Correct synthesis and integration of compiler-generated function units

Thesis by
Martin Ellis

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

University of Newcastle upon Tyne
Newcastle upon Tyne, UK

Abstract

Computer architectures can use custom logic in addition to general purpose processors to improve performance for a variety of applications. The use of custom logic allows greater parallelism for some algorithms. While conventional CPUs typically operate on words, fine-grained custom logic can improve efficiency for many bit level operations. The commodification of field programmable devices, particularly FPGAs, has improved the viability of using custom logic in an architecture.

This thesis introduces an approach to reasoning about the correctness of compilers that generate custom logic that can be synthesized to provide hardware acceleration for a given application. Compiler intermediate representations (IRs) and transformations that are relevant to generation of custom logic are presented. Architectures may vary in the way that custom logic is incorporated, and suitable abstractions are used in order that the results apply to compilation for a variety of the design parameters that are introduced by the use of custom logic.

Contents

Chapter 1

Introduction

Contents

High-level programming languages allow concepts in a given application domain to be represented and manipulated without reference to a particular computer architecture [Hor75]. Programs written in such languages are translated into an executable format by a compiler tool chain, usually including a compiler, an assembler and a linker [Sch03]. The compiler typically performs (so called) optimisations on the program to improve the performance of the executable on the targeted computer architecture (henceforth, *architecture*) [ASU86].

If the resulting executable is considered too slow, there are several alternatives for improving its performance. The original program could be modified to implement a more efficient algorithm. Parts of the high-level program can be re-written in a lower level language to exploit the underlying architecture in a way that the compiler did not. A third alternative is that the underlying architecture be modified to improve the performance of that application.

Broadly speaking, this thesis is about the extension of existing architectures with custom hardware logic to improve application performance. However, a more precise statement will require further introduction.

Extending an architecture can be accomplished by adding a *function unit* with hardware logic dedicated to a particular task. The term *hardware acceleration*

refers to the technique of improving performance by making a program use a function unit for some processing which would otherwise be done using a general purpose CPU [How06]. Performance improvements can arise from greater computational efficiency of dedicated logic [AS93], and from using a function unit to execute many parts of a computation in parallel [LEM04].

A variety of architectures that allow custom function units to provide hardware acceleration have been developed, some examples of which are described in Section 2.3.1, *Types of RFU*.

While it is certainly possible to design function units for an architecture using a *hardware description language* — such as VHDL [IEE02] or Verilog [IEE01] — there have been a number of efforts to use compilers to synthesise hardware logic from sequential implementations of algorithms written in languages based on traditional high level programming languages. This technique is sometimes referred to as *hardware/software compilation*.

Several examples of hardware/software compilers, including both those described in literature from research projects and those that are commercial products, are described in Chapter 2. Such products and literature claim varying improvements in performance and automation for a variety of applications. However, such *quantitative* issues are not the focus of this thesis.

Instead, this thesis addresses issues of a *qualitative* nature: here, the focus is on establishing the correctness criteria for the hardware/software compilation. More specifically, the aim is to ensure that a program that has been modified to use a custom function unit has similar behaviour to that of the unmodified program. Of course, to do this, it is necessary to consider the behaviour of the function unit as well as that of the program itself. By drawing from methodologies and techniques in both compiler verification and hardware verification, this thesis presents the development of a formal framework within which propositions about the behavioural similarities of such systems can be constructed and verified.

This chapter is organised as follows: Section 1.1, *Hardware Acceleration for High-Level Languages*, provides an introduction to the use of function units for hardware acceleration. It also motivates the use of hardware/software compilers for compiling certain parts of a program into a function unit design that can be used to provide hardware acceleration. Section 1.2, *Correctness of Hardware/Software Compilers*, uses existing compiler correctness concepts to introduce an intuitive notion of correctness for hardware/software compilers; describes some of the problems in their specification and verification, and illustrates why existing compiler verification techniques need to be extended for hardware/software compilers. Thus, it forms the motivation for this thesis. Section 1.3, *Thesis Outline*, outlines the approach used in this thesis: that is, how it adapts existing compiler verification techniques to suit hardware/software compilers.

1.1 Hardware Acceleration for High-Level Languages

This thesis addresses various problems of how to ensure the correctness of hardware/software compilers: that is, how to ensure a hardware logic design

and software generated by a compiler from an algorithm in a high level language implement that algorithm faithfully. The motivation for this work — explained in Section 1.2, *Correctness of Hardware/Software Compilers* — is predicated on there being sufficient reason to use hardware/software compilers in the first place.

This section motivates and introduces concepts of using such compilers to generate hardware acceleration from high level languages. It is structured as follows: Section 1.1.1 introduces the use of function units to provide hardware acceleration. Section 1.1.2 describes the use of reconfigurable logic to provide hardware acceleration. Hardware/software compilers typically target reconfigurable logic, and are introduced in Section 1.1.3. It motivates the use of hardware/software compilers by describes some of the benefits of their use as an approach to producing function unit designs.

1.1.1 Function Units

The term *function unit* is used somewhat loosely in this thesis. However, this section serves to provide at least an intuitive understanding of the terminology associated with function units as used in this thesis.

Here, the term *function unit* refers to a unit of dedicated logic that is used for computations, but that is not part of the *general purpose processor* (GPP) in a given *instruction set architecture* (ISA).

There are many existing examples of architectures that use a dedicated function unit to improve the performance of particular types of computations, and it should be clear that this idea is not new. Examples of types of applications that have been accelerated by the use of function units include:

- **floating point arithmetic**: many architectures improve the performance of such arithmetic using floating point units (FPUs);
- **multimedia** applications: examples include the *MMX* extensions — and subsequent *SSE*, *SSE2* and *SSE3* extensions — developed by Intel for the *IA-32* architecture (see [Int05], Chapters 9 to 12 respectively); the *3DNow!* [AMD00a], *3DNow! Enhanced* [AMD00b] and *3DNow! Professional* [AMD02] extensions developed by AMD;
- **cryptographic** applications: such as the *SafeNet EIP-25* cryptographic co-processor for the *ARM* architecture that provides accelerated performance for various algorithms associated with public key cryptography [ARM04];
- **digital signal processing** (DSP): for example, portable music players may include a DSP co-processor in addition to a GPP to implement part of an audio codec.

In some cases, function units have been integrated into the architecture by putting it in the same physical chip as the processor [WH95]. In other cases, the function unit has been connected by means of one of the system buses [WAL+93]. The definition used here includes both such uses, although it is noted that such a definition may make the decision as to whether a given part of an ISA constitutes a function unit or part of the GPP rather subjective.

All of the examples given above have been implemented in *ASICs* — Application Specific Integrated Circuits. ASICs are designed to perform a specific type of computation, and can perform that computation very fast and efficiently [CH00]. As such, ASICs have a fixed purpose, which cannot be altered after fabrication. Their manufacture is associated with high *non-recurring engineering* (NRE) costs and a long manufacturing cycle. Furthermore, any changes to the design requires refabrication of the ASIC.

Board-level circuits — circuits constructed from smaller components, usually connected via 'tracks' on a circuit board — are also hard to modify, sometimes requiring a redesign of the board layout or even a replacement of the board [CH00].

1.1.2 Reconfigurable Function Units

An alternative to using ASIC techniques for function units is to use a field programmable logic device, such as one or more *Field Programmable Gate Arrays* (FPGAs) [BR96]. An FPGA consists of an array of uncommitted logic blocks, connected by reconfigurable interconnect resources. Logic blocks — also known as *Configurable Logic Blocks* (CLBs) [Xil05a] or *Logic Elements* (LEs) [Alt05] [1] — contain resources that can be configured to implement combinational and sequential logic. Thus, a hardware designer may describe a circuit in a *Hardware Description Language* (HDL) and *synthesise* the design into a format that can be mapped onto the logic blocks and routing resources provided by a given FPGA. The FPGA design flow process is described further in Section 2.1.2, *FPGA Design Flow*.

FPGAs tend to fall into one of two categories: those based on *anti-fuse* technology, and those that are SRAM-based [BR96]. Anti-fuse FPGAs may be configured only once. That is, although they may be programmed 'in-the-field', they are not reconfigurable. SRAM-based FPGAs can be configured and reconfigured many times: the behaviour of its reconfigurable logic blocks and the interconnections between them can be repeatedly changed.

SRAM-based FPGAs have the advantage that they can be reconfigured in-circuit. Since they are SRAM-based, their behaviour can be changed by downloading a new configuration, just as the behaviour of a general purpose computer can be changed by loading a new program [Hau98]. Unlike anti-fuse FPGAs, most SRAM-based FPGAs lose their configuration when powered down, requiring reconfiguration when next powered up (although there are exceptions to this that use Flash or other non-volatile memory to store the FPGA configuration [Lat05]).

The concept of using FPGAs to implement 'customisable compute engines' was introduced in the late 1980s [GSAK00] and within approximately 10 years, it had been demonstrated that they could provide speed improvements by factors of between 10 and 100 over conventional high performance workstations [FGL01].

[1] The terminology varies according to both the details of the implementation and the respective manufacturer.

1.1.3 Hardware/Software compilers for Function Units

While traditionally FPGA configurations are synthesised from a design expressed in a hardware description language (HDL), it is also possible, and sometimes advantageous, to perform synthesis from an algorithm expressed in an high-level programming language (HLL), such as C. Thus the larger part of a program may be compiled to executable object code for a general purpose processor, while certain program *hot-spots* that need to run efficiently may be synthesised for the FPGA, such that the FPGA is used to provide hardware acceleration for those hot-spots. Examples of synthesis from high-level languages are given in Section 2.3.

While the purpose of using a function unit is to improve performance for a given (type of) application, it may not be obvious which parts of the application should be accelerated in order to achieve optimal (or even satisfactory) results. A developer may want to experiment with the choice of whether particular functions should be implemented in hardware or in software, and it is desirable to be able to do such experimentation with minimal changes to the program design.

Using a traditional HDL to design the function unit, and a software programming language to implement the rest of the program would require the developer to write two implementations of a function — one in the HDL and one in the HLL — in order to evaluate whether the function should be hardware accelerated.

By using a language that can be both synthesised to an FPGA and compiled efficiently, the required source code changes can be minimised. Thus synthesis from a HLL can reduce development effort. One study showed that synthesis from a C-based language yielded productivity improvements of a factor of 10 over hand-written HDL [FGL01].

1.2 Correctness of Hardware/Software Compilers

It has been claimed that — despite the potential benefits outlined in the previous section — the adoption of hardware/software compilers in the "real world" is predicated on their ability to produce hardware with comparable performance to hand-coded designs [FGL01]. That is, that they produce hardware designs for function units that make efficient use of available (reconfigurable) hardware resources.

A stronger assumption is behind the motivation for this thesis: that the adoption of hardware/software compilers is predicated not only on their ability to produce both efficient hardware designs *and* object code; but also their ability to produce correct output, and hence that engineers have confidence in their tools.

The purpose of this section is to introduce compiler correctness in the context of hardware/software compilers; and to motivate the development of the formal framework for specifying and reasoning about their correctness presented in this thesis.

Section 1.2.1, *Correctness for Hardware Acceleration*, compares the two as-

sumptions given above and provides an argument for the latter, stronger assumption pertaining to correctness. Section 1.2.2, *Correctness for Conventional Compilers* introduces concepts of formal methods for compiler development, based on the *Vienna Development Method* (VDM), a systematic approach to software development originally developed for the purpose of compiler specification and verified design [BJ78]. Some fundamental difficulties in applying such techniques directly to the development of hardware/software compilers are highlighted in Section 1.2.3, *Inadequacy of Current Techniques*. It is these difficulties that are addressed in this thesis. Section 1.2.4, *Specific Problems in Hardware/Software Compilation* describes further problems in applying formal methods to hardware/software compilers. This thesis does not address these issues directly, but resolving the issues addressed by this thesis is likely to be a prerequisite to solving those problems.

1.2.1 Correctness for Hardware Acceleration

Scope

The differences between the assumptions introduced at the start of this section are two-fold. The first difference is that the latter assumption — that made in this thesis — assumes that it is the properties of the *entire* system that should be evaluated, and not just that of a function unit. It would be pointless to use a function unit for a particular computation if the cost of initialising the function unit, and then retrieving a result from it, was not amortised by the improved performance of the function unit for that particular computation. To illustrate: it may take several CPU cycles to swap the endianness of a value, but only one cycle to achieve the same result in a reconfigurable function unit. If the CPU is being clocked several times faster than the reconfigurable logic (which is not implausible) then the resulting system may still be faster if the operation is performed in software, even though a dedicated function unit may implement that particular operation more efficiently.

The distinction between performance of the function unit itself, and that of the resulting system (which includes software and hardware) is obvious enough that drawing attention to it here seems pedantic. However, the distinction helps to clarify the second difference between the assumptions: that hardware/software compilers must be able to produce correct output.

The correctness of a hardware/software compiler is that the resulting program and function unit behave in a similar manner as the original program would have, had it been compiled entirely as software, with no hardware synthesis. This definition is not perfect, because if a program measures the time it takes to complete an operation then its behaviour may vary when the program is hardware accelerated. It also leaves open the interpretation of 'behaviour' in this context: obviously it should not refer to temporal properties of the program, because that is exactly what the use of hardware acceleration is supposed to modify!

In summary, just as it is more useful to consider the performance of the resulting system, than just that of the function unit, it is also more useful to consider the correctness of the resulting system, rather that just the correctness of the

function unit design itself. This gives confidence that a program will exhibit similar behaviour regardless of whether any given part of the program is realised in hardware or software.

Requirement for correctness

The assumption that the wider adoption of hardware/software compilers is predicated, in part, on their correctness can be justified by consideration of the implications of bugs within such a compiler.

Debugging optimised code can be difficult, even when a compiler is not exhibiting any bugs [Cop94]. When a compiler generates incorrect code, it may be necessary to trace the program to find that point that an error occurs — which may occur long before the bug manifests, and in logically unrelated code — and examining the output of the compiler, possibly stepping through the program machine instruction by machine instruction [Eis97]. Assuming the source of the bug can be found and traced to the compiler, it remains to either fix the compiler; get a new compiler; or work around the compiler bug [Cop94].

Debugging FPGA designs becomes more difficult with the increasing logic capacities of FPGAs. This is due to "limited observability": meaning that it is harder to see how a design behaves in the FPGA. Traditional hardware debugging techniques — involving logic analysers or oscilloscopes — can only be used to the extent that an FPGA package provides enough spare pins that may be connected to, and used to monitor, internal signals [GNH01].

The difficulties in managing compiler bugs and design errors in FPGAs are combined for hardware/software compilers that exhibit bugs. If a hardware/software compiler produces incorrect output then the task of finding, and working around, the error will be particularly difficult. This is because the developer may have to examine both the executable object code and the hardware design that the compiler outputs.

Complexity also arises due to the need to examine not just the individual outputs, but also the way in which the software interacts with the function unit. Compiler bugs would require strong hardware and software debugging skills to track down.

While it may be important to be able to mark regions of code as good candidates for hardware acceleration, it is possible that a developer may not have specified the hardware/software interface precisely when using a hardware/software compiler. In some cases, it would be more appropriate for the compiler to do fine-grained partitioning, perhaps by synthesising only part of an expression into hardware. It can be also be argued that they should not have to specify the interface, because this would represent unnecessary commitment to a particular amount of reconfigurable logic, thus reducing the portability of the code.

If the selection of code to synthesise to hardware is semi-automated, the programmer may not even know whether the code that they are attempting to modify was implemented in software or in a function unit. However, this approach means that the developer will not be familiar with the hardware software interface: their intuitive understanding of it will be weaker, because they did

not design it directly. Thus, debugging a hardware/software system developed this way is likely to be difficult.

Although bugs may exist in the code, it is important that the use of a hardware/software compiler does not introduce more bugs. That is, bugs in the program should be reproducible using a conventional compiler, in order that traditional debugging techniques can be used. To rephrase more positively, it is important that a hardware/software compiler preserve the semantics of the original source program correctly.

If a hardware/software repeatedly generates incorrect results, the cost and complexity of debugging errors may make the traditional approach of separate hardware and software design more cost effective.

1.2.2 Correctness for Conventional Compilers

A formal (mathematical) definition of a programming language makes it possible to formulate a precise statement of the correctness criteria of a compiler for that language [Jon76]. Hence compiler development should begin with a precise description of the source and target languages of that compiler followed by the formal development of a compiling algorithm [Bjø82].

A formal definition of a programming language can be expressed in terms of the following 'views' of the language [Jon90a]:

- **Concrete syntax**, which may be written in BNF, together with some indication of operator (or more generally, token) precedence;
- **Abstract syntax**, which defines a representation for programs which is an abstraction of the concrete syntax;
- **Context conditions**, which characterise a subset of all well-formed programs, such as programs which are both well-formed and well-typed;
- **Semantic domains**, which are used to represent an abstract state that would be manipulated by an abstract interpreter for the language;
- **Semantic functions (or rules)**, which manipulate the semantic domains according to a program's representation in the abstract syntax.

Typically, the semantic domains for the source and target languages are different. For example, a high level source language may model state in terms of a stack, a heap, local and global variables, while a low level target language may model state in terms of registers and a flat (linear) view of memory. Since the semantic domains of the models are often different, it stands to reason that the semantic functions or rules that operate upon them must also be different. The exception to this is where a language is both the source and target language of a compiler.

In order to ensure operations in the target language correctly implement operations in the source language, it is necessary to determine a relationship between both the semantic domains of the two languages, and between the operations defined by each language.

To relate the semantic domains, it may be possible to define a *retrieve* function that maps an abstract state of the target language to its equivalent in the source language. Thus, the domain of the retrieve function represents the semantic domains of the target language, and its co-domain represents the semantic

domains of the source language. Retrieve functions may be defined where a given state in the target machine language corresponds to a single state in the source language.

Where a compiler's source language is a higher level language than its target language — perhaps assembly language, which may require several operations to complete a single source level operation — it may be reasonable to define a retrieve function.

The correctness condition for a compiler is formulated as a property of both the retrieve function and the compilation algorithm. For example, a retrieve function that maps a series of stack frames for the target architecture to the environment for a given block of code in the source language must be used with a compilation algorithm that uses a stack discipline for procedure calls.

The correctness condition of a compiler specified in terms of a retrieve function, *retr*, and a compilation algorithm, *comp*, is illustrated in Figure 1.1. The source language state, σ, is a value within the semantic domains of the source language before the execution of program p. The target machine language state $m\text{-}\sigma$, is a value within the semantic domains of the target language, such that $retr(m\text{-}\sigma) = \sigma$.

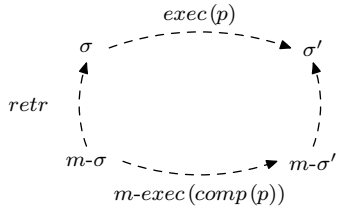


Figure 1.1: Conventional compiler correctness

The semantic rules of the source and target languages are represented as the functions *exec* and *m-exec* respectively, and are higher order functions that yield a function from initial state to final state for a given program. The correctness condition is that the retrieve function and the compilation function commute:

$$exec(p)\ (retr(m\text{-}\sigma)) = retr(\ m\text{-}exec(comp(p))(m\text{-}\sigma)\ )$$

1.2.3 Inadequacy of Current Techniques

Unfortunately, the techniques for specifying and developing compilers outlined in the previous section cannot be applied directly to hardware/software compilers. The limitations of these techniques in the context of hardware/software compilers are identified here and motivate the development of the formal framework presented in this thesis (which is introduced in Section 1.3, *Thesis Outline*).

The form of the correctness condition given above assumes a single target language for the compiler. This means the target language has a single state representation and a single set of semantic functions or rules. However, hardware/software compilers must target two languages: a hardware description

language, within which a function unit design is described; and a software language, into which the remainder of the program is compiled. In this context, the hardware description language does not exclude low-level representations such as a netlist language or an FPGA configuration bit-stream. The software language may be assembly language, object code, or a higher level language for which there exists a compiler to the relevant ISA.

Modelling hardware and software behaviour separately

It is possible to define a single 'universal' target language which can contain a representation of the parts of the program that will be compiled to software, together with parts of the program that have been synthesised into a hardware representation. It is argued here that — while this combined approach may be suitable for an implementation — adopting this approach directly is not suitable for specification and modelling in the development of hardware/software compilers. Instead, the hardware language and software language should be modelled separately.

One of the reasons for the possible discrepancy between the compiler implementation and its associated specification and modelling is that the formal model must capture not only the details of the representation of both the hardware and software language; but also the semantics of each language and the representation of state (the semantic domains) of each language. In contrast, most of these details do not appear in the implementation, which need only capture the hardware and software representations (and transformations thereof).

There are several advantages to modelling the hardware and software separately. The semantics of these languages may be very different. To be confident that the model is accurate, it should be as simple as possible to relate to the hardware and software that it describes. However, it would be difficult to model the behaviour of a computer program on a given ISA and a hardware design accurately using the same set of semantic rules.

A GPP has sequential behaviour: it steps through a program instruction by instruction, and from a given state, it is a reasonable modelling assumption that there may only be a single successor state (ignoring I/O and undefined behaviour). Conversely, a low-level model of hardware may define concurrent behaviour. Many signals may change value simultaneously. Furthermore, if exact component and signal propagation delays are not known, it may not be possible to predict the exact behaviour of the device from a given initial state. This situation can occur, for example, when elements in a logic design for reconfigurable hardware have not yet been allocated to physical resources.

In a circuit with entirely synchronous (clocked) timing, where the data-flow path between registers consists entirely of combinational logic, it may be possible to determine the 'next state' of a circuit, provided that the state is modelled as the values of each signal during each clock cycle. However, for complex circuits, a circuit that is synchronous may operate in a globally asynchronous manner, where synchronous components are composed together using a handshaking interface [RC03]. In this context, a model of the hardware that supports concurrency is beneficial because it affords a compositional approach to circuit design.

Although the same argument could be applied to software: where many threads execute in parallel, the assumption of sequential behaviour is no longer necessarily valid. However, concurrency in the source language is not considered in this thesis.

The difference between these models can be likened to different programming language paradigms. It might be considered unusual to attempt to use the semantic rules for a logic programming language when modelling an imperative language, and likewise should be considered so for modelling hardware constructs with semantics rules for a software language. Thus, the formal framework developed in this thesis uses a separate set of semantic rules to describe the behaviour of hardware and software.

Separating hardware state from software state

An argument similar to the above — that the semantics of hardware and software languages should be modelled separately — can be constructed for modelling the state of those languages.

There are a variety of ways for modelling state in software languages, depending on the level of abstraction being considered. In a hardware language, state may be modelled by the values of signals, registers and memories. Modelling the languages separately allows a more natural representation of state for hardware and software.

1.2.4 Specific Problems in Hardware/Software Compilation

The use of a formal framework such as that presented in this thesis need not be limited to compiler specification. Indeed, there are many more problems to be solved in hardware/software compiler verification.

This section describes several specific problems in hardware/software compiler verification. A framework similar to that presented here is likely to be a prerequisite for solving these problems. It is expected that such a formal framework could also be extended, and used in later stages of the formal development of a compiler.

Compiler development

A framework that includes formal definitions of a source language, a target software language and a target hardware description language may be extended to support the formal development of a verified compilation algorithm, or other transformations such as optimisations. Extending the formal framework to model these algorithms is necessary to allow verification of hardware/software compilers to the extent that current techniques can be used for the verification of conventional software compilers.

Of course, reasoning about the actual implementation of the compilation algorithm and transformations is only possible if there is a formal semantics for the implementation language. One might consider extending the formal framework to include not only formal definitions of the source and target languages, but

also a formal definition of the implementation language. However, the focus of this thesis is on compiler specification, and not on implementation correctness, and such an approach is not attempted here.

Source code portability

Another way that the formal framework could be usefully extended would be to support the identification of programs that contain either *self-modifying code* or *self-examining code*. Ideally the identification should be based on a static analysis of the program text, and not dynamically, at run-time.

Self-modifying and self-examining code can be difficult to detect statically in programs for Von Neumann architectures, and in general, it is not possible to translate self-modifying and self-examining programs automatically [JSW99]. In this sense, such programs are not portable.

Dynamic analysis to monitor changes to the program may be appropriate for where program code is generated at run time [CK94], but may not be appropriate for hardware accelerated systems in general. Some of the reasons translating such code may not be practical are: that a function unit may not be implemented in reconfigurable logic; the long synthesis times for hardware design making reconfiguration to reflect changes in the program impracticable; and the difficultly in detecting whether the function unit needs to be re-synthesised.

Self-modifying programs — or programs that exhibit self-modifying behaviour by modifying memory whose contents is later executed — include just-in-time compilers, such as Java virtual machines and .NET interpreters [MB03] and dynamic linkers that 'patch' addresses in the program area. The technique may also be used to make a program more efficient [Hen01].

Self-modification/examination is not the only behaviour that makes verified software/hardware compilation difficult. For example, if a program makes an indirect branch to a given address, and that address could have been modified at run-time, then it is necessary to ensure the branch is to a valid instruction and that the program will continue to behave in a similar manner to a non-accelerated version of the same program.

A common example of an indirect branch is a the result of compiling a `switch` (or `case`) statement. Where the cases are over consecutive enumerable values, the statement may be compiled as an indirect branch where the switch value is used as an index into a jump-table.

Target machine portability

A retargetable compiler is a compiler that can compile programs for more than one target architecture [HF95]. The target-specific parts of a retargetable compiler are independent of each other, and support for new targets may be added (and support for old targets removed) without affecting support for other targets.

Retargetable compilers can support different targets by having a *machine description* for each target architecture from which a compiler backend, or code generator, is produced [Sta02, HF95].

It seems beneficial for hardware/software compilers that target FPGAs to be retargetable in the sense that it be possible to support new FPGAs or hardware acceleration products. By making a compiler retargetable, a single compiler code base could feasibly be used to provide support for a range of hardware acceleration products based on FPGAs, where each product may provide a different type of FPGA, or different number of FPGAs. Higher end FPGAs may provide greater logic capacity, or dedicated logic units that can improve the performance of particular operations [Xil03]. For example, FPGAs may contain dedicated multipliers to provide faster multiplication.

In order to verify synthesis to FPGAs with dedicated logic units, a specification of the behaviour of each dedicated logic unit must be available.

The framework for describing hardware/software languages developed in this thesis supports two compilation targets. It is expected that it could be extended to support two targets described by machine descriptions — one for the GPP and one for the FPGA. Again, however, such an approach is not attempted here.

1.3 Thesis Outline

Some of the benefits of hardware/software compilers — particularly of those that target reconfigurable function units — were introduced in Section 1.1. The motivation for verifying the correctness of hardware/software compilers was introduced in Section 1.2. Existing techniques for compiler verification were outlined and the limitations of such techniques in the context of hardware/software compilers were discussed.

This section outlines the approach taken to extending these existing techniques as presented in this thesis, and the rationale for that approach.

1.3.1 Scope

This thesis presents an approach to the formal specification of hardware/software compilers. It focuses on those aspects of compiler specification that are specific to hardware/software compilers.

These aspects represent the compiler stages that find parallelism within the algorithm, and transformation into a combination of software and hardware intermediate representations (IRs). Specifically, the approach used here considers only hardware/software compilers for imperative, sequential programming languages.

Topics where formal specification techniques for conventional compilers can be applied directly to hardware/software receive little attention here. For the compiler frontend, these include pre-processing, lexical analysis and parsing. For the compiler backend, these include register allocation for, and code generation from an IR, for a given GPP. Topics related to other parts of the tool-chain, such as the assembler and linker are similarly out of the scope of this thesis.

Two approaches to achieving performance improvements using custom function units have already been identified here. The first is to ensure that the function

unit provides better computational efficiency for a given task than a general purpose processor. This could mean ensuring that the function unit uses fewer logic resources or clock cycles to complete a given computation than the GPP. The second is to ensure that the function unit provides sufficiently greater parallelism for a given computation that its use amortises the overhead of 'invoking' the function unit. That is, that the time to pass data to and from the function unit is less than the net improvement in execution time of dedicated logic over the GPP.

Techniques that employ the former approach include methods such as bit-width optimisations that reduce the amount of logic generated by reducing word size where it will not affect the output, thus eliminating redundant logic [BSWG01].

The latter approach requires the derivation of parallelism from a sequential implementation of an algorithm. Unlike methods that adopt the former approach, it requires the translation from a sequential language with its associated semantic rules into a language that supports parallelism (and possibly non-determinism) and hence, with very different semantics. As such, it is a fundamental technique in hardware/software compilers, and thus this approach receives more attention in this thesis than the former.

It should be clarified that considering the derivation of parallelism from sequential implementations of algorithms here is not with the intent to render existing HDLs obsolescent.

Hardware/software compilers ease the process of co-design: where it is not clear how an algorithm should be partitioned into hardware and software, such compilers reduce the overhead in re-implementing a sequential algorithm in hardware, and updating the hardware/software interface to match. However, they still require the use of a GPP in the targeted system, the implementation of which is assumed to be described in an HDL. Furthermore, just as assembly language skills may be required to analyse and tune the output of a traditional compiler, hardware design skills may be required to analyse and tune the output of hardware/software compilers.

The approach taken in this thesis assumes that there exists a means of identifying which parts of a program should be realised in software, and which parts should be realised in hardware. As existing solutions to this problem exist — such as the use of pragmas in the source language, or the use of profiling information to determine partitioning — this issue receives little attention here.

1.3.2 Hardware/Software Interface

Compilers for high-level languages typically parse the source code of a program and construct a representation of the program in an abstract syntax tree. The abstract syntax tree is then converted into a compiler *intermediate representation* (IR). The program in its IR form undergoes a series of analyses and transformations. These include optimisations (and the analyses necessary for those optimisations), and *code-lowering*: the process of reducing the semantic gap between the IR and the target architecture [Muc97]. The IR may allow the many high-level language constructs to be represented, while the target architecture may provide a relatively primitive set of operations. Code-lowering is

used to map high-level language constructs into a lower-level language, which may still be independent of the target architecture. After these transformations are complete, the compiler's code generator converts the simplified IR into object code for the target architecture.

Multiple Intermediate Representations

A compiler may use more than one IR. The use of more than one representation is useful where there is a significant semantic difference between the source language and the target language. By lowering code through several representations with decreasing levels of abstraction from the target machine, the task of compilation is divided into simpler compilation tasks between IRs.

Another reason to use more than one IR is that some representations may by more apt for a given transformation (or analysis) than another. For example, it is easier to re-order expression trees if the sub-expressions (sub-trees) have no side-effects [App97].

The framework in this thesis defines more than one representation for both of the above reasons. This approach reduces the complexity of transformation from an imperative, sequential language which has substantially different semantics to a hardware design language. The use of different IRs also simplifies the specification of transformations that make parallelism inherent in an algorithm expressed in a sequential language explicit.

Synthesis from an Intermediate Representation

It is assumed that hardware designs for a function unit are to be synthesised from the IR form of a program.

This assumption permits the use of techniques that are conducive to producing efficient hardware and software. By synthesising from an IR form of the program, existing compiler analyses and optimisations may be applied before hardware synthesis. Furthermore, it allows the use of optimisations that modify the program that cross the *phrase structure* of the language.

Examples of optimisations that may be usefully applied before hardware synthesis include dead code elimination, common sub-expression elimination, strength reduction and loop unrolling. Code elimination optimisations may prevent redundant hardware being generated and using an unnecessarily amount of logic in the function unit. This, in turn, may prevent other parts of the program being hardware accelerated. Strength reduction may allow more efficient logic to be generated. Loop unrolling may be used to achieve better parallelism in the resulting hardware.

One of the implications of performing hardware synthesis from an IR is that the part of the program representation that is synthesised to hardware may not correspond directly to the source text. However, this is the case for the object code output produced by existing software-only compilers, is not expected to present problems in verification because existing compiler verification techniques support reasoning about compilation algorithms that provide such kinds of optimisations [Jon69].

Further rationale for the assumption that the hardware/software compiler should produce a relatively low level hardware design generated from an IR is given towards the end of Section 3.3.1, in the section entitled *Summary of Hardware Representations*.

Hardware/Software Partitioning

A program that is to be compiled into a combination of hardware and software must be *partitioned* into those parts that will be compiled to object code and those that will be synthesised into a hardware design. In order to represent this partitioning, it is necessary to annotate parts of the IR to indicate which compiler backend should be used for each part of the program: that is, whether the compiler should use code generation or hardware synthesis for each part.

If partitioning is to be specified manually — perhaps by pragmas in the source language — it is necessary that each IR supports these annotations and that they be propagated through each transformation. It should be noted though that such annotations may be subject to modification by optimisations and other transformations on the IR.

Hardware synthesis from an IR may allow higher levels of automation in the selection of which parts of a program are to be synthesised into hardware. This would mean that the hardware/software interface may not necessarily be precisely defined by the programmer, but rather by analysis of the program text (probably in addition to pragmas or profiling information) to determine a suitable partitioning.

Synthesis from an IR allows sub-expressions in the source text to be synthesised into hardware, while the outer (nesting) expression is compiled into software that uses a function unit for those sub-expressions that have been realised in hardware. Thus, the partitioning of a program into hardware and software may cross the phrase structure of the source language.

Memory Interface

There are practical problems associated with a function unit efficiently sharing access to memory with a GPP. Some of these problems are discussed here. As such, the methodology used here considers techniques that can be used to transform the program such that the function unit does not require a memory interface.

If a function unit has direct access to the system memory, it is important to ensure that there can be no interference that can cause the program to behave differently to the way it would had it been compiled into software only. This may be particularly difficult where the function unit may access program memory in a Harvard architecture, or index into system memory using a program value as a memory index on a von Neumann architecture. Such behaviour would be characteristic of self-modifying or self-examining code, the problems with which were discussed in Section 1.2.4, *Specific Problems in Hardware/Software Compilation*.

A memory controller in the function unit may also require the use of some of the logic in the function unit, which may be undesirable as it introduces extra complexity and uses logic resources that could otherwise be used for computation.

1.3.3 Methodology and Thesis Structure

In order to apply formal specification techniques to analyses and transformations, it is necessary to provide a formal definition of each language (or IR). A formal definition — and thus formal semantics — of each language is required to formulate a precise statement of the behaviour of a program in its corresponding representation. Furthermore, formal definitions are required in order to formulate a precise specification of transformations: that is, what it means to preserve the semantics of a program as it is translated from one IR language into another.

Chapter 2 describes the design flow for FPGAs, which have a significant rôle in hardware/software compilation, because they are reconfigurable. It presents background literature on compiler IRs and hardware/software compilers, and also some general background in formal semantics and reasoning in the context of compiler development.

Chapter 3 addresses some of the practical issues that arise when using an machine support to model and reason about hardware compilation. In particular, it introduces the Isabelle system and Isabelle/HOL — the automated theorem proving environment and formulation of higher order logic used to denote and verify most of the formulae and propositions in subsequent chapters (exceptions to this are in reference to existing literature). The use of higher order logic to model the behaviour of hardware circuits is also introduced.

This thesis presents two IRs in the order in which similar representations could be used in a hardware/software compiler. A formal semantics of each IR is given, together with examples of program fragments expressed in that IR.

The first IR is presented in Chapter 4, and is based on both existing IRs for conventional compilers, and on existing IRs for hardware/software compilation. No specific HLL is assumed, but it is intended that it be relatively simple to target this IR as the first stage of compilation from the abstract syntax tree representation of a program in an HLL.

The translation into this IR reveals some of the inherent *instruction level parallelism* (ILP) in the program being compiled. Once made explicit, this parallelism can be exploited by generating function units that provide concurrent evaluation of program constructs that are identified as being candidates for concurrent evaluation.

The second IR — presented in Chapter 5 and Chapter 6 — is a low level language for modelling hardware logic designs. It is intended to represent a language that could be translated into a netlist format, or limited subset of an HDL, in a straight-forward manner.

The features of the netlist language that provide support for representing the primitives in the first IR are described in Chapter 6. An approach to modelling the behaviour of combinations of these primitives is discussed by appeal to

existing literature that adopts a different approach, but shares the common goal of correct hardware compilation.

Chapter 7 presents an example of a program fragment expressed in the first intermediate representation. It also presents a design in the netlist representation that is conjectured to provide a hardware implementation of that program fragment. The example supports an explanation of the what it means in general for a design in the netlist language to correctly implement part of a program in the intermediate representation.

Chapter 8 concludes with a discussion of the work presented in this thesis in the context of related literature, and describes a number of practical issues that arose during its development.

Chapter 2

Background

Contents

The use of hardware/software compilers for FPGAs has been described as a 'meeting point' between the *electronic design automation* (EDA) industry and the *high performance computing* (HPC) industry [Mor05]. In order to increase the level of design abstraction presented to hardware designers, EDA tools have been developed that allow hardware synthesis from languages based on C. This technique is sometimes known as *high-level synthesis*, and examples of such tools based on the approach include Celoxica's DK Design Suite and Mentor's Catapult C. Meanwhile, in order to improve the performance of existing applications, vendors to the HPC industry have developed tools that allow languages used for HPC applications to target hardware and software using conventional software development techniques: providing software developers with familiar compile/debug development cycle. Tools that fall into this category include SRC Computers' Carte Programming Environment.

The different skill sets and expectations of the EDA community and the HPC community account for the different tool sets. This chapter describes the fundamental concepts behind those tool sets.

Section 2.1, *FPGA Structure and Design Flow* describes the structure of an FPGA, showing how a custom logic design can be mapped into an FPGA configuration. It describes a conventional design flow for FPGA designs using EDA tools. Section 2.2, *Compiler Intermediate Representations* introduces a variety of intermediate representations used in compilers that are typical of those used in the compilation of high level languages and, consequently, in

hardware/software compilers. Section 2.3, *Hardware Acceleration using FPGAs* describes several approaches to using FPGAs for hardware acceleration, including hardware/software compilers that use such representations.

2.1 FPGA Structure and Design Flow

This section describes how custom logic designs can be reified using FPGAs. More specifically, the discussion here pertains to SRAM-based FPGAs, which are typically more useful for hardware acceleration because they can be reconfigured, due to the volatile nature of their configuration. However, there is little reason why the techniques described in this thesis should not apply to FPGAs with non-volatile configurations. Given that SRAM-based FPGAs are commodity items, and that their reconfigurability allows a traditional compile/test/debug cycle, it is reasonable to assume their use in hardware/software compilation here. Even if a non-reconfigurable FPGA were to be targeted, a design would likely be tested on an SRAM-based FPGA first.

The structure of the reconfigurable fabric in FPGAs, and the availability of additional dedicated logic within the FPGA such as multipliers, varies between different models and manufacturers. Likewise, the techniques for implementing logic designs in FPGAs — the design flow — varies. One aspect that affects the design flow is the method used for design entry. The design may be created in various ways, including the use of schematic entry; by describing the required design in an HDL; by instantiating existing IP cores for which no human readable design is readily available; through the use of hardware/software compilers; or by a combination of any of these techniques.

Rather than provide a taxonomy of FPGAs and their associated tool support, this section serves to provide illustrative examples of typical FPGAs and tool sets. This is sufficient to provide the required background for this thesis in FPGA structure and design flow.

A broader view on the topic can be found in tutorials and surveys on the topic [BR96, Hau98]. More specific information can be found by consulting manufacturers' data-sheets for specific FPGAs.

2.1.1 FPGA structure

The Xilinx Spartan-3E family [Xil05b] of FPGAs are similar to those used in existing FPGA-based hardware acceleration products. The family of FPGAs is described here briefly to illustrate the structure of a typical FPGA, and to show how a logic design can be reified in reconfigurable logic [1].

It is important to note that logic designs can be implemented in reconfigurable logic, because the ability to do so forms a major assumption of this thesis. This section is intended to provide an intuitive understanding of how a logic

[1] This particular family was selected for discussion here in favour of those known by the author to be used in existing hardware acceleration products only because of the clarity of its datasheet. Structurally, it is similar enough to those used in existing products for the purposes of the high-level discussion here. Thus, it is reasonable to assume that they may be appropriate for use in hardware acceleration products (even if they are not already).

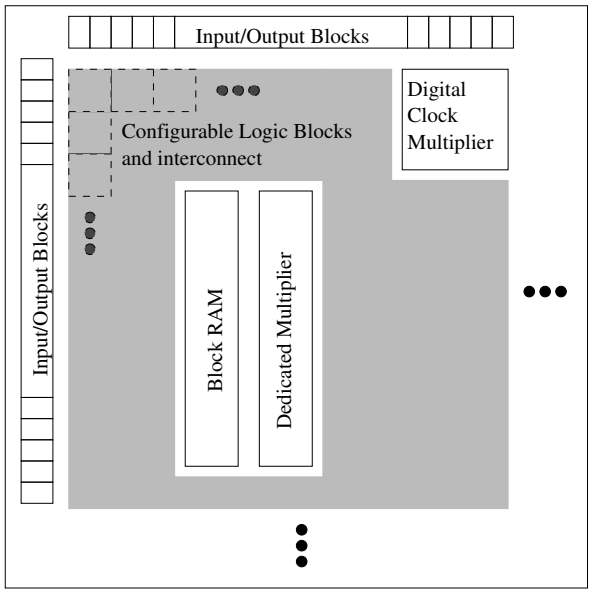


Figure 2.1: High-level view of a Spartan-3E FPGA

design can be mapped onto reconfigurable hardware. After this section, no further reference is made to the process of creating an FPGA configuration from a logic design. Instead the focus is on producing correct logic designs for hardware acceleration. Ensuring that a logic design is correctly mapped into an FPGA configuration falls outside the scope of this thesis.

Structure

The high-level structure of a Spartan-3E FPGA is shown in Figure 2.1. The design is comprised of a matrix of *configurable logic blocks*, known as CLBs, which can be used to implement both combinational and sequential logic. These can also be used for data storage, known as *distributed RAM*.

Several units of logic dedicated to specific tasks are interspersed within the matrix of CLBs. These include blocks of RAM — *Block RAMs* — which provide more space-efficient data storage than the multi-purpose CLBs. They also include dedicated multipliers, which can be used to implement faster multiplication than can be achieved using the CLBs.

The matrix is surrounded by a set of programmable input/output blocks which connect the pins of the FPGA package to the internal signals of the FPGA.

Configurable Logic Blocks are comprised of four *slices*. Each slice contains two SRAM-based *Look-Up Tables* (LUTs) and two single bit storage elements. They also contain dedicated carry logic to improve the performance of arithmetic operations, and multiplexers to simplify logic that is implemented across several slices.

Each LUT has four logic inputs and one logic output, and has sixteen bits of in-

ternal storage. They can be used to implement combinational logic, distributed RAM or shift registers.

When used to implement combinational logic, the four logic inputs form a four bit address that selects one of the sixteen stored bit values, which becomes available on the single output. Thus, any Boolean function of up to four Boolean values can be implemented using a LUT, simply by loading its sixteen bit RAM with the appropriate binary representation of the truth table for that function.

When using LUTs to implement distributed RAM, each LUT contains one bit of up to four values represented as bit vectors. The remaining bits for each value are stored in nearby LUTs, and each of the four values can be selected by applying the same four bit address to the inputs of those LUTs.

The two single bit storage elements may be used as either D-type flip-flops — to implement a register, for example — or as level-sensitive transparent latches. These are used to implement sequential logic.

The interconnections within a slice — between the LUTs and the storage elements — are reconfigurable. These connections are largely controlled by multiplexers, with configurable 'select' values. These also control connections to other slices within the same CLB.

The routing between CLBs is controlled by configurable switch matrices. Each switch matrix is connected to horizontal and vertical signal lines of various lengths that connect nearby CLBs. They also connect adjacent CLBs to allow signals to be routed from CLBs with relatively low connectivity to better connected CLBs.

2.1.2 FPGA Design Flow

The ISE Design Tools [Xil05c] provided by Xilinx support a complex design flow. The design flow supports a variety of design entry techniques, and allows different FPGAs to be targeted.

The terminology here is somewhat awkward. 'Design' may be used as a verb, to refer to the process of designing hardware logic; or as a noun, to refer to the product of this design process. 'Design flow' is commonly used to refer to development steps with tool support. Thus, it is appropriate to say that design flow typically begins with design entry, even if the process of designing logic begins much earlier.

The design entry techniques supported by the ISE Design Tools include HDL entry for VHDL and Verilog; graphical schematic input; and an IP core generator that produces various designs which can be tailored to a particular design by instantiating design templates with appropriate parameters. A design may include components described in more than one of these formats.

HDL of mixed levels of abstraction can be used. Some of a design may be described physically, at a very low level of abstraction, in terms of the locations of individual primitives on the FPGA. Other parts of a design may be described structurally, where the components are instantiated and connections between them are defined explicitly. In a structural description, an instantiated

component may be a complex device such as an *Arithmetic Logic Unit* (ALU), and need not be a primitive directly provided by the FPGA. A higher level of functional abstraction still is a behavioural description of a device, from which a structural description must be inferred.

Once a design has been entered, the next stage in the design flow is *synthesis*. This process combines the various design formats used into a single structural *netlist* format, such as the *Electronic Design Interchange Format* (EDIF) or *Native Circuit Description* (NCD). Part of this process includes behavioural synthesis that translates any parts of a design expressed in behavioural HDL into a structural format.

After synthesis, the resulting netlist may be combined with a set of design constraints into a *Native Generic Database* (NGD). In the ISE Design Tools, this process is known as *translation*. Typical uses of design constraints include requiring that a given signal be connected to a named pin on the FPGA, and applying a maximum timing constraint for a given signal.

The resulting design with associated constraints forms the input to the *map* process. This matches the components in the structural description of the design with types of FPGA primitives provided by the targeted FPGA.

Once mapped, the next stage of implementation is *Place And Route* (PAR). In the previous stage, the types of primitives required for each part of the design were identified. The PAR stage selects actual resources in the target FPGA for those primitives. 'Place' refers to the selection of specific FPGA primitives, while 'route' refers to finding a configuration for the reconfigurable interconnect that will connect the placed primitives according to the netlist.

2.2 Compiler Intermediate Representations

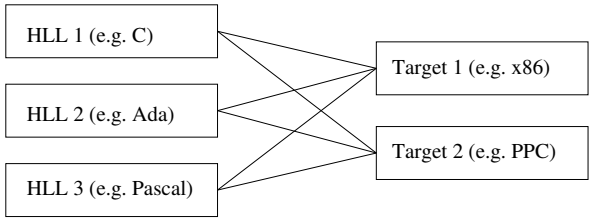
Typically, compilers for high-level languages produce assembly language output, which is then passed to an assembler in order to produce object code. After any pre-processing stages, compilation proceeds by a combination of lexical analysis and parsing, in order to build a parse tree representation of the program.

The structure of the parse tree is largely determined by the concrete and abstract syntax of the high-level language.

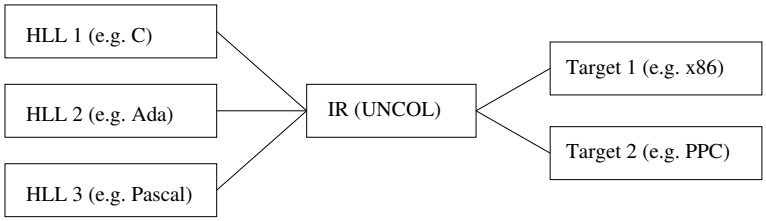
Compilers generally transform the program into a corresponding assembly language output via a series of IRs. This section considers the purpose of using IRs in compilation, and provides a brief account of IRs relevant to hardware/software compilers.

IRs serve several purposes in compilers for high-level languages. One such purpose is to allow the development of compilers that support more than one target architecture — known as retargetable compilers — and compilers that support more than one source language. Developing a suitable representation that allows this flexibility, and that can be used to produce efficient object code, is traditionally known as the UNCOL (*UNiversal Computer-Oriented Language*) problem [Con58].

Without the use of an IR, the compiler must translate directly from the source



(a) Without an IR: one compilation algorithm for each combination of source language and target architecture.



(b) Without an IR: one compilation algorithm for each source language and one for each target architecture.

Figure 2.2: A compiler IR helps to support multiple source languages and multiple target architectures.

language into the target language and thus a separate compiler is required for each combination of source language and target language. Translating a source language program into an IR first, and then translating that IR into the target language, requires one compilation algorithm for each source language and one for each target language.

To restate this: without the use of an IR, the number of compilation algorithms required to allow compilation from n source languages into m target architectures is $n*m$. Using a single IR, the number of compilation algorithms required to support n source languages and m target architectures is $n + m$. Therefore using an IR requires fewer compilation algorithms if more than one architecture and more than one source language are to be supported. Figure 2.2 illustrates this concept for three high-level languages and two target architectures.

It is often beneficial for a compiler to use more than one IR, and to translate the program between these representations during compilation. This is, in part, because some representations are better suited to performing some types of analyses and optimisations than others. For example, a low-level representation in which registers and addressing modes are explicit may be used to improve register allocation (by reducing the number of 'spills' into memory). Higher level representations may be more suited to dependence analysis [Muc97].

An additional benefit of using multiple intermediate representations is that their use divides the compilation task into several simpler tasks. Rather than compile from a high-level representation into a low-level representation in one step, the compiler can repeatedly translate the program into lower level representations, in several stages, before code generation. This reduces the complexity of a given compilation stage.

