator does not implement interactive input/output. We simply store input and output values at a particular address in memory.

5.2.2 The Microcomputer Design

The Microcomputer Design implements the DIY CPU together with functions that access the memory. The memory is a variable that is initially sent to the top-level system and is manipulated by the functions read and write (Figure 5.7). Like the CPU Design, the specification is also based on states. However the notion of state is different from the one presented in Section 5.2.1. As the memory is now part of the system, it is possible to execute one instruction completely in a single function call. In the previous section a state was a fine-grained intermediate step of the execution of a single instruction.

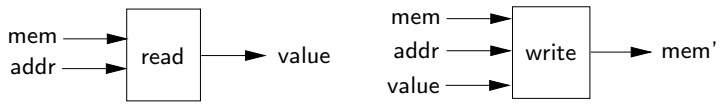


Figure 5.7: Memory functions.

The top-level system takes a memory as input and returns the resulting memory after running the program it contains (see Figure 5.8). An initial state for the DIY registers is created before running the program. The function next_state executes a single instruction on each call. If the program terminates, the system outputs the possibly modified memory.

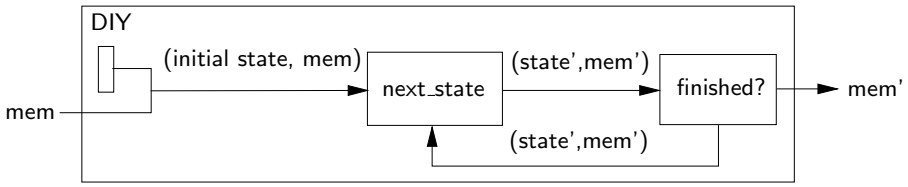


Figure 5.8: The DIY interface.

Recall that we must prove that our functions terminate. However if we have to take a memory as input and run whatever program is in it, we cannot guarantee beforehand that this program terminates, and consequently, that the DIY computing it terminates. To overcome this problem we define two criteria for termination. The first one is based on the memory position reached by the program. We define a constant address called `end_of_program`. If the DIY reaches that position, then it indicates that the program has finished. However, the program might loop and end up never reaching the termination address. So, a second criterion is used to guarantee termination. We introduce a simple counter that allows the DIY to execute 2,000,000 instructions before halting.

The top-level function takes a memory as input and calls the function `run`, which executes the program stored in the memory. The state of the system is represented by the tuple (`mem`, `ACC`, `PC`, `OV`, `N`, `Z`, `C`), which is initialised by `diy`.

```
diy(mem:word104) = run(2000000w,(mem,0w,0w,F,F,F,0w))
```

Notice that we use a tiny memory of 13 bytes only. This is mostly due to a limitation of our Verilog compiler (see a discussion on this issue in Section 5.3).

The function `run` is a recursive function. It stops whenever the program counter `PC` reaches the `end_of_program` address or the step counter is zero. Notice that the function `diy` above initialises `step` to 2,000,000. Every recursive call decrements the value of `step` and calls the function `next_state`. The measure function `(w2n o FST)` is used by `TFL` to prove `run`'s termination.

```
run(step:word32, (mem:word104,ACC:word8,PC:word16,
                  OVNZC:bool # bool # bool # word8))
    = (if (step = 0w) \/ (PC = end_of_program) then mem
       else run(step-1w,next_state(mem,ACC,PC,OVNZC)))
measuring (w2n o FST)
```

The function `next_state` executes the instruction in memory pointed by the program counter and returns the new state of the system.

```
next_state (mem:word104,ACC:word8,PC:word16,
            OVNZC:bool # bool # bool # word8)
  = let (IR,op2,op1) = (read(mem,PC),read(mem,PC+2w),read(mem,PC+1w)) in
    let addr = op2 @@ op1
    in if IR = STA_ABS then
          let mem' = write(mem,addr,ACC) in (mem',ACC,PC+3w,OVNZC)
       else if IR = LDA_IMM then
          let (OV,N,Z,C) = OVNZC in (mem,op1,PC+2w,OV,word_msb(op1),op1=0w,C)
      else if testJUMP(IR,OVNZC) then
          (mem,ACC,addr,OVNZC)
      else (mem, alu(IR,read(mem,addr),ACC,PC,OVNZC))
```

First the function reads the instruction and its two consecutive bytes (`op1` and `op2`). In the absolute addressing mode these bytes are the least and the most-significant bytes of the operand's address. If the instruction is a *store*, the function `write` is called to generate an updated memory. A *load* using immediate addressing stores the contents of `op1` in the accumulator. If the instruction is a jump, then `PC` points to (`op2 @@ op1`). The remaining instructions are executed by `alu`.

The function `alu` executes the current instruction and returns the new state of the system. As the variable `PC` points to the current instruction (using absolute addressing), it is incremented by 3 in order to skip the operands and to point to the next instruction. In this design we did not implement the instructions using implied addressing (we discuss this issue in the next section).

```
alu (IR:word8,data:word8, ACC:word8, PC:word16, OV:bool, N:bool,
     Z:bool, C:word8) =
     if IR = LDA_ABS then
        (data,PC+3w,OV,word_msb(data),data=0w,C)
     else  if IR = ADD_ABS then
        let res = ACC+data
        in (res,PC+3w,add_ov(ACC,data),word_msb(res),res=0w (* Z *),
            b2w(w2n(ACC) > (255-w2n(data))) (* C *))
     ...
     else (ACC,PC+3w,OV,N,Z,C)‘;
```

As this design implements a very small memory, we could only run tiny programs which execute two or three instructions. This contrasts with the CPU Design described above, where any program smaller than 64KB could be executed. However, in order to achieve a reasonable performance with the CPU Design, we have to implement the memory on the board in order to replace the slow serial cable connection.

		Booth	CPU only	CPU+Memory
Instructions		—	25	20
Chip area[a]		93%	34%	70%
Compilation time[b]	HOL	3 min.	3.3 hr	55 min.
	Quartus II	8 min.	5 min.	2 hr
Primitive components	Dtype	36	15	34
	Combinational	216	784	604
	Total	252	799	638

[a]We used an FPGA with 100,000 gates.
[b]Running on an Intel® Pentium® 4, 3GHz, 3.7GB RAM

Table 5.1: Case study analysis.

5.3 Result Analysis

In this section we analyse the performance of the compiler for the Booth multiplier, the CPU and the Microcomputer Designs and report the strengths and weaknesses of our approach uncovered by these case studies.

The Booth multiplier is a medium-sized design whose circuit could be employed as co-processor. The compiler had a fairly acceptable performance. Most of the total compilation time was spent by Quartus II (see Table 5.1).

Regarding the DIY machine, both designs specified state machines implemented by a long nested `if-then-else` manipulating large tuples (the states). As our specification language is intrinsically stateless, the state is a variable which has to be taken as argument by the functions. These particular features combined make the compiler to generate large logical terms that slow the compilation process down significantly. The impact of these large terms on the compilation time of both designs made us restrict our implementation to a subset of the complete DIY instruction set. For example, we have not specified any instruction related to subroutines or to the stack pointer. In particular, we did not implement instructions using immediate addressing in the Microcomputer Design in order to reduce the depth of nested `if-then-elses` in the specification (see Table 5.1).

Note that the Microcomputer Design used 70% of the chip area, which in principle would allow us to implement a larger memory. However, the tiny memory size of 13 bytes was imposed mostly by Quartus II, which was running out of memory during the compilation of long-sized variables.

The limitations of our compilation method revealed by the DIY case study show that we have to constantly create or adapt our optimisations according to the application. In particular, nested `if-then-elses` manipulating large tuples – a combination of features not tested previously – generated a term explosion. One of the causes of this explosion is the generation of large tuples from a binary Par. A possible solution to overcome this problem is the definition of constructors like, say Par_4 (and PAR_4) to generate 4-tuples and eventually reduce the size of the terms.

Another issue is the fact that pure functional languages are not suitable for state-based applications. The solution adopted by Beyer et al. [4] was to extend the Verilog pretty-printer with facilities to generate next-state functions and memories from special PVS

functions like $(\mathsf{ns} : \mathit{State} \times \mathit{Input} \rightarrow \mathit{State} \times \mathit{Output})$[2]. The Verilog code manipulates the state inside a module. The external interface deals only with *Input* and *Output*. Although this is a valid alternative to be considered, it relies on the fact that the pretty-printer is no longer an obvious syntactical transformation between two netlist notations, thus reducing our confidence in the correctness of the outcome.

In contrast to these issues, our approach provided two important facilities.

First the high level of abstraction of our source language allows us to specify a Booth algorithm as a state machine and the DIY microcomputer in a very concise way. The specification in HOL abstracts away several low-level details of implementation described by Maxfield and Brown [53]. For example, the DIY Data Book shows in full detail how the registers connect to the ALU and how the data and the address buses work[3]. Notice that we did not have to specify the bus protocol between the memory and the CPU in the Microcomputer Design. The compiler automatically generates the low-level connection between components.

Second, the compiler produced a verified circuit. In the worst case, it took HOL4 3.3 hours to complete. This is definitely a long compilation time if we consider no verification. However, the theorem prover actually verified automatically that a netlist with 799 components implements a handshaking device which computes the function diy.

We can conclude that our compiler is adequate to medium-sized projects like a co-processor, but it still has limitations considering a larger system like a microcomputer.

[2]The work proposed by Beyer et al. [4] is not proof-producing, but follows a more standard approach.

[3]The *Official DIY Data Book* comes in the CD-ROM accompanying the book.

Instruction	Description
LDA_IMM LDA_ABS	Loads the contents of a byte of data in the memory into the accumulator. Flags affected: N (set if the accumulator is negative) and Z (set if the accumulator is zero).
STA_ABS	Stores the contents of the accumulator to a byte in the memory. Flags affected: none.
ADD_ABS	Adds the contents of a byte in memory to the contents of the accumulator and stores the result in the accumulator. Flags affected: OV, N, Z and C.
ADDC_ABS	Same as ADD, except that it adds the contents of the carry flag as well.
SUB_ABS	Subtracts the contents of a byte in memory from the contents of the accumulator and stores the result in the accumulator. Flags affected: OV, N, Z and C.
SUBC_ABS	Same as SUB, except that it subtracts the carry flag as well.
AND_ABS	Applies a bit-wise AND operation to the contents of a byte in memory and the contents of the accumulator and stores the result in the accumulator. Flags affected: N and Z.
OR_ABS	Applies a bit-wise OR operation to the contents of a byte in memory and the contents of the accumulator and stores the result in the accumulator. Flags affected: N and Z.
XOR_ABS	Applies a bit-wise XOR operation to the contents of a byte in memory and the contents of the accumulator and stores the result in the accumulator. Flags affected: N and Z.
CMPA_ABS	Compares the contents of a byte in memory with the contents of the accumulator assuming that both are *unsigned* values. The accumulator and the memory are not affected. Flags affected: Z (set if the values are equal) and C (set if the accumulator is greater than the byte in memory).

