ANGLIA RUSKIN UNIVERSITY

A TYPE-SAFE APPARATUS EXECUTING HIGHER ORDER FUNCTIONS IN CONJUNCTION WITH HARDWARE ERROR TOLERANCE

JONATHAN RICHARD ROBERT KIMMITT

A thesis in partial fulfilment of the requirements of Anglia Ruskin University for the degree of Doctor of Philosophy

This research programme was carried out in collaboration with The University of Cambridge
Submitted: October 2015

Acknowledgements

This dissertation was self-funded and prepared in part fulfilment of the requirements of the degree of Doctor of Philosophy under the supervision of Dr David Greaves of The University of Cambridge, and Dr George Wilson and Professor Marcian Cirstea at Anglia Ruskin University, without whose help this dissertation would not have been possible. I am grateful to Dr John O'Donnell of The University of Glasgow and Dr Anil Madhavapeddy of The University of Cambridge for their willingness to examine the degree.

Dedication

Dedicated to my wife Christine Kimmitt

ANGLIA RUSKIN UNIVERSITY

ABSTRACT

FACULTY OF SCIENCE AND TECHNOLOGY

DOCTOR OF PHILOSOPHY

A TYPE-SAFE APPARATUS EXECUTING HIGHER ORDER FUNCTIONS IN CONJUNCTION WITH HARDWARE ERROR TOLERANCE

JONATHAN RICHARD ROBERT KIMMITT

October 2015

The increasing commoditization of computers in modern society has exceeded the pace of associated developments in reliability. Although theoretical computer science has advanced greatly in the last thirty years, many of the best techniques have yet to find their way into embedded computers, and their failure can have a great potential for disrupting society.

This dissertation presents some approaches to improve computer reliability using software and hardware techniques, and makes the following claims for novelty: innovative development of a toolchain and libraries to support extraction from dependent type checking in a theorem prover; conceptual design and deployment in reconfigurable hardware; an extension of static type-safety to hardware description language and FPGA level; elimination of legacy C code from the target and toolchain; a novel hardware error detection scheme is described and compared with conventional triple modular redundancy. The elimination of any user control of memory management promotes robustness against buffer overruns, and consequently prevents vulnerability to common Trojan techniques.

The methodology identifies type punning as a key weakness of commonly encountered embedded languages such as C, in particular the extreme difficulty of determining if an array access is in bounds, or if dynamic memory has been properly allocated and released. A method of eliminating dependence on type-unsafe libraries is presented, in conjunction with code that has optionally been proved correct according to user-defined criteria. An appropriately defined subset of OCaml is chosen with support for the Coq theorem prover in mind, and then evaluated with a custom backend that supports behavioural Verilog, as well as a fixed execution unit and associated control store.

Results are presented for this alternative platform for reliable embedded systems development that may be used in future industrial flows.

To provide assurance of correct operation, the proven software needs to be executed in an environment where errors are checked and corrected in conjunction with appropriate exception processing in the event of an uncorrectable error. Therefore, the present author's previously published error detection scheme based on dual-rail logic and self-checking checkers is further developed and compared with traditional N-modular redundancy.

keywords: FPGA, type-safety, fault-tolerance, self-checking

Contents

List of Figures

List of Tables

Glossary

λ-Calculus A formulation of universal computation based on function abstraction and application using variable binding and substitution, page: iv, v, vii, viii, 3, 4, 10, 13, 22, 23, 25, 27, 29, 31, 32, 76, 78–80, 126, 147

the halting problem Turing's halting problem [Boyer and Moore, 1984]. Turing proved no algorithm can exist which will always correctly decide, for a given arbitrary program and its input, whether the program halts when run with that input, page: 3, 22, 55

axiom A theorem that is asserted to be true with no proof, page: 8, 34

bare-metal operation An application that runs standalone on a CPU without any operating system or hardware abstraction layer, page: 4

boxed An integer or float in an array or tuple rather than directly referenced in a register. In OCaml, unboxed integers need to be distinguished from boxed objects by the least significant bit, which is one for an unboxed integer and zero for a boxed object, page: 36

camlp4 A syntax extension and pre-processor for OCaml, page: 14, 77

curry Method of reducing a function with multiple arguments to multiple functions with a single argument, page: 37

currying The process of reducing a function with multiple arguments to multiple functions with a single argument, page: 36, 37

functional computation in the form of expressions that derive a program's new state as a pure function of the old state, page: 3

garbage collection Scanning memory to find objects that are unreferenced, or only reference themselves, page: v, 13, 14, 21, 22, 31, 36, 39, 70–72, 75

imperative computation in the form of statements that change a program's state in-place, page: 3

quteRTL Verilog synthesis engine, which can perform coarse-grain elaboration, due to [Yeh, Wu and Huang, 2012], page: 86

raw FPGA fabric A flexible software execution environment consisting of a sea of gates with no dedicated processor, page: 4