2.3 Hardware Acceleration using FPGAs

In Chapter 1, the performance benefits of using function units to support GPPs were introduced. These were identified as being the result of increased computational efficiency, due to function units having logic dedicated to a particular task; and the result of increased parallelism. It is noted that, depending upon our interpretation of 'computational efficiency', these concepts may not be entirely distinct. FPGAs have been used to implement function units that provide both improved computational efficiency [AS93] and parallelism [LEM04].

In this section, the rationale for using FPGAs to implement *reconfigurable function units* (RFUs) is discussed. Different architectures of systems with RFUs are considered, along with techniques used for hardware/software compilation for such architectures. Finally, some examples of applications that have been shown to benefit from the use of RFUs are described.

Custom logic designs can provide performance improvements over GPPs. However, the performance advantages of dedicated function units must be weighed against the cost implications of their use. While GPPs are commodity items, specialised function units do not enjoy the same market share. This is because a given custom logic design typically only benefits one application, or one class of applications. Thus specialised function units are likely to be more expensive. However, FPGAs are now commodity items: although the market share for individual logic designs for FPGAs may be small, FPGAs themselves are not specialised to a particular application, and benefit from commodity economics [DeH94].

FPGAs also obviate the need for ASIC mask production, and so benefit from lower *non-recurring engineering* (NRE) costs and improved time-to-market for each application.

A given collection of fixed-purpose function units is less than optimal for most applications, because the hardware resources used to implement them are not utilised in applications that do not benefit from those function units. Although RFUs have some overhead in hardware resources — due to their reconfigurable nature — they can be used to provide performance improvements to many types of application [Hau98].

Performance improvements can also be achieved by increasing the clock frequency of a GPP [HP02]. However, this technique is of limited use. Increasing the clock frequency results in an increase in the power consumption of the GPP. This may have a negative impact on battery life for mobile devices, and cause heat dissipation problems in higher end processors. The use of FPGAs to implement function units has been shown to allow increased performance without commensurate power consumption [BJC+03].

2.3.1 Types of RFU

The way in which an RFU is integrated into a system, and the degree of coupling between RFUs and the GPPs of that system, varies between the different ISAs that include reconfigurable logic. Furthermore, while the description of FPGAs in Section 2.1 provided an intuitive understanding of how hardware

can be 'reconfigurable', the granularity of the reconfigurable logic used is not always as fine-grained as the example given there. In this section, different designs for systems containing RFUs are considered.

RFU Coupling

Some systems have included reconfigurable logic as part of the processor, where the reconfigurable logic is directly connected to the rest of the processor. Examples of such systems include PRISC (PRogrammable Instruction Set Computers) [RS94], Nano [WHG94], DISC (Dynamic Instruction Set Computer) [WH95] and OneChip [WC96]. Such processors — known as *reconfigurable instruction set processors* (RISPs) — provide instruction formats that allow the RFU to be used as if it were any other function unit on the processor [BD02]. In some systems, the RFU shares general purpose registers with the rest of the processor.

The overhead of transferring data between the RFU and other function units, and of invoking the operations implemented on the RFU, is relatively low on RISPs due to the proximity and tight coupling of general purpose and reconfigurable logic. However, the size of logic designs is limited because the reconfigurable logic shares the same die as the rest of the processor.

RFUs on RISPs are usually used to provide custom instructions on small amounts of data. This is because the tight coupling makes it efficient to transfer small amounts of data to the RFU, while the limited amount of reconfigurable logic available in a RISP limits the amount of data that the RFU can process at any given time. Examples of custom instructions include multiply and accumulate operations (MAC); variable length coding and decoding (VLC and VLD); error correction logic and other bit-level operations [AS93]. Applications that benefit from bit-level operations also include static and dynamic code generation [ATB05, Eng96].

Another approach, using a less tightly coupled RFU, is to use a co-processor with reconfigurable logic. Examples of this approach include the Garp [HW97], Napa [GS98] and Prism-II [WAL+93] architectures.

This approach permits designs to use more reconfigurable logic, compared to RISP designs. Synchronisation and data transfer with the co-processor can be achieved using similar protocols to those used for other co-processors, such as external floating point units [BD02]. The RFU then operates in parallel with the GPP. As such, the approach can reduce the overhead of invoking operations on the function unit compared to RISPs, because the RFU may perform operations that take many cycles to complete without needing to synchronise or interrupt the processor [CH00].

RFUs may also be connected as an *attached processor* via a system bus, such as the PCI bus [LTS99], or as if they were another processor in a multi-processor system. This allows for large amounts of reconfigurable logic to be used. Just as for using an RFU as a co-processor, the RFU may operate in parallel to the CPU. However, using this technique, the RFU has no access to the GPP's data cache, and synchronisation and setting up data transfers have greater overhead.

Some classifications of reconfigurable computing platforms also distinguish be-

tween RFUs that are connected via a system bus and internal to a system, and those that are connected to external stand-alone reconfigurable units [Hau98].

An RFU may be supported by a memory store dedicated to the RFU. In addition this, RFUs may have an interface to the system memory, optionally with their own address generator.

RFU Granularity

Not all reconfigurable hardware used for hardware acceleration is as fine-grained as the use of FPGAs described in Section 2.1. That section considered logic that is reconfigurable at a bit level. In contrast, *reconfigurable data path units* (RDPUs) have higher granularity.

Fine-grained architectures can be less efficient due to the routing overhead: the flexibility of allowing data paths with a width of one bit incurs significant penalty due to the large amount of routing resources required. Using wider data paths reduces routing overhead, and also the size of the configuration for the reconfigurable logic [Har01].

RDPUs have fixed purpose logic units which are implemented in dedicated logic for improved efficiency. The RaPiD architecture provides a number of ALUs, multipliers, registers and memory modules in a linear array connected by a reconfigurable 16 bit data path [ECF+97].

2.4 Logics, Meta-logics, and Logical Frameworks

In Section 1.2.2, it was noted that formal definitions of source and target languages are required to precisely formulate the correctness criteria — or the specification — of a compilation algorithm. An example of the structure of a formal definition was provided, that included definitions of the concrete and abstract syntax; context conditions; semantic domains; and semantic functions or rules.

The purpose of a formal language definition is two-fold. Firstly, the definition serves as a means to allow programmers to reason about the correctness of a program written in that language. Secondly, the definition may also form the basis of the formal development of a compilation algorithm.

The correctness property of a compilation algorithm is that it preserves the semantics of a program. Where the compilation algorithm has a different source and target language, this property must be expressed with respect to a relation between both the semantics and semantic domains of each language.

In this section, *formal frameworks* are considered in which language definitions may be described, and compilation algorithms specified and verified.

2.4.1 Formal Systems

Formal systems can be used to represent logical systems (or simply, 'logics') using syntactic techniques. This allows formal reasoning within a logical system, and also reasoning about logics represented by the formal system. The use

of exclusively syntactic techniques allows reasoning to be formulated as proofs that can be verified by computer programs, without reference to the intuition that underlies the logic.

Logics are represented within a formal system by a formal language, and a set of inference rules for reasoning about expressions within that language. Examples of logics that can be represented by formal systems include propositional logic, predicate logic [GT96] and the logic of partial functions [BFL$^+$93].

Some logics that can be defined by a formal system are sufficiently expressive that they themselves can be used as a formal system for describing logics. When defining a logic within a formal system, it is often important to distinguish between the *object logic* that is the being defined, and the *meta-logic* of the formal system used to define the object logic. When referring specifically to the formal languages of each logic, it is useful to distinguish between the *object language* and the *meta-language*.

Various definitions of the term 'formal system' exist, and there are different ways in which a formal system can be modelled [Smu61]. The term is used less specifically here than in earlier work relating to formal systems in which the syntax of the logic under construction is described using a context-free grammar. This is because the primary interest in using formal systems here is in order to construct logics to model the formal semantics of programming and hardware description languages. Assuming the underlying formal system is sound, it is the properties of logics and their ability to model the formal semantics of programming languages that is of interest.

It may, therefore, appear to be a digression to consider such foundational reasoning techniques here. However, when reasoning within an object logic using Isabelle/HOL — the theorem prover used to model the languages presented in Chapter 4 to Chapter 6 — it is necessary to be aware of the distinction between the meta-logic and the object logic.

For the purpose of the discussion that follows, a formal system is considered to define a logic in terms of the following attributes:

- an alphabet used for constructing *sentences* within the formal system;
- alphabets used to identify variables and predicates;
- a grammar that can generate expressions from these alphabets;
- a finite set of axioms, each of which is generated by the grammar;
- a finite set of inference rules for deriving new theorems.

The first three of these attributes are used to characterise a set of expressions that are considered to be meaningful within the object logic. These are discussed below in *Formal Languages*. The remaining attributes are used for reasoning about such expressions within the object logic, and are discussed in *Formal Reasoning*.

Formal Languages

Formal systems define a formal language that can be used to represent expressions in the object logic. The formal language is defined in terms of alphabets used to represent elementary terms and operators in the object logic, and in

terms of a grammar used to construct expressions from elementary terms and other expressions.

A number of different alphabets can be used to define a formal system, depending on the logic to be modelled. For example, constants, variables and predicate identifiers may be taken from different alphabets. Allowing expressions in the formal system to contain variables allows an expression to represent many sentences within the formal system, by using different instantiations for each variable.

Meta-variables can represent non-terminal elements of the grammar in an expression. Expressions that contain arbitrary terms are known as *sentential forms*. Expressions that contain no meta-variables are called *sentences*.

Sentences and sentential forms denote sets of expressions within the grammar. If there are no derivations of a given sentence or sentential form, then it denotes an empty set of expressions. On the other hand, they can denote an infinite set of expressions. For example, if $?P$ and $?Q$ are meta-variables, and $\neg$ and $\longrightarrow$ are in the alphabet of a formal system with an infinite number of identifiers, then the sentential form $((?P \longrightarrow ?Q) \longrightarrow \neg\ ?Q) \longrightarrow \neg\ ?P$ denotes an infinite set of expressions (which, in this case, are presumably theorems).

A grammar generates an enumerable, possibly infinite set, of expressions. The expressions can be viewed simply as strings over the relevant alphabets. Alternatively, they can be viewed as trees of an *abstract syntax* [McC63b].

For simple formal systems, each expression generated by the grammar is considered to be a *well-formed formula* (wff). It is assumed that there is a decision procedure (a total Boolean function) that determines whether or not a given expression is generated by the grammar: that is, whether it is in the set of well-formed formulae.

In more complex formal systems, only a subset of the expressions generated by a context free grammar are considered well-formed. For example, a grammar may allow the use of integer numbers and Boolean values in a formula, but not admit a formula in which an integer number is tested for equality with a Boolean value. Such formal systems can be described using context sensitive grammars.

Alternatively such systems may be described using a context free grammar, together with a set of Boolean functions defined over terms that represent syntactic entities in the formal system. The Boolean functions characterise the set of well-formed formulae within the formal system, and are known as *context conditions*. These are predicates in the meta-logic being used to describe the formal system that characterise terms in the formal system according to the syntactic entity that they represent.

Formal Reasoning

Formal systems define a finite set of axioms — propositions that are assumed to be true — and a set of rules for deriving new theorems from those axioms and from other theorems already derived. Thus formal systems admit formal reasoning in the object logic: a proof can be represented by a description of the derivation of a given theorem from a set of assumed axioms, theorems, and

lemmas.

Axioms are theorems, and it is assumed that each axiom is a well-formed formula. Theorems derived by applying inference rules to axioms and existing theorems are also assumed to be well-formed.

Inference rules are represented as a set of assumptions (or premises), together with a single expression which represents the conclusion of the inference rule. Expressions used in the assumptions and the conclusion of an inference rule are typically sentential forms because the use of variables allows systems with an infinite number of theorems to be constructed.

Formal systems may introduce notation for delimiting premises, and for the implication operator (typically denoted by $\supset$, $\longrightarrow$, and $\Longrightarrow$) used to separate the premises in an inference rule from its conclusion. Where only a single implication operator exists in an inference rule, it is sometimes written as a judgement, using a horizontal rule to separate the conclusion from the premises. For example, an inference rule with n premises may be denoted as follows:

$$\boxed{\textit{rule}}\ \frac{\begin{array}{l}\textit{prem-1}\\ \ldots\\ \textit{prem-n}\end{array}}{\textit{conc}}$$

Inference rules are applied using syntactic techniques, and make no reference to an interpretation of the axioms and theorems to which they are applied. This makes it possible to check proof derivations using a machine. Proof checking algorithms may be implemented as total Boolean functions, and are terminating.

A proof of a given theorem may be represented as a sequence describing the order in which inference rules can be applied to existing theorems in order to derive that theorem.

Computer Aided Verification

Reasoning within an object logic using a formal system can be an onerous task: the derivation of a proof for a proposition can be long, even for those that seem to be 'obviously true'. Furthermore, the exact derivation for the proof of a given proposition may not be obvious — some 'exploration' is often necessary in order to find an appropriate derivation.

One of the advantages of reasoning within a formal system is that proofs of theorems can be checked by a computer program. Given the complexity of constructing proofs within a formal system, it is not surprising that many programs that can verify proofs also provide some level of automation for exploring and constructing proofs.

The level of automation that can be provided by a computer program varies according to the object logic. If the object logic is *decidable*, and the computer to be used for the verification task sufficiently powerful, then proofs of theorems in the logic can be automatically derived. For example, a tableau-based decision procedure can be used to automatically determine whether a statement in propositional logic is a theorem.

Modal logics are rather more expressive than propositional logic, yet decision procedures for them can still be developed [Var97, Grä01]. Modal logics can be used to model and reason about certain behavioural properties of labelled transition systems, or state machines. Such methods have been used in model checking programs such as NuSMV [CCG+02] for verifying properties of control-path hardware circuits and concurrent systems.

The logics that are sufficiently expressive to formulate many interesting mathematical statements have no decision procedure. These include first order logic, and higher order logics.

For logics which are in general undecidable, only a limited amount of automation can be provided. Although theorem provers for such logics cannot verify, or find proofs for arbitrary theorems, they can assist the user by providing a number of tools to reduce the amount of work required to construct a proof of a given theorem. These tools include term rewriting and simplification; decision procedures for terms which are decidable; and proof management for forward and backward reasoning.

Examples of theorems provers that provide some form of automation for constructing proofs are HOL [Gor85], Coq [DFH+91], LEGO [LP92] and PVS [ORS92].

2.4.2 Formal Semantics

Formal reasoning about programming languages requires a formal system in which the definitions of programming languages can be expressed. Furthermore, in order to reason about particular programs in a given programming language, it is necessary to have a definition of that programming language expressed in a formal system[Win93].

One can, of course, reason about an algorithm independently of the languages in which it has been, or could be, implemented. This approach is typical in compiler verification literature (such as [BJ82] or [KN04]), where the focus is on the definitions of source and target languages and the compilation algorithm, rather than on the implementation of that algorithm.

The formal semantics of a programming language can be expressed using a number of different styles [Jon03], including the *operational* and *denotational* styles.

The operational technique is, to a great extent, attributable to John McCarthy [McC63a]. McCarthy introduced an interpretation function, or i function, that mapped a program and an initial memory state to final program states. Thus the type of his i function was $\mathit{Program} \times \Sigma \to \Sigma$.

The interpretation function was defined recursively in terms of an *exec* function, of type $\mathit{Stmt} \times \Sigma \to \Sigma$, that evaluates the side effect caused by a program statement; and an *eval* function, of type $\mathit{Expr} \times \Sigma \to \mathit{Value}$ that evaluates expressions. McCarthy applied this technique to provide a semantics of a subset of ALGOL [McC66].

Where operational semantics are defined such that the execution rules of the program are directed by the syntactic structure of the program, they are referred to as *structural operational semantics* [Plo81].

Denotational semantics is an alternative technique to operational semantics for formalising the meaning of programming languages. The technique was pioneered by Christopher Strachey [Str66], and later given a mathematical basis by Dana Scott [SS71, Sco70].

Using denotational semantics, the semantics of a given program is expressed as a function from initial state to final state: $\mathit{Program} \rightarrow (\Sigma \rightarrow \Sigma)$. Superficially, this appears to be a simple currying of McCarthy's i function, found by introducing an extra lambda abstraction for the program representation. However, the two models are very distinct, due to the possible recursion (or other iteration) that may occur in the program.

In the presence of unbounded recursion, the denotation of a program — the semantic function representing it — may be a partial function. Thus in this case, no type for the semantic function could be given. Dana Scott introduced the concept of computational domains to model limited program recursion, and thus allow the semantic function to be typed, and therefore, given a solid mathematical basis.

The formal semantics of a simple compiler intermediate representation are defined in Chapter 4, and of a simple low-level hardware description language in Chapter 5 and Chapter 6.

Chapter 3

Machine Support for Reasoning

Contents

Formal reasoning refers to logical reasoning based on a formal system of the kind introduced in Section 2.4.1. Using formal reasoning, expressions may be proved to be theorems within that formal system, by applying inference rules to existing axioms and theorems in a purely syntactic manner.

Formal reasoning about programming languages requires a formal system in which programming language definitions may be expressed, including the semantics of those languages. Such definitions were discussed in Section 2.4.2, and may be used to define the formal semantics of a language. Within a formal system, propositions about language definitions may be expressed, and properties of the defined languages may be verifiable using formal reasoning.

In order to reason formally about hardware/software compilation, a formal system is required in which language definitions that model both hardware and software can be expressed. There is, of course, no requirement that the same language be used to describe both hardware and software. It is only required that the semantics of the two languages be modelled in the same formal system.

There are various approaches to defining the both formal semantics of languages that model hardware, and those that model software. This chapter considers the use of various approaches to constructing formal models for hardware and software languages; how the choice of approach can affect reasoning within and

about a formal model; and also how the choice of approach can affect reasoning that relates the software and hardware models.

When compiling custom function units, it is reasonable to consider both the correctness of the compiler and that of the compiler output (consisting of object code and a hardware model for the custom function unit) with respect to an initial source text. In order to address either of these considerations formally, it is necessary to have both a formal semantics of the software and hardware languages involved, and also a means to relate those semantic descriptions. The latter is necessary to define a notion of hardware behaving 'equivalently' to software.

One complexity that arises when relating the hardware and software models — and in the translation from one to the other — is that the translation considered here is not 'complete': only a part of a program is to be implemented in hardware, while the remaining (presumably larger) portion of the program should be compiled into object code. During execution, the general purpose processor and the custom function unit need to exchange data, and modelling the behaviour of the resulting system requires that this interface is modelled.

The chapter is structured as follows: Section 3.1, *The Isabelle System* describes Isabelle, the theorem proving environment used to construct language definitions presented later in this thesis, and for reasoning about definitions in those languages. Section 3.2, *Representation of key concepts* addresses the representation of semantic objects that are relevant to both hardware and software languages, including numeric values and addressable memory stores. Finally, Section 3.3, *Representation and Reasoning about Hardware* considers how hardware logic and its behaviour may be represented in a formal system.

3.1 The Isabelle System

The logic used for modelling the languages developed in this thesis is Isabelle/HOL [NPW05]: a formulation of higher order logic included with the Isabelle theorem prover [NPW02].

3.1.1 Isabelle/Pure

The Isabelle theorem prover implements a formal system called *Isabelle/Pure*, which is used as a meta-logic to define objects logics, such as Isabelle/HOL [Pau05]. In addition to the attributes of a formal system given on page 28, Isabelle/Pure provides a number of features specifically for the purpose of defining object logics. These allow the user to:

- define a concrete syntax for object logics, using infix and 'mixfix' grammar declarations and syntax macros, each with associated precedence rules;
- define an abstract syntax for object logics, using higher-order constants (discussed below);
- define inference rules that can be applied directly to terms in the object logic, rather than the representation of those terms in the meta-logic.

Isabelle/Pure Types

The formulations of higher order logic provided by both Isabelle/HOL and Gordon's HOL system bear some similarity to that of Cambridge LCF [Pau87], the predecessor of Gordon's HOL system from which it evolved. Each system uses the formula structure of propositional logic, and terms are based on the lambda calculus [Gor00].

The type systems used by each theorem prover are based on a modified version of Church's simple type theory [Chu40]. This includes aspects of the lambda calculus and Whitehead and Russell's original type theory. One such modification, due to Robin Milner, was to move type variables from meta-language into the logic: whereas a 'term' with a type variable in Church's notation denoted a family of terms with different type instantiations, Milner used the notation to denote a single polymorphic term. A type inference algorithm which can be used to find the type of a term based on the generic type of the constants within that term is also due to Milner [Mil78]. Both Isabelle and the HOL system provide type inference in a similar manner.

The syntax of types (classified into sorts) in Isabelle is given by the following ML definition:

```
datatype typ = Type of string * typ list
             | TFree of string * sort
             | TVar of index name * sort
```

The `Type(t, Ts)` ML datatype constructor is used to apply the Isabelle type constructor `t` to a list of type operands, `Ts`. The expression `Type(‘‘fun’’, [A, B])` represents the type of functions from type `A` to type `B`. A number of built in type constructors are included, and new constructors may be introduced. Isabelle/HOL provides a number of types, including fun (as above); bool (for Boolean values); product types; disjoint sum types; natural and integer numbers, and sets.

The `TFree` datatype constructor is used for representing explicit, named type variables. Type variables may be bound according to their context. The `TVar` is used for unknown types, which can be considered as free type variables. They represent universally quantified type variables, that is schematic or generic type variables, that have not yet been instantiated. Free type variables can become bound by type unification.

Both the higher order logic based theorem provers and the LCF based theorem provers provide logics in which all functions are total functions. However, the latter category require the use of domain theoretic techniques to ensure that functions are total, by interpreting terms as members of a Scott domain [Sto77]. However, Gordon claims that this technique is 'overkill' for the purpose of hardware verification, noting that primitive recursion is usually sufficient[Gor00].

Isabelle/Pure Terms

Terms in Isabelle/Pure are represented by the following ML datatype:

```
datatype term = Const of string * typ
              | Free of string * typ
              | Var of indexname * typ
              | Bound of int
              | Abs of string * typ * term
              | op $ of term * term;
```

Constants are terminal symbols in the grammar of the (meta- or object-) logic, and are represented using the `Const` constructor for terms. The type associated with a constant represents the *generic type* of that constant used for type inference.

The `Free` and `Var` constructors denote free variables and scheme variables respectively. Free variables are named by a string, while scheme variables are identified by an `indexname`: a name with a natural number subscript. Scheme variables differ from free variables in that they may be instantiated in unification, and that the numeric subscript supports a simple renaming mechanism.

The `Const`, `Free` and `Var` constructors represent the predicate calculus style aspects of the Isabelle/Pure syntax. The remaining three constructors — `Bound`, `Abs` and the infix operator `$` — represent the lambda calculus syntax constructs. Specifically, they represent bound variables (identified by de Bruijn indices); lambda abstractions and function application respectively.

Isabelle/Pure Syntax

The Isabelle/Pure meta-logic defines a few primitive types and constants. These frequently appear in formulae, even when developing a theory in one of the object logics that Isabelle defines. As a consequence, some familiarity with the meta-logic is necessary when working with theories defined in an object logic.

The ML representation for the type of functions with domain A and co-domain B is `Type(‘‘fun’’, [A, B])`, assuming the ML identifiers correspond with the types that they define. However, ML representations are not usually seen by an Isabelle user. Instead, the user uses a concrete syntax when working with expressions in the logic. The concrete syntax for above is denoted in the concrete syntax as $A \Rightarrow B$.

Implication within the meta-logic is denoted by $P \Longrightarrow Q$, where P and Q are Boolean terms. Implication in the meta-logic can be used to represent inference rules of the object logic. In inference rules with more than one assumption, the assumptions are normally surrounded by ‘Strachey’ brackets, and delimited with semi-colons. To provide an example, an inference rule could be denoted by $[\![P \Longrightarrow Q \; ; \; P]\!] \Longrightarrow Q$. This representation is an alternative representation for the formula $P \Longrightarrow (Q \Longrightarrow (P \Longrightarrow Q))$, or simply as $P \Longrightarrow Q \Longrightarrow P \Longrightarrow Q$ because implication associates to the right. The rule form shown on page 30 is supported by the Isabelle pretty printer.

Isabelle/Pure uses the notation $\bigwedge x \; . \; P \; x$ to denote meta-level universal quantification. This is sufficient to define the usual universal ($\forall$) and existential ($\exists$) quantifiers in object logics [Pau04].

3.1.2 Isabelle/HOL

Isabelle/HOL is an object logic provided with the Isabelle system. The use of Isabelle/HOL seems appropriate for reasoning about hardware/software compilation because Isabelle/HOL has been used to formulate aspects of imperative programming languages [Nip98, Nip03]. Furthermore, it is similar to the logic provided by Gordon's HOL system[Gor85] — the logic upon which Isabelle/HOL is based[NPW05] — which was originally developed for the purpose of hardware verification [Gor00].

The Isabelle/HOL object logic defines a number of built-in types. For each type defined within the logic, one can typically also expect to find definitions of the following:

- some constants(s) that inhabit the given type;
- operators and other functions on values of that type, and functions that return values of that type;
- syntax-related definitions to improve the readability of expressions that include terms of the given type;
- proofs of some general properties of terms of the given type.

A type may also have some theorem proving tactics associated with it, although their definitions are *extra-logical*: for Isabelle/HOL, these are implemented as functions in Standard ML.

Isabelle/HOL provides various means for defining new types, described below in *Isabelle/HOL types*. Conceptually, new types are defined as subsets of existing types. In order to ensure consistency when defining new types, it is necessary to prove that the new type is *inhabited*, by showing that there is at least one value of that type [NPW05]. This property is proved by the Isabelle system automatically for all of the types introduced in this thesis.

New constants can also be defined in Isabelle/HOL. Defined constants represent values, including higher order values. They can be referred to by name, or by any syntactic translations that have been defined for that constant. Constant definitions are of the form $c \equiv e$, where c is a new constant and e is an expression. If c is a function, then its arguments may appear on the left hand side. Thus the definition $c\ x\ y \equiv e\ x\ y$ is equivalent to $c \equiv \lambda x.\ \lambda y.\ e\ x\ y$.

New definitions cannot introduce inconsistency in the logic, nor change the behaviour of existing definitions. Isabelle rejects definitions that do not satisfy this requirement. The rules for new definitions are similar to those of the HOL system, and are described in the Isabelle tutorial [NPW02, §6.1.1]. To summarise briefly, it is required that arguments (x and y in the example) are distinct (unique) variables; that any variables, or type variables, on the right hand side also appear on the left hand side. Furthermore, constant definitions cannot be recursive.

This section is intended to describe only those aspects of the Isabelle/HOL logic that are required for comprehension of the definitions and proofs presented in subsequent chapters. More comprehensive treatments can be found in the Isabelle documentation [NPW02, NPW05].

Isabelle/HOL types

Boolean values The Boolean type, *bool*, is available in Isabelle/HOL. The values are denoted by *True* and *False*. The usual logical operators are defined including the infix operators for conjunction ($\wedge$) and disjunction ($\vee$). "If and only if" is denoted by object level equality ($=$), which has an unusually high precedence — even higher than negation ($\neg$). Implication is denoted by a long right arrow ($\longrightarrow$), and is distinct from meta-level implication ($\Longrightarrow$). Boolean terms are also used in the condition of *if ... then ... else ...* expressions, which are similar to those of Standard ML, and other functional programming languages.

Higher order properties of the logic are established by *reflection* between formulae and Boolean terms, meaning that they are isomorphic [Pau90]. Universally ($\forall$) and existentially ($\exists$) quantified expressions are also of type *bool*.

Available inference rules include reflexivity, substitution, modus ponens (elimination of object-level implication) and an introduction rule for implication. The Hilbert description operator ($\varepsilon\ x.\ P\ x$) is specified, although it is not used directly in this thesis. However, its presence ensures that the logic is classical, and standard theorems of classical logic are available as derived rules.

Natural numbers The natural numbers are represented by the type *nat*. Values of this type are denoted by *0* and *Suc n* for some natural number n [1]. A natural number can also be represented using the decimal notation for natural numbers (that is, Arabic numerals, but no sign or decimal separator). Addition ($+$), subtraction (-) and multiplication ($*$) are defined [2].

For natural numbers, *0 - n = 0* is a theorem. Tactics such as `arith` can be used to prove many properties of arithmetic. However, this class of arithmetic is undecidable in general.

Pairs and Tuples Tuples of values are represented by nesting elements of a more primitive 'pair' type. Pairs associate to the right, and hence the tuple *(0, 1, 2)* is an abbreviation for *(0, (1, 2))*. Elements in a pair can be different types. They can be extracted from the pair using the functions *fst* and *snd*, for the left and right element respectively.

Lists Finite lists of values can be represented using the *list* type in Isabelle/HOL. The list type is parametrised by a single type variable, representing the type of elements in the list. Thus, all elements within a list must be of the same type.

Literal list values are denoted by square brackets containing a comma-delimited list of terms representing list elements. The list may be of arbitrary (but finite) length, and may contain no elements. The empty list is denoted by [] or by *Nil*, although the latter notation is usually only seen when performing case distinction in proofs (such as in the base case for list induction).

[1] Occasionally *Zero* appears in proofs, although this is used for distinguishing cases and is not a constant of the logic.

[2] Division, the modulus operator and the 'divides by' relation are also available, but not used here.

The *Cons* operator can also be denoted by one of two infix notations: the notation for consing a term x with a list xs which has elements of the same type as x is $x{\cdot}xs$. List can also be appended to other lists. The notation for appending a list ys to a list xs is $xs@ys$.

The standard *hd* and *tl* functions decompose the consed list into its constituent head and tail. The *hd* function is underspecified for the empty list: it returns an arbitrary element of the correct type.

Elements within a list can be accessed by their index within the list. This avoids repeated explicit use of the *hd* function. The n'th element of a list xs can be accessed by $xs_{[n]}$ or by $xs!n$. These notations are interchangeable.

Other functions on lists defined by Isabelle/HOL include higher order functions *map*, *zip* and *foldl*. The *map* function has the type $(\alpha{\Rightarrow}\beta){\Rightarrow}(\alpha\ list{\Rightarrow}\beta\ list)$, where α and β are type variables, and is used to apply a function to all elements in a list of type α.

The *zip* function $\alpha\ list{\Rightarrow}\beta\ list{\Rightarrow}(\alpha\ list{\times}\beta\ list)$ constructs a list of pairs, where the left and right elements in the n'th pair are the n'th elements of the first and second lists arguments respectively. In this thesis, the first and second list arguments always have the same number of elements.

The *foldl* function provides iteration over lists. It applies a given two argument function for each element in the list. Its type is $(\alpha{\Rightarrow}\beta{\Rightarrow}\alpha){\Rightarrow}\alpha{\Rightarrow}\beta\ \text{list}{\Rightarrow}\alpha$, and is defined by recursion on the list:
$foldl\ f\ a\ [] = a$
$foldl\ f\ a\ (x{\cdot}xs) = foldl\ f\ (f\ a\ x)\ xs$

The given function f is first applied to an initial value $a{:}\alpha$, and the element at the head of the list. The function f is then applied to the subsequent elements in the list, except that each time, the result of the previous application of f is used instead of the initial value.

The *foldl* function applies f to list elements from left to right (head to tail). Isabelle/HOL also defines a similar function *foldr*, which applies the function from right to left (tail to head).

In this thesis, list iteration is used to define functions on lists that associate the names of signals in a circuit with their values. Signal names in a circuit are assumed to be unique at a given level of abstraction. These functions can be defined using either *foldl* or *foldr*: the choice is rather arbitrary here, because of the assumption that signal values are unique, and the values of f used. Only function *foldl* is actually used here [3].

Strings Isabelle/HOL defines a string type as a list of characters. Characters are, in turn, defined by a pair of 'nibbles', each of which may have one of sixteen values. Thus, strings are represented using a single octet character encoding. Values of the string type are denoted here in a sans-serif typeface to distinguish them from identifiers: for example the string containing the single character

[3] Considering the order of elements in the list raises the question of whether the a list is the most appropriate data type for this purpose. However, the use of list iteration functions provides for executable semantic functions which aids the validation of the semantics.

'A' is written A. This notation was introduced because it is more compact than the standard Isabelle notation $''A''$.

Bit values Bit values (or signal values) within a circuit are considered to be distinct from the Boolean values. This is in contrast to the majority of the hardware verification work performed using the HOL system, in which bit values are represented as Boolean values.

Bit values are denoted by the terms **0** and **1**. The operators on bit values appear similar to the corresponding Boolean operators, but have a subscript to denote the type. They include bit-wise 'And' ($\wedge_b$), bit-wise 'Or' ($\vee_b$), bit-wise 'Exclusive Or' ($\oplus_b$), and the unary operator bit-wise 'Not' ($\neg_b$).

The *bit* type is defined in the *Word* section of the Supplemental Isabelle/HOL Library [Ska05]. A word is represented by a *bit list*. For convenience, words are interpreted as natural numbers. This is sufficient for the arithmetic used here. The left-most bit is considered the most significant: **10** represents the natural number two.

The library provides a theory about the correspondence between bit values and the numbers that they represent. As a short-hand notation, the natural number represented by a bit list bs is written $\ll bs \gg$.

Defining New Types

Type Synonyms In Isabelle/HOL, new types are defined as some, possibly improper, subset of an existing type (although axiomatic definitions are possible, that approach is subject to the usual caveats with respect to ensuring soundness). Given that all types must be inhabited, this excludes the most trivial subset: the empty set. The simplest type definition is a *type synonym*, which gives another name to an existing type. Therefore, the new type name refers to a type with exactly the same members as the original type.

A type synonym is a 'shallow' type definition, in the sense that the two type names are interchangable. Internally, the distinction between the two synonyms is often lost, and the Isabelle system usually presents the name of the original 'super'-type back to the user. The principle advantage of using type synonyms is to improve the readability of theories and proof scripts.

Type synonyms are useful whenever an existing type has the properties required to model a specific concept in a theory, and is sufficient for that purpose. For example, in order to model an abstraction of time in a hardware circuit, a type that has a well-founded order is often appropriate, where the least element can be used to represent the time that the circuit was reset. Thus, the natural numbers are often sufficient for modelling time for hardware modelling, and a new type *time* can be defined as being synonymous with the natural numbers using the **types** keyword in Isabelle/HOL:

```
types
  time = nat
```

Datatypes Datatypes can be defined to represent a class of values where each value in that class belongs to one, and only one, of a finite and fixed number

of sub-classes. Each sub-class is identified by a *datatype constructor* - conventionally an identifier with an upper case initial character. They are similar to the `datatype` construct in Standard ML, and other functional programming languages. The *bit* type is defined thusly, using a datatype declaration:
datatype *bit* =
Zero (**0**) |
One (**1**)

The type of values in each sub-class can be different. Datatypes can be parametrised with type variables, and the type of a sub-class can be defined in terms of the type parameters.

Mutually recursive datatypes are allowed, and must be declared simultaneously: declarations of mutually recursive datatypes occur in the same top-level **datatype** clause, and are separated by the **and** keyword.

Functions on datatype values can be checked for primitive recursion by ensuring that the datatype constructor of one of its arguments has been 'removed', before each recursive function application using that argument. Primitive recursive functions usually have a definition for each datatype constructor. They can also be underspecified by omitting a definition the definition for a datatype constructor, as is the case with the list function *hd*. In this case, the function behaves arbitrarily on values for which no defining clause was given. Functions on mutually recursive datatypes are also defined simultaneously.

Case distinction between datatype constructors need not always be performed on function arguments. The **case** ... **of** ... construct can be used within an expression to distinguish between datatype constructors, and define a result value for the whole expression based on the datatype constructor of a value. Case expressions are similar to the `case` construct in Standard ML: clauses for each constructor are separated by a vertical bar (|), and patterns containing the constructor are separated from the result expression with a function arrow (⇒).

Optional Values and Maps A datatype that is predefined in Isabelle/HOL is *option*, which can be used to add a 'distinguished element' to an existing type [NPW02]. The *option* datatype has two constructors: *Some 'v* and *None*. The *Some* constructor can be used to denote a value of the polymorphic type *'v*, and the *None* constructor can be used to denote the absence of such a value.

A function returning an *option* can be used to model maps in Isabelle/HOL. In this context, *None* is used to denote a value for which the map has no value. The Isabelle/HOL syntax for a map from type *'a* to type *'b* is *'a⇀'b*.

Records Isabelle/HOL provides a means for defining *records*. Records can be considered as tuples, where each component (known as a field) is identified by a name, rather than by its position in the tuple. Hence, they are similar to records in high-level programming languages. Record types are introduced using the **record** keyword, and declare a name and type for each field:
record *r_t* =
field1 :: *type1*
field2 :: *type2*

Once a record type has been declared, values of that type can be declared. For example, a value belonging to the record type above can be declared using the form $r \equiv (\!|\ \mathit{field1}{=}\mathit{val1},\ \mathit{field2}{=}\mathit{val2}\ |\!)$, assuming the values *val1* and *val2* are of type *type1* and *type2* respectively.

Field selectors for record values are functions on record values that return a value of the type of the corresponding field. The field selector has the same name as the field itself. Using the examples above, *field1 r* reduces to *val1*.

The record syntax of Isabelle/HOL is adapted to include an operator for updating records, μ. The operator is applied to a record value, a field selector and a value of the appropriate type for that field. The expression $\mu(r,\ f{\mapsto}v)$ denotes the record r updated at f with the new value v. This syntax differs from the Isabelle convention of $r(f{:=}v)$.

3.2 Representation of key concepts

Comparing the behaviour of a conventional program to that of a program which makes use of a custom function unit is a complex task if it is not easy to relate the semantic objects in the software language definition, the semantic objects in the hardware language definition, and the model of their interface. For example, consider that it might be desirable to model machine words as natural numbers in the software semantics, perhaps to enjoy the benefits of reasoning about natural numbers when considering indexed memory access. On the other hand, in the hardware model it may be more appropriate to represent a machine word as a bit vector, an abstraction closer to those provided in hardware netlist languages.

It is clear that there may be a disparity between the preferred way of modelling a concept in the software semantics, and that of the same concept in the hardware semantics. For each such disparity, there are two approaches to ensuring that it still be possible to reason about those models. The first approach is to use a common representation for each concept (such as bit vectors in the example above), and attempt to 'make do' with a less natural representation. The second approach is to use the most natural representation within each model, and attempt to resolve the differences between models when relating them. Thus there is some trade-off in the modelling: using a common representation makes relating the hardware and software models simpler at the cost of greater difficultly in reasoning about the individual models. Likewise, using the most natural representation in each model makes reasoning about it or using it easier, while making relating models more complex.

This section reviews the different approaches to representing two of the most important types of semantic objects in a compiler intermediate representation: addressable memory and machine words.

3.2.1 Memory representation

Addressable memory refers to random access memory, where the value at a location specified by a given offset may be stored or retrieved. Thus, the representation chosen for it affects the formal model of the stack, the heap,

and global data areas. The program itself is also stored in addressable memory: branch instructions can specify the address of the next instruction to execute. In the context of a von Neumann architecture, all of these memory areas are regions in a single addressable memory space.

Memory Regions

Each of these memory areas can also be considered as being divided further into memory regions. An example of a type of region within an area of memory is a branch table (or jump table). These can be stored within a global memory area, and can be addressed using the index of the required value as an offset to the base address of that jump table within the memory map. Such a region is independent from other memory regions: it does not make sense to access the region using any other base address.

Certain stack frames have a similar property, depending on how the function that created the stack frame has been compiled. If the size of the stack frame is not modified throughout the execution of a function, then it will have fixed size. Local variables in that function can be addressed via a fixed offset from the stack pointer. If that is the only way that local variables are addressed in a given function, then it can also be considered as an independent region in the sense used above.

Identification of independent memory regions is an important consideration when designing hardware/software compilers for systems where the custom function unit does not have direct high-speed access to a memory interface. In these systems, there is greater overhead in transferring data to the custom function unit, and performance improvements are achieved by pre-loading the function unit with the required data [BL00]. Transferring the data to the custom function unit, must be initiated explicitly by the program where the function unit does not have direct access to the program memory.

Independent regions of memory may be appropriate candidates for transfer to the custom function unit. This is likely to be feasible only when the region is small enough to be stored in the limited memory resources of a function unit, and when the time required to transfer data is less than the execution time saved by using a function unit.

One complexity that arises when considering stack frames is that they typically include pointers into frames higher up in the stack. However, while traditional compilers can assume that the values on the stack are stored in a linear and contiguous memory region, this assumption is not valid when certain local variables have been transferred to the custom function unit.

Where a compilation targets a custom function unit, the local variables for a given function (or group of functions) may be transferred to the function unit, and any pointers transferred to the function unit may not be valid in any memory region attached to the function unit. Thus care must be taken with values that represent pointers to memory locations, and values that are used as offsets to an index into a memory region.

The problem of pointers within the stack can be generalised to a more fundamental problem: the need to consider the type of the values that are manipu-

lated by a program. As a result, it is appropriate to consider modelling types of intermediate values used within the computation, in addition to modelling the actual values used.

Modelling Addressable Memory

Memory regions can be modelled in several ways. In order to consider which models are appropriate, it is necessary to consider the properties required in a model of memory regions. It is also necessary to consider the target architecture in order to avoid a mismatch between the model and the architecture.

One property of memory regions is that they must be able to model arrays and branch tables. For this, the model (and hence the underlying architecture) must support access using indexed addressing: that is, addresses can be constructed by adding an offset to a base address. Memory access via an 'opaque' token may be sufficient for addressing local scalar variables, but in general, is inadequate for modelling the semantics of intermediate representations that are typically used by compilers.

The simplest approach to providing indexed addressing is to use natural numbers to represent memory locations. This allows an index to be the result of an arbitrary arithmetic expression. This is the approach adopted in the languages developed in this thesis.

A memory region could be modelled as a function from such locations to the individual values stored at each location. Alternatively, it could be modelled as a list of values where the index of each element corresponds to the memory address.

Regardless of which representation is used, it should be noted the valid addresses within a memory area may not be contiguous. For example, in a system where the stack grows downwards, and the heap grows upwards, there may be a 'gap' in the memory map. Attempting to use addresses within this gap may result in a segmentation fault, which in turn, could cause a program to be terminated by the operating system. Alternatively, the target system may not have enough memory such that there is a memory location for every possible data value that can be computed as an index.

Furthermore, it is should be considered that in general, memory regions in a program are likely to contain uninitialised values. For example, values that can be observed within the current stack frame could have been stored by a function that was called prior to that currently being executed. Thus not only will some addresses refer to invalid areas of memory, others will refer to uninitialised areas of memory.

If a memory region is modelled as a function from locations to values, it is necessary to consider whether it be a total function or a partial function. Given that the memory map for a program is not contiguous, and that locations may not have been initialised, intuitively it may seem appropriate that a partial function be used to model the memory map.

Being a logic of total functions, it is not possible to define partial functions directly in Isabelle/HOL. Instead, they can be represented as a total function that returns an optional value of type α *option*, where *option* is a datatype

parametrised on a single type α, with constructors *Some* α and *None*, and where α represents the range of the partial function being represented. In Isabelle/HOL, this is referred to as a map. Where the partial function being represented evaluates to a value v for a given argument, the Isabelle/HOL representation of that function evaluates to *Some* v. Where the partial function is not defined for a given argument, the Isabelle/HOL representation of that function returns *None*.

Alternatively, a finite map could be used instead of a partial function, because the domain includes only a finite number of locations. Wildmoser provides a definition of finite maps based on lists in Isabelle/HOL [WN04]. One of the features of the definition is that it supports the generation of executable Standard ML code.

The purpose of Wildmoser's definition is to support the verification of particular programs, rather than the purpose of this work: the consideration of the properties of a compilation algorithm. As such, Wildmoser's definition is unnecessarily complex for the purpose here.

Instead, it suffices to use a total function for modelling memory here. The reasons for this are as follows. Firstly, the ability to execute them efficiently within a theorem proving environment is of limited value, because the aim here is not to verify individual programs. Thus the use of the existing finite map theories for code generation to Standard ML merely adds unnecessary complexity.

Secondly, this work makes various assumptions about the memory safety of programs to be compiled for an architecture with a custom function unit. Specifically, it is assumed that programs do not use self-modifying (or self-examining) techniques that could affect the separating out parts of the program that can be executed in a custom function unit. These properties are similar to the property that a program does not use uninitialised or attempt to access invalid memory locations, and establishing that this class of properties holds for a given problem is beyond the scope of this thesis. This means that all memory accesses are assumed to be valid here.

Under these assumptions, total functions are used to model the memory space of a program in this thesis. This is consistent with existing literature that uses Isabelle/HOL to relate the semantics of different programming language definitions [Nip98].

Approaches to the problem of establishing memory safety properties for a given program, or for all programs that have certain static properties, can be found in literature that describes proof-carrying code [Nec97, WNKN04, App01] and typed intermediate languages [SA01, XH01, Cra03].

3.2.2 Number representation

The type of semantic object used to model the memory map of a program was discussed in the previous section, and the use of a total function to represent it was considered to be adequate for the purposes here. Furthermore, it noted that, because of the requirement for address arithmetic, natural numbers form a suitable domain for that function.

The purpose of this section is twofold. Firstly, it identifies several alternative representations for numeric values, including those that are appropriate for use as the domain of the memory map. Secondly, it considers a suitable range for the function representing the memory map.

There is, of course, some overlap between these two concepts. The range of the memory map function represents the values that can be stored by a program. It should be clear that, in general, some of the values stored by a program will be numeric values, and that some of those values will represent memory locations.

The sets of values used to represent locations and values form part of the language definitions used in the specification of a compilation algorithm. Where the target of the compilation algorithm is a low-level language, perhaps with a similar level of abstraction to assembly language, these values typically model machine words in a physical system.

Machine words can represent only a finite set of values, due to their representation as a *bit vector*: an ordered set of bit values of fixed length. When producing a language definition, it is necessary to consider whether locations and values will be modelled by a finite type, or approximated by a type with an infinite number of values. The use of approximations in a formal treatment of a language definition can allow larger verification problems to be tackled, at the cost of having a definition that is not 'fully formal' [ST99].

The need to address this issue arises from the requirement to specify the semantics of operators, such as assembly language instructions, that manipulate machine values. Were it not for this requirement, it would be sufficient to represent machine words using an abstract type, and the semantic functions for primitive operators could be omitted from the definition. Nipkow provides some examples of the use of an abstract type to represent values in language definitions defined in the Isabelle/HOL logic, where the semantics of the operators on values are not defined [Nip98, NPW02].

Instead, it is necessary to define a type for machine words that is not abstract. In the Isabelle/HOL framework, it is conventional to do so by defining a subtype of an existing type [NPW02, §8.5]. Where machine words are represented using a finite type, they can be represented as a finite subset of the natural numbers or integers.

Rauch and Wolff describe two different approaches to the formalisation of machine words that store integer values that are interpreted using the twos-complement system [RW03]. Each assumes that machine words of length n represent values between -2^{n-1} and $2^{n-1}-1$, denoted here by $\mathbb{Z}_{min}$ and $\mathbb{Z}_{max}$ respectively.

In one approach, termed the *partial approach*, operators on machine words are partially defined: an operator is defined if and only if the result of the operation can be represented by a machine word. In this approach, the expression $\mathbb{Z}_{max}+1$ is undefined.

In the other approach, the *wrap-around approach*, operators are defined such that their behaviour more accurately represents the usual machine implementation of those operators. In this approach, $\mathbb{Z}_{max}+1=\mathbb{Z}_{min}$.

An alternative to the use of a finite subset of numbers is to represent machine words as a list of bit values [Mel93]. Additionally, machine words can be represented as a sub-type of bit value lists. In this case, the sub-type includes only bit value lists of the same length as a machine word. In Isabelle/HOL, this type can be denoted by $m\_word = \{x :: bit\ list.\ length\ x = n\}$, where n is the length of a machine word. The wrap-around approach is appropriate when using bit lists to represent machine words [RW03].

It is more convenient to define some operators on machine words in terms of their bit list representation, rather than in terms of the integer that it represents. For example, an operator that performs an *n+1* bit rotation on a register and the carry flag can be defined on a register *reg* represented by a bit list, and a carry flag represented by a single bit as follows:

$$rotateLeft\ reg\ carry \equiv (tl\ reg\ @\ [carry],\ hd\ reg)$$

A definition based upon the numerical interpretation of the value stored in the register would vary depending upon whether a twos-complement or unsigned interpretation was being used. Furthermore, it would be less clear that the definition represented a bit rotation because of the need to use arithmetic operators on the numeric value.

When defining semantic functions that operate on bit lists representing machine words, it is appropriate to ensure that words produced by semantic functions are the correct length, in order to ensure that the intuition underlying the semantics is faithfully represented.

The means for ensuring that semantic functions on machine words produce machine words of the correct length vary according to the formal framework being used. For example, using the Vienna Development Methodology [Jon90b], a machine word type could be defined as a bit list with invariant length. It would then be necessary to discharge a series of proof obligations in order to show that semantic functions that return machine words respect that invariant.

In formal frameworks that provide a restricted form of dependent types [Pie02], ensuring the correct length of a list can be reduced to type checking. In this approach, the type of a list can be an indexed type, where the index corresponds to the length of that list. Furthermore, the types that appear in the signatures of functions on lists include variables denoting the length of lists upon which they operate.

For example, the *hd* function can be defined with the type $\Pi n::\mathbb{N}_1.bit\ list(n) \Rightarrow bit$, where $\mathbb{N}_1$ denotes the non-zero natural numbers. The type indicates that the function operates on lists of any length *n*. In a similar manner, *tl* can be defined with the type $\Pi n::\mathbb{N}_1.bit\ list(n) \Rightarrow bit\ list(n\text{-}1)$, indicating that the resulting list from *tl* is one element shorter than the argument. Thus the task of ensuring that the *rotateLeft* function above returns a machine word of the correct length is reduced to type checking.