Table 5.2: Instructions of the DIY.

Instruction	Description
JMP_ABS	Jumps unconditionally to the address stored in the two consecutive bytes in memory following the instruction code. Flags affected: none.
JC_ABS	Jumps to a new address if the carry flag C is 1. Flags affected: none.
JNC_ABS	Jumps to a new address if the carry flag C is 0. Flags affected: none.
JN_ABS	Jumps to a new address if the negative flag N is 1. Flags affected: none.
JNN_ABS	Jumps to a new address if the negative flag N is 0. Flags affected: none.
JO_ABS	Jumps to a new address if the overflow flag OV is 1. Flags affected: none.
JNO_ABS	Jumps to a new address if the overflow flag OV is 0. Flags affected: none.
JZ_ABS	Jumps to a new address if the zero flag Z is 1. Flags affected: none.
JNZ_ABS	Jumps to a new address if the zero flag Z is 0. Flags affected: none.
SHL_IMP	Shifts the contents of the accumulator one bit to the left. A zero is inserted in the least-significant bit of the accumulator and the most-significant bit of the accumulator is stored in the carry flag C. Flags affected: N, Z and C.
SHR_IMP	Shifts the contents of the accumulator one bit to the right. The most-significant bit of the accumulator does *not* change, while the least-significant bit of the accumulator is stored in the carry flag C. Notice that this is an *arithmetic shift right*. Flags affected: N, Z and C.
ROLC_IMP	Rotates the contents of the accumulator one bit to the left through the carry flag. The least-significant bit of the accumulator stores the original value of C, while the carry flag stores the most-significant bit of the accumulator. Flags affected: N, Z and C.
RORC_IMP	Rotates the contents of the accumulator one bit to the right through the carry flag. The most-significant bit of the accumulator stores the original value of C, while the carry flag stores the least-significant bit of the accumulator. Flags affected: N, Z and C.
INCA_IMP	Adds 1 to the contents of the accumulator. Flags affected: N and Z.
DECA_IMP	Subtracts 1 from the contents of the accumulator. Flags affected: N and Z.

Table 5.3: Instructions of the DIY.

Chapter 6

Related Work

In this chapter we overview previous work on synthesis and verification of hardware using theorem provers and functional languages. Although this is not an extensive survey on hardware verification, it provides a glimpse of more closely related work. In the last section we compare them with our approach.

6.1 LAMBDA

LAMBDA (Logic And Mathematics Behind Design Automation) is a tool-set to support interactive synthesis which integrates proof and design [19, 21]. The LAMBDA theorem prover sets a goal (correctness theorem) to be proved through synthesis. The user builds the circuit incrementally by connecting primitive hardware components using the schematic interface DIALOG. Each refinement step automatically simplifies the goal. The process continues until a circuit that implements the specification is constructed.

6.2 VERITAS$^+$

The approach called *Formal Synthesis* is, like LAMBDA, based on goal-directed and interactive design that develops a circuit and its proof of correctness simultaneously [36]. VERITAS$^+$ is the logic used to describe and synthesise circuits. It is an extension of classical (non-constructive) typed higher-order logic with dependent types and subtypes [20]. The designer interactively refines a behavioural specification into a structural design via proof.

6.3 DDD

The Digital Design Derivation (DDD) is an interactive transformation system that synthesises high-level specifications into hierarchical boolean systems [44]. The specifications are tail-recursive lambda abstractions (*S-expressions* in the Lisp dialect Scheme [66]), which essentially describe finite-state machines. For example, the function factorial is defined by Johnson et al. [45] as

```
(letrec ([FACT (lambda (N ACC)
   (if (zero?  N)
```

```
        ACC
        (FACT (- N 1) (* M ACC)))) ])
(FACT X 1)
```

The tail-recursive call is interpreted as a transfer of control to `FACT` when the registers `N` and `ACC` are simultaneously updated to the value `(- N 1)` and `(* M ACC)`, respectively.

DDD automates semantic preserving transformations like *folding*, *unfolding* and the distributive law for conditionals:

$$(\mathit{if}\ T\ (F\ A)\ (F\ B)) \Longleftrightarrow\ (F\ (\mathit{if}\ T\ A\ B))$$

System factorisation laws allow more elaborate architectural decomposition transformations, which rearrange the system hierarchy into communicating modules [43]. The user guides the application of these transformations in order to refine and optimise the specification until a synthesisable design is reached.

The DDD system contributes to the interplay between *verification* (construction of a proof of correctness after the fact) and *derivation* (correct-by-construction design). According to Bose [11], the decomposition and restructuring of the system by correct-by-construction transformations deal with the uninteresting portion of the design. The smaller building blocks that remain to be checked are better addressed by verification.

6.4 Gropius

Gropius is a hardware description language defined as a subset of HOL [8, 9, 10, 67]. It comprises four languages ranging from gate-level abstractions (Gropius 0) to system level structures (Gropius 3). The algorithmic level (Gropius 2) provides control structures like if-then-else, sequential composition and while loop. The atomic commands are DFGs (data flow graphs) represented by lambda abstractions. The compiler initially combines every while loop into a single one at the outermost level of the program:

$$\mathsf{PROGRAM}\ \mathit{out\_default}\ (\mathsf{LOCVAR}\ \mathit{vars}\ (\mathsf{WHILE}\ c\ (\mathsf{PARTIALIZE}\ b)))$$

The body b of the WHILE loop is an acyclic DFG. The list *out_default* provides initial values for the output variables. The term LOCVAR declares the local variables *vars* and PARTIALIZE converts a non-recursive (terminating) DFG into a potentially non-terminating command. The compiler then synthesises a handshaking interface which encapsulates this program. Each of these hardware blocks are now regarded as primitive blocks or *processes* at the system level. Processes are connected via communication units (*k-processes*) which implement delay, synchronisation, duplication, splitting and joining of a process output data (actually there are 10 different k-processes [8]). Just like in our approach, there are previously proved theorems which provide a correct implementation for every process and k-process.

6.5 Occam Synthesis

Bowen and He also applied the compositionality principle in the compilation of occam into netlists. This work was built from previous work for software compilation [12, 69].

The compilation is based on the refinement calculus and the algebraic laws of occam. Both source and target languages are subsets of occam.

For each language constructor, there is an implementation also described in occam which *refines* the source code. The assignment can be regarded as an atomic constructor.

$$\vdash\ (x := e)\ \sqsubseteq\ C$$

C is an occam program in a particular *normal form* which represents a netlist:

$$\texttt{var}\ s, f, \underline{l}, \underline{w}\ ;\ \mathit{Init}\ ;\ (\texttt{while}\ \neg f\ \texttt{do}\ \mathit{Step})\ ;\ \mathit{Final}\ ;\ \texttt{end}\ s, f, \underline{l}, \underline{w}$$

The variables s and f represent the control wires *start* and *finish*, respectively. The *list* of variables $\underline{l}$ and $\underline{w}$ are wires connected to delays and combinational components, respectively. The procedure *Init* assigns initial values to delays and stable states to combinational components. The activity of the circuit in one clock cycle during its computation is described by *Step* in terms of multiple assignments. Finally, the procedure *Final* defines the state of the circuit after the completion of its operation. We omit details of the interpreter which links the normal form occam program to a netlist-like notation [12].

The sequential composition of two circuits is refined by the function *Merge*.

$$\vdash\ (C_1\ \mathit{and}\ C_2\ \mathit{are\ well\ defined})\ \Rightarrow\ C_1\ ;\ C_2\ \sqsubseteq\ \mathit{Merge}(C_1, C_2)$$

$\mathit{Merge}(C_1, C_2)$ transforms C_1 and C_2 into a single circuit described in the normal form provided that C_1 and C_2 are well defined, i.e. they do not share output wires.

In addition to sequential composition, conditional command, while loop and channels have also been defined in a similar way. The compilation can be fully mechanised provided that the operators are monotonic with respect to $\sqsubseteq$. For instance, $(P_1 \sqsubseteq C_1) \wedge (P_2 \sqsubseteq C_2) \Rightarrow (P_1\ ;\ P_2) \sqsubseteq (C_1\ ;\ C_2)$.

6.6 Ruby

Ruby is a language for specifying and reasoning about hardware using a relational calculus [46]. A circuit is specified as a relation which enforces that the values of its outputs are consistent with those of its inputs.

The language provides combinators to specify circuit layouts which, together with their mathematical properties, provide a calculational style for design exploration. For example, the composition of two circuits (or relations) R and S is denoted by $R; S$. The composition is defined by $x\ (R; S)\ z \iff \exists y.\ (x\ R\ y) \wedge (y\ S\ z)$. Intuitively, the composition $R; S$ *suggests* a connection between two circuits made from one side of R to the other side of S. The associativity of composition is an example of the mathematical properties of these geometric combinators, which allow a correct-by-construction design.

6.7 Functional Languages

6.7.1 μFP

The functional language μFP is an extension of Backus' FP for specifying circuits at the structural level [2, 72]. The circuits are defined in terms of primitive functions over

booleans, numbers and lists and higher-order functions, the *combining forms*, which compose hardware blocks in different structures (sequence, parallel, conditional etc). Algebraic laws allow the designer to transform an abstract specification into a concrete circuit.

6.7.2 Lava

Lava is a hardware description language embedded in the functional language Haskell [5, 47]. Like Ruby and μFP, Lava also provides combinators that capture common circuit patterns. It also takes advantage of some of the Haskell features like monads and type classes to allow multiple interpretations of a single circuit description. For example, a half adder can be defined as:

```
halfAdd (a,b) =
   do carry <- and2 (a,b)
      sum <- xor2 (a,b)
      return (carry,sum)
```

Two interpretations for this half adder are presented by Bjesse et al. [5]. The *standard* interpretation is the identity monad, which simply evaluates the function in the boolean domain. Executing the program `halfAdd(high,high)` produces the pair `(high,low)`. Alternatively, the *symbolic* interpretation generates descriptions of circuits to be provided to external tools. For example, running the half adder code under the symbolic interpretation generates the netlist

```
["b3" := And [BitVar "b1", BitVar "b2"],
 "b4" := Xor [BitVar "b1", BitVar "b2"]].
```

These circuits can be verified with respect to a relation that restricts the values of the inputs and outputs. For example, the following code poses the question whether a full adder with low carry is equivalent to a half adder.

```
question =
   do a <- newBitVar -- fresh variables
      b <- newBitVar
      out1 <- halfAdd (a,b)
      out1 <- fullAdd (low,a,b)
      equals (out1,out2)
```

The fresh variables `a` and `b` are the inputs of both circuits.