The Isabelle/HOL system does not support dependent types. The methodology for verifying the validity of machine words produced by semantic functions is less obvious in this framework, particularly where a separate type (such as *m_word*, rather than simply a *bit list* of fixed length) is used to represent machine words.

However, Paulson briefly mentions a useful relationship between dependent types and existential quantification [Pau90, §5]. An expression with dependent types can be rewritten as a proposition that a witness value exists that satisfies the properties of the dependent type.

Using the definition of the *m_word* type on the preceding page, the Isabelle system defines two constants, namely *Rep_m_word* and *Abs_m_word* (cf. [NPW02, §8.5]. *Rep_m_word* is a function of type *m_word⇒bit list*, and is specified for all values of type *m_word*. Its range is a subset of all bit lists: namely those of length *n*, where *n* is that used in the definition of *m_word*.

The *Abs_m_word* function is of type *bit list⇒m_word*. Where its argument is a bit list within the sub-type defined by the *m_word* type definition, *Abs_m_word* simply returns the corresponding value from that type. Its behaviour is under-specified on arguments that are not also a member of the sub-type.

The Isabelle system defines properties that characterise this relationship between *Abs_m_word* and *Rep_m_word*, namely that machine words are isomorphic with their representation as a bit list. Specifically, given that *m_word* represents the subset of bit lists of the same length as machine words, it asserts the following as theorems:

1. $Abs\_m\_word\ (Rep\_m\_word\ m\_w) = m\_w$
2. $bs \in m\_word \Rightarrow Rep\_m\_word\ (Abs\_m\_word\ bs) = bs$
3. $Rep\_m\_word\ m\_w \in m\_word$

The assumption in the second theorem notes that the isomorphism exists only between machine words and bit lists for bit lists that are of the correct length. Thus, where *Abs_m_word* is used to convert a semantic object of the bit list type into a machine word, it is useful to ensure that the bit list is always of the correct length.

In summary, although it is not possible to verify bit list length properties in Isabelle/HOL using type checking alone, such properties can still be verified using the theorem prover. One approach to this is to define semantic functions in terms of bit lists, rather than on machine words, and verify the bit level properties of those functions as auxiliary lemmas, before using the *Abs_m_word* function to create semantic objects of the appropriate type.

3.3 Representation and Reasoning about Hardware

This section considers how hardware logic can be represented by expressions in higher-order logic, and more specifically, the Isabelle/HOL framework which was introduced in Section 3.1.2. It also describes briefly how it is possible to use formal reasoning to analyse the behaviour of circuits represented by higher-order logic expressions.

The concepts introduced in this section form the basis of the definition of a structural netlist representation, which is introduced in Chapter 5 and further developed in Chapter 6. These chapters provide a formal semantics for the netlist representation, and are given in Isabelle/HOL. Thus, this thesis does not consider how to target commonly used HDLs, such as VHDL or Verilog, for which there are existing EDA synthesis tools.

Section 3.3.1, *Hardware Representations* discusses the rationale for using a new representation in this thesis, in favour of using an existing HDL to represent hardware. Section 3.3.2, *Representing hardware in HOL* describes how higher order logic can be used to model hardware behaviour.

3.3.1 Hardware Representations

When designing a hardware/software compiler it is necessary to select which language will be targeted for describing the hardware logic design for the RFU. As the logic design produced by such a compiler is to be implemented using reconfigurable hardware, it is appropriate to represent that design in an HDL that can be translated into a configuration for the targeted reconfigurable logic device. The generated design could be synthesisable, for example, expressed in a synthesisable subset of an HDL, or it could be a structural description of the required design.

An obvious approach for a hardware/software compiler would be to target an existing HDL, as supported directly by existing synthesis tools for reconfigurable logic. The hardware/software toolchain can then convert the logic design generated by the compiler into a reconfigurable hardware configuration using existing EDA tools, just as it converts the assembly language generated by the compiler into object code using an assembler.

The use of an existing reconfigurable logic toolchain would likely greatly reduce the cost of the development of a new hardware/software compilation toolchain. However, in the context of this thesis, the cost or convenience of compiler implementation represents only one factor in the selection and design of hardware representation.

In order to specify the correctness of a compilation algorithm precisely, a formal semantics of each language or representation used by the compiler is required.

Formal Semantics of Existing HDLs

Typically, HDLs supported by reconfigurable hardware synthesis tools do not have a formally defined semantics. Such HDLs are thus unsuited to the purpose here: the consideration of the correctness criteria of a hardware/software compiler.

However, to state that commonly used HDLs have no defined semantics would be inaccurate. For example, the simulation semantics of VHDL and Verilog — two HDLs commonly supported by tool sets for FPGAs — are described in their respective language manuals [IEE02, IEE01] albeit in natural language, rather than a form conducive to formal reasoning. Furthermore, the mapping between language constructs in these HDLs and hardware designs is described in the synthesis manual for each language [IEE04b, IEE04a].

There have been various attempts to retrospectively construct a formal semantics of these two languages. For VHDL, these include an operational semantics for a zero-delay subset [vT95]; a process algebraic approach [BM95], and an embedding in the Nqthm theorem proving system by Boyer and Moore, including a LISP implementation [Rus94].

A number of attempts to provide a tractable semantics for Verilog have also been made. In 1995, Mike Gordon identified Verilog as "a relatively simple real-world language in need of theoretical support", and provided an outline of how a formal semantics for a simplified version of Verilog, which he named *V*, might be defined [Gor95].

Schneider and Xu later described a formal semantics for a subset of Verilog that they called V^-, in recognition of Gordon's definition [SX98]. Their semantics were based on Duration Calculus, extended to allow reasoning about events with zero duration and the causality between them, and about behaviours over non-finite time intervals.

An operational semantics of a subset of Verilog was later developed, having been prototyped in the logic programming language Prolog [Bow99, BHX00]. The Prolog implementation was used in the validation of the semantics, as it provided an executable model in which the behaviour of a hardware design under those semantics could be determined mechanically. In particular, the use of Prolog allowed exploration of the different behaviours that might arise due to the non-determinism present in the semantics.

Correctness in Hardware Synthesis

The semantics of VHDL and Verilog as defined in their respective language reference manuals are 'simulation semantics', meaning that they specify an abstract interpreter for those languages. The formal semantics that have been defined retrospectively for these languages also include the notion of a simulation cycle.

An alternative to defining the semantics of HDLs in terms of a simulation cycle is to represent hardware designs directly as expressions within a formal system, such as higher order logic. This technique is the subject of Section 3.3.2.

The extent to which semantic definitions based on a simulation cycle model are conducive to formal reasoning about hardware designs appears to be rather limited. At least, such approaches appear limited when compared with the formal reasoning admitted by techniques that represent hardware designs directly as expressions within a logic. For example, Gordon's HOL system has been used to model and verify a microprocessor architectures at different levels of abstraction [Fox02, Fox01a]. However, formal reasoning using models of the simulation semantics for VHDL and Verilog appears to have been conducted only on much simpler hardware designs [KB95a].

Furthermore, there do not appear to be any results that demonstrate that the behaviour of a hardware design according to those formal semantics is consistent with the behaviour of a lower level hardware design derived from the transformations described in the synthesis manual for those languages. That is, whether the behaviour according to the formal semantics commutes with respect to both the transformations described in the synthesis manual for each language, and a model of the primitives targeted by the those transformations. Therefore, even where a formal semantics exists, it may not be clear that the design will implement the modelled behaviour when synthesised according to the rules in the synthesis manuals.

The fact that the majority of existing tools for producing configurations for reconfigurable hardware use languages with no formal semantics cannot be overlooked. Even if a hardware/software compiler were developed and shown to be correct, the process of generating a logic design for the function unit is still susceptible to any flaws elsewhere in the toolchain.

However, this problem can be mitigated by reducing the assumptions made of other components in the toolchain. For example, if a hardware/software compiler generates a netlist-level description of a logic design, then it is no longer necessary to rely on synthesis tools that generate a netlist-level representation from high-level HDLs — the bugs in the synthesis tool cannot manifest if the tool is never invoked.

Summary of Hardware Representations

In this section, a number of requirements have been identified for a hardware representation to be used for formal reasoning about hardware/software compilation. To summarise, with only limited correctness guarantees for existing synthesis tools, a representation for this purpose should:

- be sufficiently expressive that it can be used for representing logic designs for function units;
- have a formal semantics suited to reasoning about the correctness of designs;
- have both abstract representation (for manipulation in the compiler) and a concrete representation (which forms the input to subsequent tools in the toolchain); and
- have a relatively low-level representation, that can be translated in a straight-forward manner into a format suitable input for tools that map designs onto a targeted technology.

To elaborate on the third requirement: just as manipulating an assembly language representation of a program within the compiler would be inappropriate for compilation into software, it would be inappropriate to use a conventional HDL as an intermediate representation for logic designs. Representations based on text-based representations would require parsing during each compiler phase. Representations based on abstract data types allow the program representation to be annotated with information that is derived from pragmas and analyses, and with information that supports optimisations.

The last requirement listed avoids relying on a correct high level synthesis tool for hardware. This raises the question of whether one should assume a correct high level synthesis tool when developing a hardware/software compiler, and then proceed to applying similar techniques to the development of the synthesis tool. However, this only shifts most of the proof burden into a separate tool: at some point it is necessary to address the problem of reducing a high level representation into a low level representation.

A netlist level hardware representation is presented in Chapter 5 and Chapter 6, and is intended to satisfy each of these requirements. It is based on existing techniques for modelling and reasoning about the behaviour of hardware using higher order logic. These techniques are introduced in the next section.

3.3.2 Modelling Hardware in HOL

This section describes the use of higher order logic for representing and reasoning about hardware designs. It is intended to provide sufficient explanation that readers who are not familiar with the approach can follow the presentation of the netlist language in later chapters. For a more detailed account of this approach, the reader is referred to more thorough treatments [Gor86, Mel93].

Representing hardware in HOL

The use of higher order logic for specification and verification of hardware designs was originally proposed as part of the VERITAS project after previous approaches based on first order logic [HD86]. The description of the approach given here assumes slightly higher level circuit primitives: rather than considering a circuit at the transistor level, the primitives used here are logic gates. This is a more appropriate level of abstraction where the primary target technology is likely to be that of an FPGA.

In this approach, digital logic designs are represented directly as predicates within the logic. That is, components are represented by a HOL term of type *bool* — the Isabelle/HOL type for Boolean values. Free variables in the term represent the external signals of that component.

To illustrate the technique, its application to a logical And gate and Xor gate (illustrated in Figure 3.1) is demonstrated here.

Figure 3.1: *And* gate and *Xor* gate

The external signals of both components are labelled A and B and O. These become free variables in the HOL representation of each component. In Isabelle/HOL, a parametrised constant can be defined for each component. Each parameter becomes a schematic variable representing one of the external signals. These are logically free variables that can be instantiated later. Constants to represent the And gate and Xor gate can be defined as follows:

$$MAnd\ A\ B\ O \equiv A \wedge B = O \tag{3.1}$$

$$MXor\ A\ B\ O \equiv A \oplus B = O \tag{3.2}$$

The symbols $\wedge$ and $\oplus$ are infix Boolean functions with a truth table that corresponds to that of the logic gate that it is used to represent. The names *MAnd* and *MXor* are constants with an arity equal to the number of external signals, and their definitions (the right hand side of the structural equality) are the Boolean terms that represent the And and Xor gate respectively.

The intuition behind the representation is straight-forward: the Boolean values *True* and *False* correspond to the binary logic levels that signals in the circuit may take. The Boolean term on the right hand side of the structural equality has the value *True* if, and only if, the combination of signals represented by

its free variables can be simultaneously observed on the external signals of the logic gate.

For example, if either of the inputs A or B to the And gate is logical *False*, then the only way the Boolean expression *MAnd* can hold is if the value of the output, labelled O and represented by the Boolean variable O, is also *False*. Likewise, if the Boolean variable O is *True*, then the values of A and B must both be *True* in order to for the term *MAnd* to hold. In this respect, *MAnd* is *True* only when the values of A, B and O represent signals that may be simultaneously observed on the external pins of the And gate.

An important property to note is that, for these gate level primitives, the term used to represent the primitive in a circuit is identical to the term that specifies the behaviour of the primitive. In the context of the examples above, although the Boolean operators (which are simply HOL constants with an arity of two) are used to denote components within the circuit, the semantics of those operators also form the specification of the components that they denote. Thus, HOL terms can be used not only to represent circuits and components, but also to specify them.

The sections that follow show how more complex components can be represented. For more complex components, there may be a number of designs that satisfy the required behaviour of a component. In such designs, the representation of a circuit as a term in higher order logic is not necessarily identical to the term that represents a specification of its desired behaviour.

In practise, the specification of components is seldom the same as the representation of a component that satisfies that specification. The process of verification involves showing that the term that specifies a components behaviour is a logical entailment of the term that represents the structure of the component.

The variable names used to denote external signals are not significant: a HOL term representing a circuit can be modified to refer to a different set of external signals by consistently replacing each occurrence of a free variable with a new free variable.

Terms denoting circuits can be in either curried or uncurried form. In the examples above, they are shown in the curried form.

Composition

Circuits that consist of a number of components can be represented by composing the terms representing each component using conjunction. As an example, a term that represents a half-adder (illustrated in Figure 3.2) can be either be denoted as shown in Equation 3.3, or as shown in Equation 3.4.

$$MhalfAdd\ A\ B\ C\ S \equiv (A \wedge B = C) \wedge (A \oplus B = S) \tag{3.3}$$

$$MhalfAdd\ A\ B\ C\ S \equiv MAnd\ A\ B\ C \wedge MXor\ A\ B\ S \tag{3.4}$$

A term representing a number of components is *True* only when each of its conjuncts that represent a component is *True*. The underlying intuition is that the new term is *True* when the signals represented by all of the free variables in that term may be observed simultaneously in the represented circuit.

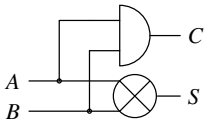


Figure 3.2: Half adder

By using the same variable names in the expression that represents the And gate as those in the expression that represents the Xor gate, the connections between those two components can be represented. Components that are connected to a common signal can be represented in HOL, by ensuring that the same variable name is used in the expressions that model those components.

Likewise, if two components are not directly connected, then the free variables in the HOL terms representing them can be modified as described above, until they contain no common free variables.

Thus far, the HOL *bool* type has been used to represent both signal values, components and their specification. It is important that components and their specifications have the type *bool*, because this is used as part of the verification process in which the specification is shown to be a logical entailment of the term representing the component,

However, there is no requirement that the type *bool* be used to represent signal values. The bit value type introduced on page 40 can be used to represent signal levels instead. In this case, the half adder can be represented as follows:

$$MhalfAdd\ A\ B\ C\ S \equiv (A \wedge_b B = C) \wedge (A \oplus_b B = S)$$

In this expression, A, B, C and S are all *bit* values, but the complete HOL term is still of type *bool*. If *MAnd* and *MXor* were defined in terms of bit values rather than Boolean values, then the definition given in Equation 3.4 would be equivalent to the above definition.

Using the syntax introduced on on page 40 to denote the integer value of a bit list v by $\ll v \gg$, the specification for the half adder can be denoted in terms of arithmetic:

$$halfAdd\ A\ B\ C\ \ S \equiv \ll A \gg\ +\ \ll B \gg = \ll [C,\ S] \gg$$

This states that the sum of A and B, which are both one bit values, is equal to the two bit output value where the carry bit C is the most significant bit. The verification condition for this component is then formulated as:

$$halfAdd\ A\ B\ C\ S \Longrightarrow MhalfAdd\ A\ B\ C\ S$$

Abstraction

Figure 3.3 shows an implementation of a full adder: its external behaviour is characterised by the possible combinations of values at the external pins — A, B, C_{in}, S and C — which represent two inputs to be added; a carry in value; the sum output and the resulting carry output respectively.

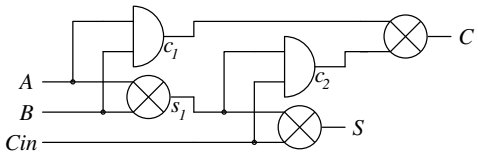


Figure 3.3: Full adder

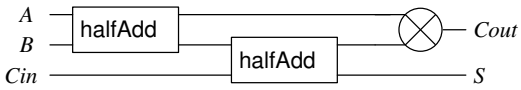


Figure 3.4: Full adder

The specification of a component should be described in terms of its external behaviour and without reference to its internal signals. In the full adder, the internal values — s_1, c_1 and c_2 — cannot be observed externally. Thus, when reusing the full adder, their values should not be of concern: any set of internal values is allowed provided the component satisfies its specification. A specification of the arithmetic properties of a full adder can be written as follows:

$$\mathit{fullAdd}\ A\ B\ C_{in}\ C\ S \equiv \ll A \gg\ +\ \ll B \gg\ +\ \ll C_{in} \gg\ =\ \ll C,\ S \gg$$

Since the specification of a component is expressed in terms of the free variables in the HOL term specifying that component, the internal values should not be visible as free variables in the term representing its structure. The following definition of a full adder shows how existential quantification is used to hide internal signals. Only the internal signals are bound by the quantifier, while the remaining terms are free.

$$\begin{aligned} \mathit{MfullAdd}\ A\ B\ C_{in}\ C\ S \equiv \exists\, & c_1,\ c_2,\ s\ . \\ & (A \wedge B = c_1) \wedge (A \oplus B = s_1)\ \wedge \\ & (C_{in} \wedge s_1 = c_2) \wedge (C_{in} \oplus s_1 = S)\ \wedge \\ & (c_1 \oplus c_2 = C_{out}) \end{aligned} \tag{3.5}$$

The above definition is more complex that it need be. It can be seen that the implementation of the full adder is simply two half adders and a further gate. A better definition would reuse the definition of the half adder, allowing the correctness of the full adder to be shown using lemmas about the correctness of the half adder implementation.

$$\begin{aligned} \mathit{MfullAdd}\ A\ B\ C_{in}\ C\ S \equiv \exists\, & c_1,\ c_2,\ s\ . \\ & \mathit{MhalfAdd}\ A\ B\ c_1\ s_1\ \wedge \\ & \mathit{MhalfAdd}\ C_{in}\ s_1\ c_2\ S\ \wedge \\ & \mathit{MXor}\ c_1\ c_2\ C_{out} \end{aligned} \tag{3.6}$$

This definition is illustrated by Figure 3.4.

The use of existential quantification can cause problems during circuit verification. The verification condition for a circuit with specification *Spec* and an

implementation *Imp* is $Imp \Longrightarrow Spec$. Where the HOL term includes existentially quantified terms to represent internal values, it is necessary to ensure that a set of witness values can be found that makes the antecedent true, and hence satisfy the implementation model.

If there no set of internal values that satisfies the model, then (since *Imp* is an existentially quantified term) the antecedent will be false, and the verification condition will become trivially true. This could lead to an incorrect hardware design apparently being verified as correct, unless care is taken to ensure that witness values can be found to satisfy the model.

Sequential logic

The discussion of the examples presented above has not addressed how to model the behaviour of a circuit over a period of time. It has been possible to avoid this issue by assuming the logic gates have zero delay, and by considering only combinational circuits that have no 'feedback' connections.

However, this class of circuits is more limited than circuits used to implement many digital logic designs, including processors and function units. Such designs typically use *sequential logic*, in order that the circuit can maintain some state information. This state can vary over time, and changes to the state are triggered by signal events. In synchronous designs, the rise or fall of a clock signal usually effects a change in state. The state may be maintained by reading and writing to a clocked register or memory. In asynchronous designs, a change in state can happen as a result of handshaking between components. Fortunately, the approach described thus far is easily extended for sequential logic in order to model the behaviour of circuits over time.

Time, being a linear property, is typically modelled as a natural number (as a *nat* in Isabelle/HOL) that is monotonically increasing. The granularity with which time is measured varies depending on the purpose of the model: in a model of a synchronous circuit, each time unit may represent one clock cycle. In this approach, combinational logic is assumed to have zero delay (or rather, the issue of signal timing is deferred until a 'place and route' process).

In a transistor level model intended for reasoning about the temporal order of signal events, each unit of time may represent a much smaller duration, perhaps smaller than the delay of a single transistor gate. Intuitively, this usage is similar to the femto-second time unit in VHDL: that is, it must be capable of finer granularity than other time units used in the model.

In order to model sequential logic, the behaviour of a signal over time can be modelled as a function of time: either as $nat \Rightarrow bool$ or as $nat \Rightarrow bit$. Where the granularity of time is sufficient to model gate delay, the behaviour of the And gate could be characterised by the following specification, in which A and B are of type $nat \Rightarrow bool$; n ranges over time, and δ represents the gate delay.

$$\forall n.\ A\ n \wedge B\ n = O\ (n+\delta)$$

Where the gate delay is ignored, and the granularity of time is the duration of a clock cycle, the definition given in Equation 3.1 can be applied, with the understanding that A, B and O now refer to the behaviours of each signal, rather than to their value directly.

Chapter 4

Intermediate Representation

Contents

Compilers may transform a program in a high-level language into one or more intermediate representations (IRs) before translating it into the target language, rather than translate the source program directly into the target language [Muc97]. The use of an IR allows the task of compilation to be decomposed into smaller stages. An IR may be considered as an 'abstract language' — a language that does not necessarily have a concrete syntax — which is used as the target language for at least one compilation stage and as the source language for the next stage.

One of the benefits of using an IR was described in Section 2.2; the use of IRs improves modularity, decoupling the front-end of a compiler from the back-end. This reduces the number of compilation algorithms that need to be implemented for retargetable compilers. This is particularly advantageous for hardware/software compilers, where the target architecture may change during the life-cycle of an application, for example, to use new components with greater reconfigurable logic capacity as they become available.

In order to provide a precise specification of each compiler stage, a formal definition of each IR is required. This chapter presents a formal definition of an IR, including its abstract syntax and semantics. It is designed such that a fragment of the program in the representation may be passed on to either a compiler's software back-end, or to the hardware back-end. This means that it must be sufficiently flexible that it be possible to generate code for

any part of a compilation unit expressed in the representation, and that it be possible to generate a logic design for those parts that are to be implemented in reconfigurable logic.

This chapter presents a formal definition of an intermediate representation, including its abstract syntax and semantics. It is designed such that a fragment of a program in the representation may be passed on to either the compiler's code generation backend, or to a backend that generates hardware designs. More specifically, the intention is that it be sufficiently flexible that any part of a compilation unit expressed in the representation may be passed to the code generator, but there is no requirement that it be possible to generate a hardware design for an arbitrary compilation unit.

In Section 4.1, *Requirements for a Hardware/Software IR*, the requirements for an intermediate representation appropriate for reasoning about retargetable hardware/software compilation are discussed. Section 4.2, *Analysis of Existing Representations* describes two existing IRs that satisfy an approximation of these requirements. These two representations form the basis of the formal definition developed here, which is presented in Section 4.3, *Formal Definition of a Hardware/Software IR.*

4.1 Requirements for a Hardware/Software IR

In designing an IR, it is appropriate to ensure that it be usable at some stage in compilation: an IR is useless if it cannot be targeted by a parser or existing compilation stage. Furthermore, the design of an IR has implications for the specification of a compilation stage in which it is used. In other words, in designing an IR to be used as the source of a compilation stage that is intended to be formally verified, it is necessary to ensure that it be possible to relate the semantics of that IR to that of the target of the compilation stage.

A similar requirement is that IRs intended for initial compilation stages have semantics that can be related to the source language, and IRs intended for late compilation stages have semantics that can be related to a model of the target architecture. These requirements are, of course, in addition to those related to the analyses or optimisations that the IR is intended to support. In this section, the requirements specific to IRs for hardware/software compilation are considered.

4.1.1 Representation of Fine-Grained Parallelism

Techniques for hardware/software acceleration may be broadly classified into two categories [BD02]. One approach is to develop hardware acceleration for specific source language constructs (such as 'case' statements). The other approach is to use data-flow analysis to group operations that could be combined and implemented in a function unit. The latter approach receives the focus here.

Using data-flow based techniques, operations that can be either combined, or executed in parallel may be grouped together and compiled for a reconfigurable function unit. In each case, the use of the function unit to execute those opera-

tions can provide hardware acceleration. In order to group together operations that may be executed in parallel, it is necessary to consider IRs that permit analyses required to identify such parallelism, and also IRs in which that parallelism may be represented explicitly.

IRs in which parallelism may be represented include the Value Dependence Graph [WCES94]; Dependence Flow Graphs [PBJ+91]; Static Single Assignment, and variations including Gated Single Assignment [OBM90], Thinned Gated Single-Assignment (TGSA) [Hav93] and Predicated Static Single Assignment (PSSA) [CSC+99].

4.1.2 Flexibility for Hardware/Software Compilation

An ideal IR is sufficiently flexible that it can be used to cover a wide variety of target architectures from a variety of source languages. In the context of hardware/software compilers, the required flexibility is that it be possible to generate object code and a hardware design that can be used for a reconfigurable function unit. Furthermore, the IR should enable an improvement in the performance of compiled programs relative to that achieved by compiling to a similar architecture without reconfigurable logic.

An IR for conventional hardware architectures may be sufficiently flexible that it can be used to target architectures that differ in a number of ways, including instruction set complexity (both RISC and CISC); capability (such as the availability of floating point hardware); supported memory addressing techniques and machine word size [Sta02]. Using such an IR, many intermediate compilation stages can be described independently of these variations, deferring these problems to a machine-specific translation compilation stage.

In designing and evaluating an IR for retargetable hardware/software compilation, the flexibility of the IR with respect to all the above variations in target architectures should be considered. However, additional design parameters are introduced when reconfigurable logic is used, and it is appropriate to evaluate the extent to which an IR provides abstractions that support retargetability for hardware/software compilation. These design parameters include those related to the capacity and type of reconfigurable logic used, and also the way in which it is integrated into the architecture. Various options for these design parameters were discussed in Section 2.3.1.

4.2 Analysis of Existing Representations

This section describes two existing intermediate representations. Both IRs allow the explicit representation of fine-grained parallelism. However, neither appears to satisfy all of the requirements for an IR for retargetable hardware/software compilation identified in the previous section. Nevertheless, both satisfy many of the identified requirements, and the two representations form the basis of the IR that is developed in Section 4.3.

The abstract syntax and semantics of each of the two representations are described here. These are accompanied by an explanation of which aspects of

the representation make it inappropriate for the formal framework developed in this thesis.

4.2.1 Static Single Assignment Form

Static Single Assignment representations allow the explicit representation of data flow and control flow dependencies, and have been described as 'the preferred intermediate representation in modern optimising compilers' [Gle04].

A procedure is in *Static Single Assignment* (SSA) form if, and only if, each variable has at most one assignment in the static program text [RWZ88]. This is a property of the program source only: a program in SSA form may still make repeated assignments to a variable during its execution due to iteration or other control flow constructs present in the program.

Abstract Syntax

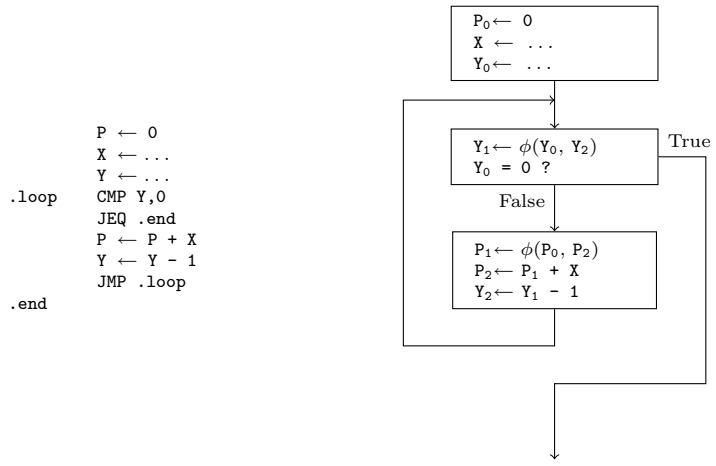
A procedure can be represented as a list of tuples where each tuple represents an operator, a number of arguments and optionally, a label. This representation is illustrated in Figure 4.1a. This can then be converted into SSA form by creating a new variable for each assignment to a variable in the original (non-SSA) program text. The new variable names created by this process are usually denoted by the original variable name with a unique subscript for each assignment.

A basic block may use a variable that has been assigned in any one of its preceding basic blocks. However, SSA form requires that the assignments in those predecessors must be to unique variables. In order to use a variable that has been 'split-up' into a number of subscripted variables, it is necessary to select the appropriate subscript of the variable in question, based on the flow of control that lead to that basic block in which it is used. The SSA form introduces $\phi-$nodes that perform that selection. A new variable subscript is introduced for each $\phi-$node to store the value selected by that $\phi-$node.

Figure 4.1 exemplifies the translation. A simple program that cumulatively computes the product of two unsigned integer values, `X` and `Y`, is shown in Figure 4.1a. It iteratively adds `X` to a variable, `P`, which is initialised to zero.

There are two assignments to `P` in the program text: the first performs the initialisation, the second increments the value for each loop iteration. In order that each assignment is to a unique variable, assignments to `P` are represented as $\texttt{P}_0$ and $\texttt{P}_2$ in SSA form in Figure 4.1b. Note that, in the tuple representation, both the block that initialises `P`, and the loop body that increments `P`, are predecessors of the loop body. The loop body must contain a $\phi-$node for `P` because its value may have been defined in either predecessor. The result of the $\phi-$node is assigned to $\texttt{P}_1$.

Figure 4.2 illustrates the same program using a more appropriate representation for SSA form. Using this notation, each basic block is a *Directed Acyclic Graph* (DAG) that represents the *Data Flow Graph* (DFG) of the basic block. Data flow through the program is represented by solid lines, while dotted lines represent control flow dependencies.



(a) *An example of a program in a tuple-based IR.*

(b) *The same program in Static Single Assignment (SSA) form.*

Figure 4.1: Static Single Assignment

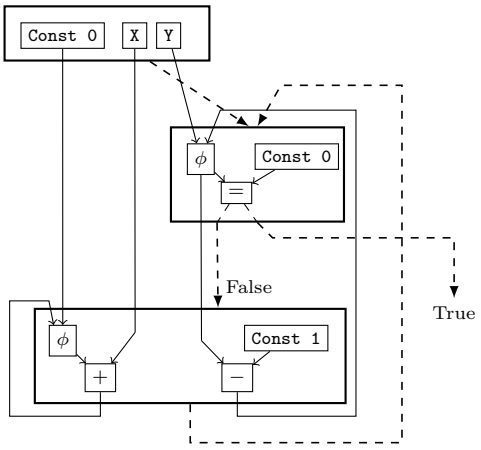


Figure 4.2: Static Single Assignment form showing data-flow and parallelism

A basic block may have either one or two immediate successors. In the case of one immediate successor, the control dependency is represented as a single dashed arrow to the next basic block. In the case of two immediate successors, the control dependencies is represented by a dashed arrow to each of the two possible successors. One of the two arrows is labelled 'True' and the other 'False'. A node in the basic block DFG is used to evaluate a Boolean condition that is used to determine which of the respective branches to take.

Using this representation, the instruction-level parallelism inherent in the algorithm is made explicit. For example, it can be seen that the addition and subtraction operations in the loop body may be computed in parallel.

Memory Access

The example given shows a computation that uses only local, scalar variables. Memory accesses that require indexing into memory require special treatment. The problem is that if the memory locations used are not known at compile time, then it is not possible to know exactly which memory operations commute. That is, it is not possible to determine whether they may be represented as parallel operations in the IR, and consequently whether it is safe to swap the order of memory accesses in the generated code.

Compilers work around this problem by using conservative transformations on the program. If it can be established that two operations definitely commute, then their order may be swapped. If there is a possibility that the operations do not commute, then a data-dependency is introduced to prevent the evaluation order being changed during the optimisation and code-generation phases of the compiler.

A naive compiler may simply assume that no memory operations commute. This is a simple approach, but may lead to the generation of inefficient code. Optimising compilers may use *region analysis* [BTV96] to identify the regions of memory that a memory access may use or modify. Where it can be established that two memory operations use or modify non-overlapping regions, those operations can be said to commute.

In order to represent the data dependencies introduced by memory operations that cannot be shown to commute, some SSA representations introduce a third type of arrow to denote dependencies between such memory operations [Muc97].

When generating the code for a basic block with no memory operations, the instruction order may be any topological sort of the DAG representing that basic block [Gle04]. For a basic block with memory accesses, the extra dependencies affect the partial order of operations, and must be taken into account.

Another approach to modelling data dependencies introduced by memory operations is to use the *functional store* approach [Ste95]. Using this technique, nodes in the DFG may represent not just operations on scalar values as shown above, but also memory operations. Thus a new type of node is introduced in the graph. While condition nodes evaluate to Boolean values and arithmetic and logic nodes evaluate to scalar values, memory write nodes evaluate to a memory region that may be indexed by location. These memory regions may then form one of the inputs to a memory read operation 'further down' the DAG.

Using the functional store approach, consecutive memory write operations are denoted by the result of the first memory write forming an input to the second. The order of memory operations is preserved because the first memory write appears earlier in every topological sort of the DAG.

Some of the literature that adopts the functional store approach assumes that memory appears as a single contiguous storage area [BG04, Gle04]. In those approaches, a basic block may use at most one memory region, and define at most one memory region.

An alternative approach is to allow a number of memory regions to be used

and defined within a single basic block. While this can be used to model a single memory region as above, it also allows compilers that perform region analyses to model 'parallel' memory accesses by determining that two operations affect non-overlapping regions, and therefore commute. In conventional compilers, parallel memory operations enable later optimisations to reorder memory operation instructions to improve the efficiency of generated code.

In hardware/software compilers, memory region analyses may allow the compiler to assign a memory region either to the main system memory, or — if the region is small enough — to memory in the function unit. This is particularly advantageous for function units without direct access to the system memory bus. In such cases, memory accesses are likely to be slow, and it is desirable to reduce the number of memory transfers between the function unit and the processor.

Formalisations

An *Abstract State Machine* (ASM) semantics of an SSA representation similar to that described above has been defined by Glesner [Gle04]. The ASM semantics allow the non-deterministic aspects of the evaluation of a program in SSA form to be modelled. Specifically, the semantics allow the DAG to be evaluated in any valid topological sort that respects the dependencies introduced by memory operations.

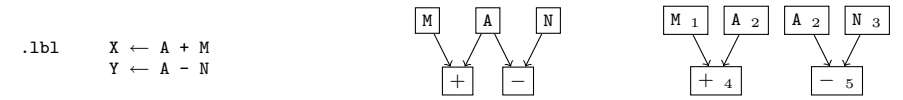
An alternative semantics formulated in the Isabelle/HOL framework is defined by Blech [Ble04, BG04]. The definition provides semantic functions that define an abstract interpreter for SSA form. In this respect, they form a denotational semantics of SSA form, describing a deterministic evaluation strategy for the representation. This is in contrast to the ASM semantics which model non-deterministic evaluation and form an operational semantics of the SSA form.

Isabelle/HOL forms the logical basis of the language definitions developed in this thesis, including the software IR developed in Section 4.3 which is based on the SSA form. Consequently, the semantics of the SSA representation as defined by Blech are described in the next section.

4.2.2 SSA in Isabelle/HOL

In the Isabelle/HOL formulation by Blech, the DAG that represents data-flow within a basic block is represented as a list of trees. The root of each tree corresponds with a exactly one node in the DAG representation. Specifically, the roots of the trees correspond to the lowermost nodes in the DAG (if it is drawn in a similar manner to Figure 4.2). These have no outward data-flow arcs to nodes within the same basic block, except arcs that cross the outer boundary of the basic block (as depicted graphically) and re-enter at the top. The leaves of each tree correspond to the uppermost nodes in the DAG. The only nodes that may be $\phi-$nodes are leaves.

In the multiplication example illustrated above, there are no *common sub-expressions*. In terms of Figure 4.2, this means that no DAG node has more than one outward (downward) data-flow arc. It is possible to represent basic



(a) *A basic block with a common sub-expression.*

(b) *The same basic block in SSA form.*

(c) *The same basic block represented as a term-graph.*

Figure 4.3: Term-graph representation of a basic block

blocks without common sub-expressions using a list of trees. The directed arcs in the DAG correspond to child-to-parent links in a graphical representation of a tree [1].

The natural representation for basic blocks using SSA form is as a DAG. However, there are advantages to using a tree representation. Most notably, a tree representation allows a simple representation in the Isabelle/HOL framework, and writing a denotational semantics as recursive functions over a tree structure appears more intuitive that writing functions over a DAG.

In general, a DAG node may have more than one outward arc, and nodes in the corresponding 'tree' list would require more than one parent. To accommodate this structure in a tree representation, the DAG is represented as a *term graph* [BN99].

Figure 4.3 illustrates the term-graph approach to representing a DAG. The tuple representation of the basic block shown in Figure 4.3a contains a sub-expression '`A`' that is common to both instructions. Figure 4.3b shows the DAG representation of the same basic block. Note that, in this format, the addition and subtraction nodes may not be used as the root node of a tree representation directly, because an '`A`' node would have to appear as a child in both the '+' tree and '−' tree. The information that the '`A`' nodes, which need to be duplicated in each tree, correspond to the same DAG node would be lost.

To represent the DAG as a term-graph, each node in the DAG is labelled with a unique identifier, in this case a number that corresponds to the variable assigned by that node. Each node in the DAG without a direct arc to another node within the same basic block is used as the root of a tree. A list of trees may then used to represent the DAG.

Each node in the trees is associated with the identifier for the corresponding operation in the DAG. The identifiers are unique within each tree. However, they are not necessarily unique across all trees because an expression may appear in more than one tree. Each tree contains all of the expression tree that is required to compute the final value of its root node.

[1] In order that a graphical representation of the trees appears visually similar to the DAG shown in Figure 4.2, the trees should be drawn with the leaves at the top and the root at the bottom of the diagram.

Formalisation

The Isabelle/HOL formulation defines a program as a list of basic blocks within the program's control flow graph (CFG):

types *cfg* = *basic_block list*

Basic blocks are identified by the ordinal number corresponding to their position in the list. Ordinals are natural numbers because programs, and hence the list, are assumed to be of finite size.

Variables are identified by values of type *id*, which is defined simply as a synonym for the type of natural numbers. Basic blocks within a program are represented using a datatype with a single datatype constructor, '*NEW*':

datatype *basic_block* = *NEW id id* (*id*×*nat*) (*id*×*nat*) (*ssa_tree list*)

The structure of the DAG corresponding to each basic block is represented as a list of trees of type *ssa_tree list*, using a term-graph representation as described on the preceding page. The structure of these trees is described below. At this point, it suffices to note that each node is associated with the *id* of the variable assigned by that node or, if the node represents a memory store operation, the memory state resulting from the evaluation of that node.

The meaning of the five values associated with the *basic_block* datatype is as follows. The first value denotes the variable in the basic block whose value is used to determine which of the two possible successor basic blocks should be executed after the current block. This value corresponds to one of the numbers that label nodes as illustrated in Figure 4.3c.

For example, in the multiplication example illustrated in Figure 4.2, the equality test would assign a variable with the result of the evaluation of the condition. Specifically, if the condition evaluates to true, the variable assigned by the condition node takes the value 1, otherwise it takes the value 0. The first value in the definition of the containing basic block would be the variable *id* associated with the node that represents the comparison.

The second value denotes the identifier associated with the node that represents the final memory state after executing the node. The node associated with that identifier is expected to be either a node representing a memory store operation or, if the basic block has no such operations, a '*MEMORY*' node that represents the initial state of the memory before the execution of the basic block.

The third and fourth values in the definition of the basic block are both pair types. These represent the necessary control flow information about the two successor basic blocks. For basic blocks with only one successor, these values may be the same.

Recall that the basic block that should be executed after the current block is determined by the result of evaluating the condition node identified by the first value in the basic block's definition. The third value determines (in a manner described below) how execution will proceed after the current basic block if the condition node evaluates to true (that is, any non-zero value). The fourth value determines how execution should proceed if the condition node evaluates to false (zero).

Within each pair, the first element is the identifier of the successor basic block

associated with the corresponding condition. Basic blocks are identified by the same type as variables: *id*. By virtue of the fact that this type is a synonym for the natural numbers, this value can be used as an index for the control flow graph, *cfg*. However, if *id* was redefined as a different type, it would be necessary to change the type used for the successor basic block.

The second value in each pair represents the 'rank' of the current basic block in the $\phi-$nodes of the corresponding successor basic block. Each $\phi-$node in a basic block will have one inward data-flow arc for each predecessor of that basic block. A $\phi-$node is defined using a list, where each element in the list is the identifier of the node associated with the source of the data-flow arc. The predecessor associated with an element in the list of a $\phi-$node is also associated with the corresponding element in the definition of all other $\phi-$nodes within the same basic block. The term 'rank' refers to the position in this list: the predecessor basic block defines its rank in the $\phi-$nodes of each successor.

This concept is better explained by considering an abstract interpreter for the representation. After evaluating a basic block, an interpreter determines the successor basic block by using the node identified by first value in the basic block's definition as a condition. It then determines whether to use the successor information from the third or fourth value (both of which are pairs), depending on the result of that condition. The first value from the pair determines the next basic block. The second value is used in the evaluation of the $\phi-$nodes in the next basic block.

The fifth value in the definition of a basic block is the *ssa_tree list* that represents the DAG that comprises the basic block. The *ssa_tree* type is declared using a datatype constructor for each node type:

datatype *ssa_tree* =
  CONST val id |
  PHI (*nat list*) *val id* |
  NODE operat ssa_tree ssa_tree val id |
  LOAD ssa_tree ssa_tree val id |
  STORE ssa_tree ssa_tree ssa_tree memory id |
  MEMORY memory id

The *id* corresponds to the name of the variable that is assigned by that node. This corresponds to the label in the term-graph representation.

Each node type that evaluates to a scalar value includes a value, *val*. Nodes that evaluate to a memory value include a *memory*, defined as:

types
  memory = *nat* ⇒ *val*

The reason for the inclusion of *val* and *memory* in the abstract syntax of the representation, is that in Blech's formulation, the semantic functions for tree evaluation recursively traverse the tree, and return a new tree in which each *val* and *memory* has been replaced with the appropriate value.

After evaluating the list of trees, a global mapping from identifiers to values is updated, by traversing the resulting trees once more, to read each *val* and *id*. The single, global memory state is also updated from the node with the *id* that corresponds to the second value in the definition of the basic block.

4.2.3 Pegasus

Pegasus is an intermediate representation designed specifically for the purpose of hardware/software compilation, and has a formally defined operational semantics [BG02b]. It is based on the Predicated-SSA and Gated-SSA representations, and therefore allows the explicit representation of fine-grained parallelism.

A representation of a program in the Pegasus IR is constructed by grouping its basic blocks into *hyperblocks*. A hyperblock is a group of instructions with a single point of entry, but any number of exit points. Each exit point represents a branch, either to another hyperblock, or back to the start of the current hyperblock.

A set of basic blocks may be grouped into a hyperblock only if they are 'reducible'. A set of basic blocks is reducible if and only if the control flow arcs between the basic blocks may be classified into forward and backward arcs, such that the following conditions hold. Firstly, the forward arcs (and the nodes that they connect) form a DAG with an entry node, where every other basic block is reachable from that entry node. Secondly, arcs that cause a basic block to dominate itself, either by branching back to itself or one of its predecessors, are classified as backward arcs. Thirdly, that the set of forward arcs and backward arcs be disjoint [Muc97].

In general, a program may be partitioned into hyperblocks in more than one way. At one extreme, each basic block can be considered as a hyperblock. This can result in a program being represented by many small hyperblocks. Alternatively, basic blocks may be grouped together into hyperblocks. In this case, the same program may be represented by fewer, but larger hyperblocks. The control flow graph may be divided into a minimal number of hyperblocks by computing minimal CFG intervals. Using this approach, inner loops that are not reducible are mapped directly into hyperblocks.

Conditional evaluation may be represented within a hyperblock, but each control flow path through the hyperblock is evaluated speculatively. This means that all the alternative branches within a hyperblock are evaluated in parallel, regardless of whether their evaluation is necessary for that particular execution of the hyperblock.

Pegasus Abstract Syntax

Pegasus represents data-flow within each hyperblock as a DAG, in a similar manner to the SSA form illustrated in Figure 4.2. A (simplified) representation of the multiplication example from the previous section in the Pegasus IR is illustrated in Figure 4.4. For the purposes of this example, it is assumed that the only variable that is 'live' on exit from the program fragment is `P`. This means that `P` is the only variable that is used later elsewhere in the program.

The boxes with a heavy outline represent hyperblocks. Note that the basic block that tests the value of `Y` has been grouped into a hyperblock with the body of the loop. Thus the body of the loop, which decrements `Y` and computes the sum of `X` and `P`, is executed speculatively in parallel with the evaluation of

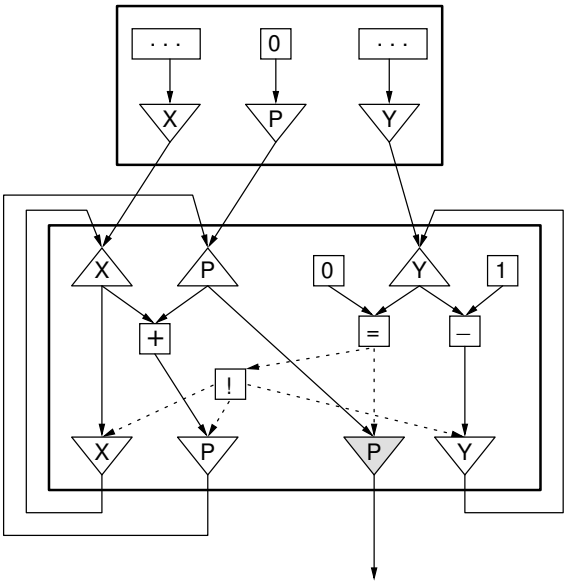


Figure 4.4: The multiplication example in the Pegasus IR.

the condition `Y = 0`. Solid lines between nodes represent data-flow of values, such as machine words. Dotted lines between nodes represent data-flow of Boolean values.

The upward-pointing triangular-shaped nodes at the top of the larger hyperblock are *merge* nodes. They are used where a variable used by a basic block may have been assigned by more than one of the basic block's predecessors. They are required here because the hyperblock with the condition and loop body uses values that may have been defined by two predecessors: itself, and the initialisation block.

The downward-pointing triangular-shaped nodes at the bottom of hyperblocks are called *eta* nodes. Eta nodes are used to 'steer' values to the basic block in which they are used. If a basic block assigns a variable that is used in n successor basic blocks, then there are n corresponding eta nodes for that variable in the basic block that makes the assignment. Each eta node is connected to a merge node in a successor basic block that uses a given variable. In the example, the assignment of `P` in the condition and loop body hyperblock may be used in two places: in the next iteration of the condition and loop body hyperblock, or any hyperblock that succeeds it (not shown).

Eta nodes may be grouped by the successor hyperblock to which they direct values. In the illustration, these groups are identified by colour. Eta nodes in the main hyperblock that pass information back into the same hyperblock are white. The eta node used to pass the value of `P` to a successor hyperblock is coloured grey. Eta nodes within a hyperblock with the same colour are referred to as *eta groups* here.

Eta nodes have one value input, one Boolean input, and one value output. When the Boolean input is true, the value at the input is passed to the merge

node to which its output is connected. When the Boolean input is false, the node is said to 'consume' its input. Where the Boolean input is not illustrated, it is implicit, and is assumed to become true when execution of the hyperblock completes. This is used in the initialisation hyperblock in the example, because it only defines values which are used by the immediately dominated hyperblock.

There is no concurrency between hyperblocks: only one hyperblock may be executed at any given time. Thus, for each execution of a hyperblock, there may only be one successor hyperblock. All eta nodes that may pass a value to that successor are assumed to do so during the execution of the basic block. This means that if one of the Boolean inputs to an eta node becomes true during the execution of a hyperblock, then the Boolean input to all eta nodes in the same eta group should be true at some point in the execution of the basic block. The term *active eta group* is used here to refer to an eta group in this state.

Only one eta group may be active at for a single execution of a hyperblock. It is assumed that the Boolean inputs to eta nodes in other eta groups are false for the duration of the hyperblock's execution. Therefore, the successor of a hyperblock is determined by the active eta group.

Merge nodes and eta nodes have a similar purpose to $\phi-$nodes in SSA representation, in the sense that they select a single value from a number of reaching definitions.

Memory Access

In contrast to the SSA formulation described in Section 4.2.2, Pegasus does not use the functional store approach for modelling memory operations. Instead, it uses an implicit global memory.

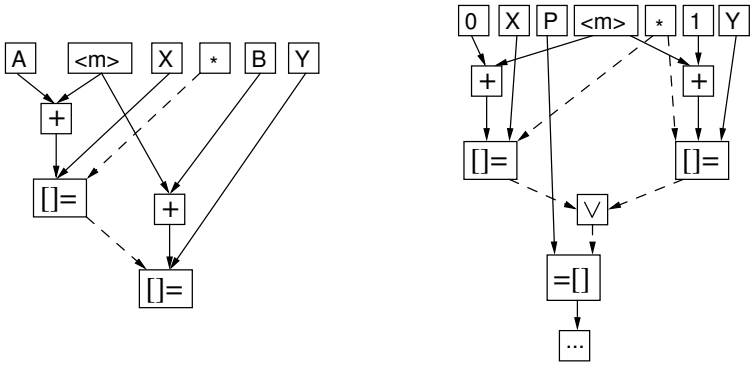
Two new types of node are introduced to represent load and store operations on the global memory, denoted by `=[]` and `[]=` respectively. Nodes representing load operations have one input value, the memory location to be addressed, and one output value, which takes the value of that memory location. Nodes representing store operations have two input values: the memory location to be addressed, and the value to be stored. Store nodes have no outputs values.

Dependencies between memory operations are denoted using a new type of edge between nodes, illustrated here using dashed lines between memory operation nodes. These edges are said to carry a token value: a memory operation may not occur until it receives a token and relevant inputs, and when the operation completes, it emits a token on any number of edge-carrying tokens.

Two further node types are introduced for managing the flow of tokens between memory operation nodes. The first is a node that emits a token at the start of the execution of a hyperblock. These are required in order that a token can be passed to the first memory operation. This node is denoted by the symbol used to represent tokens: an asterisk (`*`).

The following example code fragment requires the use of two store operations, and can be used to illustrate the token edges, and the token node:

```
M[A] ← X
```



(a) *The token value and memory operation dependencies.*

(b) *The 'V' token merge operator.*

Figure 4.5: Dependencies between memory operations in Pegasus.

```
M[B] ← Y
```

In general, the two assignments do not commute because if `A` and `B` are equal, then the order of these instructions affects the result of their execution. Thus, if the compiler cannot determine than `A` and `B` have different values, then it must preserve the order of these assignments. Figure 4.5a illustrates the representation of these store operations in the Pegasus IR. Note that the assignment to `M[B]` cannot occur until the relevant store operation receives a token from the store operation that implements assignment to `M[A]`.

The second type of node for managing token flow is the *V-node*. A V-node has a number of token inputs, and a single token output. It is used where a memory operation depends on more than one memory operation. The V-node waits until tokens are available on all of its inputs, and then emits a single token on its output.

The use of the V-node can be demonstrated by the Pegasus representation of the following code:

```
M[0] ←  X
M[1] ←  Y
...  ← *P
```

In this instance, the two store operations commute. However, if the compiler detects that `M` may alias `P`, or does not perform alias analysis, then the final instruction does not commute with either of the store instructions. Thus it has a memory dependency on both of the store instructions. Figure 4.5b shows the Pegasus representation for these instructions. A V-node is used to merge the tokens from the two store operations, and the load operation from address `P` cannot occur until both store operations have emitted tokens.

Pegasus Semantics

The formal semantics of Pegasus [BG02b] are expressed using the operational semantics style attributed to Plotkin [Plo81]. This approach provides a non-deterministic evaluation strategy for Pegasus programs, and allows parallelism in the evaluation of different sub-graphs of a hyperblock to be modelled.

The intuition behind the semantics is that nodes produce and consume data, where 'data' includes (among other things) program values and memory access tokens. Most types of node both produce and consume data, although nodes that define constants and that produce initial tokens consume no data. This style of semantics has been used to model the behaviour of asynchronous circuits [DN97] and dataflow architectures [Vee86].

The protocol between producers and consumers is implemented using a synchronous version of the *Two-Phase Bundled Data* convention [Sut89]. The bundled data corresponds to the edges between nodes in the Pegasus representation. The stages in the handshake protocol can be interpreted to determine whether or not a value is available for consumption, or whether the value on that edge has already been consumed.

The two-phase bundled data protocol works as follows. When a node produces data, it signals the data on its output, and asserts a 'request' signal. It must maintain this output signal until the consumer returns a corresponding 'acknowledge' signal. At this point the producer may return the request signal to its initial state. When the consumer is ready to for another value, it returns the acknowledge signal to its initial state.

The semantics use an abstraction of the implementation of the protocol. There is one semantic domain, σ, representing the state of all edges. The value of an edge e, denoted by $\sigma(e)$, may be any of the following:

- Numbers, for which the representation is not specified;
- Boolean values, denoted by $\boldsymbol{T}$ and $\boldsymbol{F}$;
- memory access tokens, denoted by τ;
- the undefined value, $\perp$, which represents the absence of any value that can be consumed;
- a *don't care* value, $\triangle$, used to represent the existence of an arbitrary value;
- a *wait* value used during the non-strict evaluation of Boolean expressions;
- node names, which are used to implement control flow between functions.

Two auxiliary functions are defined to simplify notation with $\perp$. These are $def(e)$, which is true when the edge e has a value available for consumption, and $erase(e)$, which is used to consume a value by updating the state such that the edge takes the undefined value. These functions are defined as:

$def(e) \equiv (\lambda\sigma.\ \sigma(e) \neq \perp)$
$erase(e) \equiv (\lambda\sigma.\ \sigma[e \mapsto \perp])$

The predicate *def* characterises edges where the corresponding request signal has been asserted, but not acknowledged. The state update function *erase* corresponds to the effect of raising the acknowledge signal, which leads to the producer ceasing to assert the request signal.

The *don't care* value is used where a node must produce a value (and hence participate in a request/acknowledge handshake), but will have no affect on the result of the program. For example, since each node in a hyperblock is evaluated speculatively, memory load operations are evaluated and expected to output a value. However, there is no purpose to fetching data from memory if it will not be used, and doing so would reduce the performance of the program. Consequently, memory load operations are predicated according to whether control flow in a sequential implementation would reach that load operation. If that predicate evaluates to false, then memory load operations emit the *don't care* value. Evaluation of unnecessary function calls is avoided similarly.

The semantics of a node that implements Boolean negation are expressed using the following rule:

$$o = \mathsf{Not}(i) \ \frac{def(i) \qquad \neg def(o)}{\sigma' = \sigma\ [o \mapsto \mathsf{Not}\ (\sigma(i))] \circ erase(i)}$$

The annotation to the left of the rule shows that the rule refers to a node that implements the Not function, with input i and output o. The precondition indicates that the semantic rule is only applicable where the node has an input value ready, $def(i)$, and that node that will consume the output value has already consumed the previous value, $\neg def(o)$. When these preconditions are met, the effect of the node is to update the output to be the Boolean negation of the input (denoted by an update of σ), and to consume the input (denoted by *erase*).

4.3 Formal Definition of a Hardware/Software IR

This section presents the semantics of an intermediate representation intended to support verifiable hardware/software compilation according to the requirements identified in Section 4.1. The IR is based on the IRs presented in Section 4.2, although these have been adapted into a form that is more appropriate to the logical framework used and developed in this thesis.

4.3.1 Abstract Syntax

The form of the semantics for the IR presented here are (superficially, at least) most similar in appearance to Blech's SSA semantics, as described in Section 4.2.2. This is because they are presented in the Isabelle/HOL framework, and consist largely of recursive functions on an abstract syntax representation of a program.

The abstract syntax is presented here using a top-down approach: starting with the top level abstract syntax construct for representing entire programs, then the representation of blocks within the program, and concluding with the nodes within a block that represent operators.

Program Representation

Programs in the intermediate representation are represented as a finite number of hyperblocks, which have a similar form to those in the Pegasus representation. Pegasus does not appear to define a construct in the abstract syntax for representing an entire program, and the formal semantics only appear to cover evaluation within a hyperblock. Hyperblocks need to be uniquely identifiable. It suffices to represent a program using a list of hyperblocks, each identified by its index in the list:

types
prog = *hblock list*

Hyperblock Representation

In Blech's formulation of SSA, each basic block may have no more than two successors. In Pegasus, hyperblocks can have more than two successors. Each eta node is allocated to a single eta group (as described on page 68), and eta groups have a one-to-one correspondence with successor hyperblocks. A hyperblock may include an arbitrary number of eta groups, and hence, may have an arbitrary number of successors. It is therefore a generalisation of the representations that only allow up to two successors, and is the form adopted here.

In general, a node may be involved in the evaluation of more than one eta group. For example, in Figure 4.4 the merge nodes labelled P and Y, the constant node 0, and the node that implements an equality test must be evaluated in order to determine which eta group is applicable, and hence which of the hyperblocks' successors should be executed next.

Blech's term-graph representation demonstrates how a basic block may be represented as a list of SSA trees. The basic blocks in that representation can be considered to be similar in structure to an eta group in Pegasus, where each eta node in that group corresponds to the root of an SSA tree [2].

However, a list of SSA trees is not in itself sufficient to represent all of the required information about an eta group. The full representation of an eta group is represented by a record type, *η-group*. Deferring the definition of that type, the abstract syntax construct for a hyperblock may be defined simply as a list of the eta groups within that hyperblock:

types
hblock = *η-group list*

As per hyperblocks within a program, eta groups within a hyperblock are identified by their position within the list that defines the hyperblock.

Eta Group Representation

The relationship between an eta group, and the successor hyperblock to which it corresponds, can be represented by identifying the successor hyperblock within

[2] Represented as DAGs, basic blocks can also be considered to be loosely similar in structure to an entire hyperblock, but this comparison is not very useful here.

the definition of the eta group. It is also necessary to include the condition under which that eta group becomes the active eta group, as defined on page 69.

It can be seen from Figure 4.4 that a single Boolean value is used as the Boolean input to all eta nodes within an eta group. The larger hyperblock in the figure shows two eta groups: one has a single, grey coloured eta node, '`P`'. The successor hyperblock associated with this eta group is not shown in the figure. The other eta group has white coloured eta nodes, and the successor hyperblock associated with that eta group is the containing hyperblock itself. The figure shows the value resulting from the evaluation of the '`=`' node forming the input to single eta node in the grey former eta group, and the value resulting from the evaluation of the '`!`' node forming the Boolean input to the eta nodes in the latter eta group.

A single Boolean value is sufficient for the Boolean input to every eta node within an eta group. This is because of the condition that the Boolean input to every eta node within an eta group must have the same value. That is, it must be true for all eta nodes in the active eta group, and false for every other eta node in the hyperblock. This condition is discussed below in *Satisfiability of Eta Group Correctness Conditions*.

The relationship between an eta group and its corresponding successor hyperblock is represented in the abstract syntax using two fields of the *η-group* record type: *next-block*, and *cond*. The *next-block* field is a natural number that identifies the successor hyperblock associated with the eta group by its index in the hyperblock list, of type *prog*, that represents the whole program. The *cond* field represents an SSA tree that evaluates to a Boolean value. This value is used as the Boolean input for all eta nodes in that eta group.

The full definition of the *η-group* type is as follows:

record *η-group* =
  next-block :: *nat*
  cond :: *boolTree*
  η-bool :: (*boolTree* × *name*) *list*
  η-word :: (*wordTree* × *name*) *list*
  η-mem :: (*memTree* × *name*) *list*

The first two fields have been described above. The other three fields require some explanation. In Blech's formulation of SSA trees, described on page 66, a single datatype was used to represent an SSA tree, with a datatype constructor for each type of node. Without the addition of appropriate functions to characterise well-formed trees, this representation allows poorly typed trees to be represented. For example, it is possible to construct an SSA tree in which nodes that would be expected to produce a scalar value, such as addition nodes, to be used as a memory operand on a node that represents a memory load or store operation.

The computed value of each node is stored in the abstract syntax construct in Blech's representation. That value may be a machine word or a representation of the memory. In order to retrieve that value, two functions are defined on the *ssa_tree* datatype. One of those functions returns the machine word only if the node evaluates to a machine word. Otherwise its behaviour is underdefined. The other returns a representation of the memory if the node evaluates to such a representation, and its behaviour is underdefined otherwise.

For this thesis, an alternative approach is preferred. Three mutually recursive datatypes are used to represent SSA trees. One datatype is used to represent trees that evaluate to Boolean values. Another is used to represent trees that evaluate to scalar values, which form an abstraction of machine words. The third is used to represent trees that evaluate to a representation of the memory. These datatypes are called *boolTree*, *wordTree* and *memTree*. These types are used in the η-*bool*, η-*word* and η-*mem* fields of the η-*group* record type.

The SSA trees are required to be mutually recursive because nodes may have a different type of input from their output. For example, a node that tests for equality has two scalar inputs and a Boolean output. The definition of the datatypes used to construct these trees is given on page 77.

The use of mutually recursive datatypes enforces a limited form of type safety. This increases the relative proportion of syntactically valid programs for which a suitable semantics can be given. This approach also reduces the need for underdefined functions in the semantics.

Eta groups, and hence hyperblocks, may include an arbitrary number of all three kinds of tree. The root element of all three kinds of tree represent eta nodes.

The only remaining aspect of the η-*group* type definition that requires explanation is the *name* associated with each tree, or viewed alternatively, with the eta node at the root of each tree.

The significance of the name associated with each eta node is perhaps best explained by reference to the graphical representation of programs in the Pegasus IR. The name associated with the eta node in the abstract syntax is not, as one might expect, the label on the eta node as shown in the graphical representation. Instead, it corresponds to the identifier of the merge node in a successor hyperblock for which the eta node is a source of data. Thus it is assumed that each eta node has exactly one outward edge. If a value is required by two merge nodes in the successor hyperblock, two eta nodes are required.

Two auxiliary functions are defined as synonyms for *fst* and *snd* for retrieving the tree and name associated with an element in a list of SSA trees in an eta group definition. These are:

$get\text{-}tree \equiv fst$
$get\text{-}\gamma\text{-}name \equiv snd$

These are intended to improve readability in the semantic functions for the representation. The name of the function, $get\text{-}\gamma\text{-}name$, is a reference to the related $\gamma-$nodes in the PDW representation [OBM90].

Satisfiability of Eta Group Correctness Conditions

There are two significant conditions on the representation of eta groups raised by the above description of their abstract syntax. The first is that a single Boolean value should form the Boolean input to all the eta nodes in a given eta group. The second is that the Boolean value used to determine the active eta group is 'one-hot'. This means that, for a given hyperblock evaluation, the Boolean input to all of the eta nodes in exactly one eta group is true, and that

the Boolean input to every eta node in every other hyperblock is false. These conditions are discussed here.

The condition that a single Boolean value should form the Boolean input to all the eta nodes in a given eta group is not, strictly speaking, a well-formedness condition of the abstract syntax for eta groups presented above. Although it is possible to graphically depict an eta group for which the constituent eta nodes have different Boolean inputs, such an eta group cannot be represented in the abstract syntax presented above.

The Pegasus representation does not have an explicit representation of eta groups [BG02b]. However, the Figure 4 in the cited paper illustrates hyperblock where eta nodes are visually grouped together according to the hyperblock to which those nodes forward values, suggesting that eta nodes are grouped at an intuitive level, even if this is not explicit in the representation.

Recall from on page 68 that an eta group is defined by a set of eta nodes that forward data to a common successor hyperblock, rather than by the Boolean condition used as inputs to those eta nodes. In terms of the Isabelle model, this means that eta groups within a hyperblock have unique values of *next-block*. Hence, an eta group can be determined from the Pegasus representation, by simply grouping eta nodes by successor hyperblocks.

The abstract syntax for an eta group is based on the assumption that a single Boolean value is sufficient for the Boolean input to every eta node within that eta group. It is necessary to justify this assumption in order for the abstract syntax to be viable for an intermediate representation.

The identification of an appropriate Boolean input for each eta group would then be left to the compilation algorithm. Furthermore, it is a correctness condition on the compiler that, for a given hyperblock evaluation, there should only be one eta group for which that input is true, and hence that such Boolean values are one-hot.

These conditions must also be satisfiable in order to justify the viability of the representation. An informal argument of the adequacy of a single Boolean value for controlling all eta nodes within an eta group, and the satisfiability of these correctness conditions is given here.

Again, recall from on page 67 that programs can be partitioned into hyperblocks in different ways. A compiler could translate each basic block into a hyperblock, because hyperblocks are a generalisation of basic blocks that can have more that two successors (that is, more than two points of exit), and provide for speculative execution. A hypothetical compiler that adopts this approach can be considered in order to demonstrate the requirements of the representation.

A basic block in Blech's representation can have either one or two successor basic blocks. If a basic block has only one successor, then it can be converted into a hyperblock with only a single eta group. The Boolean value used to control the eta nodes within that group is simply the constant *true*. If a basic block has two successors, then the translation into a hyperblock will need to create two eta groups: one for each successor. Hence, there is one eta group for each exit from the hyperblock. The eta nodes that need to be in each group are determined by variables that are live on entry to each successor.

The eta nodes in one eta group will be controlled by a Boolean value derived from the branch condition at the end of the basic block, and the other with its logical negation. The logical negation can be represented simply by connecting the branch condition to an eta node via an inverter.

Using this strategy, a single Boolean value is sufficient to for controlling each eta group. Furthermore, the one-hot property of eta groups is guaranteed, because there are only two eta groups where the Boolean value controlling each eta group is the logical negation of the other.

A more sophisticated approach that allows basic blocks to be grouped into a hyperblock is described in Section 2.3 of the cited paper about Pegasus. The technique involves representing mapping control flow between basic blocks into data flow within a hyperblock. In this strategy, an eta group is again created for each exit from the hyperblock.

A more complex example involves forming a hyperblock from two basic blocks, one of which immediately dominates the other. In this case, there are two Boolean values that control the eta groups representing the exits from the dominated block, and hence the hyperblock. One of those values is the logical conjunction of the branch condition of the dominator block and the branch condition of the dominated block. The other is the logical conjunction of the branch condition of the dominator block and the negation of the branch condition of the dominated block. The only other eta group in the hyperblock is controlled by the logical negation of the branch condition of the dominator block. Again, only one of the Boolean values controlling eta groups can be true.

Where a basic block is compiled into a hyperblock that contains more than one predecessor to that basic block, disjunction is used on the exit conditions of each predecessor. This is illustrated in the Pegasus paper.

SSA Tree Representation

The datatypes used to represent SSA trees are defined as follows:

datatype *boolTree* =
  BoolMonBoolOp boolMonadicBoolOp boolTree id
| *BoolMonNatOp boolMonadicNatOp wordTree id*
| *BoolDyNatOp boolDyadicNatOp wordTree wordTree id*
| *BoolMerge name id*
and *wordTree* =
  Const word id
| *WordOp wordOp wordTree wordTree id*
| *BoolToWord boolTree id*
| *Load wordTree memTree id*
| *WordMerge name id*
and *memTree* =
  Store wordTree wordTree memTree id
| *MemMerge name id*

Each node in an SSA tree is associated with an *id*, as required by the use of a term-graph representation. It is sufficient for the purposes here to define *id* as a synonym of the type *nat*.

Other types that appear in the type definition of SSA trees are *name* and *word*. Both of these types are also defined as synonyms of the type *nat*. This choice is rather arbitrary however. The *word* type could also be defined in terms of integers; a bit list representing a machine word; or as a restricted set of integers or natural numbers. The *name* type is used to represent the labels that identify eta nodes. Any type that provides sufficient unique identifiers would be sufficient for *name*.

Eta nodes themselves are not represented in the SSA trees. The only information that needs to be associated with eta nodes is the γ-*name*, which is already included in the η-*group* abstract syntax construct. The only place that an eta node could appear in an SSA tree is as the root node. The eta nodes are therefore elided because no extra information needs to be represented in the abstract syntax, and their presence becomes implicit.

SSA trees include merge nodes for each of the three primitive types in the IR: namely Booleans, machine words and memory regions. These nodes are represented by the *BoolMerge*, *WordMerge*, and *MemMerge* datatype constructors respectively. The name associated with a merge node represents the γ-*name* of that node. That is, it corresponds directly with the label of a merge node in the graphical notation of Pegasus.

The names associated with eta nodes — or rather, the names associated with SSA trees in the η-*group* type — are expected to refer to the names associated with these merge nodes. Programs that do not meet this assumption are not considered well-formed.

Formal definitions of small languages expressed in higher order logic often neglect to explicitly name operators that exist in the language [NPW02, Nip98]. Instead, the operators are represented in expressions as an anonymous higher order functions that determine the behaviour of the implied operator. This technique has the advantage that the language definition is abstracted over the primitive operators it provides.

In contrast, an initial set of named primitive operators is used here. The reason for naming operators explicitly is that this methodology assumes that most, if not all, operators in the IR will be associated with a hardware circuit design with equivalent behaviour. In order to specify the translation of a program represented in the abstract syntax into a hardware representation, it is necessary to name operators rather than use anonymous functions. This is because within the logic, it is not possible in general to derive a suitable hardware implementation for an operator from a term representing a function.

An initial set of primitive operators includes operators that produce both machine word and Boolean and values. Two dyadic operators that produce machine words are *Add* and *Sub*. There are both monadic and dyadic operators that produce Boolean values. A monadic operator on machine words is the *Test For Zero* operator, *TFZ*. The single monadic operator on Boolean values is *BNot*, which provides logical negation. Dyadic operators that produce Boolean values are the arithmetic comparison operators, equal, *Eq*; less than or equal, *LtEq*; and greater than, *Gt*. Thus an initial set of primitives is defined as:

datatype *wordOp* = *Add* | *Sub*

datatype *boolMonadicBoolOp* = *BNot*
datatype *boolMonadicNatOp* = *TFZ*
datatype *boolDyadicNatOp* = *Eq* | *LtEq* | *Gt*

An alternative approach would be to use anonymous functions and define extra-logical functions over the structure of the higher order terms used to represent the abstract syntax. Such an approach is taken by Gordon et al. [GIOS05] to semi-automatically derive a description of a hardware circuit for a given function in higher order logic. In practical terms, the extra-logical functions they use are functions written in Standard ML over data structures that represent terms in higher order logic.

Gordon's approach has the advantage that it admits greater automation in the derivation of hardware designs. The disadvantage of that approach is that it is not possible to reason about the extra-logical functions that operate on higher order terms, because they necessarily fall outwith the logical framework used to represent the language definition.

The focus of this work is the development of a logical framework for reasoning about the functions that derive hardware circuit designs from a program representation. Thus the use of named primitives is justified by the requirement to be able to reason about the compilation in the same logical framework as the definition of the IR, and of the hardware representation presented in Chapter 5 and Chapter 6.

4.3.2 Semantics

A denotational semantics of the IR is presented here. The semantics strongly resemble that of Blech's formulation of SSA: a function is defined to model the state transition effected by executing a block. This function is used repeatedly by another function which also determines the order in which blocks should be executed.

The semantics are presented here by first defining the semantic domains of the representation, which model the state of a program during execution. The initialisation function that determines the initial state of the semantic objects used in the execution of a program from its abstract syntax representation is also given. Finally, the semantic functions that define the relevant state transitions for a given program are specified.

Semantic Domains

In Blech's SSA formulation, each node in an SSA tree was associated with the value that had been computed for it. These values were omitted from the abstract syntax definition of this IR, as defined on page 77. Instead, the state of each block is maintained as a separate semantic object, Σ_b:

record Σ_b =
  σ-*bool* :: *name*⇒*bool*
  σ-*word* :: *name*⇒*word*
  σ-*mem* :: *name*⇒(*word*⇒*word*)

Each field in the record type determines the state of all the nodes in the basic block of a given type. That is, the σ-*bool* field maps the identifiers associated with nodes that evaluate to Boolean values to the value of that node at a given point in the execution of the block. Likewise, the σ-*word* field maps identifiers associated with nodes that produce word values to the last value computed by that node.

The σ-*mem* field allows different memory regions to be used by a basic block. Each memory region is indexed by a machine word. The result of using a previously unassigned memory location within a region, or of using an invalid memory location is underspecified. Examples of regions that might be used in a basic block include a single stack frame; the entire stack; a page in memory; or the entire memory store of the target machine.

The memory values within a basic block, represented in the σ-*mem* field, are likely to contain largely duplicated information. This is due to the use of the functional store approach, where a new memory node is must be introduced for each store operation on a given memory region.

One of the advantages of representing the state of a block separately to that of the representation of the block itself is that it is clearer that the semantic functions do not affect the structure of the program, only the state of the program. This is significant, because in hardware/software compilation self-modifying programs are particularly problematic. This is because their behaviour will vary depending on which parts of the program have been compiled as a hardware function unit and which have been compiled as object code.

The state of the entire program is simply the combined state of each of its constituent hyperblocks. Just as a program is represented by a list of its constituent hyperblocks, the state of the program is represented as a list of the current state of each hyperblock:

types
$\Sigma_p = \Sigma_b$ *list*

Thus, for a program p with current state σ_p, the i'th hyperblock is denoted by $p_{[i]}$, and its state is denoted by $\sigma_{p\,[i]}$.

The initial state of a program is determined by the function *init*-Σ. This simply creates a list of hyperblock states of the same length as the list of hyperblocks that represents the program, by recursion on the latter. No initialisation of the program memory is assumed, and therefore the initial state of each hyperblock is an arbitrary value.

init-Σ [] = []
init-Σ $(b \cdot bs)$ =
$(\!|\sigma\text{-}bool = arbitrary, \sigma\text{-}word = arbitrary, \sigma\text{-}mem = arbitrary, \ldots = ()|\!) \cdot$
init-Σ *bs*

Overview of Semantic Functions

The semantic functions defined here, and later in this thesis, are identifiable by the letter 'm' which prefixes each function name. The prefix is a reference to the fact that the function gives the meaning to an abstract syntax construct.

The remainder of each name usually denotes the syntactic construct for which the function gives a semantics.

A more commonly used notion in literature on denotational semantics, is to use "Strachey" brackets when writing semantic functions over syntactic terms and contructs. For example, the semantics of a term t would be denoted by $\mathcal{M}[\![t]\!]$.

The more commonly used notation is avoided here for two reasons. Firstly, such a notation would unnecessarily complicate the language definition within the Isabelle/HOL framework. Secondly, the semantic functions presented here do not have a one-to-one correspondence with syntactic constructs. Instead, some of the functions have a polymorphic definition, and are used to define the semantics of more than one syntactic construct. This approach results in a shorter, simpler language definition here.

The semantic functions for the IR fall into two categories: those that model the control flow from one hyperblock to the next; and those that model the data flow within a hyperblock. Naturally, there is some overlap between these categories, because data values are usually used to determine a successor hyperblock.

Top-level Semantic Functions

At the 'top-level' of the language definition, a function is introduced to define the semantics of executing a given number of hyperblocks. More accurately, the function repeatedly evaluates the state transition effected by the current hyperblock and determines the next hyperblock to be executed until it has completed a given number of repetitions. The function is called *step*, and assumes a function *mHblock* (defined on the next page) to that can be used to evaluate the state transition that results from the execution of a hyperblock. It is defined by recursion on the remaining number of iterations, as follows:

$$step\ p\ \sigma\ 0 = \sigma$$
$$step\ p\ \sigma\ (Suc\ n) = (\mathit{let}\ (curr,\ \sigma_p) = \sigma\ \mathit{in}\ step\ p\ (mHblock\ p_{[curr]}\ curr\ \sigma_p)\ n)$$

The first parameter of the function, p, is a list of the hyperblocks that represent the program. Hence, p is of type *prog* as defined on page 73. The second parameter, σ, is a pair that represents the first hyperblock to be executed, denoted by *curr*, which is an index into p; and the current state of the program, denoted by σ_p, which is of type Σ_p. The final parameter, n, denotes the number of hyperblock evaluations that should performed. The value returned by *step* is a pair of the same type as σ: it determines the next hyperblock to be executed after n hyperblock executions, and the state of the program at that point. The value returned by *mHblock* is also a pair when applied to the arguments as shown above.

The function is defined to model a known, finite number of hyperblock executions to avoid the need to introduce a fixed-point combinator into the language definition. The use of a fixed point-operator would complicate the language definition, because the semantics are defined in a logic of total functions, and it is unclear that it would serve any useful purpose here. In this respect, the *step* function can be considered as an example of an *iterated map* [Fox01b].

Hyperblock Execution

Hyperblocks are evaluated by determining the active eta group, and then evaluating the SSA tree for each eta node in that group. However, in order to determine the active eta group, it is necessary to evaluate sufficient SSA trees to find the value of the predicates that determine which eta group is active. The SSA trees that need to be evaluated are those specified in the *cond* field of the *η-group* type.

In order to determine the active eta group of a hyperblock, it is sufficient to evaluate the *cond* SSA tree within each eta group until an eta group is found where *cond* evaluates to *True*. This is sufficient because there must be a unique active eta group for any given hyperblock evaluation, and thus it can be assumed that *cond* will evaluate to *True* for exactly one eta group. The evaluation strategy represented by the semantic functions here uses this approach for finding the active eta group.

In order to evaluate a *cond* SSA tree, a function *mBoolTree* is assumed, which evaluates a Boolean SSA tree under a given hyperblock state (of type Σ_b). Using this, a function to determine the active eta group, *mHblock-next*, is defined in terms of the hyperblock and its state:

mHblock-next [] σ_b = *arbitrary*
mHblock-next ($e{\cdot}es$) σ_b =
(*if* *mBoolTree* (*cond* e) σ_b *then* e *else* *mHblock-next* es σ_b)

Once the active eta group has been determined, the *next-block* field identifies which hyperblock should be evaluated after the current hyperblock. Recall that the identity of a hyperblock is the corresponding index into a list of type *prog*. Using this, the semantic function for a hyperblock, *mHblock*, returns a pair which consists of the identity of the successor hyperblock, and a new program state. It is defined as follows:

mHblock hb curr $\sigma_p \equiv$
let *active* = *mHblock-next hb* $\sigma_{p[curr]}$; *next* = *next-block active*
in (*next*, *mη-group active curr next* σ_p)

The function *mη-group* that determines the new program state that results from the evaluation of a hyperblock. It is a function of the current eta-group, the identity of the current hyperblock, the identify of the successor hyperblock, and the current program state.

Note that once the active eta group has been determined, only that eta group needs to be evaluated in the evaluation of the hyperblock. As a result, once the active eta group has been determined, evaluation of the hyperblock becomes synonymous with evaluation of the active eta group.

It should be clear that *mη-group* must be a function of, among other things, the current eta group — the behaviour of a hyperblock must clearly be determined (in part, at least) by the operations within the hyperblock. The reason why *mη-group* must also be a function of the identity of the current and successor hyperblocks is perhaps less obvious.

The identity of the current hyperblock is required for the evaluation of the current hyperblock because the evaluation function is a function on the entire

program state. Thus, in order to use the state of the current hyperblock during evaluation, its index within the global program state must be known.

The identity of the successor hyperblock is also required for the evaluation of a hyperblock. This is because the state of the subsequent hyperblock is updated as a side-effect of evaluating each eta node in the current hyperblock. The state of the subsequent hyperblock is updated such that, when that hyperblock is evaluated, its merge nodes will evaluate to the value of the corresponding eta node in the current hyperblock.

The semantic functions that model the evaluation of a hyperblock can therefore be considered to 'read-only' with respect to the state current hyperblock, using it only for evaluation of the merge nodes. It is not necessary to reflect the value evaluated for each node in the SSA trees in a hyperblock because of the use of denotational semantics[3]. Only the state of the successor hyperblock is modified.

Eta Group Evaluation

An eta group is evaluated by evaluating its constituent SSA trees, by recursively evaluating each node in the tree until the merge nodes at the tree leaves are reached. Once each eta node at the top of each tree has been evaluated, the state of the successor hyperblock is updated, such that the corresponding merge node will take the value of the evaluated eta node.

SSA trees may be evaluated in any order. This is because of the use of the functional store approach; and the assumption that all the eta nodes in a group are assumed to have a unique label, and hence correspond to only one merge node in the relevant successor hyperblock.

When at eta group is evaluated, all the SSA trees in that eta group, regardless of whether they evaluate Boolean values, machine words or memory regions. The evaluation of all trees of a given type is modelled by recursion on the η-*bool*, η-*word* and η-*mem* lists for the given eta group.

To avoid specifying this recursion for each type of tree, a higher order function, $m\eta$-*group-t* is defined. This function has a polymorphic definition, and it represents a generic strategy for evaluating lists of SSA trees with eta node roots. Its parameters include: a list of SSA trees, which may be any of the three types of trees; a semantic function for trees of the same type as those in the first argument; the identity of the current and successor hyperblock; and the program state. It returns the state resulting from the evaluation of the given SSA trees, including updates to the state of the subsequent hyperblock.

Using such a recursive function, and assuming further semantic functions for the three different types of SSA trees with eta node roots — namely $m\eta$-*boolTree*, $m\eta$-*wordTree* and $m\eta$-*memTree* — the evaluation of an eta group is defined as follows:

[3]Using an (small-step) operational approach would allow a more natural representation of parallelism within the hyperblock, but it would be necessary to update the state of the current hyperblock after each node evaluation.

$m\eta\text{-}group\ g\ curr\ next\ \sigma_p \equiv$
$m\eta\text{-}group\text{-}t\ (\eta\text{-}bool\ g)\ m\eta\text{-}boolTree\ curr\ next$
$(m\eta\text{-}group\text{-}t\ (\eta\text{-}word\ g)\ m\eta\text{-}wordTree\ curr\ next$
$(m\eta\text{-}group\text{-}t\ (\eta\text{-}mem\ g)\ m\eta\text{-}memTree\ curr\ next\ \sigma_p))$

The semantic function for each type of SSA tree rooted at an eta node is a function of the following: the identity of the current hyperblock; that of the successor hyperblock; the name of the merge node in the successor hyperblock associated with the eta node root; the tree itself; and the program state in which it is to be evaluated. The definition of $m\eta\text{-}group\text{-}t$ is then a straight forward recursion on the list of SSA trees with eta node roots:

$m\eta\text{-}group\text{-}t\ []\ m\ curr\ next\ \sigma_p = \sigma_p$
$m\eta\text{-}group\text{-}t\ (t{\cdot}ts)\ m\ curr\ next\ \sigma_p =$
$m\ curr\ next\ (get\text{-}\gamma\text{-}name\ t)\ (get\text{-}tree\ t)\ (m\eta\text{-}group\text{-}t\ ts\ m\ curr\ next\ \sigma_p)$

In the above definition, m is the polymorphic semantic function for SSA trees with eta node roots of a given type. The three such semantic functions that $m\eta\text{-}group$ uses to instantiate m all have similar definitions.

One of those is the semantic function for SSA trees rooted with a Boolean valued eta node assumes a semantic function, *mBoolTree*, for Boolean valued SSA trees without an eta node root. This is used to evaluate the SSA tree associated with the eta node. The behaviour of the eta node itself is modelled by storing the resulting value in the $\sigma\text{-}bool$ field in the state associated with the successor hyperblock, specifically at the location identified by the $\gamma\text{-}name$ of the eta node.

$m\eta\text{-}boolTree\ curr\ next\ \gamma\text{-}name\ bTree\ \sigma_p \equiv$
$\mathsf{let}\ \sigma_b = \sigma_{p[next]}$
$\mathsf{in}\ \sigma_p[next := \mu(\sigma_b, \sigma\text{-}bool \mapsto (\sigma\text{-}bool\ \sigma_b)(\gamma\text{-}name := mBoolTree\ bTree\ \sigma_{p[curr]}))]$

The semantic functions for machine words and memory regions have respective definitions:

$m\eta\text{-}wordTree\ curr\ next\ \gamma\text{-}name\ wTree\ \sigma_p \equiv$
$\mathsf{let}\ \sigma_b = \sigma_{p[next]}$
$\mathsf{in}\ \sigma_p[next := \mu(\sigma_b, \sigma\text{-}word \mapsto (\sigma\text{-}word\ \sigma_b)(\gamma\text{-}name := mWordTree\ wTree\ \sigma_{p[curr]}))]$

$m\eta\text{-}memTree\ curr\ next\ \gamma\text{-}name\ mTree\ \sigma_p \equiv$
$\mathsf{let}\ \sigma_b = \sigma_{p[next]}$
$\mathsf{in}\ \sigma_p[next := \mu(\sigma_b, \sigma\text{-}mem \mapsto (\sigma\text{-}mem\ \sigma_b)(\gamma\text{-}name := mMemTree\ mTree\ \sigma_{p[curr]}))]$

SSA Tree Evaluation Without Eta Nodes

The strategy for evaluating each SSA tree without an eta node root is simply to recursively evaluate trees by a post-order evaluation of the tree, and then to update the state associated with the corresponding merge node in the subsequent hyperblock. The only distinction between the different types of trees is the types of values that they evaluate.

Semantic functions are assumed to exist for the primitive operators listed on page 78. The first argument to all such semantic functions is a datatype constructor that represents an operator. The semantic functions that produce

Boolean values are: *mBoolMonBoolOp*, for the monadic operator on Booleans (*BNot*); *mBoolMonNatOp* for the monadic operators on word values (*BNot*), and *mBoolDyNatOp* for dyadic operators respectively. For machine words, there are only dyadic operators, and a single function, *mWordOp*, suffices.

$$
\begin{aligned}
&mBoolTree\ (BoolMonBoolOp\ bOp\ tree\ bId)\ \sigma = \\
&\quad mBoolMonBoolOp\ bOp\ (mBoolTree\ tree\ \sigma) \\
&mBoolTree\ (BoolMonNatOp\ bOp\ tree\ bId)\ \sigma = \\
&\quad mBoolMonNatOp\ bOp\ (mWordTree\ tree\ \sigma) \\
&mBoolTree\ (BoolDyNatOp\ bOp\ tree_1\ tree_2\ bId)\ \sigma = \\
&\quad mBoolDyNatOp\ bOp\ (mWordTree\ tree_1\ \sigma)\ (mWordTree\ tree_2\ \sigma) \\
&mBoolTree\ (BoolMerge\ n\ bId)\ \sigma = \sigma\text{-}bool\ \sigma\ n
\end{aligned}
$$

$$
\begin{aligned}
&mWordTree\ (Const\ wVal\ wId)\ \sigma = wVal \\
&mWordTree\ (WordOp\ wOp\ tree_1\ tree_2\ wId)\ \sigma = \\
&\quad mWordOp\ wOp\ (mWordTree\ tree_1\ \sigma)\ (mWordTree\ tree_2\ \sigma) \\
&mWordTree\ (BoolToWord\ bTree\ wId)\ \sigma = \\
&\quad (if\ (mBoolTree\ bTree\ \sigma)\ then\ 0\ else\ 1) \\
&mWordTree\ (Load\ wTree\ mTree\ wId)\ \sigma = \\
&\quad (mMemTree\ mTree\ \sigma)\ (mWordTree\ wTree\ \sigma) \\
&mWordTree\ (WordMerge\ n\ wId)\ \sigma = \sigma\text{-}word\ \sigma\ n
\end{aligned}
$$

$$
\begin{aligned}
&mMemTree\ (Store\ vTree\ lTree\ mTree\ mId)\ \sigma = \\
&\quad (mMemTree\ mTree\ \sigma)((mWordTree\ lTree\ \sigma){:=}(mWordTree\ vTree\ \sigma)) \\
&mMemTree\ (MemMerge\ n\ mId)\ \sigma = \sigma\text{-}mem\ \sigma\ n
\end{aligned}
$$

Chapter 5

A Netlist-Level HDL

Contents

This chapter motivates and describes the development of a simple low-level hardware description language, which forms the basis of the hardware IR developed in the Chapter 6. It is based on an unpublished technical note by Richard Boulton - "A Semantics for a Simple Netlist Language" [Bou98]. Both the language presented here, and Boulton's original language, model hardware based upon a description of the structure of the circuit: the components and the connections between them. Components may be gates that implement logic functions. Additionally, in the language presented here, once a circuit has been defined, it may be reused as if it were a primitive component in subsequent component definitions.

The chapter begins by presenting the motivation for a netlist language, and identifies the features the language must provide in order to make as simple as possible the translation of a fragment of code in the software IR to an equivalent fragment of hardware IR. The abstract syntax of the language is developed in Section 5.3. Subsequent sections present a semantics given in

higher-order logic for circuits consisting only of combinational logic, which are illustrated by examples. The chapter concludes by showing an example of a correctness proof for a full adder. The treatment of bit vectors and sequential logic is deferred to the next chapter.

5.1 Need for a Netlist Language

Section 3.3 describes the use of higher-order logic for hardware verification. In that methodology, components and specifications are defined as constants in the logic. At some point, however, it is necessary to have a model that can be synthesised into a real hardware design. Current synthesis tools (for example, those provided by FPGA and ASIC manufacturers) typically synthesise VHDL or Verilog models, but not models written in HOL.

One approach to this problem is to have two versions of a design: a verifiable (or preferably verified!) model in HOL, and a similar model in a language supported by the relevant synthesis tool. Using this approach, correctness of the synthesised circuit is dependent upon the latter model being a faithful representation of the former. Even if an obvious relationship exists between the two models, maintaining the two versions is vulnerable to human error.

An alternative is to produce the input to the synthesis tool directly from the verified hardware model, expressed in a synthesisable subset of HOL [GIOS05]. This methodology seems ideal: specifications are refined into implementations, and the verified designs can then be compiled into synthesisable designs automatically. It should be noted that the designs are specific to the formulation of higher-order logic used by the compiler. The development of such a compiler proceeds by identifying a subset of HOL that can be synthesised. Pretty-printing functions for terms in that subset are then defined to yield constructs in the synthesisable HDL that are conjectured to have similar behaviour to the original HOL term.

The approach taken here is to define a simple netlist-style HDL with semantics expressed in HOL. The semantic function for circuits yields a HOL expression that characterises the behaviour of that circuit from its structural description, given in an abstract syntax. These expressions can then be used in a correctness proof of the circuit. Furthermore, a function can be defined between the (abstract) syntax of the netlist language and the (concrete) syntax of a synthesisable HDL.

The rationale for this approach is twofold. Firstly, since the abstract syntax is a structural description of the circuit, the translation to a synthesisable HDL design is conceptually simpler than the previous method, which requires translation from HOL terms to synthesisable HDL. By using a language designed for describing connections between components, the complexity of determining synthesisable HOL expressions, and their translation to a synthesisable HDL from HOL is avoided.

Secondly, the syntax of the language has been separated from the HOL logic. Thus a different semantics could be given to the language, it is possible to give a zero-delay and an event-based semantics to the same abstract syntax. Alternatively, a semantics of the language could be formulated in a different

logic, allowing component definitions (but not their correctness proofs) to be reused.

The trade-off for these benefits is that semantics functions and proofs are complicated by the fact that designs and their correctness proofs are based on a language embedded in HOL, rather than HOL itself.

5.2 Requirements from a Netlist language

A verification technique for hardware designs is described in Section 3.3.2. This technique assumes that a specification for the hardware exists. It also assumes the existence of a hardware model to be verified, or that there will be manual input to the process to produce a verified model from the specification. However, neither of these assumptions are valid in the case where a compiler is to generate both object code, and a hardware model for a function unit to support that object code. In the general case — where different parts of an expression may execute in different function units — if a hardware specification is to exist, it must be generated from a compiler intermediate representation. Generation from the IR is also necessary if the code to be compiled is to benefit from standard compiler optimisations. If the process is to be fully automatic, the compiler must also generate the synthesisable hardware model.

It is desirable for the netlist language to support most concepts in the software IR at similar levels of abstraction. This is necessary in order to simplify the translation of fragments of the software IR into the netlist language. For example, if the software IR semantics models a machine word as a list of bits, then the netlist language should support a similar representation. Furthermore, ensuring that the semantic objects in the software IR are similar to those in the netlist language reduces the proof burden when showing that the generated hardware model is 'equivalent' to the software IR fragment from which it was generated.

One problem that may arise using this approach is that it may be difficult to develop a hardware language that supports translation from a software IR. For example, if values are represented by natural numbers in the semantics of the software IR, rather than machine words, then it will be difficult to relate its semantics to those of a hardware language that uses bit strings of finite length to represent values. In this case, it would be easier to relate the semantic descriptions of the languages if the HDL supported natural numbers. However, supporting arbitrarily large natural numbers is likely to cause problems with synthesis. Although they are presented separately, in should be clear that the software IR in the previous chapter has been developed in parallel with the hardware representation that follows.

5.3 Abstract Syntax

5.3.1 An abstract syntax for hardware

In order to model elementary (or 'primitive') components, it is first necessary to decide how each component will be identified – each primitive needs a name.

It would be possible to identify primitives using a datatype constructor:
datatype *comp* = *And* | *Xor* | ...

It then might seem reasonable to define a function that, given a *comp*, would yield a semantic function describing the behaviour of that component. However, as primitives can have a varying number of external signals (*Not* has two, and *And* has three, for example), it would not be possible to make that function well typed. Instead, the signal names can be included as part of the construct:
datatype *comp* = *And a b o* | *Xor a b o* | ...

However, the datatype constructor approach makes it difficult to extend the set of primitives, and the language needs to support the definition of new components, which in turn should be usable as if they were themselves primitives.

Section 3.3.2 described the use of predicates in higher order logic to specify the behaviour of a component. The free variables in a predicate represent signal values, and the predicate is true for any context in which the free variables represent signal values that could be observed simultaneously on the external pins of a component. Here, a new type is introduced, that of a *binding* function, which binds signal names (which can be represented by character strings) to signal values:
types
binding = *sig-name*⇒*bit*

A binding represents the value of signals in a circuit at a particular point in time, and a predicate on a binding can be used to specify the behaviour of combinational logic. Unfortunately, this technique introduces an extra level of indirection when referring to signals, because a function application is necessary to find the value of each signal. Given a binding *cb*, the specification of an *And* gate given in Equation 3.1 becomes:

$$(cb\ A \wedge_b cb\ B) = cb\ O \tag{5.1}$$

Using strings to represent signal names in the netlist language, it is now possible to define a way of specifying the connections between the named pins of a component (which are the same for every instantiation of the same type of component) and signals that connect instantiations of components. A new type name is introduced to represent pins, *pin-name*. It is important to note that *sig-name* and *pin-name* are the same type. This is necessary because when a new component is defined, its external signals become named pins of that component. The purpose of distinguishing between them here is to make the following definitions clearer. A new datatype, *conn*, is then defined to represent the connection between a pin and a signal (or wire). The datatype distinguishes between the pins that serve as inputs to the component, and those that serve as outputs.

datatype *conn* =
In pin-name sig-name
| *Out pin-name sig-name* (5.2)

For the *And* gate, *A* and *B* are inputs, while *O* is an output. By specifying the direction of a pin connection, it becomes possible to formulate propositions

about the number of outputs that are driving a given signal. A well-formedness condition on the language requires that each signal be driven by only one output, avoiding the need to specify the semantics of *signal resolution*. Each instantiation of a component is described by a name that identifies the component being instantiated, and a *conn list* that shows which pins on the component (for example, *A*,*B* and *O* for the *And* gate) are connected to which signals.

record *inst* =
  Comp :: *prim-name*
  Conns :: *conn list*

A component definition is represented by a description of its external signals, Ext, and a list of instantiations of primitive components, $Insts$:

record *comp* =
  Ext :: *tyenv*
  Insts :: *inst list*

The type of Ext is *tyenv*, defined as a pair of lists of pin names: *pin-name list*× *pin-name list*. The first list represents the inputs of the design, and the second represents its outputs. Once a component has been defined, it is available as a primitive in subsequent designs, and the external signals represent the pins that will be available on the newly defined primitive. The standard functions *fst* and *snd* are used to access each item in the pair: a design *d* has inputs *fst (Ext d)*, and outputs *snd (Ext d)*. Signal names that appear in the design, but not listed as external signals are considered to be internal signals. Different instantiations of a primitive component are distinguished by their position in the $Insts$ list.

types
  design = (*prim-name* × *comp*) *list*

5.3.2 An example design of a Full Adder

Half adder

The half adder, illustrated in Figure 3.2 is described by the following component description in the netlist language:

```
halfAddCmp ≡ (|
    Ext= ([A,B],[C,S]),
    Insts = [
        (|Comp = And, Conns = [In A A, In B B, Out O C] |),
        (|Comp = Xor, Conns = [In A A, In B B, Out O S] |)
    ]
|)
```
(5.3)

The *And* and *Xor* gate are assumed to be defined primitives with the obvious behaviour.

Full adder

The full adder, illustrated in Figure 5.1 is described by the following component description:

fullAddCmp ≡ (|

```
  Ext= ([A,B,Cin],[Cout,S]),
  Insts= [
    (|Comp = halfAdd, Conns = [In A A, In B B, Out C C1, Out S S1] |),
    (|Comp = halfAdd, Conns = [In A S1, In B Cin, Out C C2, Out S S] |),
    (|Comp = Xor, Conns = [In A C1 , In B C2, Out O Cout] |)
  ]
|)
```

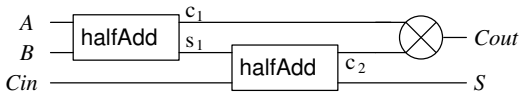


Figure 5.1: Full adder component

5.4 Netlist Semantics

Section 3.3.2 shows how circuits, represented by HOL terms of type *bool*, can be constructed by composition and abstraction. In the netlist language, components are represented by functions of type *binding⇒bool* where the term is true if it is applied to a binding that represents a set of signals that could be observed simulataneously on the external pins of the component. A new type is introduced to represent component behaviour:

types

behav = *binding⇒bool*

5.4.1 Primitives

The netlist language has an initial set of primitive components from which circuits can be composed. The initial primitives do not have a defined internal structure, but their behaviour is defined. For example, the *And* gate (as given in Equation 5.1) and the *Xor* gate are defined with the following behaviours:

$$MAnd \equiv (\lambda cb.\ (cb\ A \wedge_b cb\ B) = cb\ O) \tag{5.4}$$
$$MXor \equiv (\lambda cb.\ (cb\ A \oplus_b cb\ B) = cb\ O) \tag{5.5}$$

The constant is prefixed with the letter M to indicate that it is a semantic function: it represents the *meaning* of an *And* gate. The definition imposes a constraint on the values of the binding at *A*, *B*, and *O*.