In order to check this formula, the function `verify` takes the description of the question (under the symbolic interpretation) and generates a file containing a logical formula. The execution of `verify question >>= print` generates input to an external theorem prover like Otter [54] and Gandalf [80] and prints its result (in this case, `Valid`).

6.7.3 Lustre

Lustre is a functional language operating on *streams* [34]. A stream can be considered a finite or an infinite sequence of values of the same type. Lustre takes a *synchronous dataflow* approach. A program has a cyclic behaviour: at the nth execution cycle, all the streams are 'evaluated' to their nth value.

For example, a program which detects a positive edge on the Boolean stream

$$X = (x_1, x_2, x_3, \ldots)$$

is defined below.

```
node POSEDGE (X:bool) return (Y:bool);
let
   Y = false -> X and not pre(X);
tel
```

The components in Lustre are called *nodes*. The operator `->` initialises the stream `Y` with the value `false`. The remaining values of `Y` evaluate to true if the current value of `X` is true and its previous value `pre(X)` is false. Otherwise, `Y` is false.

Formal verification of Lustre programs is achieved by program comparison. This approach is similar to the one adopted by Lava. Given two programs (a specification and an implementation), a verification program runs the specification and the implementation in parallel over the same inputs. It returns some correctness relation between the specification and the implementation outputs. The Lustre compiler produces an automaton for this system and checks if the output is always true.

6.7.4 Hydra

Hydra is a software system which comprises a hardware description language (embedded in Haskell) and a set of tools for simulation, synthesis and timing analysis [61, 62]. The language allows different semantics to correspond to the same circuit specification — thanks to Haskell's function overloading based on type. For example, the same specification of, say, an inverter can be executed if applied to a Boolean or can generate a netlist if applied to a wire name. It also provides facilities to define general n-bit circuits (*circuit patterns*) and interactive formal equational reasoning [63].

6.7.5 Hawk

Hawk is another hardware description language based on Haskell [18]. It has been used in the specification and verification of complex microprocessor pipelines at the micro-architectural level. The designer benefits from several features like polymorphism, type-classes, higher-order functions, lazy evaluation and state monad.

The language has a built-in type *signal* used to model wires. The signals can be regarded as infinite sequences where the clock cycle is the index. For example:

```
toggle = True, False, True, False, ...
primes =  2, 3, 5, 7, 11, 13, 17, ...
```

The signal `toggle` alternates between `True` and `False` and the signal `primes` produces a prime number at every clock tick. Hawk also provides several built-in functions to construct and manipulate signals. For instance, `bundle` transforms a pair of signals into a signal of pairs:

```
bundle (primes,toggle) = (2,True), (3,False), (5,True), ...
```

The components of a microprocessor are functions from signals to signals. A simulation is produced by solving the mutually dependent equations of the specification using lazy evaluation.

The micro-architectural components satisfy simple algebraic laws, which are used to simplify a pipelined micro-architecture. These transformation laws have been mechanised in the Isabelle theorem prover [51, 52].

6.7.6 ReFLect

reFLect is a strongly typed functional language designed and implemented at Intel's Strategic CAD Labs [32]. The language has *reflection* features like quotation and anti-quotation constructs in a LISP-like fashion.

The language has being designed for applications in industrial hardware design and theorem proving. In particular, circuit descriptions are functional programs which can be simulated, exported to other design tool, or transformed by functions that traverse its abstract syntax (thanks to the quotation and anti-quotation facilities).

A hardware design language is embedded in *reFLect* and its circuits are constructed from primitives using higher-order functions in a similar approach adopted by Sheeran in μFP [72].

6.7.7 SAFL

SAFL (Statically Allocated Functional Language) is a first-order functional language [57, 71]. A user program consists of a sequence of function definitions in which all recursive calls are tail-recursive. The compiler translates each function into a single hardware block. Its high-level of abstraction allows the exploitation of powerful program analyses and optimisations not available in traditional synthesis systems. For instance, the functional properties of SAFL allow equational reasoning and therefore the application of semantic-preserving source-to-source manipulations.

6.7.8 SASL

SASL (Statically Allocated Stream Language) is an extension of SAFL which improves its I/O model and implements common functional features such as closures and lazy evaluation [24].

The idea behind this extension is to address the limitations of SAFL's call-return I/O mechanism. For instance, no pipelining or state held between calls are possible. SASL's I/O model treats input and output as lazily evaluated lists (called *streams*). Instead of dealing with a potentially infinite lists, *lazy* lists store only the information required to generate the lists. For instance, the input streams are not read until they are needed.

SASL's functions process a combination of scalar values and lazy lists. Contrary to the evaluation of lists, scalar variables are evaluated eagerly.

6.7.9 SHard

Shard [68] is a prototype compiler which transforms a functional subset of Scheme [66] into hardware. The compiler supports tail and non tail-recursive functions. Non tail-recursive

functions are transformed into tail-recursive ones by converting them to continuation-passing style (CPS) [1].

The compilation is carried out by several phases of source-to-source transformations based on control flow analysis. The outcome of the main compilation phases is mapped to a dataflow circuit which connects instances of 9 generic hardware components. Each component implements a generic computation like parallel composition or conditional.

This intermediate representation can either be simulated in Scheme or implemented in VHDL for synthesis in an FPGA. No formal verification is performed.

6.8 Comparative Analysis

LAMBDA and VERITAS$^+$ are approaches based on theorem proving systems. Both synthesis and verification are performed simultaneously in an interactive way. Each new project has to be verified under user guidance. In our approach we pre-verified (interactively, but only once) primitive composable circuits, which allows the synthesis and the verification to be carried out automatically, except for the proof of termination. DDD follows a slightly different approach. It combines synthesis by algebraic refinement (semantic preserving transformations) and direct proof. However, its synthesis process also depends on substantial user guidance.

Ruby's emphasis is on circuit layout. The language provides a large set of geometric combinators which allow correct-by-construction design exploration. Our approach does not address layout issues, but emphasises the functional correctness.

Gropius has a similar verification method based on pre-proved theorems which provide correct implementations. In addition to that, Gropius also offers a large variety of interfaces. The key difference to our approach is the lack of composable theorems to certify the correctness of top-level systems combined by the k-processes. The theorems prove the correctness of each sub-system.

Several approaches use a functional language as a hardware description language. Lustre, SHard, SAFL and SASL model hardware systems at the behavioural level. Hawk specifications are at the micro-architectural level. The other functional languages described above model circuits at the structural level, which is equivalent to our *target* language.

The main difference between Lustre, Hawk and SASL and our source language is the usage of streams. In Lustre, streams have a special impact on loops. While our source language uses tail-recursive function calls, the elements of a Lustre stream represent iterations. For each function call, either one step of the iteration is executed or the end of the loop is reached (the environment resets the loop).

Our approach is partially inspired by SAFL, especially the ideas in Richard Sharp's PhD [71]. However, instead of proposing a concrete functional language, our source language is a subset of higher-order logic. The compilation of SAFL into circuits applies the compositionality principle to primitive hardware components. However, SAFL synthesis is not based on correct-by-construction transformations and the compiler has not been verified. Inspired by the SAFL's compilation method, we proposed a similar compilation which, instead of using composable circuits, uses composable theorems.

The same compositionality principle adopted by SAFL (and which inspired our compilation) was previously used by Bowen and He for the compilation of occam. They regard

compilation as a task of program refinement following a previous approach for software compilation [69]. The theorems are identical to ours in nature (with respect to compositionality), but the underlying concepts and formalism are different. The theorems were proved from refinement calculi and algebraic laws of occam. Our proofs are in higher-order logic and are built from axioms, primitive constants and primitive and derived rules of inference of HOL. Their approach is also amenable to mechanisation, which has been done in Prolog.

6.8.1 Summary

The comparative analysis is summarised in Table 6.1.

	Level of Abstraction		Verification	
	Structural	Behavioural	Automatic	Semi-automatic[a]
LAMBDA	√			√
VERITAS$^+$		√		√
DDD		√		√
Gropius		√	√[b]	
occam		√	√	
Ruby	√			√
μFP	√			√
Lava	√		√	
Lustre		√	√	
Hydra	√			√
Hawk	√[c]			√
reFLect	√			√
SAFL		√		
SASL		√		
SHard		√		
HOL4		√	√[d]	

[a] Semi-automatic or manual verification.
[b] Except at the top-level system.
[c] Actually, micro-architectural level.
[d] Semi-automatic proof of termination sometimes required.

Table 6.1: Comparative analysis.

In what follows, we describe the main features that distinguish our approach.

High level of abstraction. Our source language is the subset of higher-order logic which constitutes a pure first-order functional language. The idea is to allow the designer to focus on a solution at a higher level of abstraction without having to manipulate and reason about circuits at the gate level.

Automatic verification. The translation of a tail-recursive function to a circuit is done by proof. Our compiler automatically generates a circuit in a theorem which certifies its correctness. There is no need for interactive user guidance in the verification process (except for termination, although such proofs are produced mostly automatically thanks to the TFL package [74]).

Theorem prover as a compiler. Our compiler is built on top of the HOL4 system [59]. The designer can use all the facilities provided by HOL4 to interactively prove properties of the tail-recursive programs, which are mathematical functions in HOL. Alternatively, the circuit development can start from a specification in higher-order logic, which can be formally refined interactively to tail-recursive definitions. Once this refinement or abstraction is achieved, the compiler automatically links (by the correctness theorems) the top level specification to the generated circuit. In either case, the lowest level of abstraction involved in *interactive* verification is that of a tail-recursive function.

Chapter 7

Conclusion

We have presented an approach to create formally verified circuits. In this project we faced two main challenges: the translation of a high-level functional language to gate-level netlists; and the development of a fully automatic verification method which proves that a circuit implements a function.

Instead of verifying the compiler, we verify the correctness of five circuit constructors. First we developed circuit constructors for each source language constructor. Circuit constructors are parameterised circuits which are built from undefined sub-circuits. They do not compute anything in particular, but we proved that they implement general computations like sequential composition or conditional commands. These theorems have been proved interactively following a standard approach. However, they satisfy an interesting property. The theorems have a particular structure which allows HOL4 to automatically compose them in order to prove new, more elaborate, but still composable theorems. At the end of the process, we reach a theorem which precisely states the correctness of the synthesised design. Circuits are extracted from the correctness theorems.

The main outcome of this work is a compiler which automatically generates a theorem stating that a circuit implements the source code. The compiler is a proof-of-concept which supports the ideas we developed to mechanically compose proofs. We have tested our ideas with some fairly substantial case studies, and more examples have been done by our colleagues in Utah [76].

7.1 Lessons Learnt

The *undefined values* and the *combinational loop* problems (sections 4.2 and 4.4) are good examples of the limitations of formal methods. The meaning of a formally verified system must be interpreted with care. We must have a clear understanding of the scope and the limitations of a mathematical model. By applying formal methods to our approach, we do not claim the development of perfect, infallible systems. Proofs of correctness are only as good as their underlying model and therefore *nearly* guarantee that an implementation meets its specification.

The mathematical model employed must capture the real world behaviour precisely. For example, if our real world is an FPGA, then it seems reasonable to deal with Boolean values only. However, if the real world is a simulator or any other technology in which intermediate values between T and F could strongly affect a circuit's behaviour, then we

should use a four valued logic. For our purposes, the simpler model based on Boolean values seems to be adequate. Although our model also does not formalise detailed aspects of a circuit like transistor-level behaviour, electrical effects or signal propagation time, our methodology could, in principle, still be used with any detailed hardware models and eventually capture aspects related to say, combinational loops.

Our experiences have shown that neither a single mathematical model nor an elaborate specification is able to cover all aspects involved in the production of totally reliable systems. See the seminal papers by Cohn [17] and Hall [35] on these issues. Nevertheless, we can still find examples that, we hope, illustrate how effective formal methods can be. We found few bugs when running our circuits from the FPGA. One of the most difficult to catch happened during the tests of the circuit implementing the encryption algorithm TEA. The circuit was apparently computing the encryption and the decryption correctly, but only occasionally did it satisfy the formally verified property that a message does not change by an encryption followed by a decryption. After debugging the compiler, we found out that it was caused by translating the HOL4 *arithmetic* shift right to the Verilog *logical* shift right. Another bug occurred during tests of the Booth multiplier. The multiplication was working only for few random values. We later discovered that it was caused by a bug in the software used to connect HOL4 to the serial cable. In spite of finding bugs in our system, we never found any bug related to the components which were formally verified.

7.2 Final Remarks

We have developed a proof-producing compilation method based on composable theorems. We show how automatic composition of proofs can raise the level of mechanisation in hardware verification. The user is no longer required to carry out proofs at the gate level.

As future work, more optimisations should be proposed to make the compiler feasible to larger projects. Alternatively, the same technology could be applied to different domains. For instance, our approach is currently being used to verify software. Konrad Slind, Guodong Li and Scott Owens from the University of Utah are developing a compiler to translate a subset of HOL to a subset of the ARM machine code [75].

It is too early to claim that this method will be considered practical in developing compilers. Our compiler is based on an incipient technology which does not scale to industrial-sized applications yet. However it is just the first attempt to produce a fully automatic verification of circuits with respect to a functional program. Its contributions to hardware verification make us believe it is a technology worth at putting more efforts on in future.

Appendix A

The DIY Specifications

This chapter presents the complete specifications of the DIY microcomputer for both designs presented above. We include the whole programs here for completeness and to give an idea of the scale that can be handled by our compiler.

Some functions in ML are used to define the specifications. The function `Define` takes a high-level specification of a HOL function and defines it in the logic. The function `hwDefine2` extends `Define` by returning, in addition to the usual function definition, some auxiliary theorems like those stating the correctness of the circuit, or the ones specifying the function in the intermediate language. These auxiliary theorems are used by the compiler in a later phase of the development process (not shown here). The expression `(e::l)` denotes a list whose head is `e` and tail is `l`, while (`val v = e;`) defines the constant `v` to have the value of `e`.