5.4.2 Abstraction and Instantiation Semantics

Two methods of abstraction are introduced in Section 3.3.2, both of which need to be reformulated slightly for the deeper embedding of the netlist language. The first is the use of existential quantification to hide signals that are internal to a component. Recall that the *binding* type is a function from signal names to values, and that behaviour is represented by the type *behav*: a function

from *binding* to *bool*. In order to hide a value, it is necessary to define a new behaviour, introducing an existential quantifier for the hidden variable. The *internalise* function is defined as a recursive function that takes a list of internal signals and a component's behaviour.

$$internalise\ []\ be = be$$
$$internalise\ (s{\cdot}ss)\ be = internalise\ ss\ (\lambda cb.\ \exists\, new\text{-}sig.\ be\ (cb(s := new\text{-}sig)))$$

If there are no internal signals, the resulting behaviour is identical to the behaviour supplied, that is, the bindings that satisfy the supplied behavioural model are those that satisfy the result. If there are internal signals, the new behaviour binds the name of the hidden signal to the existentially quantified variable. The resulting behavioural model is unaffected by the named signal and is satisfied by any binding that satisfies the model provided. Additionally, the resulting model is also satisfied by any binding that differs only in the values of the named internal signals.

The second method of abstraction introduced in Section 3.3.2 is behavioural abstraction where a component is referred to by name, rather than simply duplicating the internal structure of that component in each design in which it is used. By associating the names of previously defined components with their external behaviour, it is possible to define and reason about a complex component without reference to the internal structure of the components from which it is defined. A new type is introduced to associate the names of defined components with their external behaviour.

types $dynenv = prim\text{-}name \Rightarrow behav$

Note that it is called *dynenv*, short for dynamic environment, because it models the dynamic behaviour of components (as opposed to simply maintaining their static properties, such as connections) — the fact that it changes during the elaboration of a design is coincidental. Using a behavioural model of a component, without reference to its internal structure has already been seen in the treatment of primitive components. To find the semantics of a design, an initial *dynenv* is constructed that associates the name of each primitive with its behaviour. As new components are defined, their behaviour is added to the dynamic environment. Instantiations of previously defined components can therefore be treated similarly to instantiations of primitives, and in the sequel, the term primitive is used to refer to both (reserving the term 'initial set of primitives' to refer to components for which no internal structure is defined).

5.4.3 Composition Semantics

As in Section 3.3.2, composition of components is modelled by conjunction. The constraint imposed on signal values by two components is that the same signal values must simultaneously satisfy the constraints imposed by both components. A function called *combine* is defined to represent the composition of two circuits, c_1 and c_2. Its result is the behaviour of the two circuits composed together.

$$combine\ c_1\ c_2 \equiv \lambda bind.\ c_1\ bind \wedge c_2\ bind$$

In a shallow embedding in HOL, different values could be constrained by simply using different free variables. However, in the definition of *MAnd*, A, B and O are not free — they are the names of pins on the And gate and are fixed. A gate that can only be connected to signals with a fixed name clearly isn't very useful! A function, *bindPins*, is used to provide *signal renaming*: to rename signals to the pins to which they are connected.

$$\begin{array}{l} bindPins :: (sig\text{-}name \times sig\text{-}name)\ list \Rightarrow binding \Rightarrow binding \\ bindPins\ cs\ cb \equiv foldl\ (\lambda cb'\ (pn,\ sn).\ cb'(pn := cb\ sn))\ cb\ cs \end{array} \tag{5.6}$$

The first argument of *bindPins* is a list of pairs of signal names. The first name in each pair is a pin name, while the second is the name of a signal. The second argument, a *binding*, represents the values of the signals within a component. The result is a binding where the signal value for each pin has been replaced by the corresponding value for the signal to which it is connected. This binding can be applied to a primitive component.

Two auxiliary functions are defined to support the use of *bindPins*. The first, $Mconn$, returns a pair where the first element is a pin name and the second is the signal name to which that pin is connected. The definition is simple, because discarding the datatype constructor of the connection that indicates the pin's direction, and constructing a pair, is all that is required.

$Mconn :: conn \Rightarrow (pin\text{-}name \times sig\text{-}name)$
$Mconn\ (In\ pn\ sn) = (pn,\ sn)$
$Mconn\ (Out\ pn\ sn) = (pn,\ sn)$

The result of $Mconn$ is ready to be used as an element of a list that is used as the first argument to *bindPins*. The full list is found by applying $Mconn$ to every connection on a component being instantiated. Thus the first argument to *bindPins* can be found by applying a second auxiliary function, $Mconns$, to the component's instantiation:

$Mconns \equiv map\ Mconn \circ Conns$

The behaviour of a given instantiation of a primitive (or previously defined component) can now be found by using a dynamic environment to find the behaviour of the primitive. The *bindPins* function described previously can then be used to find the primitives behaviour in the context of the component being defined. This means finding the constraint on signal values that the primitives instantiation imposes, by mapping signal names in the component to pin names of the primitive. Thus the behaviour of a components instantiation can be found by the following function:
$Minst :: dynenv \Rightarrow inst \Rightarrow behav$
$Minst\ denv\ cmp \equiv$
$\lambda bind.\ (denv\ (Comp\ cmp))\ (bindPins\ (Mconns\ cmp)\ bind)$

5.4.4 Component Semantics

Thus far, an initial set of primitives has been given. The semantics of component instantiation has been shown and seen to be similar for both primitives and defined components. It is now possible to show how a new component can be defined in order that it may later be used as a primitive in subsequent component definitions or as a top level design entity. A function *Minsts* is defined

to find the behaviour of a list of component instantiations. It can be applied to the *Insts* field of a component record.
Minsts is ≡ (*foldl combine* (λ*b*. *True*)) *o* (*map* (*Minst is*))

The expression *map* (*Minst is*) yields a list of behaviours, one for each component instantiation. By combining these behaviours, the signal constraints imposed by the new component can be found. Suppose *is* is an empty list, a degenerate component definition that does not instantiate any primitives or connect any signals. The *map* expression in *Minsts* is clearly an empty list, then $Minsts[] = foldl\ combine\ (\lambda b.True))\ [] = (\lambda b.True)$. An empty list of component instantiations imposes no constraints on signals. If *is* is non-empty, then the constraint will be the (distributed) conjunction of constraints imposed by each component instantiation. Although *Minsts* represents the behaviour of a list of component instantiations, its result is not the behaviour that should be added to the dynamic environment. Recall that the dynamic environment associates primitives and components with their *external* behaviour, and that internal signals between component instantiations in a new component definition should not be visible from another component. Assuming a function *pinsComp* that returns a list of signals that are internal to a component, the behaviour of a new component *cmp*, that uses other components and primitives from a given dynamic environment *denv*, is given by the semantic function *Mcomp*:

Mcomp cmp denv ≡ *internalise* (*pinsComp cmp*) (*Minsts denv* (*Insts cmp*)) (5.7)

The signals that are internal to a component can be found from the list of external signatures (given in the abstract syntax for a component) and the signals named in the connections of each component instantiation. The signals used by a given component instantiation are given by the *pinsInst* function, which uses an auxiliary pinExpr function to get the signal name from each pin connection:

sigExpr (*In pn sn*) = *sn*
sigExpr (*Out pn sn*) = *sn*

pinsInst ≡ *map sigExpr* ∘ *Conns*

The external signals of a component are given by *fst (Ext d)* $\cup_l$ *snd (Ext d)*, where $\cup_l$ is a set-theoretic union operation on lists that removes duplicates. *pinInsts*, defined below gives a (duplicate free) list of all the signals named by a given list of component instantiations. Finally, assuming a similar set-theoretic minus operation on lists, $-_l$, the internal signals are given by *pinsComp*, the set of all signals minus the set of external signals in a component definition.

pinsInsts cs ≡ *foldl op* $\cup_l$ [] (*map pinsInst cs*)

pinsComp d ≡ *pinsInsts* (*Insts d*) $-_l$ (*fst* (*Ext d*) $\cup_l$ *snd* (*Ext d*))

5.5 Semantics of a Full Adder design

To illustrate the semantics of the netlist language, this section shows the result of applying the semantic functions to two sample component definitions, namely the half adder and the full adder definitions given previously.

5.5.1 Half Adder Semantics

The component definitions assume that *And* and *Xor* gates have been defined as primitives. Using the behavioural definitions of these gates given in Equation 5.4 and Equation 5.5 respectively, an initial environment (of type *dynenv*) is constructed as follows:

$$Mprim\ p \equiv ((\lambda prim\ bind.\ False)(\mathsf{And} := MAnd, \mathsf{Xor} := MXor))\ p$$

The behaviour of the half adder component, whose definition is given in Equation 5.3, is found by applying the semantic function for a component to that definition and the environment of primitives.

$$MhalfAdd = Mcomp\ halfAddCmp\ Mprim$$

Unfolding the definition of *Mcomp* yields:

$$\ldots = internalise\ (pinsComp\ halfAddCmp)\ (Minsts\ Mprim\ (Insts\ halfAddCmp))$$

Recall that pinsComp yields a list of signals that are internal to a component. It does this by finding which signals a component uses, but does not declare as an external signal. By examining the diagram of the half adder Figure 3.2 it can be seen that it has no internal signals, and expanding the definition of pinsComp shows that it yields the empty list, because all the signals named in the component definition, given by pinsInsts (Insts halfAddCmp), are listed as external signals. Thus the following yields the empty list:

$$pinsInsts\ (Insts\ halfAddCmp) -_l (fst\ (Ext\ halfAddCmp) \cup_l snd\ (Ext\ halfAddCmp))$$

Since there are no internal signals, and $internalise\ []\ b = b$, the *MhalfAdd* expression can now be reduced to:

$$\begin{aligned} MhalfAdd &= (Minsts\ Mprim\ (Insts\ halfAddCmp)) \\ &\ldots = (foldl\ combine\ (\lambda b.\ True) \circ map\ (Minst\ Mprim))\ (Insts\ halfAddCmp) \end{aligned} \tag{5.8}$$

Unfolding *Minsts* shows that the next step must be to apply *Minst* to find the behaviour of each (primitive) component instantiation. The behaviour of each instantiable component is given in the dynamic environment *Mprim*, which forms the first argument to *Minst*. The first primitive instantiation in the half adder is that of the *And* gate. The function *Minst* is applied, and the name of the component to be instantiated (*And*) is replaced by its behaviour given by the dynamic enviroment, *Mprim*:

$$Minst\ Mprim\ (\!|Comp = \mathsf{And},\ Conns = [In\ \mathsf{A}\ \mathsf{A},\ In\ \mathsf{B}\ \mathsf{B},\ Out\ \mathsf{O}\ \mathsf{C}],\ \ldots = ()|\!) \tag{5.9}$$

$$\begin{aligned} \ldots = (\lambda cb.\ MAnd\ (bindPins\ (Mconns \\ (\!|Comp = \mathsf{And},\ Conns = [In\ \mathsf{A}\ \mathsf{A},\ In\ \mathsf{B}\ \mathsf{B},\ Out\ \mathsf{O}\ \mathsf{C}]|\!))\ cb)) \end{aligned}$$

The *Mconns* function then pairs the pin names of the *And* gate (*A*, *B* and *O*) with the signals names to which it is connected (*A*, *B* and *C*). Note that a signal sharing the same name as a pin is no more problematic than an argument sharing the same name as a formal parameter in a high-level programming language — the two names are separate in the semantics. [1]

[1] This is not to say that there are no issues with sharing names. In a language with call-by-value-return semantics, such as Ada, what should the value of a variable passed as output parameters in two different argument positions be after executing the function? Likewise, connecting the same signal to different pins may result in an inconsistent model — what happens if the same signal is connected to both the input and output of an inverter?

$\ldots = (\lambda cb.\ MAnd\ (bindPins\ [(A,A),\ (B,B),\ \ (O,C)]\ cb))$
The *bindPins* function updates the *binding* that is passed to *MAnd*. It sets the value of the binding at O, used by the definition of *MAnd*, to be the value of the binding at C. Note that where a pin has the same name as the signal to which it is connected, we have $cb = cb(A := cb\ A)$, and $cb = cb(B := cb\ B)$, and consequently the signal renaming can be expressed as follows:

$$\ldots = (\lambda cb.\ MAnd\ (cb(O := cb\ C)))$$

The definition of *MAnd* can then be unfolded, and the resulting expression simplified:

$$\ldots = (\lambda cb.\ (cb\ A \wedge_b cb\ B) = cb\ C) \tag{5.10}$$

The method for finding the semantics of the *Xor* instantiation is similar to that of the *And* gate, and the details are omitted here.

$$\begin{aligned} &Minst\ Mprim\ (\!|Comp = Xor,\ Conns = [In\ A\ A,\ In\ B\ B,\ Out\ O\ S]|\!) = \\ &\qquad (\lambda cb.\ (cb\ A \oplus_b cb\ B) = cb\ S) \end{aligned} \tag{5.11}$$

Now the behaviour of the two instantiated primitives has been found, it is possible to return to Equation 5.8 and subsitute in a list of the behaviours of the two primitives that are instantiated half adder, as shown in Equation 5.12.

$$\begin{aligned} &(foldl\ combine\ (\lambda b.\ True) \circ map\ (Minst\ Mprim))\ (Insts\ halfAddCmp) = \\ &\quad foldl\ combine\ (\lambda b.\ True)\ [\\ &\qquad (\lambda cb.\ (cb\ A \wedge_b cb\ B) = cb\ C), \\ &\qquad (\lambda cb.\ (cb\ A \oplus_b cb\ B) = cb\ S)\ ] \end{aligned} \tag{5.12}$$

The combine function is then applied iteratively to $(\lambda b.\ True)$ and the behaviours in the list. Combining $(\lambda b.\ True)$ with the *And* gate behaviour (Equation 5.10), simplifies to just the behaviour of the *And* gate, because $(\lambda b.\ True) \wedge (\lambda b.\ P\ b) = (\lambda b.\ P\ b)$. Applying combine again to this term, and the behaviour of the *Xor* gate gives the conjunction of the two behaviours (Equation 5.13).

$$MhalfAdd = (\lambda cb.\ (cb\ A \wedge_b cb\ B) = cb\ C \wedge (cb\ A \oplus_b cb\ B) = cb\ S) \tag{5.13}$$

5.5.2 Full Adder Semantics

This section shows how the semantics of the full adder can be derived from it's structural definition. The focus is on the features of the language that were not illustrated in the half adder example in the previous section, notably: the treatment of internal signals; and the use of a previously defined component in a new component.

The full adder is composed of two half adders and an Xor gate. The half adders are instantiated by treating the half adder design as a primitive, rather than by simply duplicating the design twice, as part of the full adder design. This allows the semantics of the half adder, *MhalfAdd* — as found in the previous section — to be reused here. In order to use it as a primitive, it is first necessary to define a new dynamic environment including the semantics of both the existing

primitives, and the semantics of the half adder. This is done by using a function update to the existing primitive environment:

$$env2 \equiv Mprim(\mathsf{halfAdd} := MhalfAdd) \tag{5.14}$$

The semantics of the full adder can then be found by applying *Mcomp* to the definition of the full adder, and the new environment. Note that the environment should define a semantics for each type of primitive listed in the *Insts* field in the component's definition.

$$\begin{aligned} MfullAdd &= Mcomp\ fullAddCmp\ env2 \\ &\ldots = internalise\ (pinsComp\ fullAddCmp)\ (Minsts\ env2\ (Insts\ fullAddCmp)) \end{aligned} \tag{5.15}$$

As in the previous section, the signals that are internal to the component definition, denoted by *pinsComp fullAddCmp* are found first. Although *pinsComp* provides a list of signals — more specifically, $(pinsComp\ fullAddCmp) = [\mathsf{S_1}, \mathsf{C_1}, \mathsf{C_2}]$ — their order is unimportant here, and noting that $set\ (pinsComp\ fullAddCmp) = \{\mathsf{S_1}, \mathsf{C_1}, \mathsf{C_2}\}$ is sufficient.

The evaluation of the second argument is to *internalise* follows. Unfolding *Minsts* (as in Equation 5.8) and applying *Insts* to the full adder component definition:

$$\begin{aligned} &Minsts\ env2\ (Insts\ fullAddCmp) = \\ &\quad foldl\ combine\ (\lambda b.\ True)\ (map\ (Minst\ env2)\ (Insts\ fullAddCmp)) \\ &\ldots = foldl\ combine\ (\lambda b.\ True) \\ &\quad (map\ (Minst\ env2) \\ &\quad\ [(\!|Comp = \mathsf{halfAdd},\ Conns = [In\ \mathsf{A}\ \mathsf{A},\ In\ \mathsf{B}\ \mathsf{B},\ Out\ \mathsf{C}\ \mathsf{C_1},\ Out\ \mathsf{S}\ \mathsf{S_1}]|\!), \\ &\quad\ (|Comp = \mathsf{halfAdd},\ Conns = [In\ \mathsf{A}\ \mathsf{S_1},\ In\ \mathsf{B}\ \mathsf{C_{in}},\ Out\ \mathsf{C}\ \mathsf{C_2},\ Out\ \mathsf{S}\ \mathsf{S}]|), \\ &\quad\ (|Comp = \mathsf{Xor},\ Conns = [In\ \mathsf{A}\ \mathsf{C_1},\ In\ \mathsf{B}\ \mathsf{C_2},\ Out\ \mathsf{O}\ \mathsf{C_{out}}]|)]) \end{aligned}$$

It can be seen that a previously defined component, the half adder, can be used as if it were just another primitive by simply referring to it by name. The next step is to apply Minst to each 'primitive', instantiated in a full adder. The difference between this step, and the steps presented in Equation 5.9 to Equation 5.12 is that here the env2 environment is used, in order to make use of the environment containing the half adder and the initial primitives, introduced in Equation 5.14. Again, Minst is used to find the semantics of each component, and foldl and combine are used to collect the resulting terms into conjuncts:

$$\begin{aligned} &\ldots = (\lambda bind. \\ &\quad MhalfAdd\ (bind(\mathsf{A} := bind\ \mathsf{A},\ \mathsf{B} := bind\ \mathsf{B},\ \mathsf{C} := bind\ \mathsf{C_1},\ \mathsf{S} := bind\ \mathsf{S_1})) \wedge \\ &\quad MhalfAdd\ (bind(\mathsf{A} := bind\ \mathsf{S_1}\ ,\ \mathsf{B} := bind\ \mathsf{C_{in}},\ \mathsf{C} := bind\ \mathsf{C_2},\ \mathsf{S} := bind\ \mathsf{S})) \wedge \\ &\quad MXor\ (bind(\mathsf{A} := bind\ \mathsf{C_1},\ \mathsf{B} := bind\ \mathsf{C_2},\ \mathsf{O} := bind\ \mathsf{C_{out}}))) \end{aligned}$$

Unlike the presentation of the half adder semantics, the MXor expression has not been unfolded here to ease comparison with the component definition. The arguments to internalise can now be substituted into Equation 5.15:

$$\begin{aligned} &MfullAdd = internalise\ [\mathsf{S_1}, \mathsf{C_1}, \mathsf{C_2}]\ \ (\lambda bind. \\ &\quad MhalfAdd\ (bind(\mathsf{A} := bind\ \mathsf{A},\ \mathsf{B} := bind\ \mathsf{B},\ \mathsf{C} := bind\ \mathsf{C_1},\ \mathsf{S} := bind\ \mathsf{S_1})) \wedge \\ &\quad MhalfAdd\ (bind(\mathsf{A} := bind\ \mathsf{S_1}\ ,\ \mathsf{B} := bind\ \mathsf{C_{in}},\ \mathsf{C} := bind\ \mathsf{C_2},\ \mathsf{S} := bind\ \mathsf{S})) \wedge \\ &\quad MXor\ (bind(\mathsf{A} := bind\ \mathsf{C_1},\ \mathsf{B} := bind\ \mathsf{C_2},\ \mathsf{O} := bind\ \mathsf{C_{out}}))) \end{aligned}$$

The aim is introduce an existentially quantified variable for each internal signal, and ensure that all references to the value of that signal 'within' the component refer to the quantified variable, irrespective of the value of a signal with the same name 'outside' the component.

Application of internalise introduces the existentially quantified variable by recursion on the list of internal signals. The following expression shows the result of one unfolding of internalise, in which only one signal has been 'internalised'. Note that the binding of signal values used in the inner lambda expression has been updated, such that references to those internal signals will refer to the existentially quantified variables instead.

$$
\begin{array}{l}
\ldots = \mathit{internalise}\ [C_1,\ C_2]\ \ (\lambda \mathit{bind}.\ \exists\, s1. \\
\quad (\lambda \mathit{bind}. \\
\quad\quad \mathit{MhalfAdd}\ (\mathit{bind}(A := \mathit{bind}\ A,\ B := \mathit{bind}\ B,\ C := \mathit{bind}\ C_1,\ S := \mathit{bind}\ S_1)) \wedge \\
\quad\quad \mathit{MhalfAdd}\ (\mathit{bind}(A := \mathit{bind}\ S_1\ ,\ B := \mathit{bind}\ C_{in},\ C := \mathit{bind}\ C_2,\ S := \mathit{bind}\ S)) \wedge \\
\quad\quad \mathit{MXor}\ (\mathit{bind}(A := \mathit{bind}\ C_1,\ B := \mathit{bind}\ C_2,\ O := \mathit{bind}\ C_{out}))) \\
\quad (\mathit{bind}(S_1 := s1)))
\end{array}
$$

Using the rule $(\lambda \mathit{bind}.\ (\mathit{bind}(x := a))\ x) = (\lambda \mathit{bind}.\ a)$*, the above expression can be simplified:*

$$
\begin{array}{l}
\ldots = \mathit{internalise}\ [C_1,\ C_2]\ \ (\lambda \mathit{bind}.\ \exists\, s1. \\
\quad\quad \mathit{MhalfAdd}\ (\mathit{bind}(A := \mathit{bind}\ A,\ B := \mathit{bind}\ B,\ C := \mathit{bind}\ C_1,\ S := s1)) \wedge \\
\quad\quad \mathit{MhalfAdd}\ (\mathit{bind}(A := s1\ ,\ B := \mathit{bind}\ C_{in},\ C := \mathit{bind}\ C_2,\ S := \mathit{bind}\ S)) \wedge \\
\quad\quad \mathit{MXor}\ (\mathit{bind}(A := \mathit{bind}\ C_1,\ B := \mathit{bind}\ C_2,\ O := \mathit{bind}\ C_{out})))
\end{array}
$$

Further unfolding of internalise quantifies the remaining internal signals, and the term reduces to the following expression, completing the evaluation of the component's semantics.

$$
\begin{array}{l}
\ldots = \\
\quad (\lambda \mathit{bind}.\ \exists\, c1\ c2\ s1. \\
\quad\quad \mathit{MhalfAdd}\ (\mathit{bind}(A := \mathit{bind}\ A,\ B := \mathit{bind}\ B,\ C := c1,\ S := s1)) \wedge \\
\quad\quad \mathit{MhalfAdd}\ (\mathit{bind}(A := s1\ ,\ B := \mathit{bind}\ C_{in},\ C := c2,\ S := \mathit{bind}\ S)) \wedge \\
\quad\quad \mathit{MXor}\ (\mathit{bind}(A := c1,\ B := c2,\ O := \mathit{bind}\ C_{out})))
\end{array}
$$

5.6 Correctness of a Full Adder design

An important requirement of the netlist language is that it should be possible to reason about the correctness of components described in the language, with respect to a specification. This is also essential for showing the correctness of a compiler that targets the netlist language. It is necessary for the specification of a component to be given in the meta-language used to describe the semantics of the netlist language: in this case, HOL. This is because the netlist language does not provide the features of a specification language.

The next sections describe how the full adder design example used throughout this chapter was mechanically verified using a theorem prover. For all the main features of the netlist language – component instantiation; composition; signal hiding and renaming; and abstraction of primitives – they show how a component design using those features can be verified.

5.6.1 Half Adder correctness

In order to show the correctness of the half adder, it is necessary to show that its semantics, given in Equation 5.13, satisfies its specification. The verification condition to be proven is as follows (its explanation is below),

where the premise (above the line) represents the semantics of the half adder model, and the conclusion (below the line) represents its specification:

$$\boxed{\mathit{halfAddCorrect}}\ \frac{\mathit{MhalfAdd}\ (\mathit{cb}(\mathsf{A} := a,\ \mathsf{B} := b,\ \mathsf{C} := c,\ \mathsf{S} := s))}{\ll[a]\gg + \ll[b]\gg = \ll[c,\ s]\gg}$$

The term $\mathit{cb}(\mathsf{A} := a,\ \mathsf{B} := b,\ \ldots)$ is used to refer to a binding in which the value of signal A is a, the value of signal B is b, and so on. The statements about the value of the binding at given signals could be written more conventionally using separate premises as follows:

$$\frac{\begin{array}{l}\mathit{MhalfAdd}\ \mathit{cb}\\ \mathit{cb}\ \mathsf{A} = a\\ \mathit{cb}\ \mathsf{B} = b\\ \mathit{cb}\ \mathsf{C} = c\\ \mathit{cb}\ \mathsf{S} = s\end{array}}{\ll[a]\gg + \ll[b]\gg = \ll[c,\ s]\gg}$$

However, the former notation is used here as it is more concise. Furthermore, it is clear than the signal terms can be permuted if they refer to different signals, for example: $\mathit{cb}(\ldots, \mathsf{A} := a, \mathsf{B} := b) = \mathit{cb}(\ldots, \mathsf{B} := b, \mathsf{A} := a)$. An alternative way of viewing the verification condition is by unfolding the semantics of the half adder (using Equation 5.13):

$$\frac{(a \wedge_b b) = c \wedge (a \oplus_b b) = s}{\ll[a]\gg + \ll[b]\gg = \ll[c,\ s]\gg}$$

The task is to prove the judgement above. With only four *bit* variables, it is clear that the circuit is trivial, and examining each combination of values is certainly the simplest, if not the only, way to complete the proof.

Recall from Section 3.1.2 that the bits are not represented by Boolean values in this work, but as a type *bit* with two datatype constructors **0** and **1**. Thus the verification condition cannot be discharged using Boolean simplification alone. Instead, case distinction must be performed on each *bit* variable.

5.6.2 Full Adder correctness

Verification of the full adder requires proving the following statement of correctness:

$$\boxed{\mathit{fullAddCorrect}}\ \frac{\begin{array}{l}(\lambda \mathit{cb}.\ (\exists\ \mathit{c1}\ \mathit{c2}\ \mathit{s1}.\\ \mathit{MhalfAdd}\ (\mathit{cb}(\mathsf{A} := \mathit{cb}\ \mathsf{A},\ \mathsf{B} := \mathit{cb}\ \mathsf{B},\ \mathsf{C} := \mathit{c1},\ \mathsf{S} := \mathit{s1})) \wedge\\ \mathit{MhalfAdd}\ (\mathit{cb}(\mathsf{A} := \mathit{s1},\ \mathsf{B} := \mathit{cb}\ \mathsf{C}_{\mathit{in}},\ \mathsf{C} := \mathit{c2},\ \mathsf{S} := \mathit{cb}\ \mathsf{S})) \wedge\\ \mathit{MXor}\ (\mathit{cb}(\mathsf{A} := \mathit{c1},\ \mathsf{B} := \mathit{c2},\ \mathsf{O} := \mathit{cb}\ \mathsf{C}_{\mathit{out}})))\ \mathit{cb}\end{array}}{\ll[\mathit{cb}\ \mathsf{A}]\gg + \ll[\mathit{cb}\ \mathsf{B}]\gg + \ll[\mathit{cb}\ \mathsf{C}_{\mathit{in}}]\gg = \ll[\mathit{cb}\ \mathsf{C}_{\mathit{out}},\ \mathit{cb}\ \mathsf{S}]\gg}$$

The correctness proof is more complex than that of the half adder because of the existential quantifier; the use of a previously defined component rather than merely initial primitives; and the extra signals. Although its clear that the full adder is simple enough to apply the (somewhat brute force) technique of building a truth table as in the previous section, this section provides a more interesting proof.

Firstly, the proof shows how the correctness proof of a larger component may use as a lemma the correctness statement of a smaller component that it instan-

tiates. This allows the instantiated component to be replaced by a different component satisfying the same specification, and — so long as the replacement has been verified — the correctness proof of the larger (instantiating) component need not be modified. Secondly, the proof allows the enjoyment of one of the most compelling benefits of using theorem proving instead of model checking: the process of proving its correctness provides extra insight into the design, suggesting an alternative implementation.

Rather than proving statement of correctness above directly, the proof proceeds by proving the following, slightly simpler, lemma where the existential quantifier has been removed:

$$\frac{\begin{array}{l} \mathit{MhalfAdd}\ (\mathit{cb}(\mathsf{A} := \mathit{cb}\ \mathsf{A},\ \mathsf{B} := \mathit{cb}\ \mathsf{B},\ \mathsf{C} := \mathit{c1},\ \mathsf{S} := \mathit{s1})) \wedge \\ \mathit{MhalfAdd}\ (\mathit{cb}(\mathsf{A} := \mathit{s1},\ \mathsf{B} := \mathit{cb}\ \mathsf{C}_{\mathit{in}},\ \mathsf{C} := \mathit{c2},\ \mathsf{S} := \mathit{cb}\ \mathsf{S})) \wedge \\ \mathit{MXor}\ (\mathit{cb}(\mathsf{A} := \mathit{c1},\ \mathsf{B} := \mathit{c2},\ \mathsf{O} := \mathit{cb}\ \mathsf{C}_{\mathit{out}})) \end{array}}{\ll[\mathit{cb}\ \mathsf{A}]\gg + \ll[\mathit{cb}\ \mathsf{B}]\gg + \ll[\mathit{cb}\ \mathsf{C}_{\mathit{in}}]\gg = \ll[\mathit{cb}\ \mathsf{C}_{\mathit{out}},\ \mathit{cb}\ \mathsf{S}]\gg}$$

It is helpful to 'label' terms representing the constraint on signal values imposed by each component that is instantiated. These labels can be used throughout the proof to distinguish between the two half adders, and to clarify what proof steps are being taken. The terms are simply the conjuncts of the lemma's premise:

halfAdd1: $\mathit{MhalfAdd}\ (\mathit{cb}(\mathsf{A}{:=}\mathit{cb}\ \mathsf{A},\ \mathsf{B}{:=}\mathit{cb}\ \mathsf{B},\ \mathsf{C}{:=}\mathit{c1},\ \mathsf{S}{:=}\mathit{s1}))$
halfAdd2: $\mathit{MhalfAdd}\ (\mathit{cb}(\mathsf{A}{:=}\mathit{s1},\ \mathsf{B}{:=}\ \mathit{cb}\ \mathsf{C}_{\mathit{in}},\ \mathsf{C}{:=}\mathit{c2},\ \mathsf{S}{:=}\mathit{cb}\ \mathsf{S}))$
xor: $\mathit{MXor}\ (\mathit{cb}(\mathsf{A}{:=}\mathit{c1},\ \mathsf{B}{:=}\mathit{c2},\ \mathsf{O}{:=}\mathit{cb}\ \mathsf{C}_{\mathit{out}}))$

Using the specification of the half adder, a mathematical relationship between signal values can be stated. From the *halfAdd1* instantiation and its specification, the sum of externals signals A and B, when each is interpreted as the natural number 0 or 1, is the value of the bit vector $[\mathit{c1},\ \mathit{s1}]$ when interpreted as a natural number in the range 0–3 (most significant bit first).

$\ll[\mathit{cb}\ \mathsf{A}]\gg + \ll[\mathit{cb}\ \mathsf{B}]\gg = \ll[\mathit{c1},\ \mathit{s1}]\gg$ **using** *halfAdd1*
$\ll[\mathit{s1}]\gg + \ll[\mathit{cb}\ \mathsf{C}_{\mathit{in}}]\gg = \ll[\mathit{c2},\ \mathit{cb}\ \mathsf{S}]\gg$ **using** *halfAdd2*

Adding the respective sides of the above equations together gives:

$$\ll[\mathit{cb}\ \mathsf{A}]\gg + \ll[\mathit{cb}\ \mathsf{B}]\gg + \ll[\mathit{s1}]\gg + \ll[\mathit{cb}\ \mathsf{C}_{\mathit{in}}]\gg = \ll[\mathit{c1},\ \mathit{s1}]\gg + \ll[\mathit{c2},\ \mathit{cb}\ \mathsf{S}]\gg$$

Subtracting $\ll[\mathit{s1}]\gg$ from each side makes the left hand side equal to that in the consequent of the required lemma:

$$\ll[\mathit{cb}\ \mathsf{A}]\gg + \ll[\mathit{cb}\ \mathsf{B}]\gg + \ll[\mathit{cb}\ \mathsf{C}_{\mathit{in}}]\gg = \ll[\mathit{c1},\ \mathbf{0}]\gg + \ll[\mathit{c2},\ \mathit{cb}\ \mathsf{S}]\gg \tag{5.16}$$

It remains to be shown that $\ll[\mathit{c1},\ \mathbf{0}]\gg + \ll[\mathit{c2},\ \mathit{cb}\ \mathsf{S}]\gg = \ll[\mathit{cb}\ \mathsf{C}_{\mathit{out}},\ \mathit{cb}\ \mathsf{S}]\gg$. Given *xor*, which can be expanded to show that $\mathit{cb}\ \mathsf{C}_{\mathit{out}} = \mathit{c1} \oplus \mathit{c2}$, this is very nearly the required result. However, if *c1* and *c2* both have the value **1**, then the sum of the terms on the left hand side cannot be expressed in the two bit output of the full adder, $[\mathit{cb}\ \mathsf{C}_{\mathit{out}},\ \mathit{cb}\ \mathsf{S}]$. Fortunately, this situation can never occur, i.e. we always have: $\mathrm{c1} = \mathbf{0} \vee \mathrm{c2} = \mathbf{0}$. This result can be seen by inspection of the schematic in Figure 5.1 and case analysis on the inputs of the *halfAdd1*. If both inputs are **1**, then $\mathit{s1} = \mathbf{0}$ *(using halfAdd1)*, and therefore $\mathit{c2} = \mathbf{0}$ *(using halfAdd2)*. On the other hand, if one of the inputs A or B is **0**, then the carry output of the first adder, *halfAdd1* will be **0**, i.e. $\mathit{c1} = \mathbf{0}$.

The observation that $c1 = \mathbf{0} \vee c2 = \mathbf{0}$ provides some insight into the design of the full adder. It shows that the Xor gate may be replaced by an Or gate, because these two components share the same truth table under the condition that at least one of the inputs is $\mathbf{0}$. While the observation is not particularly relevant for the case that the design will be implemented by look-up tables in an FPGA, it is encouraging to note that improved insight into the design of a component gained by theorem proving is possible, even for simple designs.
The proof is concluded by considering the cases $c1 = \mathbf{0}$ and $c2 = \mathbf{0}$. If $c1 = \mathbf{0}$ then $c2 = cb\ C_{out}$, and substituting these values into Equation 5.16 gives the consequent of the lemma because the term with $c1$ evaluates to $\ll[\mathbf{0}, \mathbf{0}]\gg$, and hence zero, and can be eliminated from the equation. In the case where $c2 = \mathbf{0}$, $c1 = cb\ C_{out}$, and again the required result can be seen though simple reasoning over arithmetic of binary numbers.

This concludes the proof of the lemma, and the required correctness theorem, *fullAddCorrect*, can now be proven. Informally, we reason that a given set of values for $s1$, $c1$ and $c2$ either satisfies the premise in the lemma or it does not. If there exists a set of values that satisfy the premise, then these values can be used as witness values in the main theorem, since the theorem uses the same model and specification. If there does not, then the lemma is trivially true, because the antecedent is always false. Likewise, if no set of values satisfying the model exists, the premise of the main theorem is false, and it too is trivially true. Thus this proof step is justified regardless of whether or not the model is satisfiable (satisfiability is proved separately).

Chapter 6

Sequential and Iterated Logic

Contents

The netlist language introduced in the previous chapter provides a means to reason about the behaviour of combinational logic. It provides a collection of named primitive components, and supports the definition of circuits by allowing components to be instantiated in a design, and the connections between those instantiations defined. It also has the capability to model abstraction: new components may be defined by associating a circuit definition with a name, which may then be used as if it were another primitive component. Furthermore, abstraction is also achieved by hiding internal signals within a component, such that they are not visible as external signals of a component.

Unfortunately, the language is rather too simple for the purpose of reasoning about hardware/software compilation. It does not model *sequential logic*, and therefore provides no support for reasoning about computations that occur in stages, requiring memory to store intermediate results. In the context of reconfigurable hardware, such computations are typically implemented using *synchronous logic*, although asynchronous techniques have also been used.

Another limitation of the netlist language as presented in Chapter 5 is that it is difficult to work with values that are represented by more than one bit.

The design and verification of a full adder component with two input values and a carry input was presented: each input represented by one bit. Typically, arithmetic computations involve operations on larger values, represented by *bit vectors*. These are ordered lists of bit values, that may be interpreted as a numeric value. It would be cumbersome to design and verify circuits that operate on bit vectors using the netlist language as described. Such circuits are often composed of repeating structures. For example, the ripple-carry adder — as used in many FPGA designs due to FPGAs providing dedicated carry logic — is composed of several full adders connected together in a row. This concept is referred to as *iterated logic* here.

This chapter addresses these limitations by extending the netlist language. As is conventional for reconfigurable hardware — and for current computer architectures in general — the extension is based on synchronous logic. It is expected, however, that an alternative extension based on asynchronous logic could be formulated, because the semantics are still given in higher order logic. Higher order logic has been used to verify the implementation of a D-type flip-flop expressed at the CMOS gate level [Gor86]. At this level of abstraction, the flip-flop implementation is essentially a delay-sensitive asynchronous circuit.

Synthesisable HDLs such as VHDL and Verilog provide support for representing both sequential logic and iterated logic. The extensions to the netlist language introduced in this chapter may be mapped into one of these higher level HDLs relatively straight-forwardly, because the extensions continue to represent a subset of the features available in higher level HDLs.

6.1 Approaches to Modelling Sequential Logic

This section surveys existing approaches to modelling sequential logic in the formal semantics of hardware description languages. Each approach involves introducing a notion of time in the formal semantics. Section 6.1.1, *Summary of Approaches to Modelling Time* discusses factors that were considered when selecting an approach to extending the netlist language, as presented in Chapter 5, to support sequential logic. It also classifies existing approaches into two categories which are described in Section 6.1.2, *Constructive Approaches* and Section 6.1.3, *Declarative Approaches*. The section concludes by describing the selected approach and why it is appropriate for the netlist language in Section 6.1.4, *Selection of an Approach to Modelling Time*.

6.1.1 Summary of Approaches to Modelling Time

While there are many different ways in which the netlist language could be extended to support sequential logic, and hence the notion of time, it is necessary to select one approach in order that the semantics be well-defined. An attempt to use more than one technique could lead to inconsistencies. It is necessary, then, to consider which approach to use for the netlist language. The different approaches considered here fall into two categories, which are referred to here as *constructive approaches* and *declarative approaches* [1].

[1] This choice of terminology borrowed from [Fox01b]

The constructive approaches share one aspect in common: in these approaches, the semantics have a well-defined mechanical approach to determining the values of semantic domains for a given point in time, based on the values of those domains for previous points in time.

In contrast, declarative methods define a predicate that characterises the behaviour of a circuit over time. While constructive approaches must define the state at each point in time based solely on the history of state values, declarative approaches may use expressions about signal values at arbitrary points in time, including those that occur in the future.

Declarative approaches tend to support greater non-determinism, because most of the constructive approaches define states as a function on previous states. Although this statement holds for many examples of formalisations of time, a stronger statement than this would be an over-generalisation. An operational semantics that determines a set of possible next states could provide a mechanical means of deriving all possible subsequent states, in such a way that the actual next state is non-deterministic.

The decision as to whether a constructive or a declarative approach would be more appropriate for the netlist language is based on several factors. The netlist language as it stands represents a commitment to various aspects of a logical framework. Its semantics are expressed as semantic functions from the abstract syntax of the netlist language into predicates in the meta language, Isabelle/HOL, allowing reasoning about the behaviour of hardware in higher order logic. In selecting an approach, it is necessary to consider this existing commitment to an approach within the logical framework, and the extension must be 'compatible' with the existing model.

As an example of how appropriate modifications to the language can be identified, one can consider whether the benefits of the current netlist representation are preserved when it is modified to model time. The relational model provided zero-delay semantics, allowing tableau-based reasoning for combinational logic. This approach was used to verify the design of a half-adder in the previous chapter. The ability to verify logic in this way is convenient, and ideally, the extension to the representation should preserve the benefits of a simple way to reason about combinational logic.

Thus one criterion for selecting an approach to extending the semantics to allow the modelling of sequential logic is that it should not entail losing the ability to reason simply about combinational logic.

6.1.2 Constructive Approaches

Constructive approaches to modelling hardware behaviour allow the behaviour of a circuit, as modelled by the semantics of a hardware language, to be computed from prior states of the circuit and a description of any stimuli that are applied to the external signals of the circuit. Hence, for hardware languages defined using a constructive approach, it is possible to develop a simulator to predict the behaviour of a given circuit.

IEEE Standardised HDLs

The IEEE standards that define VHDL and Verilog describe a constructive method for simulating their respective HDLs. The semantics of these languages are complex, and written in English: the standards do not provide formal semantics for the languages. Verification of designs is usually performed by using a test-bench, also written in an HDL.

Hardware designers typically consult the manuals for their synthesis tool to determine which HDL constructs may be used to denote a particular hardware construct. Hardware not designed in this manner may not be synthesisable, and even if it is, it may not behave as intended.

The semantics described in the Language Reference Manuals [IEE02, IEE01] for these languages model the simulation semantics of the language, without reference to hardware. Although separate manuals describe the hardware intended to be synthesised from certain language constructs [IEE04b, IEE04a], it is not clear how accurately hardware synthesised according to the latter manuals behaves as per the simulation semantics of the former.

Thus, it is not clear that a hardware design expressed in one of the above HDLs will behave as per their deterministic simulation semantics, which are an approximation to the behaviour of the actual hardware. In contrast to the semantics of these HDLs, the temporal behaviour of the actual hardware may be non-deterministic. The behaviour of hardware varies according to the details of the hardware implementation of the design, the target technology and the operating environment, which may not be known precisely at design time.

Formal verification of designs using the Language Reference Manuals is not possible because they do not provide formal semantics for the languages. This makes the standard semantics of these languages inappropriate for the purposes here.

Formal semantics for these languages (or subsets thereof) have been constructed retrospectively and attempt to formalise the simulation semantics of the language [KB95a, SX98, Gor95, BM95, Rus94].

The semantics describe an iterated simulation cycle. The simulation cycle considers a number of 'hardware threads' which are executed until they reach a '`wait`' statement. As they execute, they may schedule signal *transactions*, allowing the value of a signal to be updated after an optional delay which can be used to model the delay of physical circuits [KB95b]. The signal updates are stored in a (chronological) delta-queue, and if a simulation cycle schedules new transactions, they are merged with existing future transactions.

One formulation of the operational semantics of VHDL defines the following semantics domains [vT95]:

$$\gamma = (name)\text{-}set \quad \sigma = (name \times value)\text{-}set$$
$$\tau = time \to \sigma \qquad \theta = time \to (\gamma \times \sigma)$$

Time is modelled using the natural numbers. Signal names and signal values are of type *name* and *value*, and are represented by strings and Boolean values respectively. Whenever an event occurs on a signal, such as a signal rise or

fall, the value is included in the set of events, γ. The state of the circuit, σ, is represented by a finite set of name-value pairs, and associates signal names with their current value. Transactions that have been scheduled to occur in the future (to model delay) are represented as name-value pairs denoting the new value the signal should take after the transition. More than one transaction may be scheduled for a given point in time, and the future of (scheduled) transactions, τ, is modelled as a function from time to a set of transactions. The history of the behaviour of a circuit is a function of time that yields the state of the circuit, and the set of signals for which an event took place at that time.

While some formalisations appear to model a large subset of the relevant HDL, hardware verification using such semantics is difficult, and requires a large amount of effort from the user [vT95].

Other HDLs

Constructive formal semantics exist for HDLs other than those standardised by the IEEE. Many of these have the advantage that formal semantics were considered during language design, rather than being developed retrospectively as in the case of VHDL and Verilog.

The formal semantics of Core ELLA adopts a process algebraic approach, and has a simple timing model that can be used to describe synchronous hardware [BGMW94]. A shallow embedding [BGHT90] of these semantics were expressed in the HOL system [Gor85, Gor88], with the intention of being used primarily for 'program' verification [BGG$^+$92].

In contrast to ELLA, the Balsa HDL is used for describing asynchronous hardware [EB02]. Given its lack of formal semantics, and the difficulties in synthesising asynchronous logic to current reconfigurable hardware, it is inappropriate for the purposes here.

Iterated Maps

Anthony Fox describes an algebraic technique for specification and verification of microprocessors [Fox01b]. The methodology involves defining two functions, namely an initialisation function and a next state function:

$$\begin{aligned} init &: \Sigma \rightarrow \Sigma \\ next &: \Sigma \rightarrow \Sigma \end{aligned}$$

The initialisation function, *init*, maps arbitrary states onto valid initial states for the circuit. This may be used to specify the reset circuitry that ensures a hardware design initialises to an appropriate state, or, in the context of microprocessors, to specify a possible flush of the instruction pipeline, if necessary. In the context of reconfigurable hardware that allows an initial state to be programmed, the initialisation function could simply represent an abstraction of this feature of the hardware.

The next state function, *next*, maps the state of a circuit at one point in discrete time to the next, and thus models how the state of a circuit varies over time.

A state function, $G : \mathbb{N} \to \Sigma \to \Sigma$, can be used to model the state of a circuit given a point in discrete time (indexed by natural numbers) and an initial state. Where such a function is defined in terms of an initialisation function and a next state function, it is known as an *iterated map*. An iterated map is therefore be defined as:

$$\begin{aligned} G\ 0\ \sigma &= init\ \sigma \\ G\ (t+1)\ \sigma &= next\ (G\ t\ \sigma) \end{aligned}$$

Since *next* is a function of a previous state, an iterated map represents a deterministic model of the hardware, and is appropriate for modelling synchronous logic. Fox states that the model is not intended to model the behaviour of a physical circuit accurately, but rather to provide a useful abstraction of the hardware, and notes that the approach could be used to model asynchronous hardware, although at a level of temporal abstraction at which components behave synchronously [Fox01b].

The iterated map approach to modelling has been used to specify and verify the ARM instruction set and 'micro-architecture' [Fox01a, Fox02].

6.1.3 Declarative Approaches

Unlike the constructive approaches to modelling hardware behaviour, declarative approaches do not specify explicitly how the behaviour of a circuit will vary over time. Rather, they specify predicates that characterise the relation between the states at different times. However, this does not preclude the possibility of determining a class of circuits for which that relation is a function, and a level of temporal abstraction such that the behaviour of the circuit can be simulated using constructive techniques. This property appears to be easier to detect from a static description of a circuit if the circuit is described structurally (as in [Hut93]), rather than as an arbitrary relation.

Structural HDLs

The Ruby hardware description language provides a declarative approach to describing the structure of combinational and synchronous, logic [JS90a]. It uses a relational approach to modelling circuit behaviour. For combinational logic, circuit behaviour is modelled as a relation between signal values. Where the relation holds for given signal values, the model admits those values as signals that may be simultaneously observed in that circuit. This is similar to the approach described in Section 3.3.2.

For sequential logic, a circuit is modelled as a relation on sequences of signal values. This is in contrast to methods based on higher order logic, which typically use functions rather than sequences to model behaviour over time [HD86].

In Ruby, circuits are described by an expression that corresponds to a structural representation of the circuit. The expression denotes a relation that charac-

terises sets of signal values, or sequences of signal values, that are consistent with that structure. Therefore, as with the technique for modelling hardware described in Section 3.3.2, the representation describes both the structure and the behaviour of the hardware.

Higher order functions are used to compose circuits from their constituent components. These include composition operator (written '`;`') that provides relational composition, used to represent connecting the output of one component to the input of another. Other functions include `row` and `col`, which are used to describe rows and columns of similar components.

Reasoning about Ruby 'programs' is possible through the use of a relational calculus [Jon90c, JS90b]. Relations are also typed, meaning that only signals with similar structure may be connected. This avoids problems that would arise in attempting to give a semantics for the composition of components with incompatible interfaces.

A Ruby interpreter exists allowing the simulation of hardware described by a Ruby program [Hut93]. The interpreter is limited to a subset of Ruby designs. The interpreter cannot, for example, simulate designs where more than one output is used to drive the same signal. Likewise, it cannot be used to simulate circuits with combinational loops. However, this does not present many problems when working with synchronous logic because latches are provided as primitives in Ruby, and do not need to be described using combinational logic.

Pebble is a hardware description language that has been developed more recently than Ruby. It was developed to be a simple HDL that supports run-time reconfigurable systems [LM98]. A formal semantics for Pebble has been developed as a shallow embedding in *Synchronous Receptive Process Theory* (SRPT) [Bar93], and is based on a deterministic constructive approach [HH05]. As such, a description of them would have been more appropriate in the previous section. However, Pebble itself is essentially a structural language like Ruby, and a declarative semantics could also be given. One of the limitations of Pebble is that it only supports the development of systems with a single clock.

Ruby and Pebble influenced the development of Quartz [PL05]. Quartz combines higher order combinators for structural descriptions, but provides more sophisticated type system. It is strongly typed and supports block composition, type inference, polymorphism and overloading. Like Pebble, it also supports run-time reconfiguration.

Reasoning in Quartz is similar to that used for Ruby circuits, and correctness preserving transformations may be applied to a design as part of the refinement process. A shallow embedding of the Quartz semantics has been modelled in the Isabelle theorem prover, although not using the Isabelle/HOL logic as used for this thesis.

'Lifting' Signal Values in Higher Order Logic

Where a semantics of a hardware description language is represented in higher order logic, it is possible to model the behaviour of the circuit using a function from time to signal value for each signal. A function of this type is referred to here as a *signal behaviour*. To clarify, a signal value represents the signal

level at a single point in time, whereas a signal behaviour represents all values that a signal takes during the operation of a circuit. Thus, where the natural numbers are used to represent time, and Boolean values are used to represent signal values, a signal behaviour is of the type $\mathbb{N} \to \mathbb{B}$.

Modelling the behaviour of a single signal has little merit, of course. Modelling the behaviour of a number of signals in a circuit can be achieved by using a free variable for each signal behaviour as described in Section 3.3.2. However, the semantics of the netlist language used here does not use free variables to model signals. While this avoids the need to refer to terms in the meta-language when considering circuit structure, it means a level of indirection is required, called a *binding*, that maps signal names to their values.

When extending the netlist language to support temporal modelling, there are different ways to modify the language semantics to support time. Two approaches to modelling *temporal bindings* are considered here.

The first approach is to use a function, $\theta : time \to (name \to value)$, that maps from a given point in time to the binding for all signals at that point in time. Applying such a function to a time yields a binding that represents a snapshot of the state of each signal at a given point in time. This approach is similar to the history, θ, in van Tassel's operational semantics for VHDL described on page 106.

The second approach is to add indirection to the signal value, using a function, $\sigma : name \to (time \to value)$, that associates each signal name with its behaviour during the operation of the circuit. Applying such a function to a name yields a signal behaviour, as defined above.

Most approaches that use higher order logic to model circuit behaviour use free variables in an expression to represent signals, thereby requiring only one level of indirection, rather than two. As a consequence, there seems to be little precedent to indicate which approach would be more appropriate for the netlist language here.

Levels of Temporal Abstraction

When using higher-order values to represent signal behaviour over time, it is necessary to decide what each unit of time should represent. A common approach is that each unit represents a very short period of time, less than a gate delay for example. This way, the behaviour of a gate, which implements a function f, with inputs $i_1, \ldots, i_m$, output o and delay δ, may be represented as $\forall t.\ f\ i_1(t) \ldots i_m(t) = o(t + \delta)$.

An alternative approach is to assume that time represents the number of clock cycles since the start of the circuit's operation. In this case, it makes little sense to refer to the value of the clock signal, as its value varies throughout the cycle. Such an approach is limited to simple synchronous logic, however. Here, 'simple' refers to the fact that it could not be used to model the behaviour of, say, synchronous systems in which the clock cycle is divided into separate phases, in a way that captures every signal transition. Systems that divide the clock cycle into phases include, for example, most ARM microprocessors [Fur00]. However, abstractions can be developed at the cost of using

models that are less faithful to the behaviour of the actual device compared to the previous approach.

For reconfigurable computing platforms, simple synchronous logic may be sufficient, and the latter approach may be adequate. Assuming that a compilation algorithm targets this class of synchronous logic, the use of a simpler temporal model should allow simpler reasoning about compilation of a logic design.

One of the advantages of using higher order logic is that it can be used to model temporal abstraction. Just as a retrieve function (as introduced in Section 1.2.2) may be used to relate machine states with abstractions of those states, a *time mapping* [Mel93] or *retiming function* [HT90] may be used to correlate time in an abstract model with that in a more faithful 'concrete' model.

6.1.4 Selection of an Approach to Modelling Time

Ideally, a compiler intermediate representation should be independent of source and target languages. A compiler IR intended to represent hardware should ideally be able to target any hardware description language. In order to reason about the correctness of compilation from a software-based IR into a hardware representations, the formal framework used should support reasoning over the semantics of both the source and target representations. This means that it must be possible to describe the formal semantics of each language within the same formal system.

The formal system used in this thesis is higher order logic, specifically the formulation provided by the Isabelle/HOL system. Higher order logic may be used to model the semantics of both constructive and declarative hardware description languages. More precisely, it may be used to model both constructive and declarative semantics of hardware representations, since a representation may be given more than one semantics.

The constructive and declarative approaches to modelling the temporal behaviour of hardware using higher order logic can be summarised as follows. The constructive approaches use higher order logic for denoting and reasoning about the semantics of hardware description languages. The VHDL semantics by van Tassel described on on page 106 fall into this category. The declarative approaches typically use higher order values to represent the values of signals, and how they may behave over time, directly within the logic. This approach was described in *'Lifting' Signal Values in Higher Order Logic* beginning on page 109.

The approach adopted here is to support temporal modelling of circuit behaviour by defining a semantics for the netlist language using the declarative style, using higher order values to represent the behaviour of signals over time. In contrast to more traditional approaches, signals are not modelled as free variables within logical expressions. This is consistent with the semantics developed in the previous chapter, which use a 'binding' function to associate signal names with the value of that signal.

Higher order logic is naturally suited to representing the concepts required for reasoning about hardware compilation. In the logic, signal values represent first

order terms. A signal behaviour is a function from time to signal value, and therefore is a second order entity. A temporal binding that maps signal names to signal behaviours, of the same type as θ or σ as defined on page 110, is either a curried function on time yielding a signal binding, or a curried function on signal names yielding the behaviour of a given signal. Thus, a circuit behaviour is a third order entity.

In Section 5.4, the notion of a *circuit behaviour* was introduced. In that context, a circuit behaviour is a predicate that characterises which signal bindings are valid in a given combinational circuit; that is, which sets of signal values may be simultaneously observed on a given set of signals. Using temporal bindings, a circuit behaviour is Boolean function on temporal bindings that characterises which temporal bindings are valid for a given circuit. This means a circuit behaviour is a fourth order entity.

To provide an example of a circuit behaviour, consider a delay component, with input A and output B, where the output signal is equal to the input signal with a delay of one unit of time introduced. The circuit behaviour for this device can be written $\lambda\ \sigma.\ (\forall\ t.\ \sigma\ \mathsf{A}\ t = \sigma\ \mathsf{B}\ (t+1))$.

In order to verify the output of a hardware compiler, it is necessary to show that the generated hardware design will behave in the same way as the hardware design represented by the intermediate hardware representation from which it was generated. The property that two circuits admit the same circuit behaviour is a Boolean function on pairs of circuit behaviours — this property is a fifth order entity.

Higher order logic has been used to represent the semantics of hardware description languages with even the most complex timing models, such as those defined by the IEEE standards. However, those models can be more appropriate for reasoning about the language itself rather than the correctness of a specific hardware design [KB95b].

In work that focuses on the development of verified hardware, rather than verified compilers, descriptions of modelling hardware using higher order logic refer to different entities at each order [HD86]. Thus higher order logic provides a flexible formal system within which hardware components corresponding to operations in a software IR could be developed and verified within the same framework, and it becomes possible to reason about a compilation algorithm that targeted those verified components.

6.2 Temporal Modelling of Sequential Logic

This section presents an extension to the netlist language introduced in Chapter 5 in order to model sequential logic. The extension requires changes to the language definition as presented: the semantic domains and semantic functions must be modified to model the behaviour of signals whose values vary over time. It also requires the introduction of new circuit primitives. The abstract syntax remains unchanged.

This section is structured as follows: Section 6.2.1, *Adding Synchronous Logic to the Netlist Language* describes the extension to the netlist language to sup-

port sequential logic by introducing synchronous primitives into the netlist language, based on the assumption that a hardware/software compiler will target only synchronous logic designs. The semantics are modified to include the notion of (abstract) time, where one 'unit' of time represents one clock cycle.

6.2.1 Adding Synchronous Logic to the Netlist Language

Modifying existing definitions to model time

The language extension described here uses the *name⇒time⇒bit* currying to represent temporal signal bindings, as described in *'Lifting' Signal Values in Higher Order Logic* of Section 6.1.3. A new type must be introduced to represent time, and the type of a signal binding must be modified to represent a temporal binding, rather than a combinational signal binding:

types
time = *nat*
binding = *sig-name⇒time⇒bit*

Circuit behaviour is still represented by the type *binding⇒bool*. A 'lift' function is defined in order to be able to reuse combinational circuit behaviours already defined using the combinational (in other words, non-temporal) signal bindings (of the type *name⇒bit*) of the original netlist language.

$$lift \equiv \lambda cb\ tb.\ \forall t.\ cb\ (\lambda n.\ tb\ n\ t) \tag{6.1}$$

The function takes a combinational behaviour (of type (*name⇒bit*)⇒*bool*) and a temporal binding (of type *name⇒time⇒bit*) and is true or false according to whether the temporal binding satisfies the non-temporal behaviour at every point in time. By applying *lift* to a combinational behaviour, and hence making a partial function application, the function can be used to convert a combinational behaviour into a temporal behaviour of type (*name⇒time⇒bit*)⇒*bool*. To demonstrate the effect of the *lift* function, it is applied to the behaviour of an And gate, as defined in the previous chapter:

$$\begin{aligned} TAnd &= lift\ MAnd \\ \ldots &= lift\ \ (\lambda bind.\ (bind\ \mathsf{A} \wedge_b bind\ \mathsf{B}) = bind\ \mathsf{O}) \\ \ldots &= (\lambda tb.\ \forall t.\ (tb\ \mathsf{A}\ t \wedge_b tb\ \mathsf{B}\ t) = tb\ \mathsf{O}\ t) \end{aligned}$$

Thus, *tb* A *t* represents the bit value of signal A at time *t* in the temporal binding *tb*. It should be noted that a temporal behaviour found via the lift function represents a zero-delay version of the combinational behaviour to which it is applied.

In order to model a combinational delay, an alternative function could be defined. Such a function would require an another argument to identify which pins of the component represent inputs and which pins represent outputs. This could be achieved using a static environment, that associates primitive names with the interface (that is, the pin out) of the corresponding primitives.

(a) Synchronous delay primitive.

(b) Synchronous delay primitive (with initialisation).

Figure 6.1: Synchronous unit-delay primitives

An initial set of synchronous delay primitives

In contrast to combinational logic, which may be defined using relations between signals values, sequential logic may be defined as a relation on signal behaviours, or alternatively, signal values at a particular times. An initial set of sequential primitives appropriate for hardware/software compilation may be defined by adopting those used by existing techniques for compiling functions into synchronous logic [GIOS05]. This requires translating free variables in each definition in the cited work into a signal look-up in a temporal binding. For example, the following components specify two delay primitives. The input of each component, D, is reflected in its output, Q, with a one unit time delay.

$$TDel \equiv \lambda tb.\ \forall t.\ tb\ \mathsf{D}\ t = tb\ \mathsf{Q}\ (t + 1)$$
$$TDelT \equiv \lambda tb.\ tb\ \mathsf{Q}\ 0 = \mathbf{1} \wedge TDel\ tb$$

The first component, *TDel*, represents a simple unit delay device. The output of the component at time zero is unspecified. The second component, *TDelT*, represents a similar component but for the fact that its output is specified to be initialised to particular value (in this case, **1**) at time zero. The *TDel* and *TDelT* primitives are illustrated in Figure 6.1a and Figure 6.1b respectively. These devices can be implemented using a D-type flip-flop (ibid).

A typical D-type flip-flop device has four pins: two inputs, namely D and a clock signal, and two outputs, Q and $\bar{Q}$. Such a device is illustrated in Figure 6.2a. For a rising-edge triggered D-type device, the output Q takes the value of D at each rising clock edge. The complement of that value is always output on $\bar{Q}$. This output is maintained until the next rising clock edge. In contrast, a falling-edge triggered device maintains the value between consecutive downward transitions on the clock signal. Thus, D-type devices may be used to implement a single-bit memory device.

At the level of temporal abstraction used here, each time unit represents one clock signal. As noted in *Levels of Temporal Abstraction* on page 110, it is not possible to model the clock signal directly at this level of abstraction. In other words, the granularity of time in the model is too coarse to model the rise and fall of the clock signal. As a consequence, it is not possible to model the distinction between the behaviour of a component that has specific behaviour at a rising clock edge, and a similar component that has similar behaviour at a falling clock edge.

It can be seen then, that the distinction between a rising-edge triggered D-

(a) Traditional D-type flip-flop.

(b) Netlist language D-type flip-flop with enable.

Figure 6.2: D-type flip flops

type flip-flop and a falling-edge triggered D-type flip-flop cannot be modelled at the level of temporal abstraction used here. Furthermore, the model of either such a device cannot include a clock pin, because the behaviour of the clock signal cannot be modelled. Therefore, in order to add either type of flip-flop to the netlist language, it is necessary to form an alternative, more abstract, description of the behaviour of these devices that does not refer to the clock signal.

One approach to removing the reference to the clock signal is to assume that the clock pin is connected to a live clock with a period of one unit of time. In the case, the clock pin is no longer required in the model and it becomes implicit. For the netlist language, the $\bar{Q}$ pin is also removed in order to simplify the model of these components.

After making these amendments to the descriptions of the D-type devices, the behaviour of each device may be modelled at the required level of abstraction. The new devices both satisfy the specification *TDel* given above. The proof that the behaviour of a D-type device defined using a more fine-grained model of time satisfies the *TDel* specification was mechanised in the HOL system in the 1980s by Tom Melham [Mel93].

Register primitives

The addition of *TDel* to the netlist language as a primitive component is justified by the fact that both types of D-type flip-flop appear as 'primitives' in typical FPGAs and other reconfigurable hardware devices targeted by reconfigurable compilers (in fact, only one type is required to implement a design in this netlist language). Such devices, such as the Spartan-3E FPGA discussed in Section 2.1.1 allow initialisation values for (distributed) RAM to be stored in the device configuration, and provide reset circuitry to perform device initialisation based on that configuration. This feature justifies the inclusion of the *TDelT* component as defined above.

While using components provided by existing reconfigurable hardware as justification for their inclusion as primitives in the netlist language, it is appropriate to introduce a primitive that is more sophisticated than the single unit delay primitives, and that models the storage components typically found in reconfigurable hardware more accurately. This allows simpler compilation of algorithms for which such a device may be useful in the implementation. However,

it does increase the complexity of the netlist language and the assumptions it makes about the capabilities of the targeted reconfigurable logic.

To this end, the *TDel* device can be extended to include a *clock enable* signal. In contrast to the clock signal, which had to be removed from the abstraction, the enable signal is expected to transition, at most, once per clock cycle. An abstraction of the D-type flip-flop with clock enable is shown in Figure 6.2b.

The *TDel* abstraction is based on a model in which a D-type flip-flop is clocked on every clock cycle. This specialisation was necessary in order that the clock signal could be omitted from the abstraction, but limited the device to provide a delay of exactly one clock cycle. Adding a clock enable signal allows the D-type flip-flop to store a bit value indefinitely: from each clock cycle in which the enable signal is high, to the next cycle in which it is high. Hence, if the clock enable is kept high, the device behaves as per the *TDel* specification.

A (rising edge) D-type flip-flop latches its input value on the rising edge of the clock signal. The 'rising edge' property of the clock signal cannot be expressed in the abstract model of time used in the semantics of the netlist language. It can, however, be expressed in a model that uses concrete time, where time is represented at a finer level of granularity that the clock period. Using a model of concrete time, the rising edge of the clock signal, *ck*, at time *t+1* may be represented by the predicate: $\neg\ ck\ t \wedge ck\ (t+1)$.

6.2.2 An Abstract Register

Gordon et al. define an 'abstract register' which latches its input value on the rising edge of a non-clock signal, *S*, and maintains that value on its output until the next clock cycle that *S* makes a rising transition [GIOS05]. This rising edge property is represented by the same predicate as that of a rising edge in concrete time given above, although its meaning is different when used to refer to abstract time. In the netlist language semantics, $\neg\ s\ t \wedge s\ (t+1)$ refers to a signal that is low at clock cycle *t*, but makes an upwards transition to high at clock cycle *t+1*.

A design that implements an abstract register is presented in this section. One of the components used to implement the abstract register is a device that detects a rising edge on a non-clock signal. This section describes a component from the cited work that accomplishes this called 'Posedge', and describes its correctness proof. The design for an abstract register is then developed, based on the specification provided in the same work.

Posedge - a device to detect rising edges

A component, referred to as *Posedge*, is specified which detects a rising edge in a non-clock signal. It has a single bit input, *A*, which is the signal being monitored for a rising edge. It also has a single bit output, *B*, which is high when *A* has made a rising transition in that clock cycle. That is, *B* is high at time *t+1* if and only if *A* is low at time *t* and high at time *t+1*.

The specification of *Posedge*, *posedgeSpec*, is defined in terms of an auxiliary function *posedge*:

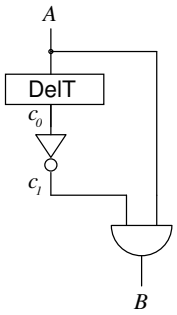


Figure 6.3: *Posedge*: a component for detecting rising edges on signal values.

$posedge\ s\ t \equiv s\ t = \mathbf{0} \wedge s\ (t + 1) = \mathbf{1}$
$posedgeSpec\ a\ b \equiv \forall t.\ \mathit{if}\ posedge\ a\ t\ \mathit{then}\ b\ (t + 1) = \mathbf{1}\ \mathit{else}\ b\ (t + 1) = \mathbf{0}$

The device that Gordon et al. define to satisfy this specification is illustrated in Figure 6.2.2. Its definition in the netlist language is as follows:

$$
\begin{array}{l}
posedgeCmp \equiv \\
(|Ext = ([A], [B]), \\
\quad Insts = \\
\quad\quad [(|Comp = DelT,\ Conns = [In\ D\ A,\ Out\ Q\ C_0], \ldots = ()|), \\
\quad\quad\ (|Comp = Not,\ Conns = [In\ A\ C_0,\ Out\ B\ C_1], \ldots = ()|), \\
\quad\quad\ (|Comp = And,\ Conns = [In\ A\ A,\ In\ B\ C_1,\ Out\ O\ B], \\
\quad\quad\quad \ldots = ()|)], \\
\quad \ldots = ()|)
\end{array}
$$

Applying the semantic functions that have been modified to include the notion of time to this design gives the following behaviour:

$$
\begin{array}{l}
MPosedge = \\
(\lambda bi.\ \exists\, c1\ c0. \\
\qquad c0\ 0 = \mathbf{1} \wedge \\
\qquad (\forall t.\ bi\ A\ t = c0\ (t + 1) \wedge \\
\qquad\quad (\neg_b\ c0\ t) = c1\ t \wedge (bi\ A\ t \wedge_b c1\ t) = bi\ B\ t))
\end{array}
$$

The correctness condition of the model is that this behavioural model satisfies the specification, *posedgeSpec*. The required verification condition is:

$$MPosedge\ bi \longrightarrow posedgeSpec\ (bi\ A)\ (bi\ B)$$

The correctness proof proceeds by unfolding both the specification and its auxilliary function, using the identity:

$$
\begin{array}{l}
posedgeSpec\ (bi\ A)\ (bi\ B) = \\
\quad (\forall t. \\
\qquad \mathit{if}\ bi\ A\ t = \mathbf{0} \wedge bi\ A\ (t + 1) = \mathbf{1} \\
\qquad \mathit{then}\ bi\ B\ (t + 1) = \mathbf{1} \\
\qquad \mathit{else}\ bi\ B\ (t + 1) = \mathbf{0})
\end{array}
$$

The proof that the model satisfies the specification can be decomposed, and proven using three auxiliary lemmas. In order to demonstrate how the required lemmas are identified, the following, more general form of the universally quantified part of the above expression can be considered: $\mathit{if}\ a \wedge b\ \mathit{then}\ c\ \mathit{else}\ d$.

This more general form can be rewritten as $(a \wedge b \longrightarrow c) \wedge (\neg\ (a \wedge b) \longrightarrow d)$, and hence, as $(a \wedge b \longrightarrow c) \wedge (\neg\ a \vee \neg\ b \longrightarrow d)$ using De Morgan's laws in the second conjunct. A further rewrite, this time using the properties of disjunction and implication, gives $(a \wedge b \longrightarrow c) \wedge (\neg\ a \longrightarrow d) \wedge (\neg\ b \longrightarrow d)$, and shows the three conjucts, each of which can be proven as a separate lemma.

The lemmas used for proving the correctness of *Posedge* can be found by instantiating the rewritten version of the general form to match *posspecSpec*. Thus, the following lemmas are used in the correctness proof (each lemma is proven under the assumption that *MPosedge* holds of each binding *bi*):

$\forall\, t.\ bi\ A\ t = \mathbf{0} \wedge bi\ A\ (t + 1) = \mathbf{1} \longrightarrow bi\ B\ (t + 1) = \mathbf{1}$
$\forall\, t.\ bi\ A\ t = \mathbf{1} \longrightarrow bi\ B\ (t + 1) = \mathbf{0}$
$\forall\, t.\ bi\ A\ (t + 1) = \mathbf{0} \longrightarrow bi\ B\ (t + 1) = \mathbf{0}$

The proof of all three lemmas follow the same format. Similar to the proof of correctness of the full adder, the behavioural expression in the antecedent of the verification condition can be rewritten in a simpler form without existentially quantified signals. The only difference in proof technique between that used for the full adder and that used for *Posedge* is that the following rule is also applied to the behavioural model:

$(\forall\, x.\ P\ x \wedge Q\ x) = ((\forall\, x.\ P\ x) \wedge (\forall\, x.\ Q\ x))$

For example, the third lemma requires deriving the following proof rule, which shows the assumptions of the lemma explicitly.

$$\frac{\begin{array}{l} c0\ 0 = \mathbf{1} \wedge \\ \forall\, t.\ bi\ A\ t = c0\ (t + 1) \wedge \\ \forall\, t.\ (\neg_b\ c0\ t) = c1\ t \wedge \\ \forall\, t.\ (bi\ A\ t \wedge_b c1\ t) = bi\ B\ t \end{array}}{\forall\, t.\ bi\ A\ (t + 1) = \mathbf{0} \longrightarrow bi\ B\ (t + 1) = \mathbf{0}}$$

In all three lemmas, for a given value t in the consequent, the universally quantified variables in the premise need to be specialised to t, $t+1$ and $t+1$ respectively.

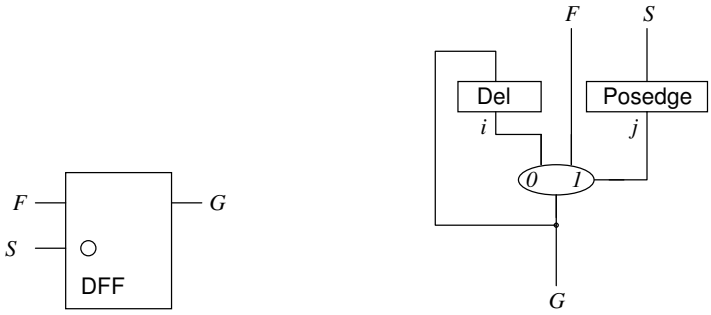
DFF - an abstract register

An abstract register is a storage component that allows a value to be stored for an arbitrary number of clock cycles. The model used here has two inputs *F* and *S*, and an output *G*. The register latches its input value on the rising edge of *S*. That is, it latches *F* when *S* was low on the previous clock cycle and high on the current cycle. The latched value is propagated to the output, where it is maintained until the next clock cycle for which there is a rising edge on *S*. The device and its pins are illustrated in Figure 6.4a.

The specification of an abstract register for the netlist language is adapted from that specified by Gordon et al. [GIOS05]. The component is named *DFF*, as per the cited work. However the specification here differs. It is specified as:

dffSpec f g s $\equiv \forall\, t.\ g\ (t + 1) =$ (*if posedge s t then f* $(t + 1)$ *else g t*)

The difference between this definition and the original are as follows. In the original definition *posedge s t* characterises the times t at which signal s is high,



(a) *DFF*: a component for detecting rising edges on signal values.

(b) *DFF*: A design implementing the specification.

Figure 6.4: Design of an abstract register

but low at time *t-1*. It appears that each use of this predicate is in the form *posedge s (t+1)*. In this work, *posedge s t* characterises the times t at which signal s is low, but high at time *t+1*. The specification has been modified according, which simplifies some of the formulae. Aside from this difference, this specification may be alpha-reduced to the original, where the external signals *F*, *G* and *S* correspond to the original 'd', 'f' and 'sel' respectively.

A design that satisfies this specification is illustrated in Figure 6.4b. Its representation in the netlist language is:

$$
\begin{array}{l}
\mathit{dffCmp} \equiv \\
(\!|\mathit{Ext} = ([\mathsf{F},\ \mathsf{S}],\ [\mathsf{G}]), \\
\quad \mathit{Insts} = \\
\quad\quad [(\!|\mathit{Comp} = \mathsf{Posedge},\ \mathit{Conns} = [\mathit{In}\ \mathsf{A}\ \mathsf{S},\ \mathit{Out}\ \mathsf{B}\ \mathsf{J}],\ \ldots = ()|\!), \\
\quad\quad\ (|\!|\mathit{Comp} = \mathsf{Mux}, \\
\quad\quad\quad\ \mathit{Conns} = [\mathit{In}\ \mathsf{A}\ \mathsf{I},\ \mathit{In}\ \mathsf{B}\ \mathsf{F},\ \mathit{In}\ \mathsf{S}\ \mathsf{J},\ \mathit{Out}\ \mathsf{O}\ \mathsf{G}], \\
\quad\quad\quad\ \ldots = ()|\!|), \\
\quad\quad\ (|\!|\mathit{Comp} = \mathsf{Del},\ \mathit{Conns} = [\mathit{In}\ \mathsf{D}\ \mathsf{G},\ \mathit{Out}\ \mathsf{Q}\ \mathsf{I}],\ \ldots = ()|\!|)], \\
\quad \ldots = ()|\!|)
\end{array}
$$

The following behavioural description results from applying the semantic functions to this definition.

$$
\begin{array}{l}
\mathit{Mdff} = \\
(\lambda \mathit{bi}.\ \exists j\ i.\ \mathit{MPosedge}\ (\mathit{bi}(\mathsf{A} := \mathit{bi}\ \mathsf{S},\ \mathsf{B} := j)) \wedge \\
\qquad\qquad (\forall t.\ \mathit{if}\ j\ t = \mathbf{0}\ \mathit{then}\ \mathit{bi}\ \mathsf{G}\ t = i\ t\ \mathit{else}\ \mathit{bi}\ \mathsf{G}\ t = \mathit{bi}\ \mathsf{F}\ t) \wedge \\
\qquad\qquad (\forall t.\ \mathit{bi}\ \mathsf{G}\ t = i\ (t + 1)))
\end{array}
$$

Here only the semantics of the primitive components in the netlist language, including the delay component, have been expanded. The semantics of the *Posedge* component have not been 'unfolded'. This allows properties of the *Posedge* component to be used in the verification of the *DFF* design. This is consistent with the approach taken in the previous chapter, where the verifica-

tion of a full adder design involved the use of properties of the half adder. The required verification condition for the *DFF* is:

$Mdff\ bi \longrightarrow dffSpec\ (bi\ F)\ (bi\ G)\ (bi\ S)$

As with the correctness proof for the *Posedge* device, the proof of this device can be decomposed, and simplified by the use of three auxiliary lemmas. The three cases to consider are:

1. where j is high, due to a rising edge on S, in which case it is necessary to show that F is propagated directly to G;
2. where the *previous* value of S was high, and hence that j is low and i has the same value as the previous cycle; and
3. where the *current* value of S is low, and hence that j is low and i has the same value as the previous cycle.

In the second and third cases, i is selected for the output because j is low. If j is low, it can be seen that i has the same value as the output G due to the behaviour of the multiplexer. It can also be seen that i has the same value that G took in the previous cycle. Ensuring that i maintains its value from the previous cycle ensures that the output is latched between rising edges.

6.3 Iterated Logic

The purpose of the netlist language is that it be able to represent hardware designs that implement operations in the software IR, such that part of a program's representation in the IR may be translated into the netlist language. In turn, the netlist language forms an intermediate representation for the hardware design before it is translated into synthesisable HDL, or a device specific netlist language. By defining a formal semantics for both the software IR and the netlist language, it becomes possible to specify a translation algorithm from the former to the latter.

Thus far, the designs in the netlist language all represent small components with only a few external signals, each of which represents one bit. However, many of the operations in the software IR, such as addition and subtraction, are defined as operations on machine words. In order to relate the semantics of the two languages, as is necessary when specifying a compilation algorithm, it is useful to raise the level of abstraction of the netlist language such that a value in the software IR semantics may be represented by a single value in the netlist language semantics. This requires that values in the netlist language can represent *bit vectors*.

It is also useful to introduce a mechanism in the netlist language that allows components to be defined in terms of repeating structures, because many word level operations are most conveniently described in this way. For example, a ripple carry adder — a common design in FPGAs due to many FPGAs providing dedicated carry logic for this purpose — may be implemented by 'chaining' a number of full adder components together in a row. A ripple carry adder that adds two four-bit bit vectors is illustrated in Figure 6.5.

The term used here to refer to logic designs composed of repeating structures is 'iterated logic'. Adding support to the netlist language for specifying, describing and reasoning about iterated logic requires changes to both its abstract

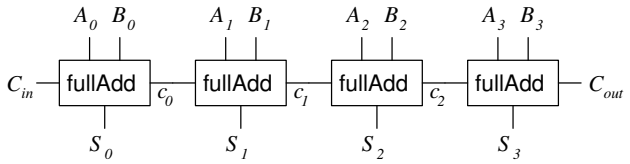


Figure 6.5: A four-bit ripple adder design, based on the existing full adder component.

syntax and its semantics. It also requires further consideration of which circuits should be considered well-formed, and hence, which circuits should have a defined semantics in the language.

This section describes how the language can be extended to support bit vectors and iterated logic. Specifically, the extension allows new components to be defined by composing existing components together in a row.

Section 6.3.1, *Adding Bit Vectors to the Netlist Language* describes the aims of the extension introduced here, and introduces the concepts of the row abstraction used here by means of examples. Section 6.3.1, *Abstract Syntax for Bit Vectors* presents a set of changes to the abstract syntax to allow components to be defined in terms of a row of existing components by introducing a new construct to define rows, called *row*. Section 6.3.2, *Bit Vector Semantics* gives a semantics to the *row* construct, providing an explanation of how the semantics can be modified to model iterated logic.

6.3.1 Adding Bit Vectors to the Netlist Language

Specifying and Modelling Operations on Bit Vectors

One of the problems involved in describing hardware that operates on machine words is developing an abstraction that can represent a (data) bus on which machine words can be represented and propagated. Such a bus may involve a large number of signals. In order to address the extension of operations on bit values to operations on bit vectors, the following specification of the full adder component, as given in Section 5.6.2, may be considered:
$\ll[cb\ A]\gg + \ll[cb\ B]\gg + \ll[cb\ C_{in}]\gg = \ll[cb\ C_{out},\ cb\ S]\gg$

Without the ability to refer to the value of a bit vector using a single identifier, the following specification would be required to describe the behaviour of the the ripple carry adder illustrated in Figure 6.5:
$\ll[cb\ A_3,\ cb\ A_2,\ cb\ A_1,\ cb\ A_0]\gg + \ll[cb\ B_3,\ cb\ B_2,\ cb\ B_1,\ cb\ B_0]\gg + \ll[cb\ C_{in}]\gg$
$= \ll[cb\ C_{out},\ cb\ S_3,\ cb\ S_2,\ cb\ S_1,\ cb\ S_0]\gg$

Ideally, it should be possible to use a single identifier to denote either a single bit, as before, or a whole bit vector. Thus, it should be possible to specify the component as follows:
$\ll cb\ A\gg + \ll cb\ B\gg + \ll[cb\ C_{in}]\gg = \ll[cb\ C_{out}] \cdot cb\ S\gg$

This requires some context to denote that A, B and S are each bit vectors of the required length.

It can be seen, therefore, that the use of a single identifier to refer to a bit vector (or machine word) can make specifications easier to read. It also makes it possible to directly relate a value in the netlist language semantics with a value in the software IR, which also allows a single identifier to denote a machine word.

Unlike the specification, the implementation of the ripple carry adder cannot be described in this way however. In order to describe the design in terms of its constituent full adder components, it is necessary to be able to refer to individual bits within the input and output bit vectors.

This highlights two further requirements for the netlist language. Firstly, that it be possible to refer to individual signals within a bit vector in the description of a component that operates on bit vectors. Secondly, that given a signal whose value is represented by a single bit (a 'bit signal') and a signal whose value is represented by a bit vector (a 'bit vector signal'), it is required that it be possible to connect the bit signal to any individual bit within the bit vector signal.

The required types of connections between signals are those that connect:

1. a bit signal with any other bit signal;
2. a bit signal with an individual bit within a bit vector signal; and
3. an individual bit within a bit vector signal with any other individual bit within a bit vector signal; and
4. a bit vector signal with any other bit vector signal of equal length.

Individual bits within bit vectors are identified by a zero-based index into the bit vector. In the abstract syntax, the n-th bit of a bit vector signal A is denoted by A_n. In the semantics, bit vectors are represented by lists, and the n-th element of a list v is denoted by $v_{[n]}$.

In order to ensure that bit signals cannot be connected to bit vector signals, each signal is typed. The type of bit vectors is indexed by a natural number according to the length of the bit vector. The types of connection listed above are considered well typed. Connections between a bit signal and an entire bit vector, or between two bit vectors with different sizes are considered poorly typed. No attempt is made here to provide a semantics for circuits that include poorly typed connections.

Pins on components are also typed, according to the types of signals to which they may be connected. Every pin in all the component definitions presented thus far have been of the bit type. In a (further) abuse of terminology, pins may also have a bit vector type. In the ripple carry adder specification, the terms A, B, and S represent pins of bit vector type. The same rules for connecting signals to other signals apply to connecting pins to signals. As before, pins are never connected directly to other pins, and may only be connected by a named signal.

Abstract Syntax for Bit Vectors

The abstract syntax of the netlist language must be modified in order that a design be able to refer to bit vectors, bit signals and to indexed signals within a bit vector. Instead of referring to pins and signals by name, they may be

referred to using 'pin expressions' or 'signal expressions'. Both are defined by the type

datatype *expr* =
  Var pin-name
| *Idx pin-name nat*

An expression is either a simple reference to a bit signal or bit vector, such as *Var* A, or it is a reference to an indexed bit within a bit vector, such as *Idx* A *0*. As expressions occur frequently in component designs, a more compact notation is adopted, using *'A'* for the former expression and the subscript notation for the latter: *'A_0'*. Note that the surrounding single quotation marks here are part of the notation. They are used to distinguish names from expressions. Thus, the terms A and *'A'* are of different types.

To satisfy the requirement of the language having the four connection types listed on the facing page, connections are modelled as a connection between a pin expression and a signal expression. The pin expression refers the named pins on the component being instantiated, and the signal expression refers to the named signals that connect the different component instantiations within the design.

datatype *conn* =
  In expr expr
| *Out expr expr*

As with the original *conn* type defined in Equation 5.2, the first term for each datatype constructor is associated with the pin, and the second is associated with the signal to which it is connected.

Adding a Type System

In the netlist language definition presented in Chapter 5, the list of internal signals in a design was derived from signals named within a design but not declared to be external signals in that design. The introduction of bit vectors into the language introduces some complications here.

An occurrence of a signal name within a design may refer to either a bit signal or a bit vector signal. However, in order to define well-formedness conditions for designs, the type of each signal referred to must be known statically.

Furthermore, when deriving the semantics of a given design, it appears to be useful for the resulting semantics to define a length of the existentially quantified internal bit vectors. This allows proofs about the components behaviour to use case distinction on the finite set of values that a fixed size bit vector may take.

Using the technique of comparing the signal references within a design to the list of named external signals is also inconvenient for designs that refer to individual bits within bit vectors. In general, it is not clear what the size of a bit vector should be, based only on the subscripts used.

It may possible to develop a semantics where the width of a bit vector is not known at design time, by effectively making bit vectors dynamically typed. To ensure that the netlist language may be used to target statically typed HDLs, this approach is rejected in favour of requiring that each the length of each bit

vector signal be declared in the design. Thus, types are introduced into the abstract syntax as follows:

datatype *ty* =
  Bit_T
| Vec_T *nat*

Bit signals have type Bit_T in the netlist language, abbreviated here to $\mathbb{B}$. Bit vector signals have type Vec_T n, abbreviated to $\mathbb{B}^n$, where n is the length of the bit vector.

The (meta-level) type *tyenv*, which represents the abstract syntax used to denote the external signals of a design is modified to include the type of each signal:

types
  tyenv = (*pin-name*×*ty*) *list* × (*pin-name*×*ty*) *list*

Finally, component definitions are modified to include the types of internal signals:

record *comp* =
  Ext :: *tyenv*
  Internals :: (*pin-name*×*ty*) *list*
  Insts :: *inst list*

6.3.2 Bit Vector Semantics

One of the requirements of the iterated logic extension to the netlist language identified earlier is that is must be possible for a signal name to refer to either a bit signal or a bit vector signal. This means that there must be two kinds of signal value. One kind of signal value is a temporal binding for a bit signal, as in the formulation of the netlist language described in Section 6.2. The other kind of signal value is a temporal binding for a bit vector. To support the two kinds of signal value, a new type called *val* is introduced to represent the value of a signal, to support this.

datatype *val* =
  Ind *nat*⇒*bit*
| *Vec* (*nat*⇒*bit*) *list*

The *Ind* datatype constructor denotes a value that represents the temporal behaviour of a bit signal. The *Vec* datatype constructor denotes a value that represents the temporal behaviour of a bit vector signal. The temporal binding for a bit vector is modelled as (*nat*⇒*bit*) *list* rather than *nat*⇒(*bit list*) because the length of the bit vector should not vary with time.

Signal names must now be associated with values. The *binding* type is modified become a *value binding* that associate signal names with the different kinds of values:

types
  binding = *sig-name*⇒*val*

Interpreting Signal Values and Behaviours

Two interpretation functions are defined on values. The function ${}^{v}\ll val\gg_{b}$ interprets the value *val* as a bit signal behaviour. The function simply returns

the value, with the *Ind* constructor removed. The result is of the type *nat⇒bit*. The function is under-specified: applying it to a *Vec* value gives an arbitrary signal behaviour about which no useful properties can be proven.

The function $^{v}\ll val\gg_{l}$ interprets *val* as a bit vector signal. The function simply returns the value, with the *Vec* constructor removed. The result is of the type (*nat⇒bit*) *list*. The function is under-specified: applying it to a *Bit* value gives an arbitrary bit vector signal behaviour.

Formally, these functions are defined as follows. The functions are left under-specified by omitting the cases that give arbitrary results.
$^{v}\ll Ind\ tb\gg_{b} = tb$
$^{v}\ll Vec\ tbs\gg_{l} = tbs$

As with the original combinational netlist language semantics and the temporal logic extension, a circuit behaviour is represented by the type *binding⇒bool*. A temporal behaviour, of type *sig-name⇒nat⇒bit* may be lifted into a behaviour based on a value binding using the *liftValT* function:

$liftValT\ tbehav \equiv \lambda vb.\ tbehav\ (\lambda s.\ ^{v}\ll vb\ s\gg_{b})$

This function may be composed with the *lift* function defined in Equation 6.1. The composition of these functions allows a behaviour in the original combinational netlist language semantics to be lifted directly into a behaviour in the iterated logic extension. This obviates the need to redefine the behaviour of simple combinational primitives such as logic gates.

A further function is defined that evaluates pin expressions within a given value binding. It occurs frequently in the netlist language semantics, and in the expressions that denote the behaviours derived from the definitions of components using those semantics. Therefore, like the interpretation functions defined above, it is given a similarly terse notation. The value of a pin expression *e* under value binding *vb* is denoted by *vb 'e'?*, where the query function (*?*) is defined by:

$vb\ `s'? = vb\ s$
$vb\ `s_i'? = (\mathsf{case}\ vb\ s\ \mathsf{of}\ Ind\ tb \Rightarrow arbitrary \mid Vec\ tbs \Rightarrow Ind\ tbs_{[i]})$

Connections

Two auxiliary functions are defined on the *conn* datatype to retrieve the expressions that define a connection. The *pinExpr* function returns the pin expression associated with a connection. The name of the resulting expression is expected to be the name of one of the external signals of the device being instantiated. The *sigExpr* function returns the signal expression associated with a connection.

$pinExpr\ (In\ pe\ se) = pe \qquad sigExpr\ (In\ pe\ se) = se$
$pinExpr\ (Out\ pe\ se) = pe \qquad sigExpr\ (Out\ pe\ se) = se$

In the combinational netlist language of Chapter 5, the *bindPins* function (defined in Equation 5.6) is used to map a binding in which names refer to the pins on the device being instantiated into a binding in which names refer to signals that connect the various devices instantiated within a design.

In order to support the four connection types required, this function needs to be modified. An auxiliary function, *bindPin*, is also introduced to support the new definition:

```
bindPin conn vb ≡
case pinExpr conn of
  'pn' ⇒
   case sigExpr conn of
         'sn' ⇒ vb(pn := vb sn)
       | 'sn_i' ⇒ vb(pn := vb 'sn_i'?)
| 'pn_i' ⇒
   case sigExpr conn of
         'sn' ⇒ vb(pn := Vec (v≪vb pn≫l[i := v≪vb sn≫b]))
       | 'sn_j' ⇒ vb(pn := Vec (v≪vb pn≫l[i := v≪vb sn≫l[j]]))
```

bindPins cs b ≡ *foldl* (λ*b cs*. *bindPin cs b*) *b cs*

The semantic function for components, *Mcomp* (originally defined in Equation 5.7), is also modified, to reflect the inclusion of the list of internal signals explicitly in the *comp* record.
Mcomp des denv ≡ *internalise* (*Internals des*) (*Minsts denv* (*Insts des*))

6.3.3 Adding a Row Construct to the Netlist Language

In the definition of the netlist language given in Chapter 5, a design that includes many instantiations of an existing component must list each instantiation separately. For example, a definition of the ripple carry adder illustrated in Figure 6.5 would need to include four instantiations of the full adder. The design also needs three named internal signals to form the carry chain.

For a design with only four component instantiations, defining each instantiation by hand is not a particularly arduous task, particularly if the design need only be defined once. However, verifying the device would be somewhat tedious. This problem becomes more apparent for typical machine word sizes.

Some hardware/software compilers perform *bit-width* analyses to reduce the word size required to implement a given operation in the program [BSWG01]. In turn, this can lead to a reduction in the amount of reconfigurable logic required to implement a particular operation. This is achieved in two ways. The first is to find bits that are constant at a given part of the computation, and hence do not need to be computed. The second is to find bit values in the computation that cannot affect the result of the computation. In both cases, the logic that computes the constant, or unused, bits may be removed from the hardware design. A hardware/software compiler that implements such optimisations may generate one design for each software IR operation, for all possible operand bit widths, up to that of the GPP machine word size. In order to verify compilation of a compiler that generates a component that operates on operands of varying bit widths, it is necessary to verify the component for each such bit width.

In the case of typical operand bit widths, and varying operand bit widths, it is clearly infeasible to verify components for all software IR operations for all possible bit widths individually. It may be appropriate to define a function that generates netlist language designs that operate on operands of arbitrary bit width. The arguments of the function would include the name of a component to be instantiated, and the number of instantiations of that component that should be made. The semantic functions of the language may then be applied

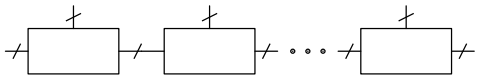


Figure 6.6: Generic model of a row of components

to the resulting designs. Using this approach, it would then be necessary to verify that the resulting designs were correct, for all such bit widths. This requires that the specification of the component be parametrised on the width of its operands.

An alternative approach is to introduce a semantic function that simply generates the semantics (or behaviour, in netlist language terms) of the resulting component. Unlike the first approach — in which a netlist language design is generated, to which the semantic functions must then be applied — the resulting behaviour of a function defined using this approach may be used directly, without further application of the semantic functions.

The second approach is taken here. Although in both cases, it is reasonable to expect that the verification of the function producing the required designs or behaviours would involve induction on the bit width of the operation's arguments.

Generic Row Abstraction

A class of iterated logic circuits is considered here in which the composition of identical components into rows is the only repeating structure used. Rows of components are considered to be of the generalised form illustrated in Figure 6.6. The pins on the repeated component are grouped into those that appear 'on the left'; those that appear 'on the right'; and those that appear neither on the left nor the right. In order to form a chain of components, there must be an equal number of pins on both the left and right of the repeated component, because each pin on the right of a component is connected to the corresponding pin on the left of the next component. The types of the corresponding pins must also match. Pins that do not appear on the left or the right do not form part of the internal chain, and become external signals in the resulting design.

Applying this to the ripple carry adder example, the C_{in} pin is considered to be on the left of the repeated component — in this case, the full adder — and the C_{out} pin is considered to be on the right. In the resulting ripple carry adder, the types of these pins are the same as that of the full adder. This means they can only be connected to bit signals. The A, B and S correspond to external signals and refer to bit vectors, which can be connected to bit vector signals.

Rows of identical components are sufficient to describe addition and subtraction, including the ripple carry example from earlier in this section; logical operations such as 'logical And' and 'logical Or'; and also components that test for zero, equality and four inequality operators, $<$, $\leq$, $>$ and $\geq$.

The test for zero operation, for example, can be implemented using a row of Or gates, as shown in Figure 6.7. In this design, the signal F is intended to be

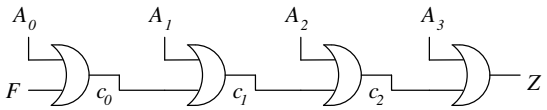


Figure 6.7: An inefficient 'test-for-zero' component, defined as a row of Or gates.

connected to a constant **0** input. In this environment, the output Z is **0** when the value on the input A represents the value zero, and **1** otherwise. In this design, the Or gate is the repeated component and the left, right and external signals are B, O and A respectively.

This implementation is somewhat inefficient, requiring time proportional to the word size before its output is guaranteed to become stable, and its inclusion here is merely intended to provide a further example of a component defined in terms of a row of repeated components.

A more efficient implementation can be implemented using a 'tree' of Or gates, requiring a delay proportional to $log_2 n$, where n is the word size. Although such an Or tree could be defined by instantiating each component individually, a better approach would be to further extend the netlist language to support the definition of trees, in a similar manner to that used here to define rows.

From the examples above, it can be seen that each pin on a component that is neither categorised as a 'left pin' or a 'right pin' has a corresponding bit vector type pin in the design resulting from composing many instantiations of that component in a row. Returning to the Or row example: the A pin in the specification of the Or gate corresponds to $A_0, \ldots, A_n$ in the resulting row, where n is the number of instantiations of the Or gate in the design. This approach makes it possible to refer to the individual bit signals, using the subscript notation, or to refer to the entire bit vector signal by omitting the subscript.

By limiting the pins that are not connected to the internal chain to the bit type, a bit vector may be used to represent those signals in the resulting design. This provides the desired property of being able to relate a single value in the netlist language with a single value in the software IR.

In contrast, the pins that do appear in the internal chain do not need to be externally visible. These should be existentially quantified in the behaviour of the device, such that their values cannot be observed externally. The pins on the left of the leftmost component instantiation, and the pins on the right of the rightmost instantiation, are exceptions to this. These should be visible in the interface of the component. However, unlike the other external pins, these should be of the same type as that in the original component.

Adding a Language Construct for Defining Rows

The only way to define new components in the netlist language as presented in Chapter 5 is via the *comp* construct, used to instantiate existing components in a new design individually, and define connections between those instantiations.

The iterated logic extension to the netlist language provides two different ways to define new components. One way is to use the *comp* construct as before. The other is to define a 'row schema' that describes how a many instantiations of a component may be connected together in a row. A row schema is defined by a new language construct in the abstract syntax, called *row*. A new component, based on that row schema, can then be instantiating for a given row length, using another language construct: *RowOf*.

The *row* construct allows row schemas to be defined as a row of instantiations of an existing component according to the conditions described above. The abstract syntax of the row construct is defined as:

record *row* =
  RowExt :: *tyenv*
  RowLeft :: *tyenv*
  RowRight :: *tyenv*
  RowComp :: *prim-name*

The value of the *RowComp* field names an existing component to be repeated to construct a row. The *RowLeft* and *RowRight* fields represent the type of the signals that form the internal chain in the row, and the external signals at each side of the chain. Note that if a signal is listed as an input in *RowLeft*, that is, if the signal is named in *fst RowLeft*, then the corresponding signal listed in *RowRight* should be an output, that is, named in *snd RowRight*. The *RowExt* field gives the name and type of each input and output bit signal of the component that will become a bit vector in the resulting design. Each signal listed in *RowExt* must be of the bit type.