A.1 The CPU Design

```
(*---------------------------------------------------------------------------*)
(*  Instructions                                                             *)
(*---------------------------------------------------------------------------*)
val STA_ABS_def = Define `STA_ABS = 153w:word8`;
val LDA_ABS_def = Define `LDA_ABS = 145w:word8`;
val LDA_IMM_def = Define `LDA_IMM = 144w:word8`;
val JNO_ABS_def = Define `JNO_ABS = 238w:word8`; (* not overflow    *)
val JO_ABS_def  = Define `JO_ABS  = 233w:word8`; (* overflow        *)
val JNC_ABS_def = Define `JNC_ABS = 230w:word8`; (* not carry       *)
val JC_ABS_def  = Define `JC_ABS  = 225w:word8`; (* if carry        *)
val JNN_ABS_def = Define `JNN_ABS = 222w:word8`; (* not negative    *)
val JN_ABS_def  = Define `JN_ABS  = 217w:word8`; (* negative        *)
val JNZ_ABS_def = Define `JNZ_ABS = 214w:word8`; (* not zero        *)
val JZ_ABS_def  = Define `JZ_ABS  = 209w:word8`; (* zero            *)
val JSR_ABS_def = Define `JSR_ABS = 201w:word8`; (* subroutine      *)
val JMP_ABS_def = Define `JMP_ABS = 193w:word8`; (* unconditional   *)
val ADD_ABS_def  = Define `ADD_ABS  = 17w:word8`;
val ADDC_ABS_def = Define `ADDC_ABS = 25w:word8`;
val SUB_ABS_def  = Define `SUB_ABS  = 33w:word8`;
val SUBC_ABS_def = Define `SUBC_ABS = 41w:word8`;
val AND_ABS_def  = Define `AND_ABS  = 49w:word8`;
val OR_ABS_def   = Define `OR_ABS   = 57w:word8`;
val XOR_ABS_def  = Define `XOR_ABS  = 65w:word8`;
```

```
val CMPA_ABS_def = Define `CMPA_ABS = 97w:word8`;
val SHL_IMP_def  = Define `SHL_IMP  = 112w:word8`;
val SHR_IMP_def  = Define `SHR_IMP  = 113w:word8`;
val ROLC_IMP_def = Define `ROLC_IMP = 120w:word8`;
val RORC_IMP_def = Define `RORC_IMP = 121w:word8`;
val INCA_IMP_def = Define `INCA_IMP = 128w:word8`;
val DECA_IMP_def = Define `DECA_IMP = 129w:word8`;

val instructions =
        [LDA_IMM_def, LDA_ABS_def, STA_ABS_def, JC_ABS_def, JMP_ABS_def,
         JN_ABS_def, JNC_ABS_def, JNN_ABS_def, JNO_ABS_def, JNZ_ABS_def,
         JO_ABS_def, JSR_ABS_def, JZ_ABS_def, ADD_ABS_def, ADDC_ABS_def,
         SUB_ABS_def, SUBC_ABS_def, AND_ABS_def, OR_ABS_def, XOR_ABS_def,
         CMPA_ABS_def, SHL_IMP_def, SHR_IMP_def, ROLC_IMP_def, RORC_IMP_def,
         INCA_IMP_def, DECA_IMP_def];

(*--------------------------------------------------------------------------*)
(* b2w                                                                      *)
(* Converts a Boolean into an 8-bit word                                    *)
(*--------------------------------------------------------------------------*)
val (b2w_def,_,b2w_dev0,b2w_comb,_)= hwDefine2
  `b2w(b) = if b then (1w:word8) else (0w:word8)`;

(*--------------------------------------------------------------------------*)
(* Constants                                                                *)
(* There are four steps to execute an instruction using absolute addressing:  *)
(* 1. Fetch the instruction (FETCH),                                        *)
(* 2. Load the least-significant byte of the operand (LD_LSB)               *)
(* 3. Load the most-significant byte of the operand (LD_MSB)                *)
(* 4. Execute the instruction (RUN)                                         *)
(*--------------------------------------------------------------------------*)
val FETCH_def  = Define `FETCH  = 0w:word4`;
val LD_LSB_def = Define `LD_LSB = 1w:word4`;
val LD_MSB_def = Define `LD_MSB = 2w:word4`;
val RUN_def    = Define `RUN    = 3w:word4`;

val constants = FETCH_def :: LD_LSB_def :: LD_MSB_def ::
                RUN_def :: instructions;

(*--------------------------------------------------------------------------*)
(* abs                                                                      *)
(* 8-bit numbers ranging from 128 to 255 (interpreted as unsigned) correspond *)
(* to the negative values from -128 to -1. This function converts numbers     *)
(* from 128 to 255 into its absolute value in the signed system.            *)
(* Example. abs(255) = 1 (unsigned 8-bit numbers whose value is 255 is      *)
(* interpreted as -1 in the signed system.                                  *)
(*--------------------------------------------------------------------------*)
val (abs_def,_,abs_dev0,abs_comb,_) = hwDefine2
   `abs(n:num) = (255-n)+1`;

(*--------------------------------------------------------------------------*)
(* add_ov                                                                   *)
(* Takes two signed 8-bit numbers and checks if their addition generates    *)
(* overflow.                                                                *)
(*--------------------------------------------------------------------------*)
val (add_ov_def,_,add_ov_dev0,add_ov_comb,_) = hwDefine2
```

```
   ‘add_ov (a:word8,b:word8)
       = (word_msb(a) /\ word_msb(b) /\ (abs(w2n a)+abs(w2n b) > 128)) \/
          ((w2n(a) < 128) /\ (w2n(b) < 128) /\ (w2n(a)+w2n(b) > 127))‘;

(*------------------------------------------------------------------------*)
(* sub_ov                                                                 *)
(* Takes two signed 8-bit numbers and checks if their subtractoin generates   *)
(* overflow.                                                              *)
(*------------------------------------------------------------------------*)
val (sub_ov_def,_,sub_ov_dev0,sub_ov_comb,_) = hwDefine2
   ‘sub_ov (a:word8,b:word8)
       = (word_msb(a) /\ (w2n(b) < 128) /\ (abs(w2n a)+(w2n b) > 128)) \/
         ((w2n(a) < 128) /\ word_msb(b) /\ ((w2n a)+abs(w2n(b)) > 127))‘;


(*------------------------------------------------------------------------*)
(* isJUMP                                                                 *)
(* Tests an instruction code to identify if it is a jump                  *)
(*------------------------------------------------------------------------*)
val (isJUMP_def,_,isJUMP_dev0,isJUMP_comb,_) = hwDefine2
   ‘isJUMP (ins:word8) = w2n(ins) > 192‘;

(*------------------------------------------------------------------------*)
(* alu                                                                    *)
(*------------------------------------------------------------------------*)
val (alu_def,_,alu_dev0,alu_comb,_) = hwDefine2
   ‘alu (data:word8,ACC:word8,PC:word16,
         IR:word8, step:word4,
         SP:word16, TMP:word32, OV: bool, N:bool, Z:bool, C:word8)
    = if IR = LDA_ABS then
         (
         (* read  *) F,       (* write *) T,     (* addr  *) PC,
         (* data  *) data,    (* ACC   *) data,  (* PC    *) PC+1w,
         (* IR    *) IR,      (* step  *) FETCH, (* SP    *) SP,
         (* TMP   *) TMP,     (* OV    *) OV,   (* N     *) word_msb(data),
         (* Z     *) data=0w, (* C     *) C
         )
      else if IR = ADD_ABS then
        let result = ACC+data
        in
          (F,T,PC,data,
           (* ACC *) result,  PC+1w, IR, FETCH, SP, TMP,
           (* OV *) add_ov(ACC,data), (* N  *) word_msb(result),
           (* Z  *) result=0w,        (* C  *) b2w(w2n(ACC) > (255-w2n(data)))
           )
      else if IR = ADDC_ABS then
        let result = ACC+data+C
        in
          (F,T,PC,data,
           (* ACC *) result, PC+1w, IR, FETCH, SP, TMP,
           (* OV *) add_ov(ACC,data+C), (* N  *) word_msb (result),
           (* Z  *) result=0w,
           (* C  *) b2w(w2n(ACC) > (255-(w2n(data)+w2n(C))))
           )
      else if IR = SUB_ABS then
        let result = ACC-data
```

```
  in
    (F,T,PC,data,
     (* ACC *) result, PC+1w, IR, FETCH, SP, TMP,
     (* OV *) sub_ov(ACC,data),  (* N  *) word_msb(result),
     (* Z  *) result=0w,        (* C  *) b2w(w2n(data) > w2n(ACC))
     )
else if IR = SUBC_ABS then
   let result = ACC - (data + C)
   in
     (F,T,PC,data,
     (* ACC *) result, PC+1w, IR, FETCH, SP, TMP,
     (* OV *) sub_ov(ACC,data+C), (* N  *) word_msb (result),
     (* Z  *) result = 0w,
     (* C  *) b2w(w2n(data)+w2n(C) > w2n(ACC))
     )
else if IR = AND_ABS then
   let result = word_and data ACC
   in
     (F,T,PC,data, result,  PC+1w, IR, FETCH, SP, TMP,
     (* OV *) OV,        (* N  *) word_msb(result),
     (* Z  *) result=0w, (* C  *) C
     )
else if IR = OR_ABS then
   let result = word_or data ACC
   in
     (F,T,PC,data, result, PC+1w, IR, FETCH, SP, TMP,
     (* OV *) OV,        (* N  *) word_msb(result),
     (* Z  *) result=0w, (* C  *) C
     )
else if IR = XOR_ABS then
   let result = word_xor data ACC
   in
     (F,T,PC,data, result, PC+1w, IR, FETCH, SP, TMP,
     (* OV *) OV,        (* N  *) word_msb(result),
     (* Z  *) result=0w, (* C  *) C
     )
else if IR = CMPA_ABS then
   (F,T,PC,data, ACC, PC+1w, IR, FETCH, SP, TMP,
   (* OV *) OV,       (* N  *) N,
   (* Z  *) ACC=data, (* C  *) b2w(w2n(ACC) > w2n(data))
   )
else if IR = SHL_IMP then
   let result = ((7><0) (ACC << 1):word8):word8
   in
      (F,T,PC-1w,data,result,PC,IR,FETCH,SP,TMP,
      (* OV *) OV,          (* N  *) word_msb(result),
      (* Z  *) result = 0w, (* C  *) b2w(word_msb ACC)
      )
else if IR = SHR_IMP then
   let result = ((7><0) (ACC >> 1):word8):word8
   in
      (F,T,PC-1w,data,result,PC, IR, FETCH, SP, TMP,
      (* OV *) OV,          (* N  *) word_msb(result),
      (* Z  *) result = 0w, (* C  *) b2w(word_lsb ACC)
      )
else if IR = ROLC_IMP then
```

```
         let result = (((7><0) (ACC << 1):word8):word8) + C
         in
            (F,T,PC-1w,data,result,PC, IR, FETCH, SP, TMP,
            (* OV *) OV,           (* N  *) word_msb(result),
            (* Z  *) result = 0w,  (* C  *) b2w(word_msb ACC)
            )
      else if IR = RORC_IMP then
         let result = ((8><1)
              ((word_ror ((ACC @@ (((0><0) C):word1)):word9) 1):word9)):word8
         in
            (F,T,PC-1w,data,result,PC, IR, FETCH, SP, TMP,
            (* OV *) OV,          (* N  *) word_msb(result),
            (* Z  *) result = 0w, (* C  *) b2w(word_lsb ACC)
            )
      else if IR = INCA_IMP then
         let result = ACC+1w
         in
            (F,T,PC-1w,data,result,PC, IR, FETCH, SP, TMP,
            (* OV *) OV,         (* N  *) word_msb(result),
            (* Z  *) result = 0w,(* C  *) C
            )
      else if IR = DECA_IMP then
         let result = ACC-1w
         in
            (F,T,PC-1w,data,result,PC, IR, FETCH, SP, TMP,
            (* OV *) OV,          (* N  *) word_msb(result),
            (* Z  *) result = 0w, (* C  *) C
            )
      else
         (F,T,PC,data,data,PC+1w, IR, FETCH, SP, TMP,
         (* OVNZC *) OV,N,Z,C
         )
‘;


(*-------------------------------------------------------------------------*)
(* testJUMP                                                                *)
(* Takes the instruction code and the status registers and check if a jump  *)
(* has to be executed.                                                     *)
(*-------------------------------------------------------------------------*)
val (testJUMP_def,_,testJUMP_dev0,testJUMP_comb,_) = hwDefine2
   ‘testJUMP (INS:word8,OV:bool,N:bool,Z:bool,C:word8) =
      ((INS=JC_ABS)  /\ (C=1w))  \/ ((INS=JNC_ABS) /\ (C=0w))  \/
      ((INS=JN_ABS)  /\ N)       \/ ((INS=JNN_ABS) /\ ~N)      \/
      ((INS=JO_ABS)  /\ OV)      \/ ((INS=JNO_ABS) /\ ~OV)     \/
      ((INS=JZ_ABS)  /\ Z)       \/ ((INS=JNZ_ABS) /\ ~Z)      \/
      (INS=JMP_ABS)‘;

(*-------------------------------------------------------------------------*)
(* implied_addr                                                            *)
(* Tests if an instruction uses implied addressing mode.                   *)
(*-------------------------------------------------------------------------*)
val (implied_addr_def,_,implied_addr_dev0,implied_addr_comb,_) = hwDefine2
   ‘implied_addr(INS) =
       (INS=INCA_IMP) \/ (INS=DECA_IMP) \/ (INS=SHL_IMP)  \/ (INS=SHR_IMP)  \/
       (INS=ROLC_IMP) \/ (INS=RORC_IMP)‘;
```

```
(*-----------------------------------------------------------------------------*)
(* DIY Calculator                                                              *)
(*                                                                             *)
(*  The input:                                                                 *)
(*                                                                             *)
(*   reset :bool (active low)                                                  *)
(*   data  :word8 (from memory)                                                *)
(*   regs  :word8 # word16 # word8 # word4 # word16 # word32                    *)
(*          # bool # bool # bool # word8                                        *)
(*                                                                             *)
(*  The output:                                                                *)
(*                                                                             *)
(*   read  :bool (active low)                                                  *)
(*   write :bool (active low)                                                  *)
(*   addr  :word16 (address)                                                   *)
(*   data  :word8 (to memory)                                                  *)
(*   regs  :word8 # word16 # word8 # word4 # word16 # word32                    *)
(*          # bool # bool # bool # word8                                        *)
(*                                                                             *)
(*  The registers (regs):                                                      *)
(*                                                                             *)
(*   ACC  :word8 (accumulator)                                                 *)
(*   PC   :word16 (program counter)                                            *)
(*   IR   :word8 (instruction register)                                        *)
(*   step :word4 (state)                                                       *)
(*   SP   :word16 (stack pointer)                                              *)
(*   TMP  :word32 (temporary register)                                         *)
(*   OV   :bool (overflow flag)                                                *)
(*   N    :bool (negative flag)                                                *)
(*   Z    :bool (zero flag)                                                    *)
(*   C    :word8 (carry flag)                                                  *)
(*-----------------------------------------------------------------------------*)

val (diy_def,_,diy_dev0,diy_comb,_) = hwDefine2
   `diy (reset:bool,data:word8,ACC:word8,PC:word16,
         IR:word8, step:word4,
         SPTMPOVNZC: word16 # word32 # bool # bool # bool # word8)
    = if (~reset) then (* C.11-C.12 *)
         (F,T,0w:word16,data,ACC,1w:word16,0w:word8,0w:word4,SPTMPOVNZC)
      else if step = FETCH then
         let (SP:word16,TMP:word32,OVNZC) = SPTMPOVNZC
         in
           if (~(isJUMP(data)) \/ testJUMP(data,OVNZC)) then
              (* The instruction code is stored in IR.
                 Request the LS byte of the operand's addr
              *)
              (
              (* read  *) F,
              (* write *) T,
              (* addr  *) PC,
              (* data  *) data,
              (* ACC   *) ACC,
              (* PC    *) PC+1w,
              (* IR    *) data,
              (* step  *) if implied_addr(data) then RUN else LD_LSB,
```

```
            (* remaining *) SPTMPOVNZC
            )
         else (* Jump which failed the test: skip the operands *)
            (
            (* read  *) F,
            (* write *) T,
            (* addr  *) PC+2w,
            (* data  *) data,
            (* ACC   *) ACC,
            (* PC    *) PC+3w,
            (* IR    *) data,
            (* step  *) FETCH,
            (* SP    *) SPTMPOVNZC
            )
      else if step = LD_LSB then
         let (SP:word16,TMP:word32,OVNZC) = SPTMPOVNZC
         in
           (* The LS byte of the operand's addr is stored in TMP
              Request the MS byte of the operand's addr
           *)
           (
           (* read  *) F,
           (* write *) T,
           (* addr  *) PC,
           (* data  *) data,
           (* ACC   *) ACC,
           (* PC    *) PC+1w,
           (* IR    *) IR,
           (* step  *) LD_MSB,
           (* SP    *) SP,
           (* TMP   *) (0w:word24) @@ data,
           (* remaining *) OVNZC
           )
      else if step = LD_MSB then
         let (SP:word16,TMP:word32,OVNZC) = SPTMPOVNZC
         in
           (* The MS+LS is placed at the addr bus in order to read
              the operand.
           *)
           (
           (* read  *) IR=STA_ABS,     (* If IR=STORE, disable read *)
           (* write *) ~(IR = STA_ABS),(* If IR=STORE, enable write *)
           (* addr  *) (data @@ ((7 >< 0) TMP):word8):word16,
           (* data  *) ACC,            (* If IR=STORE data=ACC else don't care *)
           (* ACC   *) ACC,
           (* PC    *) if isJUMP(IR) then ((data @@ ((7><0) TMP):word8)+1w) else PC,
           (* IR    *) IR,
           (* step  *) if isJUMP(IR) then FETCH else RUN,
           (* SP    *) SP,
           (* TMP   *) TMP,
           (* OVNZC *) OVNZC
           )
       else alu(data,ACC,PC,IR,step,SPTMPOVNZC)
`;
```