As an example, the row schema that describes the way that the Or gates illustrated in Figure 6.7 are connected may defined as follows:

orRow ≡ (|
  RowExt = ([(A,𝔹)], []),
  RowLeft = ([(B,𝔹)], []),
  RowRight = ([], [(O,𝔹)]),
  RowComp =Or |)

A new abstract syntax construct is also required to allow new components to be defined either by using the *comp* construct or by instantiating a row schema for a given length using *RowOf*. A new construct is defined that forms the top-level of the abstract syntax tree for component definitions:

datatype *defn* =
  OneOf comp
| *RowOf row nat*

Note that the values associated with the *RowOf* constructor are a row schema, and a natural number indicating the required length of the row.

6.3.4 Semantics of the Row Construct

Semantics of a Row Item

The *row* abstract syntax construct includes information about the signal direction and type of each pin. Recall that the signals listed in the first element of the pairs that constitute the *RowExt*, *RowLeft* and *RowRight* fields are inputs,

and those listed in the second element of each pair is an output. This information is necessary to ensure that a circuit definition is well-typed, and does not describe a circuit in which two output pins drive a single signal. However, once a circuit has been checked for well-formedness, the information is no longer required: it has no effect on the semantics derived for a component.

The separation of inputs and outputs, and inclusion of type information in the abstract syntax complicates the semantics, due to the need to 'ignore' the typing information, and collect input and output signals into a single list. An auxiliary function is defined to concatenate a list of inputs and a list of outputs, and remove the signal types from the resulting list:

$$
\begin{array}{l}
\mathit{remTypes}\ \mathit{te} \equiv \\
\mathit{foldl}\ (\lambda \mathit{ss}\ \mathit{pair}.\ \mathit{fst}\ \mathit{pair} \cdot \mathit{ss})\ []\ (\mathit{fst}\ \mathit{te})\ @\ \mathit{foldl}\ (\lambda \mathit{ss}\ \mathit{pair}.\ \mathit{fst}\ \mathit{pair} \cdot \mathit{ss})\ []\ (\mathit{snd}\ \mathit{te})
\end{array}
$$

As in Chapter 5, a semantic object is derived for each abstract syntax construct. The semantic object associated with a *row* definition is partially determined by the behaviour of the component being instantiated, named in the *RowComp* field. The named component must have been defined previously, such that there is a dynamic environment *de* that associates the name with its behaviour. The semantic object associated with a *row* definition *ri* is the three-tuple defined by the following function:

$$
\begin{array}{l}
\mathit{MrowItem}\ \mathit{ri}\ \mathit{de} \equiv \\
(\mathit{de}\ (\mathit{RowComp}\ \mathit{ri}),\ \mathit{remTypes}\ (\mathit{RowExt}\ \mathit{ri}), \\
\quad \mathit{zip}\ (\mathit{remTypes}\ (\mathit{RowLeft}\ \mathit{ri}))\ (\mathit{remTypes}\ (\mathit{RowRight}\ \mathit{ri})))
\end{array}
\tag{6.2}
$$

The first element in the tuple is the behaviour of the component to be repeated. The second element is a list of pin names that should be externally visible as bit vectors in the resulting row semantics. The third element in the tuple is a list of pairs, in which each pair associates a pin on the left of the repeated component with the name of the corresponding pin on the right of a similar component to which it may be connected.

Generating Signal Names for Internal Signals

The *internalise* function introduces an existential quantifier for a named signal in the semantics of a component. Thus far, the signals to be internalised are those named within a design. The signal names to be internalised must be distinct from signals that are declared to be externally visible, which become pins.

In order to existentially quantify internal signals when composing a row of components, it is necessary to have a signal name to refer to each internal signal, in order to use this technique. Consideration of the row of two components illustrated in Figure 6.8 reveals why this is non-trivial. None of the signal names used in the design may be used as a name for the internal signals, because they are already being used to refer to the externally visible signals.

The netlist language requires a means to generate new signal names that may be used to name the internal signals. One way to solve this problem would be to define a function that, given a list of signal names already used in a design, generates a new unused name. However, using the functional approach used here, it would be inconvenient to have to define such a function.

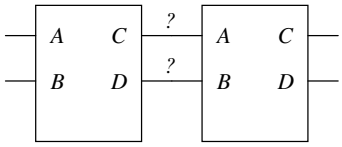


Figure 6.8: The problem of generating names for internal signals.

Instead, a technique from compiler design is adapted whereby a value in an intermediate representation may refer to either a named identifier in the source program, or to a temporary location generated by the compiler. In the source language, only the identifiers named in the source program may be referenced explicitly. The temporary identifiers used are not available in the source program, and are distinct from identifiers in the source language.

The netlist language analogue of this concept requires the introduction of a new type of signal expression, *sigTemp*. Expressions of this type may either be a signal expression that appears in a component definition, or it may be a temporary signal expression:

datatype *sigTemp* =
  Sig *expr*
| *Temp* *expr*

A new type of signal binding, *stBind*, is defined to represent signal bindings that include both signals which have been explicitly named in the design, and signals with automatically generated temporary names. The values of temporary signals are kept independent of the values of named signals by using a datatype constructor, *sigTempT*, in the domain of the new signal binding. The domain of the new type of signal binding is effectively the disjoint sum of signal names and temporary names. The behaviours admitted by a component are, as usual, a predicate on the relevant binding:

datatype *sigTempT* = *SigT* | *TempT*
types
  stBind = (*sig-name*×*sigTempT*)⇒*val*
  stBehav = *stBind*⇒*bool*

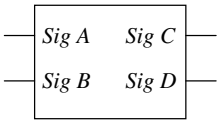
Signal bindings may be converted into an *stBind* binding using the function *biToSTbi*. An *stBind* may also be converted to the usual signal binding using the function *stBiToBi*, although this conversion loses the values of temporary signals, the names of which are not available in a *binding*, which does not include temporary signals.

biToSTBi bi ≡ λ(*s*, *t*). *if t* = *SigT then bi s else arbitrary*
stBiToBi bi ≡ λ*s*. *bi* (*s*, *SigT*)

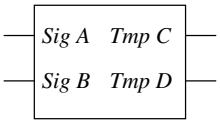
Two further functions convert between behaviours defined in terms of *binding* and behaviours defined in terms of *stBind*:

beToSTBe be ≡ λ*bi*. *be* (*stBiToBi bi*)
stBeToBe be ≡ λ*bi*. *be* (*biToSTBi bi*)

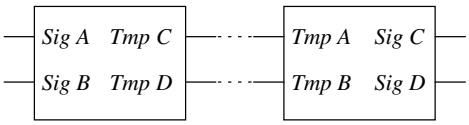
A temporary name can now be generated for each of the internal signals shown in Figure 6.8. The generated temporary name may reuse the signal names listed in the design, because the values of temporary signals are disjoint from those of signals named in the design.



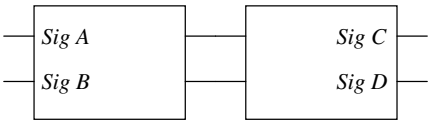
(a) Initial component.



(b) Preparing an row for composition.



(c) A row and a new component ready for composition.



(d) Row resulting from composition.

Figure 6.9: Row composition in stages

The pins that will be connected to internal signals when the two components are composed must be bound to the temporary signal names, before the components can be composed. The *tempSig* function is used to bind a pin name to a temporary signal name, where the pin name is used for the temporary value. The *tempSigs* function is the extension of *tempSigs* to lists of signal names:
tempSig be sn $\equiv \lambda bi.$ *be* $(bi((sn,\ SigT) := bi\ (sn,\ TempT)))$
tempSigs ss be $\equiv$ *foldl tempSig be ss*

Constructing Rows in the Semantics

The semantics of a row of a component are derived by a series of functions that are recursive on the n, the number of components in a row. The base case for these functions, where n=*0*, denotes a row with one component instantiation.

The intuition behind the process is as follows. An initial component is 'laid down', where the pins on its left represent signals that will be externally visible in the resulting design (Figure 6.9a). The pins on the right will only be externally visible in the resulting design if the row only has one component. Otherwise, they form the start of the internal chain.

As discussed on page 128, the pins that are not on the left or right of the component will be externally visible as part of a bit vector, and each such pin on the initial component is bound to the lowest index of a corresponding bit vector.

The row is built up incrementally, by successively adding new row elements. A new element is composed with the existing row as follows: firstly, each pin on the right of the existing row is bound to a temporary value with the same signal name (Figure 6.9b). This is necessary to avoid the those pin names conflicting

with the pin names on the new component. Secondly, each pin on the left of the new component is bound to a temporary value with the same signal name (Figure 6.9c). This is necessary to avoid the pin names on the left of the new component conflicting with the pins on the left of the existing row. Finally, an existentially quantified signal behaviour is introduced for each corresponding pair of pins, and the two pin names denoted by each pair of temporary signal names is bound to that behaviour (Figure 6.9d). Pins that to not form part of the internal chain are bound to an index within the corresponding externally visible bit vector.

After composing as many row elements as is required, the resulting *stBehav* behaviour may be converted into a *behav* behaviour, which does not have temporary signals, using the *stBeToBe* function.

Starting a Row

The first element in a row is created by binding each repeating external pin to the lowest index in a bit vector corresponding to that pin.

For this purpose, a function *tSigToVec* is defined that binds a given pin, *s*, to a given index in a bit vector of the same name as the pin for a given behaviour.

The behaviour is of the *stBehav* type that can include temporary signal names in addition to conventional signal names, and the index n is an argument, in order that the function be reusable for adding later row elements.
$tSigToVec\ n\ s\ be \equiv \lambda bi.\ be\ (bi((s,\ SigT) := stBiToBi\ bi\ `s_n'?))$

In general, a component may have many external pins. A further function is defined to bind lists of pins, *ss*, to corresponding bit vectors:
$tNamesToIdx\ ss\ n\ be \equiv foldl\ (\lambda be\ s.\ tSigToVec\ n\ s\ be)\ be\ ss$

A function is then defined that returns the (*stBehav*) behaviour of the initial row using a semantic object derived from the application of *MrowItem* to a *row* definition:

$rowBase\ mir \equiv \mathsf{let}\ (be,\ ts,\ ss) = mir\ \mathsf{in}\ tNamesToIdx\ ts\ 0\ (beToSTBe\ be)$

Extending a Row

In order to build up the row, the left-hand pins on a new row element must be bound to the temporary signals to which the pins on the right of the existing row are bound.

The third element in a tuple returned by *MrowItem* is a list of pairs, where each pair associates a pin on the left of the component with the corresponding pin on the right. Where this list is denoted by some value *ss*, the term *map fst ss* denotes the pins on the left of the component. Thus, in order to add a new component to the row, these signals are bound to temporary signals:
$rowElemN\ be\ ts\ ss\ n \equiv tNamesToIdx\ ts\ n\ (tempSigs\ (map\ fst\ ss)\ be)$

Note that is necessary to bind each external signal that is part of a bit vector to the corresponding bit in that bit vector for every row element using *tNamesToIdx*.

As the row elements are composed, the pins on the right of the previous row must be connected to the pins on the left of the new row element, and an existential quantifier introduced to hide these signals, which now form part of the internal chain. The *internaliseTemps* function takes the list of pairs that associate corresponding pins from the left and right, introduces an existentially quantified signal for each pair, and binds the pins to the quantified signal:

```
internaliseTemps [] be = be
internaliseTemps (lr·ss) be =
(let (l, r) = lr
 in internaliseTemps ss (λbi. ∃ n. be (bi((l, TempT) := n, (r, TempT) := n))))
```

Again, a further function is defined to apply these functions to the semantic object derived from *MrowItem*. This function performs the actual composition of the new element using conjunction. The behaviour of the resulting row admits signal behaviours only if they are admitted by both the existing row and the new component.

```
rowRec beh n mir ≡
let (be, ts, ss) = mir
in internaliseTemps ss
    (λbi. rowElemN (beToSTBe be) ts ss n bi ∧ tempSigs (map snd ss) beh bi)
```

Deriving the Semantics of a Row

The *rowBase* and *rowRec* functions may be used to recursively define the semantics of a row of length n, with the caveat that the result is of the type *stBehav* rather than *behav*. The recursion is defined by:

```
rowOfInt 0 mir = rowBase mir
rowOfInt (Suc n) mir = rowRec (rowOfInt n mir) (Suc n) mir
```

However, after composing a row, no temporary signal values are referred to in the resulting behaviour, and it the behaviour can be converted to a *behav* type behaviour without loss as follows:

```
rowOf n ri de ≡
let (be, ts, ss) = MrowItem ri de in stBeToBe (rowOfInt n (be, ts, ss))
```

Finally, it is also appropriate to define a semantic function that derives the behaviour of a row of components directly from the *defn* abstract syntax object:

```
Mdefn (OneOf c) de = Mcomp c de
Mdefn (RowOf ri n) de = rowOf n ri de
```

Chapter 7

Compilation Correctness

Contents

An intermediate representation typical of those found in contemporary optimising compilers was presented in Chapter 4, together with a formal definition of its semantics. It was designed to support hardware/software compilation, and hence designed such that it be possible to compile a program fragment in that representation into either assembly language, via a code generation backend, or converted into a design for a function unit.

Chapter 5 and Chapter 6 describe the development of a hardware representation with a formal semantics. This chapter discusses what it means for a function unit design expressed in the hardware representation to provide hardware acceleration for a hyperblock in the intermediate representation, and the correctness conditions that the design must satisfy.

7.1 Data Flow Between Hyperblocks

In order to state the correctness criteria for the compilation of a hyperblock in the intermediate representation into a design in the netlist language, it is useful to select a convention for propagating data values between circuits that implement hyperblocks. One reason why it is useful to select a convention is that it allows one to construct a generalised formal statement of the compilation

correctness criteria for an arbitrary hyperblock. This means that, by assuming a particular convention for data flow between hyperblocks, it is possible to state what it means for a circuit to correctly implement a hyperblock in general, and without reference to a specific hyperblock.

Another reason to select a convention for propagating data flow between circuits that implement hyperblocks is that it supports a compositional approach to reasoning about translation correctness. Two circuits can be composed together more simply if they both share a convention for synchronisation and data flow.

The convention used here for propagating values between circuits that implement hyperblocks is based on a synchronous version of the Two Phase Bundled Data Convention [Sut89]. The convention used here differs slightly from other descriptions of synchronous versions of this protocol [FDG+93, BG02a, GIOS05].

The use of this protocol is based on some assumptions about the compilation strategy in use, which are described in Section 7.1.1. A brief overview of the Two Phase Bundled Data Convention is given in 7.1.2. A description of the variant of this protocol used here, and an explanation of why it differs from the cited synchronous versions of this protocol follows in Section 7.1.3.

For brevity, this chapter henceforth uses the term 'hyperblock' to refer to a conceptual hyperblock in the intermediate representation; to a netlist implementation of that conceptual hyperblock; and to its hardware reification: a circuit implementation of that hyperblock. It is intended that context be sufficient to avoid any ambiguities. Furthermore, although the hardware semantics describes signal values in terms of the binary values **0** and **1**, the logical equivalent of those constants may be used to refer to signal values.

7.1.1 Assumptions

The compilation from the IR into the hardware representation could be approached in a variety of different ways. One approach might be to allow values to propagate between hyperblocks asynchronously, in the sense that values arrive at the merge nodes of a given hyperblock at different times. The evaluation of a hyperblock may be able to proceed before all values have propagated from a preceding hyperblock. This can occur when the results of those expressions are not required by the later hyperblock execution, or are not required immediately. The approach would support lenient evaluation: if a value is not required for a particular evaluation of the hyperblock, then there is no need to wait for it to propagate from a predecessor hyperblock.

In a description of CASH [BG02a] — a compiler framework that uses Pegasus as the basis of its intermediate representation — it is stated that only one hyperblock is actively switching at any one time. This suggests that values are propagated between hyperblocks simultaneously. However, the examples in the cited paper, and that used in the description of the semantics of Pegasus [BG02b], do not appear to illustrate how this is invariant is maintained when more than one value is propagated between consecutive hyperblocks.

In order to consider the correctness criteria for a hardware/software compiler

that uses the intermediate representation presented in this thesis, it is assumed that there is only one active hyperblock at a time. A single evaluation function for a hyperblock *mHblock* is defined on page 82 which gives a denotational semantics for the hyperblock construct. The evaluation strategy defined by the semantics involves evaluating each hyperblock sequentially: there is no consideration of two hyperblocks being evaluated concurrently, as would be appropriate for modelling the evaluation of a hyperblock before all the values from a preceding hyperblock were available.

Without the assumption of a single active hyperblock, it would be necessary to consider the parallel evaluation of two hyperblocks, and the implications of how the state transitions in one hyperblock affecting the state of the other. Hence, this assumption greatly simplifies the statement of the correctness criteria given in this chapter.

Specifying a convention for propagating data values between hyperblocks and limiting the implementation to having only one active hyperblock at a time has some disadvantages. The resulting circuits could be less efficient because of the requirement that only one hyperblock be active at a time, and also because this requirement reduces the opportunities for lenient evaluation across hyperblocks, as described above.

This approach could also limit the types of optimisations that can be performed on the resulting design, because there could be more efficient implementations based on different data flow conventions. Furthermore, the use of a single data flow convention between hyperblocks limits opportunities for inter-hyperblock optimisations.

Even if optimisations were applied after compilation — translating the design to a form that does not use the Two Phase Bundled Data Convention — it is likely to be difficult to significantly optimise the interfaces between circuits that implement hyperblocks in those optimisations. The complexity would arise as a result of modifying the hardware interface provided by a hyperblock, and yet trying to ensure that a number of related hyperblocks still exhibit the same behaviour after the optimisations have been applied. Thus, there appears to be a trade-off between the benefits of compositional reasoning about the behaviour of hyperblocks and the benefits of aggressive optimisations at the netlist level.

No convention for data flow within a hyperblock implementation is assumed here. By not mandating the use of any particular convention within a hyperblock, a greater variety of optimisations can be applied to a hardware implementation of a hyperblock. However, this work only considers the correctness criteria for compilation of a complete hyperblock. There may be some benefit in assuming some convention for intra-hyperblock data flow if compilation correctness is to be considered at a lower level of abstraction, such as at the expression level.

7.1.2 Two Phase Bundled Data Convention

The mechanism used to propagate values between circuits considered here is based upon a synchronous version of the Two Phase Bundled Data convention. This is consistent with the approach used in related literature: including CASH,

and more recently, work by Gordon et. al on compiling HOL expressions into hardware [GIOS05].

The Two Phase Bundled Data Convention involves the use of two handshaking signals used to control data flow between two circuits (or two communicating parts of a single circuit): a *load* input to signal that the input values to a circuit have been set up and are valid, and a *done* output which the circuit uses to indicate that it has completed the evaluation of its inputs, and that the results of the evaluation can be read from the outputs of the circuit. In this context, a hyperblock is implemented by a circuit.

The protocol works as follows: a positive transition is provided by the environment on the *load* signal. In the synchronous implementations described by the cited texts, this means the environment asserts a low (*False*) signal on *load* for at least one clock cycle, and then asserts a high (*True*) signal. The environment must wait until the hyperblock is asserting *True* on its *done* output to ensure that the hyperblock is ready, before it asserts the rising edge on *load*. When the hyperblock detects this transition, it asserts False on *done*, indicating that it is busy. When this happens, the environment may assert *False* on load again. When the hyperblock evaluation has completed — meaning that all of its outputs are valid and can be read — then the hyperblock should assert *True* on *done* to indicate that the environment can read those values.

7.1.3 Hyperblock Synchronisation

The Two Phase Bundled Data convention requires that each circuit using that convention have a pair of signals used for synchronisation: the *load* and *done* signals. The *done* signal of one circuit is connected to the *load* signal of the next circuit in a computation pipeline. If circuits do not form a simple pipeline, and there is more than one possible circuit that should activate when another circuit completes its computation, then some control flow logic is required to ensure the correct *load* signal is asserted for the next step in the computation.

In the intermediate representation used here, a hyperblock can have more than one predecessor, and more than one successor hyperblock. Hence, the hyperblocks in the intermediate representation do not in general map into a simple computation pipeline. This means that extra control flow logic is required, in order to 'trigger' the correct hyperblock. The correct hyperblock is determined by the semantics of the intermediate representation, and in particular the value of *mHblock-next* as defined on page 82.

The control logic could be implemented either as a centralised control unit that reifies a finite state machine, providing control logic based either on a micro-instruction set, or in logic that implements a similar finite automata. Alternatively, the control flow logic can be implemented in a distributed fashion, where each hyperblock is responsible for triggering the correct hyperblock once it has completed its evaluation.

A distributed control strategy is considered here, in which each hyperblock has one *load* signal for each predecessor hyperblock, and one *done* signal for each successor hyperblock. It is a requirement that a hyperblock never has more than one *load* signal asserted during any evaluation of that hyperblock. This

is satisfied by two restrictions: firstly, that there is only ever a single active hyperblock, and secondly, that within each block there is only a single active eta group at any time.

Conceptually, this approach seems fitting with the use of the Two Phase Bundled Data Convention because it avoids the use of a centralised control unit. It ensures that all reasoning about the behaviour of the hyperblock is determined by the hardware implementation of the hyperblock itself. This avoids the need to consider the specification of a centralised control unit, which would reduce the compositionality of compilation correctness proofs.

7.2 Example Hyperblock

An example of a hyperblock in the intermediate representation is presented in this section. The example hyperblock is used to illustrate the synchronisation mechanism described in the previous section, and in the discussion of the correctness criteria for hyperblock compilation which are discussed in the following section, Section 7.3, *Correctness Criteria*.

7.2.1 Overview

The hyperblock described here is based on the multiplication example used to describe various representations in Chapter 4. The example involves a naïve implementation of multiplication based on repeated addition: an integer number, X, is added repeatedly to an accumulator, P. The number of times that X is added to P is determined by a non-negative integer, Y, which is decremented on each loop iteration. The calculation is complete when Y reaches zero.

Figure 7.1 illustrates a hyperblock implementation of the loop body of this algorithm. It is similar to Figure 4.4 on page 68, but differs in two ways. Firstly, the loop header is not illustrated here. The loop header is a separate hyperblock, and for the purposes of this example, consideration of only a single hyperblock is sufficient. This is due to the assumption of a convention for dataflow between hyperblocks and the associated synchronisation.

Secondly, each node in the hyperblock has been annotated with a natural number. This is the *id* used in the term-graph representation, in order that where the same node appears in more than one tree in the abstract syntax, it can be identified as such by its unique identifier. The use of unique identifiers to identify nodes using a term-graph representation was discussed in Section 4.2.2, *SSA in Isabelle/HOL*.

7.2.2 Intermediate Representation of the Hyperblock

In order to simplify the presentation of the example hyperblock, a few constants are declared before providing a definition of the hyperblock in the abstract syntax of the intermediate representation. In addition to simplifying the expressions that define the hyperblock, they allow each part of the definition to be described separately.

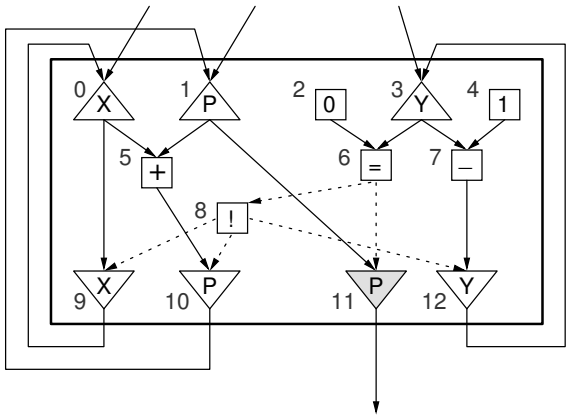


Figure 7.1: Hyperblock from the Figure 4.4 with node identifiers used in the term-graph representation.

First, a number of constants are used to identify variables which, together, form the inputs and outputs of the hyperblock:

loop-x ≡ *0*
loop-y ≡ *1*
loop-p ≡ *2*
cont-p ≡ *0*

The constants *loop-x*, *loop-y* and *loop-p* identify the hyperblock inputs, X, Y and P respectively. The constant *cont-p* identifies the hyperblock output, P. More accurately, it identifies the value that forms the input to the merge node for P in the successor hyperblock, which is not shown. Note that these values are identifiers for variables, and are used as values in domains of the hyperblock state fields, Σ_b (defined on page 79). These values need not correspond with the identifiers required for the term-graph representation: they need only be unique with respect to the other variable names used in the same hyperblock which are of the same type (where 'type' is a word value, Boolean or functional memory store).

The variable identifiers are values of the type *name*, as described in the section *SSA Tree Representation* on page 77. They can be used in the abstract syntax representation of the merge nodes of the hyperblock:

loop-x-m ≡ *WordMerge loop-x 0*
loop-y-m ≡ *WordMerge loop-y 3*
loop-p-m ≡ *WordMerge loop-p 1*

These definitions represent the merge nodes at the top of the illustration of the hyperblock. They include the term-graph identifiers for each node, as illustrated on the diagram. The term-graph identifiers can be assigned arbitrarily, with the restriction that they are unique within a given hyperblock.

Nodes with the term-graph identifiers 6 and 7 operate on the value of the Y input. These nodes — and the nodes below them in the abstract syntax tree (or

the nodes above them in the figure) — can be defined in the abstract syntax:

$activeGrp \equiv BoolDyNatOp\ Eq\ (Const\ 0\ 2)\ loop\text{-}y\text{-}m\ 6$
$eg1Y \equiv WordOp\ Sub\ loop\text{-}y\text{-}m\ (Const\ 1\ 4)\ 7$

The *activeGrp* tree is used to determine the active eta group for a given hyperblock evaluation. It compares the value of Y with the constant zero. The result forms a predicate to one of the eta groups, which determines whether that eta group is the active one. The negation of the result forms an equivalent predicate for the other eta group. The use of one value and its negation ensures the 'one-hot' property of these values.

The *eg1Y* tree is used to determine the next value of Y in the computation. It is evaluated speculatively, regardless of whether the decremented value of Y is required for another iteration.

The accumulation is implemented by the node with term-graph identifier 5. It can be defined similarly:

$eg1P \equiv WordOp\ Add\ loop\text{-}x\text{-}m\ loop\text{-}p\text{-}m\ 5$

Using these definitions, the two eta groups for the hyperblock can be defined. The eta group that is active while there are still iterations to be performed is defined as follows.

$loop_1 \equiv (\!|$
  $next\text{-}block = 1,$
  $cond = BoolMonBoolOp\ BNot\ activeGrp\ 8,$
  $\eta\text{-}bool = [],$
  $\eta\text{-}word = [\ (loop\text{-}x\text{-}m,\ loop\text{-}x),\ (eg1Y,\ loop\text{-}y),\ (eg1P,\ loop\text{-}p)\ ],$
  $\eta\text{-}mem = [\ ]\ |\!)$

Note that the *cond* field is negated with respect to the other eta group, $loop_2$, which is defined below. The *cond* field in $loop_2$ is active when there are further iterations to perform. Hence, it is active when no further iterations are required, and serves only to forward the value of P to the next hyperblock.

$loop_2 \equiv (\!|$
  $next\text{-}block = 2,$
  $cond = activeGrp,$
  $\eta\text{-}bool = [],$
  $\eta\text{-}word = [\ (loop\text{-}p\text{-}m,\ cont\text{-}p)\ ],$
  $\eta\text{-}mem = [\ ]\ |\!)$

Finally, the hyperblock is defined simply as a list of the eta groups that it contains (page 73):

$example\text{-}hb \equiv [loop_1,\ loop_2]$

7.2.3 Netlist Implementation

Figure 7.2 illustrates a netlist design which is conjectured to form a netlist reification of the example hyperblock. This netlist design is discussed in conjunction with the intermediate representation form of the hyperblock in the next section which describes what it means for a netlist design to correctly

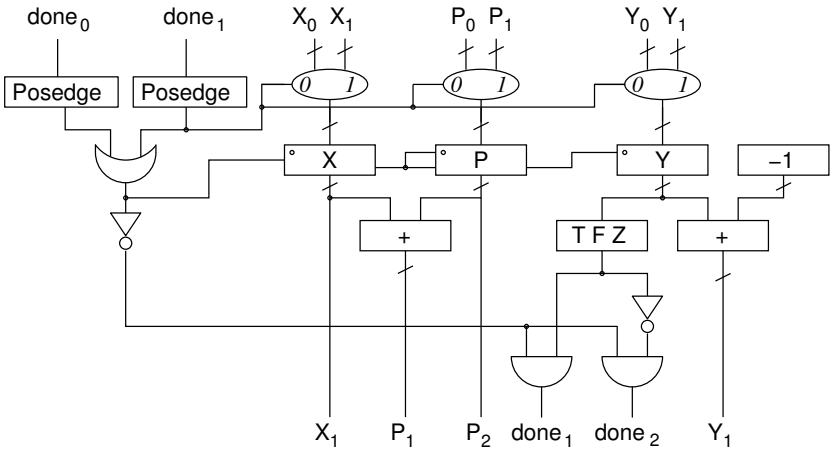


Figure 7.2: Netlist implementation of the example hyperblock in Figure 7.1.

implement a hyperblock in the intermediate representation. The netlist implementation is intended for explanatory purposes only, and has been constructed such that it can be explained in terms of the netlist language constructs presented earlier in this thesis.

No formal reasoning about the correctness, or otherwise, of the netlist implementation is attempted here: the focus is on constructing a generalised relationship between the behaviour of a hyperblock in the intermediate representation, and that of a netlist language implementation of that hyperblock with respect to the semantics associated with those representations. For similar reasons, the efficiency of the illustrated design is of no concern here. This should come as no surprise, given the choice of multiplication algorithm used in the example.

While much of the notation in Figure 7.2 should be familiar from previous chapters, some aspects of the notation used here require a little explanation. The explanation here first addresses the notation itself, before proceeding to describe the design and its relationship to the example hyperblock.

Notation

Lines between components that have a short, slanted line crossing them each represent a bus with the width of one machine word. All other connecting lines represent bit signals. Where a signal name at the bottom of the diagram matches one of those at the top of the diagram, this indicates that those two lines denote the same signal. This means, for example, that the output of the register holding the value for X is directly connected to the input to the multiplexer which selects between X_0 and X_1.

The three multiplexers at the top are an extension of the multiplexer — as used in the netlist definition of the *DFF* component (page 119) — to bit vectors or more specifically, to machine words. These can be constructed in the netlist language using the *row* construct (page 129), in conjunction with the multi-

plexer. The multiplexers are used to select input values for the hyperblock from the output values of the hyperblocks that immediately precede it in the program control flow.

The three rectangular boxes immediately below these multiplexers are abstract registers (page 118), which are an extension of the *DFF* construct to machine words.

The *TFZ* construct implements a *Test For Zero*, which could be implemented, albeit rather inefficiently, using the *row* type as shown in Figure 6.7 (page 128). The '-1' construct simply indicates that each bit signal within the bus is connected to a source that outputs a constant **1** value, and hence the value of signals across the bus represent the value *-1* in twos complement form.

At this point, the significance of selecting an appropriate representation for binary numbers (Section 3.2.2, *Number representation*) becomes apparent. The $\ll v \gg$ notation that has been used thus far to indicate the value of a bit vector is an interpretation of a bit vector as a natural number representation, rather than as a twos complement representation. The choice of the simplest definition of bit vector value has simply deferred the need to resolve the discrepancies between number representation in the intermediate representation and that in the netlist language until such time that the semantics of these representations needs to be related.

Design

The netlist implementation of the hyperblock illustrated in Figure 7.2 demonstrates the distributed control flow scheme described in Section 7.1.3, *Hyperblock Synchronisation*. In this scheme, a hyperblock may be activated by a rising transition on the *done* signal of one of a number predecessor hyperblocks.

The $done_0$ and $done_1$ signals in the top left corner respectively represent the *done* signals from the loop header hyperblock, and from the example 'loop body' hyperblock itself. Each *done* signal is connected directly to a *load* signal of a successor hyperblock, and hence there are no signals labelled as a *load* signal: the load signals are simply the *done* signals from different predecessor hyperblocks.

The $done_1$ signal near the bottom right of the diagram is connected directly to that with a similar label in the top right of the diagram. This is used by the hyperblock implementation to trigger the next loop iteration when a further loop iteration is required. The $done_2$ signal adjacent to it is raised when no further loop iterations are required, and hence when control flow should proceed to the successor of the loop body hyperblock.

The hyperblock is triggered when there is a rising transition on either the $done_0$ or the $done_1$ signal. The *Or* gate below the *Posedge* components provides this behaviour. A rising transition on either signal causes a rising transition on the clock enable signal of the *DFF*-based registers. These registers then latch the inputs of the hyperblock, either from the loop header hyperblock, or from the previous iteration of the loop body hyperblock.

One of the two *done* signals is connected to the select input of each of the multiplexers controlling the inputs to the hyperblock. When $done_0$ has a rising transition (and $done_1$ is low), the multiplexers select X_0, Y_0 and P_0 from the

loop header hyperblock. When $done_1$ has a rising transition, the multiplexers select X_1, Y_1 and P_1 from the previous iteration of the hyperblock. The latter set of signals are driven by the signals with the same name that are shown at the bottom of the diagram.

The inverter shown below the *Or* gate is used to prevent either of the *done* signals becoming high during a clock cycle with a rising transition on $done_0$ or $done_1$. This allows the input values to be latched, and the output values to be computed. This design assumes that the accumulator, P, can be incremented, and the counter, Y, decremented and (simultaneously) tested for zero within a single cycle. If it were not possible to perform all of the necessary computations within the hyperblock in a single cycle, a delay could be inserted on the output of the inverter.

Only one *done* signal can be active at any point, because the *And* gates that drive these signals share a common input which is inverted for only one of those gates.

7.3 Correctness Criteria

This section considers the correctness criteria for the translation of a hyperblock into a netlist design. It describes the properties that a netlist design must satisfy in order that it can be considered a correct implementation of a hyperblock. These properties are specific to the control strategy outlined in Section 7.1 for communication between circuits implementing hyperblocks.

It would be possible to define a different control strategy based on the same intermediate representation and netlist language. However, the use of a different control strategy would require a different set of properties for reasoning about translation correctness to those described here.

The aim is to show how the semantic functions and semantic domains of the intermediate language and of the netlist language are related. The concept is similar to that illustrated in the commutativity diagram of Figure 1.1, where the *exec* function represents the evaluation function of a hyperblock in the intermediate representation and σ represents the state of a program in the netlist language. However, the concept differs from that illustrated in that the figure, which suggests that the semantics of the lower level language has a denotational semantics. Here, the netlist language is the lower level language, and its semantics are defined as relations on signal values.

Compilation correctness is considered at the level of individual hyperblocks: the values of variables in the intermediate representation must be related to the signal values in the netlist representation only at the start and end of a hyperblock evaluation. No constraints are defined on intermediate values which are calculated during the evaluation of a given hyperblock but are not an output of that hyperblock. In the text that follows, a program variable is considered to be any value that is propagated between hyperblocks.

7.3.1 Auxiliary Functions

The correctness conditions consider the relationship between signal values in the netlist language and the values of program variables in the intermediate representation. A number of auxiliary definitions are defined here which support the correctness conditions that follow in Section 7.3.2.

A number of functions are defined on the abstract syntax of the intermediate representation. These fall into two categories: those that derive information about the control flow of the program, and those that determine what program variables are used in a given program.

Three functions relating to the control flow of a program are defined. The *precedes* function determines whether a hyperblock *hb* is an immediate predecessor of the hyperblock with index *hbi*.

$$precedes\ hb\ hbi \equiv hbi\ mem\ map\ next\text{-}block\ hb$$

The functions *preds* and *succs* are defined as follows:

$$preds\ p\ hbi \equiv [i \leftarrow [0..<|p|]\ .\ precedes\ p_{[i]}\ hbi]$$
$$succs\ p\ hbi \equiv map\ next\text{-}block\ p_{[hbi]}$$

The first argument to each function is a program, of type *prog*, as defined as a list of hyperblocks (see page 73). The second argument is an index of one of the hyperblocks in that list. The function *preds* returns a list in which each element is the index of a hyperblock that immediately precedes the hyperblock identified by the arguments. The function *succs* returns a list in which each element is the index of a hyperblock which is a successor to that identified by the arguments.

The other auxiliary functions defined on the abstract syntax are related to determining the variable identifiers used by a program. A program variable is identified by a pair of values. The first value is the number of the hyperblock in which that variable is used, that is, the index of the hyperblock within the program which includes the merge node for that variable. The second value is the name associated with that merge node. Recall from page 75 that the name associated with an eta node in the abstract syntax is that of the merge node to which it forwards a value.

The first such auxiliary function is *egVars*, which returns the variables assigned by a given eta group. The first argument to the function is one of the field selectors for η-*group* type (defined on 74), namely η-*bool*, η-*word* or η-*mem*. These return a list of pairs: the second element of each is the name of a variable assigned by that eta group, and that of a merge node in a successor hyperblock. The second argument is an eta group, of the record type η-*group* .

$$egVars\ fld\ g \equiv map\ (\lambda n.\ (next\text{-}block\ g,\ n))\ (map\ snd\ (fld\ g))$$

The function returns a list of pairs identifying program variables, one for each eta node in the field identified by the given selector. The first element of each pair is the value of the *next-block* field for the given η-*group*, and the second is one of the *name* values from the selected field. Hence, *egVars* η-*bool g* denotes a list of all Boolean program variables assigned by the eta group *g*.

Recall that each eta group corresponds to a single successor hyperblock, and the successor hyperblock to which each eta group corresponds is unique within

a hyperblock. Hence, the value of the *next-block* field is unique between each eta group within a hyperblock. Appending the program variables assigned by each eta group within a hyperblock gives a 'disjoint union' of program variables assigned by that hyperblock:

$$hbVars\ fld\ [] = []$$
$$hbVars\ fld\ (g{\cdot}gs) = egVars\ fld\ g\ @\ hbVars\ fld\ gs$$

Determining the program variables used in a whole program is a little more complicated than determining the program variables assigned by a single hyperblock. In general, a program variable may be assigned by any number of preceding hyperblocks. That is, in general, a merge node may select a value from a number of eta-nodes in preceding hyperblocks. The list union operator, $\cup_l$, is used to eliminate duplicates arising from combining the results of *hbVars* for all hyperblocks in a program.

$$progVars\ []\ f = []$$
$$progVars\ (b{\cdot}bs)\ f = hbVars\ f\ b \cup_l progVars\ bs\ f$$

Two further functions are defined to facilitate determining the variables used within a given program: *prgWords* returns a list of all variables that are represented by a machine word, and *prgBools* returns a list of all variables that are represented by a single bit.

$$prgWords\ p \equiv progVars\ p\ \eta\text{-}word$$

$$prgBools\ p \equiv progVars\ p\ \eta\text{-}bool$$

Programs with functional memory store values are not considered here. The set of components defined for the netlist language as presented is not sufficiently rich to support these values, and their use is dependent on analyses that determine potential overlaps between memory regions.

The auxiliary functions defined above are functions on the abstract syntax of the netlist language. The remaining auxiliary functions are defined on values in the semantic domains of the intermediate representation and the netlist language.

The functions ir_w and ir_b return natural numbers and a Boolean values respectively. The first argument for both functions is a program variable represented as a pair, as described above. The second argument represents the program state. They are defined as follows:

$$ir_w\ var\ \sigma_p \equiv \sigma\text{-}word\ \sigma_{p[fst\ var]}\ (snd\ var)$$

$$ir_b\ var\ \sigma_p \equiv \sigma\text{-}bool\ \sigma_{p[fst\ var]}\ (snd\ var)$$

A similar pair of functions is defined for the semantic domains of the netlist language, namely nl_w and nl_b. These return the types *bit list* and *bit* respectively. The first argument is a signal name; the second is a signal value binding, of the *binding* type defined on page 124, and the third parameter is the time at which the value of the signal is of interest.

$$nl_w\ sn\ vb\ t \equiv map\ (\lambda w.\ w\ t)\ {}^{v}{\ll}vb\ sn{\gg}_l$$

$$nl_b\ sn\ vb\ t \equiv {}^{v}{\ll}vb\ sn{\gg}_b\ t$$

Finally, a function *hasPosedge* is defined, which determines whether a given bit value has a rising transition between two points in time, t_1 and t_2.

$hasPosedge\ s\ t_1\ t_2 \equiv \exists\, t{\geq}t_1.\ t < t_2 \wedge posedge\ s\ t$

7.3.2 Correctness Conditions

In the context of this thesis, the output of a hardware/software compiler can be considered in two parts. One part is an executable program or a representation of a program that can be further translated into an executable format. The other part is a hardware design for a function unit that provides hardware acceleration for that program.

In order to allow for translation verification, it is assumed here that the compiler provides further output which includes a limited amount of information about how the hardware design relates to the executable program. Specifically, it is assumed that the compiler outputs information that maps variable identifiers in the intermediate representation to the signal identifiers in the hardware representation. This means that if the generated function unit uses a given program variable, then that mapping should define the name of the hardware signal which propagates the value of that variable.

In addition to this, it is also assumed that the compiler outputs information about which signals in the function unit design are 'load' or 'done' signals, and which hyperblocks those signals are used to synchronise.

The compiler output is modelled as follows:

record *cc-out* =
  rfu :: *design*
  prg :: *prog*
  load :: $(nat \times nat) \rightharpoonup$ *sig-name*
  vm-bool :: $(nat \times name) \Rightarrow$ *sig-name*
  vm-word :: $(nat \times name) \Rightarrow$ *sig-name*

The value of the *rfu* field is the hardware design for the function unit. The *prg* field represents the whole program, including those parts which are intended to be implemented as a hardware design in the function unit. The reason for including the whole program here is that the correctness conditions use information about the control flow and program variables derived from this program.

The *load* field is a map that identifies load/done signals between hyperblocks. It is defined on pairs of hyperblock indices. The first hyperblock index in the pair is the index of a hyperblock that is an immediate predecessor of the hyperblock identified by the second index in the pair. This means that the 'done' signal from the first hyperblock is connected to the 'load' signal of the second hyperblock. The value of the map at a given pair of hyperblocks is the name of the signal that connects the *done* output with the *load* input.

The *vm-bool* and *vm-word* fields are functions that map variable identifiers in the program to signals in the hardware design. The domain of each function is a program variable identifier (as defined in Section 7.3.1), and the range of the functions are the signal names that propagate the values of those variables. More specifically, in terms of Figure 7.2 they are the signal names that form

the input to the DFF components that are labelled with variable names and the output of the multiplexers at the top of the diagram.

The functions can be used to define what it means for the program variables in the intermediate representation to correspond with signal values at a given point in time. A predicate *varsCond* is defined to characterise this property, again with the caveat that functional store variables are not considered here. It is defined on the compiler output *co*; the signal value binding *vb*; a point in time *t*, and the state of program with respect to the semantic domains in the intermediate representation σ_p:

$$
\begin{array}{l}
\mathit{varsCond}\ \mathit{co}\ \mathit{vb}\ t\ \sigma_p \equiv \\
(\forall\, w.\ w\ \mathit{mem}\ \mathit{prgWords}\ (\mathit{prg}\ \mathit{co}) \longrightarrow \ll nl_w\ (\mathit{vm\text{-}word}\ \mathit{co}\ w)\ \mathit{vb}\ t \gg = ir_w\ w\ \sigma_p) \wedge \\
(\forall\, b.\ b\ \mathit{mem}\ \mathit{prgBools}\ (\mathit{prg}\ \mathit{co}) \longrightarrow (nl_b\ (\mathit{vm\text{-}bool}\ \mathit{co}\ b)\ \mathit{vb}\ t = \mathbf{1}) = ir_b\ b\ \sigma_p)
\end{array}
$$

Intuitively, the correctness condition can be described as follows: if an abstract interpreter for the intermediate representation is evaluating a program in steps of a single hyperblock evaluation, then the *varsCond* property should hold at the end of each such step. A more precise statement of this property requires stating precisely the arguments for which the property should hold, and also the conditions under which it is required to hold.

The correctness of a hardware implementation of a hyperblock is based on certain assumptions about the environment of the hyperblock implementation. For example, a hyperblock is not expected to produce meaningful results if a rising transition occurs on a 'load' signal unless the hyperblock is asserting *True* on all 'done' signals. These assumptions related to load/done signal values are characterised by a predicate *hbAssm*, which is defined on the compiler output *co*; a signal value binding *vb*; the time *t* at which the hyperblock evaluation starts, and the time *t'* at which the hyperblock evaluation is complete.

$$
\begin{array}{l}
\mathit{hbAssm}\ \mathit{co}\ \mathit{vb}\ t\ t'\ i \equiv \\
\quad (\forall\ s.\ s\ \mathit{mem}\ \mathit{succs}\ (\mathit{prg}\ \mathit{co})\ i \longrightarrow \\
\qquad (\exists\ d.\ \mathit{load}\ \mathit{co}\ (i, s) = \mathit{Some}\ d \longrightarrow {}^{v}\!\ll \mathit{vb}\ d \gg_b t = \mathbf{1})) \wedge \\
\quad (\exists!\ p.\ p\ \mathit{mem}\ \mathit{preds}\ (\mathit{prg}\ \mathit{co})\ i \longrightarrow \\
\qquad (\exists\ l.\ \mathit{load}\ \mathit{co}\ (p, i) = \mathit{Some}\ l \longrightarrow \\
\qquad \mathit{posedge}\ {}^{v}\!\ll \mathit{vb}\ l \gg_b t\ \wedge \neg\ \mathit{hasPosedge}\ {}^{v}\!\ll \mathit{vb}\ l \gg_b (t{+}1)\ t' \wedge \\
\qquad (\forall\ d.\ d \in \mathit{ran}\ (\mathit{load}\ \mathit{co}) \wedge d \neq l \longrightarrow \neg\ \mathit{hasPosedge}\ {}^{v}\!\ll \mathit{vb}\ d \gg_b t\ t')))
\end{array}
$$

The time *t* refers to the start of a rising transition on one of the load signals: that is, a load signal that has value **0** at time *t* and value **1** at time *t+1*. The time *t'* refers to the time at which the hyperblock raises a 'done' signal to indicate completion of the hyperblock evaluation.

The first conjunct (defined in the first two lines of the *hbAssm* definition) represents the assumption that the hyperblock is asserting **1** on all done signals at time *t*. The third line introduces a bound variable *p* which is the hyperblock index of a unique hyperblock that has asserted **1** on one of the load signals of the current hyperblock. The fourth and fifth lines define the assumption that the corresponding load signal *l* has a rising transition starting at time *t* and does not have a subsequent rising transition during the evaluation of the hyperblock. The last line states that there are no rising edges on any other load signals in the entire design during the hyperblock evaluation.

The last line is based on the requirement that only a single hyperblock be active at any time. Hyperblocks are required to be quiescent when they are not active: they must not create a rising transition on a done signal that is not due to the completion of a hyperblock evaluation triggered by a done signal. Were this not a requirement, a spurious rising transition on a done signal would result in more than one active hyperblock. This could trigger a series of hyperblock evaluations which could result in more rising load transition on a hyperblock. This property is not formalised here.

The predicate *hbAssm* defines assumptions on signal values of signals used for control flow. A predicate *hbCorr* is defined which characterises the correct behaviour of control flow signals assuming that the *hbAssm* holds.

$$
\begin{aligned}
&\mathit{hbCorr}\ \mathit{co}\ \mathit{vb}\ t\ t'\ \mathit{hbi}\ \mathit{next} \equiv \\
&\quad (\forall\ d \in \mathit{ran}\ (\mathit{load}\ \mathit{co}).\ \neg\ \mathit{hasPosedge}\ {}^{v}\!\ll\!\mathit{vb}\ d\!\gg_b\ t\ t' \wedge \\
&\qquad \mathit{posedge}\ {}^{v}\!\ll\!\mathit{vb}\ d\!\gg_b\ t' = (\mathit{load}\ \mathit{co}\ (\mathit{hbi},\ \mathit{next}) = \mathit{Some}\ d))
\end{aligned}
$$

The property is defined as a function of the compiler output *co*; the signal value binding *vb*; the start time of the rising transition on a load signal t for the current hyperblock evaluation; the start of the rising transition on the done signal *t'*; the index of the current hyperblock within the program *hbi*, and the index of the next hyperblock to be evaluated *next*.

The first conjunct represents the requirement that the hyperblock should not effect a rising transition on any done signal for the duration of the hyperblock evaluation. This applies to all done signals in the design. The second conjunct states that time *t'* should be the start of a rising transition on the correct done signal. More verbosely, it states that at time *t'* the property *posedge* holds of a given load/done signal if and only if that signal connects a done output of the current hyperblock with the correct load input on the next hyperblock to be evaluated.

A state transition effected by a hyperblock in the intermediate representation must commute with the state transitions that occur on its hardware implementation. Using the above definitions, a statement about the behaviour that the function unit is required to exhibit, in order for the state transitions effected by a single hyperblock to commute can be formulated:

$$
\begin{aligned}
&\mathit{commute\text{-}at}\ \mathit{co}\ \sigma_p\ {\sigma_p}'\ \mathit{hbi}\ \mathit{next}\ \mathit{vb}\ t\ t' \equiv \\
&\quad \mathit{hbAssm}\ \mathit{co}\ \mathit{vb}\ t\ t'\ \mathit{hbi} \wedge \mathit{varsCond}\ \mathit{co}\ \mathit{vb}\ (t{+}1)\ \sigma_p \longrightarrow \\
&\quad \mathit{hbCorr}\ \mathit{co}\ \mathit{vb}\ t\ t'\ \mathit{hbi}\ \mathit{next} \wedge \mathit{varsCond}\ \mathit{co}\ \mathit{vb}\ t'\ {\sigma_p}'
\end{aligned}
$$

In the above definition, *co* is the compiler output, of type *cc-out*. The intermediate representation states σ_p and $\sigma_{p'}$ are program states before and after the evaluation of a hyperblock respectively. The terms *hbi* and *next* refer to the index of the hyperblock being evaluated, and the index of the hyperblock which should be evaluated immediately after the current evaluation. The signal value binding is denoted by the term *vb*, and t and *t'* represent the times (in the netlist language semantics) at which the hyperblock evaluation starts and finish respectively.