A.2 The Microcomputer Design

```
(*-------------------------------------------------------------------------*)
(*  Instructions                                                           *)
(*-------------------------------------------------------------------------*)
val STA_ABS_def = Define `STA_ABS = 153w:word8`;
val LDA_ABS_def = Define `LDA_ABS = 145w:word8`;
val LDA_IMM_def = Define `LDA_IMM = 144w:word8`;
val JNO_ABS_def = Define `JNO_ABS = 238w:word8`; (* not overflow    *)
val JO_ABS_def  = Define `JO_ABS  = 233w:word8`; (* overflow        *)
val JNC_ABS_def = Define `JNC_ABS = 230w:word8`; (* not carry       *)
val JC_ABS_def  = Define `JC_ABS  = 225w:word8`; (* if carry        *)
val JNN_ABS_def = Define `JNN_ABS = 222w:word8`; (* not negative    *)
val JN_ABS_def  = Define `JN_ABS  = 217w:word8`; (* negative        *)
val JNZ_ABS_def = Define `JNZ_ABS = 214w:word8`; (* not zero        *)
val JZ_ABS_def  = Define `JZ_ABS  = 209w:word8`; (* zero            *)
val JSR_ABS_def = Define `JSR_ABS = 201w:word8`; (* subroutine      *)
val JMP_ABS_def = Define `JMP_ABS = 193w:word8`; (* unconditional   *)
val ADD_ABS_def  = Define `ADD_ABS  = 17w:word8`;
val ADDC_ABS_def = Define `ADDC_ABS = 25w:word8`;
val SUB_ABS_def  = Define `SUB_ABS  = 33w:word8`;
val SUBC_ABS_def = Define `SUBC_ABS = 41w:word8`;
val AND_ABS_def  = Define `AND_ABS  = 49w:word8`;
val OR_ABS_def   = Define `OR_ABS   = 57w:word8`;
val XOR_ABS_def  = Define `XOR_ABS  = 65w:word8`;
val CMPA_ABS_def = Define `CMPA_ABS = 97w:word8`;
val SHL_IMP_def  = Define `SHL_IMP  = 112w:word8`;
val SHR_IMP_def  = Define `SHR_IMP  = 113w:word8`;
val ROLC_IMP_def = Define `ROLC_IMP = 120w:word8`;
val RORC_IMP_def = Define `RORC_IMP = 121w:word8`;
val INCA_IMP_def = Define `INCA_IMP = 128w:word8`;
val DECA_IMP_def = Define `DECA_IMP = 129w:word8`;

val instructions =
        [LDA_IMM_def, LDA_ABS_def, STA_ABS_def, JC_ABS_def, JMP_ABS_def,
         JN_ABS_def, JNC_ABS_def, JNN_ABS_def, JNO_ABS_def, JNZ_ABS_def,
         JO_ABS_def, JSR_ABS_def, JZ_ABS_def, ADD_ABS_def, ADDC_ABS_def,
         SUB_ABS_def, SUBC_ABS_def, AND_ABS_def, OR_ABS_def, XOR_ABS_def,
         CMPA_ABS_def, SHL_IMP_def, SHR_IMP_def, ROLC_IMP_def, RORC_IMP_def,
         INCA_IMP_def, DECA_IMP_def];

(*-------------------------------------------------------------------------*)
(* Constants                                                               *)
(* End of program is the highest address in memory available.              *)
(* We have no space for a big memory, so if a program reaches address 12,  *)
(* then it means it is finished. This constant is used as a termination    *)
(* condition                                                               *)
(*-------------------------------------------------------------------------*)
val end_of_program_def = Define `end_of_program = (12w:word16)`;

val constants = end_of_program_def :: instructions;

(*-------------------------------------------------------------------------*)
(* b2w                                                                     *)
(* Converts a Boolean into an 8-bit word                                   *)
(*-------------------------------------------------------------------------*)
```

```
val (b2w_def,_,b2w_dev0,b2w_comb,_)= hwDefine2
  `b2w(b) = if b then (1w:word8) else (0w:word8)`;

(*--------------------------------------------------------------------------*)
(* abs                                                                      *)
(* 8-bit numbers ranging from 128 to 255 (interpreted as unsigned) correspond *)
(* to the negative values from -128 to -1. This function converts numbers     *)
(* from 128 to 255 into its absolute value in the signed system.              *)
(* Example. abs(255) = 1 (unsigned 8-bit numbers whose value is 255 is        *)
(* interpreted as -1 in the signed system.                                  *)
(*--------------------------------------------------------------------------*)
val (abs_def,_,abs_dev0,abs_comb,_) = hwDefine2
   `abs(n:num) = (255-n)+1`;

(*--------------------------------------------------------------------------*)
(* add_ov                                                                   *)
(* Takes two signed 8-bit numbers and checks if their addition generates      *)
(* overflow.                                                                *)
(*--------------------------------------------------------------------------*)
val (add_ov_def,_,add_ov_dev0,add_ov_comb,_) = hwDefine2
   `add_ov (a:word8,b:word8)
       = (word_msb(a) /\ word_msb(b) /\ (abs(w2n a)+abs(w2n b) > 128)) \/
          ((a <+ 128w) /\ (b <+ 128w) /\ (w2n(a)+w2n(b) > 127))`;

(*--------------------------------------------------------------------------*)
(* sub_ov                                                                   *)
(* Takes two signed 8-bit numbers and checks if their subtractoin generates   *)
(* overflow.                                                                *)
(*--------------------------------------------------------------------------*)
val (sub_ov_def,_,sub_ov_dev0,sub_ov_comb,_) = hwDefine2
   `sub_ov (a:word8,b:word8)
       = (word_msb(a) /\ (b <+ 128w) /\ (abs(w2n a)+(w2n b) > 128)) \/
         ((a < 128w) /\ word_msb(b) /\ ((w2n a)+abs(w2n(b)) > 127))`;


(*--------------------------------------------------------------------------*)
(* write                                                                    *)
(* Takes a memory, address and value as input and returns a new memory after  *)
(* writing the value in the received address                                *)
(* This function writes from address 3 to 12 only due to lack of memory for   *)
(* compiling the longer IFs                                                 *)
(*--------------------------------------------------------------------------*)
val (write_def,_,write_dev0,write_comb,_) = hwDefine2
    `write(mem:word104,addr:word16,data:word8):word104 =
       if addr=3w then
           ((((103 >< (8+(3*8))) mem):word72 @@
            data):word80) @@
           ((((3*8)-1) >< 0) mem): word24
       else if addr=4w then
           ((((103 >< (8+(4*8))) mem):word64 @@
            data):word72) @@
           ((((4*8)-1) >< 0) mem): word32
       else if addr=5w then
           ((((103 >< (8+(5*8))) mem):word56 @@
            data):word64) @@
           ((((5*8)-1) >< 0) mem): word40
```

```
       else if addr=6w then
           ((((103 >< (8+(6*8))) mem):word48 @@
            data):word56) @@
           ((((6*8)-1) >< 0) mem): word48
       else if addr=7w then
           ((((103 >< (8+(7*8))) mem):word40 @@
            data):word48) @@
           ((((7*8)-1) >< 0) mem): word56
       else if addr=8w then
           ((((103 >< (8+(8*8))) mem):word32 @@
            data):word40) @@
           ((((8*8)-1) >< 0) mem): word64
       else if addr=9w then
           ((((103 >< (8+(9*8))) mem):word24 @@
            data):word32) @@
           ((((9*8)-1) >< 0) mem): word72
       else if addr=10w then
           ((((103 >< (8+(10*8))) mem):word16 @@
            data):word24) @@
           ((((10*8)-1) >< 0) mem):word80
       else if addr=11w then
           ((((103 >< (8+(11*8))) mem):word8 @@
              data):word16) @@
           ((((11*8)-1) >< 0) mem):word88
       else if addr=12w then
           data @@
           ((((12*8)-1) >< 0) mem):word96
       else mem‘;

(*--------------------------------------------------------------------------*)
(* read                                                                     *)
(* This function takes a memory and an address and returns the value stored *)
(* in that addres.                                                          *)
(*--------------------------------------------------------------------------*)
val (read_def,_,read_dev0,read_comb,_) = hwDefine2
    ‘read(mem:word104,addr:word16) =
         ((UNCURRY word_extract) (7+(w2n addr)*8,(w2n addr)*8) mem):word8‘;

(*--------------------------------------------------------------------------*)
(* testJUMP                                                                 *)
(* Takes the instruction code and the status registers and check if a jump  *)
(* has to be executed.                                                      *)
(*--------------------------------------------------------------------------*)
val (testJUMP_def,_,testJUMP_dev0,testJUMP_comb,_) = hwDefine2
   ‘testJUMP (INS:word8,OV:bool,N:bool,Z:bool,C:word8) =
      ((INS=JC_ABS)  /\ (C=1w))  \/ ((INS=JNC_ABS) /\ (C=0w))  \/
      ((INS=JN_ABS)  /\ N)       \/ ((INS=JNN_ABS) /\ ~N)      \/
      ((INS=JO_ABS)  /\ OV)      \/ ((INS=JNO_ABS) /\ ~OV)     \/
      ((INS=JZ_ABS)  /\ Z)       \/ ((INS=JNZ_ABS) /\ ~Z)      \/
      (INS=JMP_ABS)‘;

(*--------------------------------------------------------------------------*)
(* alu                                                                      *)
(* The system state is characterised by the tuple (memory,ACC,PC,OVNZC).    *)
(* This function takes the instruction register, a data (operand),          *)
(* the accumulator, the program counter and the status register (OVNZC) and *)
```

```
(* returns the new value of part of the state (ACC,PC,ONVZC)                    *)
(*-------------------------------------------------------------------------------*)
val (alu_def,_,alu_dev0,alu_comb,_) = hwDefine2
   `alu (IR:word8,data:word8,
         ACC:word8,PC:word16,OV: bool,N:bool,Z:bool,C:word8) =

         if IR = LDA_ABS then
            (data,PC+3w,OV,word_msb(data),data=0w,C)

         else  if IR = ADD_ABS then
            let res = ACC+data
            in
            (res,PC+3w,add_ov(ACC,data),word_msb(res),res=0w (* Z *),
                b2w(w2n(ACC) > (255-w2n(data))) (* C *))

         else if IR = ADDC_ABS then
            let res = ACC+data+C
            in
            (res,PC+3w,add_ov(ACC,data+C),word_msb(res),res=0w (* Z *),
                b2w(w2n(ACC) > (255-(w2n(data)+w2n(C)))) (* C *))

        else if IR = SUB_ABS then
            let res = ACC-data
            in
            (res,PC+3w,sub_ov(ACC,data),word_msb(res),res=0w (* Z *),
                b2w(data >+ ACC) (* C *))

         else if IR = SUBC_ABS then
            let res = ACC-(data+C)
            in
            (res,PC+3w,sub_ov(ACC,data+C),word_msb(res),
               res=0w (* Z *),
               b2w((w2n(data)+w2n(C)) > w2n(ACC)) (* C *))

         else if IR = AND_ABS then
            let res = word_and ACC data
            in
            (res,PC+3w,OV,word_msb(res),res=0w (* Z *),C)

         else if IR = OR_ABS then
            let res = word_or ACC data
            in
            (res,PC+3w,OV,word_msb(res),res=0w (* Z *),C)

         else if IR = XOR_ABS then
            let res = word_xor ACC data
            in
            (res,PC+3w,OV,word_msb(res),res=0w (* Z *),C)

         else if IR = CMPA_ABS then
            (ACC,PC+3w,OV,N,ACC=data (* Z *),b2w(ACC >+ data))

         else (ACC,PC+3w,OV,N,Z,C)`;


(*-------------------------------------------------------------------------------*)
```

```
(* next_state                                                                  *)
(* The system state is characterised by the tuple (memory,ACC,PC,OVNZC).      *)
(* This function takes a state and returns the new state after executing one  *)
(* instruction.                                                               *)
(*----------------------------------------------------------------------------*)
val (next_state_def,_,next_state_dev0,next_state_comb,_) = hwDefine2
   `next_state (mem:word104,
         ACC:word8,PC:word16,OVNZC:bool # bool # bool # word8)
    = let (IR,op2,op1) = (read(mem,PC),read(mem,PC+2w),read(mem,PC+1w)) in
      let addr = op2 @@ op1
      in if IR = STA_ABS then
            let data = write(mem,addr,ACC) in
            (data,ACC,PC+3w,OVNZC)
         else if IR = LDA_IMM then
            let (OV,N,Z,C) = OVNZC in
            (mem,op1,PC+2w,OV,word_msb(op1),op1=0w,C)
         else if testJUMP(IR,OVNZC) then
            (mem,ACC,addr,OVNZC)
         else (mem, alu(IR,read(mem,addr),ACC,PC,OVNZC))`;


(*----------------------------------------------------------------------------*)
(* run                                                                        *)
(* This function takes a step counter and a state (memory,ACC,PC,OVNZC) and   *)
(* runs the program in memory until it reaches the end_of_program address.    *)
(* As every function must terminate, we have to prevent this program from     *)
(* executing an infinite loop. So, we run the DIY a finite number of times.   *)
(* It finishes after running n steps or after reaching end_of_program address.*)
(*----------------------------------------------------------------------------*)
val (run_def,_,run_dev0,run_comb,run_tot) = hwDefine2
   `(run(step:word32, (mem:word104,ACC:word8,PC:word16,
                   OVNZC:bool # bool # bool # word8))
        =
        (if (step = 0w) \/ (PC = end_of_program) then mem
         else run(step-1w,next_state(mem,ACC,PC,OVNZC))))
     measuring (w2n o FST)`;

(*----------------------------------------------------------------------------*)
(* diy                                                                        *)
(* Takes a memory and runs the program it contains. If the program does not   *)
(* terminate in 2,000,000 steps (instructions executed) it returns the        *)
(* partial results computed up to that point.                                 *)
(*----------------------------------------------------------------------------*)
val (diy_def,_,diy_dev0,diy_comb,_) = hwDefine2
   `diy(mem:word104) = run(2000000w,(mem,0w,0w,F,F,F,0w))`;
```

Bibliography

[1] Andrew Appel and Trevor Jim. Continuation-passing, closure-passing style. In *Proceedings of the ACM Conference on Principles of Programming Languages*, pages 293–302, 1989.

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

[3] Christoph Berg, Christian Jacobi, and Daniel Kroening. Formal verification of a basic circuits library. *Proceedings of the International Conference on Applied Informatics, Innsbruck (AI 2001)*, pages 252–255, 2001.

[4] Sven Beyer, Christian Jacobi, Daniel Kroening, and Dirk Leinenbach. Correct hardware by synthesis from PVS.
`http://www-wjp.cs.uni-sb.de/publikationen/BJKL02.pdf`, 2002.

[5] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. *ACM SIGPLAN Notices*, 34(1):174–184, January 1999.

[6] Paul E. Black. Is "implementation implies specification" enough? *12th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'99), Nice, France*, 1999.

[7] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *FM 2006: Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer-Verlag, 2006.

[8] Christian Blumenröhr. A formal approach to specify and synthesize at the system level. In *GI Workshop Modellierung und Verifikation von Systemen*, pages 11–20, Braunschweig, Germany, 1999. Shaker-Verlag.

[9] Christian Blumenröhr and Dirk Eisenbiegler. Performing high-level synthesis via program transformations within a theorem prover. In *Proceedings of the Digital System Design Workshop at the Euromicro 98 Conference, Västeras, Sweden*, pages 34–37, Universität Karlsruhe, Institut für Rechnerentwurf und Fehlertoleranz, 1998. Online at:
`http://www.ubka.uni-karlsruhe.de/cgi-bin/psgunzip/1998/informatik/37/37.pdf`.

[10] Christian Blumenröhr and Viktor Sabelfeld. Formal synthesis at the algorithmic level. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99*, volume 1703 of *Lecture Notes in Computer Science*, pages 187–201, Bad Herrenalb, Germany, September 1999. Springer.

[11] Bhaskar Bose. *DDD-FM9901: Derivation of a Verified Microprocessor*. PhD thesis, Indiana University, USA, 1994. Computer Science Department.

[12] Jonathan P. Bowen and Jifeng He. An approach to the specification and verification of a hardware compilation scheme. *The Journal of Supercomputing*, 19(1):23–39, 2001.

[13] R. S. Boyer and J. S. Moore. A theorem-prover for a computational logic. In Mark E. Stickel, editor, *10th International Conference on Automated Deduction*, volume 449 of *LNCS*, pages 1–15, Kaiserslautern, Germany, July 1990. Springer-Verlag.

[14] Shiu-Kai Chin. Verified functions for generating signed-binary arithmetic hardware. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(12):1529–1558, 1992.

[15] Alonzo Church. A simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[16] Avra Cohn. Correctness properties of the Viper black model: the second level. Technical Report UCAM-CL-TR-134, University of Cambridge, Computer Laboratory, The U.K, May 1988.

[17] Avra Cohn. The notion of proof in hardware verification. *Journal of Automated Reasoning*, 5(2):127–139, 1989.

[18] Byron Cook, John Launchbury, and John Matthews. Specifying superscalar microprocessors in Hawk. In *Workshop on Formal Techniques for Hardware*, Maarstrand, Sweden, June 1998.

[19] Simon Finn, Michael P. Fourman, Michael Francis, and Robert Harris. Formal system design—interactive synthesis based on computer-assisted formal reasoning. In Luc Claesen, editor, *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, Volume 1*, pages 97–110, Houthalen, Belgium, November 1989. Elsevier Science Publishers, B.V. North-Holland, Amsterdam.

[20] F.K. Hanna, N. Daeche, and G. Howells. Implementation of the Veritas Design Logic. In V. Stavridon, T.F. Melham, and R.T. Boute, editors, *Proceedings of the International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 77–94, Nijmegen, 1992. North-Holland.

[21] Michael P. Fourman. Proof and design. In Manfred Broy, editor, *Deductive Program Design*, volume 152 of *NATO Advanced Science Institute, Series F: Computer and System Sciences*, pages 397–439. Springer, Marktoberdorf Germany, July 1996. Proceedings of the NATO Advanced Study Institute on Deductive Program Design, also available as LFCS report ECS-LFCS-95-319.

[22] Anthony Fox. Verifying ARM6 multiplication.
`http://www.cl.cam.ac.uk/users/acjf3`.

[23] Anthony C. J. Fox. Formal verification of the ARM6 micro-architecture. Technical Report 548, The Computer Laboratory, University of Cambridge, England, November 2002.

[24] Simon Frankau. *Hardware Synthesis from a Stream-Processing Functional Language*. PhD thesis, University of Cambridge, the Computer Laboratory, Cambridge, England, 2005.

[25] Simson Garfinkel. History's Worst Software Bugs. Wired News.
`http://www.wired.com/news/technology/bugs/0,69355-0.html`, 2005.