The correctness statement above is formulated using implication. If the environment of the hyperblock does not maintain the invariant denoted by *hbAssm*, then arbitrary behaviour is admissible. However, if the assumptions about the behaviour of the environment hold, then an implementation must implement

the correct synchronisation behaviour, as denoted by *hbCorr*, and effect a state transition ensures signal values representing program variables correspond with the variables in the intermediate representation state.

It remains to formulate the property that characterises whether the evaluation of an arbitrary hyperblock in the netlist language to commute with an implementation of that hyperblock in the netlist language. This can be expressed as follows:

commute co hbi σ_p *vb t* ≡
(*let* (*next*, $\sigma_p{}'$) = *mHblock* (*prg co*!*hbi*) *hbi* σ_p *in*
  ∃ *t*′. *commute-at co* σ_p $\sigma_p{}'$ *hbi next vb t t*′)

The above condition forms part of the correctness criteria for a hyperblock implementation. The aforementioned condition on quiescent behaviour is also required for correctness. These conditions specify what it means for a hardware implementation of a hyperblock to correctly implement a hyperblock in the intermediate representation.

Chapter 8

Discussion and Conclusions

Contents

Compiler correctness is one of the requirements for producing robust programs in high level languages. Compilers that target a reconfigurable function unit — or other, less volatile, technology — are no exception to this. However, existing techniques that support rigorous compiler development cannot be applied to hardware/software compilers directly. This is unsurprising: many of the theoretical foundations pertaining to compiler development that underlie the approach adopted in this thesis were developed in the late 1970s, approximately ten years before FPGAs began to be used for the implementation of custom function units. Even then, initial efforts were undertaken without the support of a hardware/software compiler.

8.1 Formulation of the IR

An intermediate representation based on the Static Single Assignment form has been presented in Chapter 4. It is based on similar representations: namely a formulation of SSA by Jan Olaf Blech [Ble04], and Pegasus, a representation based on hyperblocks [BG02b]. Each of these involves representing the data-flow within a basic block as a graph in which nodes denote computations, and edges denote data dependencies.

All three definitions use a different approach to representing the data-flow graph. Blech uses a term-graph representation. The term graph representation

is convenient for expressing semantics as recursive functions over an abstract syntax tree. Blech's term-graph approach for representing SSA form has been adapted for the definition presented here.

This section provides a comparison of the features of each representation, and summarises contributions made in adapting the existing representations in order to make them more amenable to verifiable hardware/software compilation.

8.1.1 Comparison of SSA Representations

Abstract Syntax of the SSA Representation

Each node in Blech's term graph has the type *SSA_tree*, regardless of whether the node represents a value or a memory store. A single primitive recursive semantic function, *eval_tree*, is used to recursively evaluate trees.

In contrast, in the definition given here, there are three different types of node in a tree: those that evaluate Boolean values, (scalar) machine word values, and those represent a memory region. Three mutually recursive functions are used to evaluate term-graphs: *mη-boolTree mη-wordTree* and *mη-memTree* (page 84). Each function evaluates a tree of a given type.

The distinction between these two approaches is that the definition here uses abstract syntax to enforce certain type safety properties: it is not possible to represent a tree in which a memory store is used where a scalar value is expected. In Blech's formulation, the abstract syntax admits SSA trees that are not type safe in this respect, and their behaviour is left underspecified.

The **record** type in Isabelle/HOL is used to represent blocks in the abstract syntax of the representation developed in this thesis (page 74). This appears to provide an improvement over the use of a single datatype constructor with a number of anonymous fields (page 65), because the field names of the record type provide a mnemonic indicative of their purpose.

Semantic Domains

Another distinction between the two representations is that Blech's formulation stores intermediate values in the abstract syntax object representing the node. The definition given in Chapter 4 uses a separate semantic domain object, Σ_b (page 79), to represent the state for the evaluation of a tree.

An advantage of this approach is that it is clear from the signatures of the semantic functions that they preserve the structure of the SSA trees, and hence that the part of the program represented by that SSA tree is invariant during program evaluation. This is an important property for hardware/software compilation, because program modification would require re-synthesis of the function unit logic.

Another benefit of the use of a separate semantic object to represent the evaluation state is that it is straight-forward to retrieve the values of nodes representing live variables after a block has been evaluated (by a simple function application). In contrast, where values are associated with the SSA node that

evaluated them, these values must be retrieved by a recursive function after a block has been evaluated.

Flow Control

The IR developed in Chapter 4 provides improved support for modelling flow control in comparison to Blech's representation. Blocks may have an arbitrary number of successor blocks, and are selected by one of a number of Boolean predicates. Only one Boolean predicate is expected to evaluate to *true*, and this is used to determine the successor block.

This provides for a simpler representation of constructs such as switch statements, where there are more than two possible successor blocks. Additionally, where a block has only a single successor, it avoids redundancy in the representation: whereas a block must have exactly two successors in Blech's representation, in the formulation given here, a successor need be specified only once if it is the only successor.

8.2 Co-design of an IR and Netlist Language

This thesis identifies the need for formal reasoning to improve the level of rigour in the development of hardware/software compilers. To support formal reasoning, an approach is advocated that involves extending existing techniques to suit the requirements of hardware/software compiler verification.

The verification of a compilation algorithm requires a formal definition of source and target languages (§1.1.2). When considering targeting a hardware description language, it may be more appropriate to target a low level language in preference to a high level HDL with no tractable formal semantics and a more complex mapping to the targeted technology. Otherwise, the task of verified compilation is not likely to be solved, but merely deferred until a later stage (§ 3.3.1, § 5.1).

With this strategy in mind, the formal semantics of an intermediate representation and a target HDL were developed and presented in this thesis. The target HDL is relatively low level, and intended to admit a straight-forward translation into a target-independent netlist language supported by a toolchains for reconfigurable hardware.

8.2.1 Developing a Netlist Language

The development of a new hardware representation for this thesis raises the question of whether it would be wiser to re-use an existing hardware representation, and attempt to derive a relationship between the existing hardware representation and an intermediate representation. A number of factors, which are described here, affected the decision to opt for the development of a new representation.

One of the most significant attributes of the IR and netlist languages is that they have been developed together with the intention that it be simple to

relate one to the other. The languages are designed to provide a similar set of abstractions: Section 3.2 described how these abstractions can be selected in order to admit direct comparison of constructs in the IR and hardware representation, specifically those that represent machine words and memory regions.

Although the netlist language presented in this thesis has not been developed sufficiently to represent memory regions, it is anticipated that an extension to support this feature would benefit from the analysis presented in Section 3.2 of the desired properties of those abstractions. However, sufficient theoretical support has been developed in order that the value of machine words can be represented as natural numbers in the IR, and the compared with their corresponding bit vector representation in the netlist language (§6.3). Hence, by developing a new representation, it has been possible to construct a hardware representation that offers abstractions that can be related directly to those the IR.

Another significant factor in the decision to design a new language was that of tool support. It is desirable to be able to use an automated theorem prover to improve confidence in theoretical results, to help finding errors in a proof, and to assist with proof exploration. A prerequisite for relating the semantics of the IR and netlist language using a theorem prover is that the semantics of each be formulated in the same theorem proving environment — in this case, Isabelle/HOL.

Mechanised reasoning about the relationship between the IR and an existing hardware description language with formal semantics [EB02, Mil85] could be achieved by one of two methods. One method would be to reformulate the semantics of a hardware description language in the logical framework used to define the semantics of the IR. The other method would be to formulate the semantics of the IR in a logical framework used to describe an existing HDL.

Both of these methods have disadvantages: in particular, they lack the benefits of having an IR and netlist explicitly designed for the purpose of hardware/software co-design. By reformulating the semantics of an existing HDL within Isabelle/HOL, it would be difficult to ascertain whether a faithful representation of the original semantics had been constructed. Furthermore, the use of an approach which is based on the VERITAS technique (§ 3.3.2) of modelling hardware in higher order logic has resulted in a semantic definition that fits relatively naturally in the Isabelle/HOL framework. That is, while it would be possible to formulate the semantics of an HDL with an operational semantics in the Isabelle/HOL framework, the benefits of the VERITAS approach would be lost.

The other method — that of formulating the IR in the logical framework used to describe the semantics of an existing HDL — also had certain disadvantages, at least at the time the decision was made to construct a new representation. At that time, there did not appear to be a readily available formal semantics of a hardware representation constructed in a theorem proving environment that would have been appropriate for direct comparison with the desired formulation of the IR used here.

The Quartz language [PL05] was being developed at Imperial College concur-

rently with the development of the netlist language for this thesis. Quartz has many of the features that were desired for the target hardware language considered here, for example, unlike the Ruby HDL [Hut93] ports on a component are typed to indicate the direction of their associated signals. However, it was not clear the extent to which the abstract objects used in the Quartz system correspond with primitives provided by targeted hardware device.

Quartz also appears to have been developed as a new object logic in Isabelle, rather than as a theory within the Isabelle/HOL. Another problem that would have likely arisen in an attempt to re-use it for the purposes here is that the IR presented here relies on many of the constructs and results provided by the Isabelle/HOL framework, such as mutual recursive function definitions.

8.3 Formulation of the Netlist Language

The netlist language presented in this thesis is based on a technical note by Richard Boulton [Bou98], but bears a number of differences to the original.

Boulton's work was based on the HOL system [Gor85], rather than Isabelle/HOL. This distinction seemed to have little impact on the actual formulation. Although some differences have been introduced here, the decisions to make these changes have largely been either based on arbitrary choice, or in order to adapt to the different proof libraries provided by Isabelle/HOL. An example of the latter was the use of the Isabelle/HOL *bit* type instead of the *bool* type for signal values. However, with some extra work, it is anticipated that a formulation using the *bool* type could have been produced, if greater similarity with the original semantics were desired.

8.3.1 Representation of Signal Bindings

Boulton showed the results of his HOL proof procedures to apply and 'unfold' his semantics to a design in the netlist language. This process has been repeated here, although the actual unfoldings used differ to the original. For example, Boulton defines a `get` function to retrieve the value of a named signal from a signal binding. The function does not appear in the semantics here: instead the signal binding is a function that can be applied directly to signal names.

The `get` function allows abstraction from the actual representation of the signal binding. It appears possible to represent the binding as a function in HOL, or as a list of pairs that associate signal names with their values, with few changes to the semantics. However, this property appeared unnecessary for the formulation here and hence, in the simple formulation presented in Chapter 5, the function was removed by using signal bindings in expressions directly where signal values are used.

In retrospect, the decision to remove the `get` function may have been unwise. The proof procedures used to apply and unfold the semantic definitions are complicated by the difficultly in preventing unwanted beta reduction in Isabelle/HOL when performing simplification of other parts of an expression. For this purpose, it may have been simpler to define a similar *get* function in Isabelle/HOL, and remove its definition from the simplification set used during

some of the simplifications. This problem occurred mostly when using function updates on signal bindings as described in Section 5.6.1.

The missing *get* function was reintroduced in Chapter 6 (§6.3.2) as the query function (*?*). Its purpose is to provide support for different signal types within the same binding, in order to avoid having one signal binding for each type of signal. This benefit of applying a function to a signal binding was not apparent from Boulton's original work, which describes netlists in which each signal represents only a single bit of data.

8.3.2 Theorem Proving Techniques

In addition to the proof tactics that were developed to apply and unfold the semantics, a proof tactic was developed to simplify the process of reasoning about combinational logic. The tactic is given list of signal names, and applies case distinction on the *bit* values corresponding to those signal names. For n signals, this results in 2^n subgoals, representing each combination of values that the names signals could take (irrespective of the constraints imposed by the circuit design). Simplification can usually be used to prove these subgoals automatically. This technique is simple, if rather naïve, but effective for small circuits. However, the approach becomes slow on larger circuits due to the number of unnecessary splits.

One of the difficulties that arises when using a signal binding to represent the value of all signals, rather than a variable for each signal, is that Isabelle/HOL has better support for case distinction on a single variable. A workaround for this problem is to assert that a new variable is equal to the value of a signal binding for some signal, then substitute this variable into the expression that requires case distinction, and finally perform case distinction on that new variable.

The netlist language was introduced in stages: starting with combinational logic, then extended to support temporal logic, and finally support for iterated logic. The 'lift' functions, *lift* (§6.2.1) and *liftValT* (§6.3.2), proved useful when experimenting with these different language definitions, because semantic objects developed in the simpler formulations could be re-used in the more complex definitions, without having to reformulate all primitives and components by hand each time.

Another difficulty that may arise when using signal bindings is that it is not clear if it would be possible to use the polymorphism in the Isabelle/HOL type system to support polymorphism in the netlist language. However, while polymorphism is desirable in an HDL designed for human use (such as that provided by Quartz), its advantages are less clear in a low level representation such as the netlist language, which is intended to represent an intermediate state of hardware/software compilation.

8.3.3 Temporal Abstraction in Higher Order Logic

Tom Melham has described a set of proofs related to temporal abstraction in higher order logic [Mel93]. The proofs were used to prove that a model of a

D-type flip-flip satisfies the specification of a device similar to the *TDel* unit delay device described in Section 6.2.1 (see page 115). The theorems in the original set of proof scripts have been proved in Isabelle/HOL as part of the research for this thesis, although the new proofs have not been presented here.

The original proofs were constructed in the mid-1980s using the HOL system. These have been re-written as tactic-based proof scripts for Isabelle/HOL. The theorems proved in Isabelle/HOL have the same form as those of the original proofs. For example, where the original proofs used universal quantification, object level quantification was used in the Isabelle/HOL proofs. This ensured term-for-term correspondence in the theorems proved.

There were several complications involved in 'porting' them to Isabelle/HOL. Firstly, object-level universal quantification made the theorem proving more complex. In order to use the automatic unification in Isabelle, it is necessary to use terms with meta-level quantification. Thus, although it was desirable to prove the theorems in Melham's theory exactly as they appeared, the form of those theorems was perhaps not the most appropriate form for proving in the Isabelle system. However, all of the required theorems were proved exactly as they appeared in the original, albeit that some of the Isabelle proof text was more cumbersome that it could have been otherwise.

Other problems encountered while porting the proofs were due to the intuition behind the proofs not always being clear. In part this was because the tactics used were unfamiliar from the perspective of a HOL novice. The description of at least one tactic used in the HOL version could not be found in the HOL manual. This may have been because it was undocumented, had been removed from the HOL system, or had been renamed.

The HOL proofs used a larger number of tactics than were required for the Isabelle proofs. In general, the tactics used appeared to have a very specific use. Fewer tactics were used to construct my Isabelle proofs. In part this was due to a lack of familiarity with the system and the available tactics. Interestingly, the proof scripts for each version are of similar length. This seemed to be because the use of more powerful proof tactics compensated for the rather unnatural expression of theorems in the Isabelle system. It is difficult to make an exact comparison of the actual number of tactic applications made, although the HOL proofs tend to contain more commands on each line.

Isabelle is intended to allow 'literate proofs' to be developed, that is, proofs that convey the intuition of a proof in the way that a pencil-and-paper proof might. However, for non-trivial proofs, producing a literate proof can be difficult without a significant amount of experience with Isabelle, even with a tactic based script or accurate pencil-and-paper proof available. It was encouraging to note that it was possible to present the full adder correctness proof presented in Chapter 5 in a form that was very similar to a proof written by hand.

8.3.4 Well-Formed Circuits

The netlist language definition does not include a formal definition of the context conditions (§1.2.2) that characterise the set of netlists for which the semantics is intended to define behaviour. Hence, there are netlist descriptions that

conform to the abstract syntax of the language for which the semantics should not be applied, because they will not give define a meaningful behaviour. It would be useful to define context conditions for the netlist language to define the set of circuits considered well-formed.

Hoare observed that the VERITAS approach to modelling hardware in higher order logic can lead to misunderstandings of the behaviour of a device being modelled, because the inputs and outputs of a component cannot be determined from the terms representing a circuit [HG88]. The netlist language defined in this thesis uses the *In* and *Out* annotations on the external pins of a device to indicate this, and can be used to define context conditions.

Stylistically, it may have been more appropriate to erase the *In*/*Out* typing annotations prior to applying the semantic functions in the netlist language. In the final version of the netlist language presented in the latter half of Chapter 6, writing semantic functions on abstract objects that included typing information became more cumbersome. The *remTypes* function (§6.3.4, page 130) was introduced to simplify the semantic functions by erasing the type annotations.

An example of a class of circuits that should be excluded by the context conditions includes circuits were a signal is driven by more than one output pin of a component (§5.3.1, page 90). This is a property of circuits that include tri-state buffers. VHDL allows the definition of signal resolution functions to provide a meaningful semantics to this type of circuit. However, it would be simpler to exclude them from consideration. This is reasonable, because synthesis tools for many FPGAs convert tri-state logic into an equivalent design based on multiplexers.

The original inspiration to annotate signals in the netlist language with a direction came from Boulton's formulation of a netlist language, although even there, the idea was not novel. Alternatives to such annotations were considered. One such alternative was to model combinational logic as a function, and iterate the application of this function for each clock cycle. This is essentially the iterated map approach (see *Top-level Semantic Functions*, page 81).

An advantage of this approach is that it provides a deterministic and executable model that is simple to test using validation conjectures. In contrast to the relational approach to modelling circuits, it is clear that there can only be one successor state.

During correspondence with Oliver Pell — who was working on Quartz — Pell explained that the Quartz compiler converted relational descriptions into functional circuit descriptions. While it was not clear (to either of us) whether this approach was useful in improving the executability of circuit model, he noted that the conversion "destroy[ed] a lot of the useful simplicity of the original relational description". Development of Quartz required similar issues to those encountered in the development of the netlist language to be be addressed. This included modelling the signal direction, and ensuring well-formedness in that respect. Quartz also appears to have avoided the issue of modelling multiple signal drivers, presumably because it was also intended for targeting FPGAs.

It is important to distinguish between using functions to specify combinational behaviour, as described above, with using functions to specify sequential behaviour, as used in the work of Gordon et al. on hardware compilation from

HOL [GIOS05]. In their hardware compilation technique, there is no representation of the generated circuit that has signal directions annotations.

The HOL compilation work has an advantage over the approach used in this thesis, in that no new proof tactics need to be developed in order to obtain a representation of a circuit as a higher order logic term that can be used for verification. However, their compiler is specific to the HOL system and its particular formulation of higher order logic. In contrast, the formulation of the netlist language separates the abstract syntax from the semantics, and an alternative semantics for the abstract semantics could be defined in a different formal system. However, it is unclear whether there would be any advantage of doing so.

Bibliography

[Alt05] Altera Corporation, 101 Innovation Drive, San Jose, CA 95134. Cyclone II Device Handbook, Volume 1, Jul. 2005.

[AMD00a] Advanced Micro Devices, Inc. 3DNow! Technology Manual, 2000.

[AMD00b] Advanced Micro Devices, Inc. AMD Extensions to the 3DNow! ® and MMX ® Instruction Sets, 2000.

[AMD02] Advanced Micro Devices, Inc. AMD Athlon ® Processor x86 Code Optimization Guide, 2002.

[App97] Andrew W. Appel. Modern Compiler Implementation in ML: Basic Techniques. Cambridge University Press, 1997.

[App01] Andrew W. Appel. Foundational Proof-Carrying Code. In LICS 2001: Proceedings of the 16th Annual Symposium on Logic in Computer Science, pages 247–256, Boston, MA, Jun 2001. IEEE Computer Society Press.

[ARM04] ARM Limited. SafeNet EIP-25 Datasheet, 2004. Last modified date: 25 February 2004.
`http://www.arm.com/miscPDFs/1753.pdf`.

[AS93] Peter M. Athanas and Harvey F. Silverman. Processor Reconfiguration Through Instruction-Set Metamorphosis. IEEE Computer, 26(3):11–18, 1993.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. Compilers: Principles, Techniques and Tools. Addison-Wesley, Reading, Massachusetts, 1986.

[ATB05] Georg Acher, Carsten Trinitis, and Rainer Buchty. CPU-independent Assembler in an FPGA. In FPL'05: Proceedings of the 15th International Conference on Field Programmable Logic and Applications, pages 519–522, 2005.

[Bar93] Janet E. Barnes. A mathematical theory of synchronous communication. Dphil thesis, Oxford University Computing Laboratory, 1993.

[BD02] Francisco Barat and Rudy Lauwereins Geert Deconinck. Reconfigurable Instruction Set Processors from a Hardware/Software Perspective. IEEE Transactions on Software Engineering, 28(9):847–862, 2002.

[BFL+93] Juan C. Bicarregui, John S. Fitzgerald, Peter A. Lindsay, Richard Moore, and Brian Ritchie. Proof in VDM: A Practitioner's Guide. Formal Approaches to Computing and Information Technology (FACIT). Springer-Verlag, Dec 1993.

[BG02a] Mihai Budiu and Seth Copen Goldstein. Compiling Application-Specific Hardware. In Proceedings of the 12th International Conference on Field Programmable Logic and Applications, Montpellier (La Grande-Motte), France, September 2002.

[BG02b] Mihai Budiu and Seth Copen Goldstein. Pegasus: An Efficient Intermediate Representation. Technical Report CMU-CS-02-107, Carnegie Mellon University, May 2002.

[BG04] Jan Olaf Blech and Sabine Glesner. A Formal Correctness Proof for Code Generation from SSA Form in Isabelle/HOL. In Proceedings der 3. Arbeitstagung Programmiersprachen (ATPS) auf der 34. Jahrestagung der Gesellschaft für Informatik. Lecture Notes in Informatics, September 2004.

[BGG+92] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. Van Tassel. Experience with Embedding Hardware Description Languages in HOL. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience, volume A-10 of IFIP Transactions, pages 129–156, Nijmegen, The Netherlands, June 1992. North-Holland/Elsevier.

[BGHT90] R. Boulton, M. Gordon, J. Herbert, and J. Van Tassel. The HOL Verification of ELLA Designs. Technical Report 199, University of Cambridge Computer Laboratory, August 1990. Revised version in Proceedings of the International Workshop on Formal Methods in VLSI Design, Miami, Florida, January 1991.

[BGMW94] Howard Barringer, Graham Gough, Brian Monahan, and Alan Williams. A Process Algebraic Semantics for Core ELLA. Technical Report UMCS-93-2-1, Manchester University, Nov 1994.

[BHX00] Jonathan P. Bowen, He Jifeng, and Xu Qiwen. An Animatable Operational Semantics of the Verilog Hardware Description Language. In John A. McDermid Shaoying Liu and Michael G. Hinchey, editors, ICFEM'00: Proceedings of the 3rd IEEE International Conference on Formal Engineering Methods, pages 199–207. IEEE Computer Society Press, 2000.

[BJ78] Dines Bjørner and Cliff B. Jones, editors. The Vienna Development Method: The Meta-Language, volume 61 of Lecture Notes in Computer Science. Springer-Verlag, 1978.

[BJ82] Dines Bjørner and Cliff Jones. Formal Specification and Software Development. Prentice-Hall, 1982.

[BJC+03] Francisco Barat, Murali Jayapala, Tom Vander Aa Henk Corporaal, Geert Deconinck, and Rudy Lauwereins. Low Power Coarse-Grained Reconfigurable Instruction Set Processor. In Peter Y. K.

Cheung, George A. Constantinides, and José T. de Sousa, editors, FPL'03: Proceedings of the 13th International Conference on Field Programmable Logic and Applications, volume 2778 of Lecture Notes in Computer Science. Springer, Sept. 2003.

[Bjø82] Dines Bjørner. Rigorous Development of Interpreters and Compilers, chapter 9, pages 271–320. In [BJ82], 1982.

[BL00] Francisco Barat and Rudy Lauwereins. Reconfigurable Instruction Set Processors: A Survey. In 11th IEEE International Workshop on Rapid System Prototyping (RSP 2000), page 168, June 2000.

[Ble04] Jan Olaf Blech. Eine formale Semantik für SSA Zwischensprachen in Isabelle/HOL. Diplomarbeit. Universität Karlruhe (TH), Institut für Programmstrukturen und Datenorganisation, Mar 2004.

[BM95] Peter T. Breuer and Natividad Martínez Madrid. A Native Process Algebra for VHDL. In Proceedings of European Design Automation Conference with EURO-VHDL '95 on EURO-DAC '95 with EURO-VHDL '95, pages 420–426. IEEE Computer Society Press, 1995.

[BN99] Franz Baader and Tobias Nipkow. Term Rewriting and All That. Cambridge University Press, Aug 1999.

[Bou98] Richard Boulton. A Semantics for a Simple Netlist Language. Technical report, February 1998.

[Bow99] Jonathan P. Bowen. Animating the Semantics of VERILOG using Prolog. Technical Report 176, United Nations University, UNU/IIST, P.O.Box 3058, Macau, China, 1999.

[BR96] Stephen Brown and Jonathan Rose. Architecture of FPGAs and CPLDs: A Tutorial. IEEE Design and Test of Computers, 13(2):42–57, 1996.

[BSWG01] Mihai Budiu, Majd Sakr, Kip Walker, and Seth C. Goldstein. BitValue Inference: Detecting and Exploiting Narrow Bitwidth Computations. Lecture Notes in Computer Science, 1900:969–??, 2001.

[BTV96] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 171–183, New York, NY, USA, 1996. ACM Press.

[CCG+02] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An Open-Source Tool for Symbolic Model Checking. In CAV'02: Proceeding of the 14th International Conference on Computer-Aided Verification, 2002.

[CH00] K. Compton and S. Hauck. Reconfigurable Computing: A Survey of Systems and Software, 2000.

[Chu40] Alonzo Church. A Formulation of the Simple Theory of Types. Journal of Symbolic Logic, 5:56–68, 1940.

[CK94] Bob Cmelik and David Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. SIGMETRICS Perform. Eval. Rev., 22(1):128–137, May 1994.

[Con58] Melvin E. Conway. Proposal for an UNCOL. Communications of the ACM, 1(10):5–8, 1958.

[Cop94] Max Copperman. Debugging Optimized Code Without Being Misled. ACM Transactions on Programming Language Systems, 16(3):387–427, 1994.

[Cra03] Karl Crary. Toward a foundational typed assembly language. In Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 198–212. ACM Press, 2003.

[CSC+99] Lori Carter, Beth Simon, Brad Calder, Larry Carter, and Jeanne Ferrante. Predicated Static Single Assignment. In IEEE PACT, pages 245–255, 1999.

[DeH94] André DeHon. DPGA-coupled microprocessors: Commodity ICs for the early 21st century. In Duncan A. Buell and Kenneth L. Pocek, editors, FCCM'94: Proceedings of the 2nd IEEE Workshop on FPGAs for Custom Computing Machines, pages 31–39, Apr. 1994.

[DFH+91] G. Dowek, A. Felty, H. Herbelin, G.P. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The Coq Proof Assistant User's Guide Version 5.6. Rapport Technique 134. Technical report, INRIA, Dec 1991.

[DN97] Al Davis and Steven M. Nowick. An Introduction to Asynchronous Circuit Design. Technical Report UUCS-97-013, University of Utah, Sept 1997.

[EB02] Doug Edwards and Andrew Bardsley. Balsa: An Asynchronous Hardware Synthesis Language. The Computer Journal, 45(1):12–18, 2002.

[ECF+97] Carl Ebeling, Darren C. Cronquist, Paul Franklin, Jason Secosky, and Stefan G. Berg. Mapping Applications to the RaPiD Configurable Architecture. In Kenneth L. Pocek and Jeffrey Arnold, editors, IEEE Symposium on FPGAs for Custom Computing Machines, pages 106–115, Los Alamitos, CA, 1997. IEEE Computer Society Press.

[Eis97] Marc Eisenstadt. My Hairiest Bug War Stories. Commun. ACM, 40(4):30–37, 1997.

[Eng96] Dawson R. Engler. VCODE: a retargetable, extensible, very fast dynamic code generation system. In PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation, pages 160–170. ACM Press, 1996.

[FDG+93] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. A Micropipelined ARM. In T. Yanagawa and P. A. Ivey, editors, Proceedings of VLSI 93, pages 5.4.1–5.4.10, 1993.

[FGL01] Jan Frigo, Maya Gokhale, and Dominique Lavenier. Evaluation of the Streams-C C-to-FPGA Compiler: An Applications Perspective. In FPGA '01: Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays, pages 134–140, New York, NY, USA, 2001. ACM Press.

[Fox01a] Anthony C.J. Fox. A HOL specification of the ARM instruction set architecture. Technical Report 545, University of Cambridge Computer Laboratory, June 2001.

[Fox01b] Anthony C.J. Fox. An algebraic framework for modelling and verifying microprocessors using HOL. Technical Report 512, University of Cambridge Computer Laboratory, April 2001.

[Fox02] Anthony C.J. Fox. Formal verification of the ARM6 micro-architecture. Technical Report 548, University of Cambridge Computer Laboratory, November 2002.

[Fur00] Steve Furber. ARM System on Chip Architecture. Addison-Wesley, second edition, 2000.

[GIOS05] Mike Gordon, Juliano Iyonda, Scott Owens, and Konrad Slind. A Proof-Producing Hardware Compiler for a Subset of Higher Order Logic. May 2005.

[Gle04] Sabine Glesner. An ASM Semantics for SSA Intermediate Representation. In Proceedings of the 11th International Workshop on Abstract State Machines, volume 3052 of Lecture Notes in Computer Science. Springer, 2004.

[GNH01] Paul Graham, Brent Nelson, and Brad Hutchings. Instrumenting Bitstreams for Debugging FPGA Circuits. In FCCM'01: The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, pages 41–50, 2001.

[Gor85] Michael J.C. Gordon. HOL: A Machine Oriented Formulation of Higher Order Logic. Technical Report UCAM-CL-TR-68, Cambridge University, Jul 1985. Revised version (Jan 2001).

[Gor86] Mike Gordon. Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware. In George Milne and P. A. Subrahmanyam, editors, Formal Aspects of VLSI Design, pages 153–178. Elsevier Science, 1986.

[Gor88] Michael J. C. Gordon. HOL: A Proof Generating System for Higher-Order Logic, pages 73–128. Kluwer Academic Publishers, 1988.

[Gor95] Mike Gordon. The Semantic Challenge of Verilog HDL. In LICS'95: Proceedings of the 10th Annual IEEE Symposium on Logics in Computer Science, pages 136–145, Jun 1995.

[Gor00] Michael J.C. Gordon. From LCF to HOL: A Short History. In Gordon Plotkin, Colin P. Stirling, and Mads Tofte, editors, Proof, Language, and Interaction. MIT Press, 2000.

[Grä01] Erich Grädel. Why are modal logics so robustly decidable?, pages 393–408. World Scientific, 2001.

[GS98] Maya B. Gokhale and Janice M. Stone. NAPA C: compiling for a hybrid RISC/FPGA architecture. In FCCM'98: Proceedings of the 6th IEEE Symposium on FPGAs for Custom Computing Machines, pages 126–135, Apr. 1998.

[GSAK00] Maya B. Gokhale, Janice M. Stone, Jeff Arnold, and Mirek Kalinowski. Stream-Oriented FPGA Computing in the Streams-C High Level Language. In FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines, page 49, Washington, DC, USA, 2000. IEEE Computer Society.

[GT96] Winfried K. Grassman and Jean-Paul Tremblay. Logic and Discrete Mathematics: A Computer Science Perspective. Prentice Hall, 1996.

[Har01] Reiner Hartenstein. Coarse Grain Reconfigurable Architectures. In ASP-DAC'01: Proceedings of the 6th Asia and South Pacific Design Automation Conference, pages 564–570, 2001.

[Hau98] Scott Hauck. The Roles of FPGAs in Reprogrammable Systems. Proceedings of the IEEE, 86(4):615–638, Apr 1998.

[Hav93] Paul Havlak. Construction of Thinned Gated Single-Assignment Form. In 1993 Workshop on Languages and Compilers for Parallel Computing, pages 477–499, Portland, Ore., 1993. Springer Verlag.

[HD86] F. K. Hanna and N. Daeche. Specification and Verification using Higher-Order Logic: A Case Study. In George Milne and P. A. Subrahmanyam, editors, Formal Aspects of VLSI Design, pages 153–178. Elsevier Science, 1986.

[Hen01] Greg Henry. Flexible High-Performance Matrix Multiply via a Self-Modifying Runtime Code. Technical Report CS-TR-01-46, The University of Texas at Austin, Department of Computer Sciences, Dec 2001.

[HF95] David R. Hanson and Christopher W. Fraser. A Retargetable C Compiler: Design and Implementation. Addison-Wesley, 1 edition, 1995.

[HG88] C. A. R. Hoare and M. J. C. Gordon. Partial Correctness of C-MOS Switching Circuits: An Exercise in Applied Logic. In Yuri Gurevich, editor, LICS'98: Proceedings of the Third Annual IEEE Symposium on Logic in Computer Science, pages 28–36. IEEE Computer Society Press, July 1988.

[HH05] Adrian Hilton and Jon G. Hall. Developing critical systems with PLD components. In FMICS '05: Proceedings of the 10th international workshop on Formal Methods for Industrial Critical Systems, pages 72–79. ACM Press, 2005.

[Hor75] James J. Horning. Yes! high level languages should be used to write systems software. In ACM 75: Proceedings of the 1975 annual conference, pages 206–208, New York, NY, USA, 1975. ACM Press.

[How06] Denis Howe. Free Online Dictionary of Computing. `http://foldoc.doc.ic.ac.uk/`, 2006. Definition of accelerator.

[HP02] John L. Hennessy and David A. Patterson. Computer Architecture: A Quantitative Approach. Computer Architecture and Design. Morgan Kaufmann, 3 edition, 2002.

[HT90] N. A. Harman and J. V. Tucker. The Formal Specification of a Digital Correlator. In K. McEvoy and J. V. Tucker, editors, Theoretical Foudnations of VLSI Design, Cambridge Tracts in Theoretical Computing Science, pages 161–262. Cambridge University Press, 1990.

[Hut93] Graham Hutton. The Ruby Interpreter. Research Report 72, Chalmers University of Technology, May 1993.

[HW97] John R. Hauser and John Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In Kenneth L. Pocek and Jeffrey Arnold, editors, IEEE Symposium on FPGAs for Custom Computing Machines, pages 12–21, Los Alamitos, CA, 1997. IEEE Computer Society Press.

[IEE01] Institute of Electrical and Electronic Engineers, IEEE Press. IEEE Standard Verilog Hardware Description Language (1364-2001), 2001.

[IEE02] Institute of Electrical and Electronic Engineers, IEEE Press. IEEE Standard VHDL Language Reference Manual (1076-2002), 2002.

[IEE04a] Institute of Electrical and Electronic Engineers, IEEE Press. IEEE Standard for Verilog Register Transfer Level (RTL) Synthesis: (1364.1-2002), 2004.

[IEE04b] Institute of Electrical and Electronic Engineers, IEEE Press. IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis: (1076.6-2004), 2004. Revision of IEEE Std 1076.6-1999.

[Int05] Intel Corporation. IA-32 Intel ®Architecture Software Developer's Manual. Volume 1: Basic Architecture. P.O. Box 5937, Denver, CO 80217-9808, September 2005.
`ftp://download.intel.com/design/Pentium4/manuals/25366517.pdf`.

[Jon69] C.B. Jones. A Proof of the Correctness of Some Optimizing Techniques. Technical report, 1969.

[Jon76] C.B. Jones. Formal Definition in Compiler Development. Technical Report 25.145, IBM Laboratory, Vienna, February 1976.

[Jon90a] C.B. Jones. A Small Language Definition, chapter 9, pages 235–256. Prentice Hall International, 1990.

[Jon90b] C.B. Jones. Systematic Software Development in VDM. Prentice Hall, 1990.

[Jon90c] Geraint Jones. Designing Circuits by Calculation. Technical Report TR-10-90, Oxford University, Apr 1990.

[Jon03] C.B. Jones. Operational Semantics: Concepts and their Expression. Information Processing Letters, 88:27–32, 2003.

[JS90a] Geraint Jones and Mary Sheeran. Circuit design in Ruby. In J. Staunstrup, editor, Formal methods for VLSI design, chapter 1. Elsevier, 1990.

[JS90b] Geraint Jones and Mary Sheeran. Relations and Refinement in Circuit Design. Technical Report PRG-TR-13-90, Oxford University, 1990.

[JSW99] Adrian Johnstone, Elizabeth Scott, and Tim Womack. Reverse Compilation of Digital Signal Processor Assembler Sources to ANSI-C. In ICSM'99: Proceedings of the 15th IEEE International Conference on Software Maintenance, page 316, 1999.

[KB95a] Carlos Delgado Kloos and Peter T. Breuer, editors. Formal Semantics for VHDL. VLSI, Computer Architecture and Digital Signal Processing. Kluwer Academic Publishers, 1995.

[KB95b] Carlos Delgado Kloos and Peter T. Breuer. Introduction, chapter 0, pages 1–8. In VLSI, Computer Architecture and Digital Signal Processing [KB95a], 1995.

[KN04] Gerwin Klein and Tobias Nipkow. A Machine-Checked Model for a Java-Like Language, Virtual Machine and Compiler. Technical report, National ICT Australia, Sydney, Mar 2004.

[Lat05] Lattice Semiconductor Corporation, 5555 N.E. Moore Court, Hillsboro, Oregon 97124-6421. LatticeXP Family Handbook, Sept. 2005.

[LEM04] Kelvin T. Leung, Milos Ercegovac, and Richard R. Muntz. Exploiting Reconfigurable FPGA for Parallel Query Processing in Computation Intensive Data Mining Applications. In ICDE'04: 20th International Conference on Data Engineering, Mar. 2004.

[LM98] Wayne Luk and Steve McKeever. Pebble: A language for parameterized and reconfigurable hardware design. In Reiner W. Hartenstein and Andres Keevallik, editors, Lecture Notes in Computer Science 1482. Field-Programmable Logic: From FPGAs to Computing Paradigm. FPL'98: Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications, pages 9–18. Springer-Verlag, 1998.

[LP92] Zhaohui Luo and Robert Pollack. LEGO Proof Development System: User's Manual. Technical Report ECS-LFCS-92-211, Edinburgh University, 1992.

[LTS99] Ronald Laufer, R. Reed Taylor, and Herman Schmit. PCI-PipeRench and the SwordAPI: A System for Stream-based Reconfigurable Computing. In FCCM'99: Proceedings of the 7th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, pages 200–208, 1999.

[MB03] Jonas Maebe and Koen De Bosschere. Instrumenting Self-modifying Code. In AADEBUG'03: Proceedings of the Fifth International Workshop on Automated Debugging, Sep 2003.

[McC63a] John McCarthy. A Basis for a Mathematical Theory of Computation. In P. Braffort and D. Hirschberg, editors, Computer

Programming and Formal Systems, pages 33–70. North-Holland, Amsterdam, 1963.

[McC63b] John McCarthy. Towards a Mathematical Science of Computation. In Proceedings of IFIP Congress 1962: Munich, Germany, pages 21–28. North-Holland, 1963.

[McC66] J. McCarthy. A formal description of a subset of ALGOL. In T.B. Steel, editor, Formal Language Description Languages for Computer Programming, pages 1–12, Amsterdam, 1966. North-Holland. Proceedings of the IFIP Working Conference, 1964.

[Mel93] Tom F. Melham. Higher Order Logic and Hardware Verification. Cambridge Tracts in Theoretical Computing Science. Cambridge University Press, 1993.

[Mil78] Robin Milner. A Theory of Type Polymorphism in Programming. Journal of Computer and System Sciences, 17:348–375, 1978.

[Mil85] George J. Milne. CIRCAL and the Representation of Communication, Concurrency, and Time. ACM Transactions on Programming Language Systems, 7(2):270–298, 1985.

[Mor05] Kevin Morris. SRC Code. FPGA and Structured ASIC Journal, 2005. `http://www.fpgajournal.com/`.

[Muc97] Steven S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann, 1997.

[Nec97] George C. Necula. Proof-Carrying Code. In POPL 1997: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 106–119, Paris, France, jan 1997.

[Nip98] Tobias Nipkow. Winskel is (almost) Right: Towards a Mechanized Semantics Textbook. Formal Aspects of Computing, 10:171–186, 1998.

[Nip03] Tobias Nipkow. Jinja: Towards a Comprehensive Formal Semantics for a Java-like Language. In Proceedings of the Marktobberdorf Summer School 2003. IOS Press, 2003. To appear.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. Isabelle/HOL — A Proof Assistant for Higher-Order Logic, volume 2283 of LNCS. Springer, 2002.

[NPW05] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. Isabelle's Logics: HOL, 2005.

[OBM90] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation, pages 257–271. ACM Press, 1990.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, CADE'92: Proceedings of the 11th International Conference on Automated Deduction, volume 607 of Lecture Notes in Artificial Intelligence, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[Pau87] Lawrence C. Paulson. Logic and Computation: Interactive proof with Cambridge LCF. Cambridge University Press, 1987.

[Pau90] Lawrence C. Paulson. A Formulation of the Simple Theory of Types (for Isabelle). In P. Martin-Löf and G. Mints, editors, COLOG'88: International Conference on Computer Logic, LNCS 417, pages 246–274. Springer, 1990.

[Pau04] Lawrence C. Paulson. Introduction to Isabelle, Mar 2004. With Contributions by Tobias Nipkow and Markus Wenzel.

[Pau05] Lawrence C. Paulson. The Isabelle Reference Manual, 2005. With Contributions by Tobias Nipkow and Markus Wenzel.

[PBJ+91] K. Pingali, M. Beck, R. Johnson, M. Moudgill, and P. Stodghill. Dependence Flow Graphs: an Algebraic Approach to Program Dependencies. In A. Nicolau, D. Gelernter, T. Gross, and D. Padua, editors, Advances in Languages and Compilers for Parallel Processing, pages 445–467. MIT Press, Cambridge, MA, 1991.

[Pie02] Benjamin C. Pierce. Types and Programming Languages. MIT Press, 2002.

[PL05] Oliver Pell and Wayne Luk. Quartz: A Framework for Correct and Efficient Reconfigurable Design. In RECONFIG'05: Proceedings of the 2005 International Conference on Reconfigurable Computing and FPGAs, 2005.

[Plo81] Gordon D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[RC03] Andrew Royal and Peter Y. K. Cheung. Globally Asynchronous Locally Synchronous FPGA Architectures. In Peter Y. K. Cheung, George A. Constantinides, and José T. de Sousa, editors, FPL'03: Field Programmable Logic and Application, volume 2778 of Lecture Notes in Computer Science, pages 355–364. Springer, 2003.

[RS94] R. Razdan and M. D. Smith. A High-Performance Microarchitecture with Hardware-Programmable Functional Units. In Proceedings of the 27th Annual International Symposium on Microarchitecture, pages 172–180, November 1994.

[Rus94] David M. Russinoff. Specification and verification of gate–level VHDL models of synchronous and asynchronous circuits. In E. Börger, editor, Speicification and Validation Methods. Oxford University Press, Oxford, 1994.

[RW03] Nicole Rauch and Burkhart Wolff. Formalizing Java's Two's-Complement Integral Type in Isabelle/HOL. In Thomas Arts and Wan Fokkink, editors, FMICS'03: Proceedings of the 8th International Workshop on Formal Methods for Industrial Critical Systems, volume 80 of Electronic Notes in Theoretical Computer Science (ENTCS), pages 40–56, Røros, Norway, June 2003. Elsevier.

[RWZ88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In POPL'88: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles

of Programming Languages, pages 12–27, New York, NY, USA, 1988. ACM Press.

[SA01] Kedar N. Swadi and Andrew W. Appel. Typed Machine Language and its Semantics, 2001.

[Sch03] Robert Schiele. Building and Using a Cross Development Tool Chain. In Proceedings of the GCC Developers Summit, pages 213–222, May 2003.

[Sco70] Dana S. Scott. Outline of a Mathematical Theory of Computation. Technical Report PRG-2, Oxford University, Nov 1970.

[Ska05] Supplemental Isabelle/HOL Library, Oct 2005. Word Theory by Sebastian Skalberg. See `http://isabelle.in.tum.de/library/HOL/Library/document.pdf`.

[Smu61] Raymond M. Smullyan. Theory of Formal Systems. Princeton University Press, 1961.

[SS71] Dana S. Scott and Christopher Strachey. Toward a Mathematical Semantics for Computer Languages. Technical Report PRG-6, Oxford University, Aug 1971.

[ST99] Donald Sannella and Andrzei Tarlecki. Algebraic Methods for Specification and Formal Development of Programs. ACM Computing Surveys (CSUR), 31(3es):10, 1999.

[Sta02] Richard M. Stallman. GNU Compiler Collection Internals. Free Software Foundation, Inc., 2002. For GCC 3.2. First released in 1988.

[Ste95] Bjarne Steensgaard. Sparse functional stores for imperative programs. In IR'95: ACM SIGPLAN Workshop on Intermediate Representations, pages 62–70, 1995.

[Sto77] Joseph E. Stoy. Denotational Semantics: The Scott-Strachey Approach to Programming Languages. MIT Press, 1977.

[Str66] Christopher Strachey. Towards a Formal Semantics. In T.B. Steel, editor, Formal Language Description Languages For Computer Programming, pages 197–220. North-Holland, Amsterdam, 1966.

[Sut89] Ivan E. Sutherland. Micropipelines. Communications of the ACM, 32(6):720–738, 1989.

[SX98] Gerardo Schneider and Qiwen Xu. Towards a Formal Semantics of Verilog Using Duration Calculus. Lecture Notes in Computer Science, 1486:282–??, 1998.

[Var97] Moshe Y. Vardi. Why is modal logic so robustly decidable? In Descriptive Complexity and Finite Models: Proceedings of a DIMACS Workshop, number 31 in DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 149–184. American Mathematical Society, Jan 1997.

[Vee86] Arthur H. Veen. Dataflow machine architecture. ACM Computing Surveys (CSUR), 18(4):365–396, 1986.

[vT95] John P. van Tassel. An Operational Semantics for a Subset of VHDL, chapter 3, pages 71–106. In Kloos and Breuer [KB95a], 1995.

[WAL+93] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, and S. Ghosh. PRISM-II Compiler and Architecture. In Duncan A. Buell and Kenneth L. Pocek, editors, IEEE Workshop on FPGAs for Custom Computing Machines, pages 9–16, Los Alamitos, CA, 1993. IEEE Computer Society Press.

[WC96] R. Wittig and P. Chow. OneChip: An FPGA Processor with Reconfigurable Logic. In Kenneth L. Pocek and Jeffrey Arnold, editors, IEEE Symposium on FPGAs for Custom Computing Machines, pages 126–135, Los Alamitos, CA, 1996. IEEE Computer Society Press.

[WCES94] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. Value Dependence Graphs: Representation Without Taxation. In Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 297–310. ACM Press, 1994.

[WH95] M. J. Wirthlin and B. L. Hutchings. DISC: the Dynamic Instruction Set Computer. In John Schewel, editor, Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing, Proc. SPIE 2607, pages 92–103, Bellingham, WA, 1995. SPIE – The International Society for Optical Engineering.

[WHG94] Michael J. Wirthlin, Brad L. Hutchings, and Kent L. Gilson. The nano processor: A low resource reconfigurable processor. In Duncan A. Buell and Kenneth L. Pocek, editors, FCCM'94: Proceedings of the 2nd IEEE Workshop on FPGAs for Custom Computing Machines, pages 23–30, 1994.

[Win93] Glynn Winskel. The Formal Semantics of Programming Languages. MIT Press, 1993.

[WN04] Martin Wildmoser and Tobias Nipkow. Certifying Machine Code Safety: Shallow versus Deep Embedding. In Proc. 17th Int. Conf. on Theorem Proving in Higher Order Logics (TPHOLs 2004). Springer Verlag, 2004. 16 pages.

[WNKN04] Martin Wildmoser, Tobias Nipkow, Gerwin Klein, and Sebastian Nanz. Prototyping Proof Carrying Code. In Proc. 3rd IFIP Int. Conf. Theoretical Computer Science (TCS 2004), 2004.

[XH01] Hongwei Xi and Robert Harper. A Dependently Typed Assembly Language. In Proceedings of the sixth ACM SIGPLAN international conference on Functional programming, pages 169–180. ACM Press, 2001.

[Xil03] Xilinx, Inc. Using Embedded Multipliers in Spartan-3 FPGAs, May 2003. Xilinx Data-sheet 467 (v1.1).

[Xil05a] Xilinx, Inc. Spartan-3 FPGA Family: Functional Description, Aug. 2005. Xilinx Data-sheet 099-2.

[Xil05b] Xilinx, Inc. Spartan-3E FPGA Family: Functional Description, Nov. 2005. Xilinx Data-sheet 312.

[Xil05c] Xilinx, Inc. Using the ISE Design Tools for Spartan-3 Generation FPGAs, May 2005. Xilinx Application Note 473.