[26] Michael J. C. Gordon. HOL: A machine oriented formulation of higher order logic. Technical Report 68, The Computer Laboratory, University of Cambridge, 1985.

[27] Michael J. C. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. J. Milne and P.A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 153–177. North-Holland, 1986.

[28] Michael J. C. Gordon and Thomas F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[29] Mike Gordon. *From LCF to HOL: a short history*, pages 169–186. MIT Press, 2000. In Proof, Language, and Interaction: Essays in Honour of Robin Milner, Gordon Plotkin, Colin Stirling and Mads Tofte, editors.

[30] Mike Gordon, Juliano Iyoda, Scott Owens, and Konrad Slind. Automatic formal synthesis of hardware from higher order logic. In Ranzo Lazic and Rajagopal Nagarajan, editors, *Proceedings of the 5th International Workshop on Automated Verification of Critical Systems (AVoCS 2005)*, volume 145, pages 27–43, University of Warwick, UK, September 2005.

[31] GPL-Cver.
`http://www.pragmatic-c.com/gpl-cver`.

[32] Jim Grundy, Tom Melham, and J. O'Leary. A reflective functional language for hardware design and theorem proving. Research report prg-rr-03-16, Oxford University Computing Laboratory, 2003.

[33] GTKWave.
`http://home.nc.rr.com/gtkwave`.

[34] Nicolas Halbwachs and Pascal Raymond. A tutorial of Lustre.
`http://www-verimag.imag.fr/SYNCHRONE`, 2002.

[35] Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, 1990.

[36] F.K. Hanna, M. Longley, and N. Daeche. Formal synthesis of digital systems. In L. Claesen, editor, *Applied Formal Methods for Correct VLSI Design*, pages 153–170. North-Holland, 1989.

[37] John Harrison. The HOL Light Theorem Prover. `http://www.cl.cam.ac.uk/~jrh13/hol-light`.

[38] John Harrison. HOL Light: A tutorial introduction. In Mandayam Srivas and Albert Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *LNCS*, pages 265–269. SV, 1996.

[39] John L. Hennessy and David A. Patterson. *Computer Architecture. A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[40] Warren A. Hunt, Jr. *FM8501: A Verified Microprocessor*. PhD thesis, University of Texas, Austin, Texas, USA, 1985.

[41] Warren A. Hunt, Jr. and Bishop C. Brock. A formal HDL and its use in the FM9001 verification. In C.A.R. Hoare and M.J.C. Gordon, editors, *Mechanized Reasoning and Hardware Design*, pages 35–47. Prentice Hall International Series in Computer Science, 1992.

[42] Warren A. Hunt, Jr. and Erik Reeber. Formalization of the DE2 language. In Dominique Borrione and Wolfgang J. Paul, editors, *Correct Hardware Design and Verification Methods, 13th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2005, Saarbrücken, Germany, October 3-6, 2005, Proceedings*, volume 3725 of *Lecture Notes in Computer Science*, pages 20–34. Springer, 2005.

[43] Steven D. Johnson. Manipulating logical organization with system factorizations. In Miriam Leeser and Geoffrey Brown, editors, *Hardware Specification, Verification and Synthesis*, volume 408 of *Lecture Notes in Computer Science*, pages 260–281. Springer, 1990.

[44] Steven D. Johnson and Bhaskar Bose. DDD – A System for Mechanized Digital Design Derivation. Technical Report TR323, Indiana University, IU Computer Science Department, 1990. Available on the Internet at: `http://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR323`.

[45] Steven D. Johnson, Robert M. Wehrmeister, and Bhaskar Bose. On the interplay of synthesis and verification: Experiments with the FM8501 processor description. In Luc J.M. Claesen, editor, *Formal VLSI Specification and Synthesis — VLSI Design Methods I*, volume 1, pages 117– 136. North-Holland, 1990.

[46] Geraint Jones and Mary Sheeran. Circuit design in Ruby. Lecture notes on Ruby from a summer school in Lyngby, Denmark., September 1990. `http://www.cs.chalmers.se/~ms/papers.html`.

[47] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

[48] Matt Kaufmann and J Strother Moore. ACL2: An industrial strength version of Nqthm. In S. Faulk and C. Heitmayer, editors, *11th Annual Conference on Computer Assurance*, pages 23–34, Gaithersburg, Maryland, 1996. National Institute of Standards and Technology.

[49] Christoph Kern and Mark R. Greenstreet. Formal verification in hardware design: A survey. *ACM Transactions on Design Automation of Electronic Systems*, 4(2):123–193, 1999.

[50] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 42–54, New York, NY, USA, 2006. ACM Press.

[51] John Matthews and John Launchbury. Elementary microarchitecture algebra. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy*, volume 1633 of *Lecture Notes in Computer Science*, pages 288–300. Springer, July 1999.

[52] John Robert Matthews. *Algebraic specification and verification of processor microarchitectures*. PhD thesis, Oregon Graduate Institute, 2000.

[53] Clive "MAX" Maxfield and Alvin Brown. *The Definitive Guide to How Computers do Math*. John Wiley & Sons, Inc., Hoboken, New Jersey, USA, 2005.

[54] William McCune and Larry Wos. Otter — The CADE-13 competition incarnations. *Journal of Automated Reasoning*, 18(2):211–220, April 1997.

[55] Thomas F. Melham. *Higher Order Logic and Hardware Verification*. Cambridge University Press, Cambridge, England, 1993. Cambridge Tracts in Theoretical Computer Science 31.

[56] Robin Milner. Logic for computable functions; description of a machine implementation. Technical Report STAN-CS-72-288, A.I. Memo 169, Stanford University, 1972.

[57] Alan Mycroft and Richard Sharp. Hardware/software co-design using functional languages. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, pages 236–251, Genova, Italy, April 2001. Springer-Verlag. LNCS Vol. 2031.

[58] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[59] Michael Norrish and Konrad Slind (project administrators). The HOL4 System. SourceForge website. `http://hol.sourceforge.net`.

[60] Objective Caml.
`http://caml.inria.fr/ocaml`.

[61] John O'Donnell. Hardware description with recursion equations. In *Proceedings of the IFIP 8th International Symposium on Computer Hardware Description Languages and their Applications*, pages 363–382, Amsterdam, April 1987. North-Holland.

[62] John O'Donnell. Overview of Hydra: A concurrent language for synchronous digital circuit design. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*. IEEE Computer Society Press, 2002.

[63] John O'Donnell and Gudula Rünger. Derivation of a logarithmic time carry lookahead addition circuit. In *Journal of Functional Programming*, volume 14, number 6. Cambridge University Press, 2004.

[64] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A Prototype Verification System. In *CADE-11: Proceedings of the 11th International Conference on Automated Deduction*, pages 748–752, London, UK, 1992. Springer-Verlag.

[65] ProofPower. `http://www.lemma-one.com/ProofPower/index/index.html`.

[66] Jonathan A. Rees and William D. Clinger. The revised^3 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 21(12):37–79, December 1986.

[67] V. Sabelfeld, C. Blumenröhr, and K. Kapp. Semantics and transformations in formal synthesis at system level. In Dines Bjorner, Manfred Broy, and Alexandre Zamulin, editors, *Perspectives of System Informatics, 4th International Andrei Ershov Memorial Conference, Novosibirsk, Russia*, LNCS 2244, pages 149–156, Universität Karlsruhe, Institut für Rechnerentwurf und Fehlertoleranz, 2001. Springer-Verlag. `http://link.springer.de/link/service/series/0558/bibs/2244/22440149.htm`.

[68] Xavier Saint-Mleux, Marc Feeley, and Jean-Pierre David. *SHard: a Scheme to Hardware Compiler*, pages 39–50. University of Chicago Technical Report TR-2006-06, *Seventh Workshop on Scheme and Functional Programming*, Portland, USA, 2006.

[69] Augusto Sampaio. *An Algebraic Approach to Compiler Design*. PhD thesis, University of Oxford, 1993.

[70] Jun Sawada. *Formal Verification of an Advanced Pipelined Machine*. PhD thesis, University of Texas, Austin, Texas, USA, 1999.

[71] Richard Sharp. *Higher-Level Hardware Synthesis*. PhD thesis, University of Cambridge, the Computer Laboratory, Cambridge, England, 2002.

[72] Mary Sheeran. muFP, A language for VLSI design. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 104–112. ACM, ACM, August 1984.

[73] Konrad Slind. Function definition in higher order logic. In *Theorem Proving in Higher Order Logics*, number 1125 in Lecture Notes in Computer Science, pages 381–398, Turku, Finland, August 1996. Springer-Verlag.

[74] Konrad Slind. *Reasoning about terminating functional programs*. PhD thesis, Institut für Informatik, Technische Universität München, Germany, 1999.

[75] Konrad Slind, Guodong Li, and Scott Owens. A proof-producing software compiler for a subset of higher order logic.
`http://www.cs.utah.edu/~slind/sw-compiler`, 2006.

[76] Konrad Slind, Scott Owens, Juliano Iyoda, and Mike Gordon. Proof producing synthesis of arithmetic and cryptographic hardware. *Formal Aspects of Computing*. To appear.

[77] Konrad Slind, Scott Owens, Juliano Iyoda, and Mike Gordon. FAQ for proof producing synthesis in HOL. In Mary Sheeran and Tom Melham, editors, *Sixth International Workshop on Designing Correct Circuits*, pages 183–199, Vienna, Austria, March 2006. ETAPS 2006. A Satellite Event of the ETAPS 2006.

[78] Mandayam K. Srivas, Harald Rueß, and David Cyrluk. Hardware verification using PVS. In Thomas Kropf, editor, *Formal Hardware Verification - Methods and Systems in Comparison*, volume 1287 of *Lecture Notes in Computer Science*, pages 156–205. Springer, 1997.

[79] Kong Woei Susanto and Thomas F. Melham. Formally analyzed dynamic synthesis of hardware. *The Journal of Supercomputing*, 19(1):7–22, 2001.

[80] Tanel Tammet. Gandalf. *Journal of Automated Reasoning*, 18(2):199–204, April 1997.

[81] Translating HOL Functions to Hardware — the HOL files. Available in the directory `hol98/examples/dev` at `http://hol.sourceforge.net`.

[82] David Wheeler and Roger Needham. TEA, a tiny encryption algorithm. In *Fast Software Encryption: Second International Workshop*, volume 1008 of *LNCS*, pages 363–366. Springer Verlag, 1999.

[83] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*. Cambridge University Press, Cambridge, England, 1925–1927. Three volumes.