SAT solver Method of solving complex boolean networks by reducing to a boolean satisfaction problem. This can be used to solve constraints, assertions, equivalence or synthesis problems., page: 16

scrubbing Cycling through memories refreshing the contents and simultaneously correcting correctable errors, page: 68

type punning The deliberate or accidental loss of type safety at interfaces to functions or foreign libraries, page: 7

uncurrying Method of reducing curried functions back to a single function with multiple arguments, page: 36, 45

untyped Lacking any checking or distinction between types, page: 3

Verilog Verification in logic, a popular register transfer level logic description [Bhasker et al., 2002] (RTL) hardware description language, page: 8

Xilinx A leading manufacturer of field-programmable gate-array devices, page: 8, 21, 31, 39, 46, 49, 54, 56, 57, 59–62, 69, 86, 88, 103

Acronyms

ABC (And-inverter graph (AIG), binary decision diagram [Filliatre, 2010] (BDD), and conjunctive normal form (CNF)) based synthesis [Brayton and Mishchenko, 2010], page: 15, 16, 58, 60, 61

ABI application binary interface, page: 36

AIG And-inverter graph, page: xii, 15, 60

API application programmer's interface, page: 37, 57, 147

ARM Advanced *Reduced instruction-set computer (RISC)* Machine architecture, page: 31, 35, 86

ASIC application-specific integrated circuit [Chinnery and Keutzer, 2002], page: 11, 19, 57, 60, 102

ASSP application-specific standard product, page: 102

AST abstract syntax tree, page: 76, 77

BDD binary decision diagram [Filliatre, 2010], page: xii, 15, 58

BLIF Berkeley Logic Interchange Format [Berkeley, 1992], page: 15

CAM Categorical Abstract Machine, page: 31, 34

CNF conjunctive normal form, page: xii, 15

CompCert Compiler Certified to be formally verified [Leroy, 2009a], page: 13, 14, 19, 25

Coq Calculus of constructions [Bertot, 2006], [Huet, Kahn and Paulin-Mohring, 2002], page: 4, 8, 12, 13, 16, 19–21, 24–26, 31, 32, 34–37, 40, 42, 44–47, 76, 102, 103, 118, 126

CRC Cyclic redundancy-check, page: 86

CTMR Cascaded triple-modular redundancy, page: 68

DWC Duplication with compare [Johnson et al., 2008], page: 68, 69

ECC error correcting code (a class of codes for parallel implementation in memories), page: 62, 69

EDIF electronic design interchange format [Stanford and Mancuso, 1989], page: 57, 59, 61, 62

Flocq *unified library for proving floating-point algorithms in Coq* [Boldo and Melquiond, 2011], page: 19, 25, 36

FPGA Field-Programmable Gate Array [Xilinx, 2009b], page: x, 2, 4, 7, 11, 15, 18–22, 24, 31, 34–36, 46, 48, 49, 51, 54–57, 62, 68, 71, 78, 86, 90, 101–103

GUI Graphical User Interface, page: 13

HLS High-level synthesis [Cong et al., 2011], [Cong et al., 2012], page: 15, 16, 35, 36, 103

HOL higher order logic [Slind and Norrish, 2008], page: 12–14

LRTT Left to Right Tree Traversal, page: 77

MMU memory management unit, page: 73, 74

MPU memory protection unit, page: 75

OCaml Object-oriented categorical abstract meta-language [Remy and Vouillon, 1998], page: 8, 14, 16, 19–21, 25, 26, 31–37, 40, 42, 44–47, 57, 68, 71–73, 76, 77, 96–98, 101, 147, 155

PEG **parsing expression grammar**, page: 76, 77

RAM Random Access memory, page: 6, 35, 56, 57, 60–62, 68, 69, 72

REPL Read-Evaluate-Print-Loop [Findler et al., 1997], page: 14, 103

RISC Reduced instruction-set computer, page: xii, 31

RTL register transfer level logic description [Bhasker et al., 2002], page: xi, 15, 16, 54, 59, 61, 62

SECD Stack, Environment, Code and Dump [Landin, 1964], page: 14, 31, 34

SEU single-event upset, page: 69

TMR Triple-modular redundancy [Miller and Carmichael, 2008], page: 60, 62, 68, 69, 86, 90, 96

XST Xilinx synthesis technology, page: 15, 57, 60

A TYPE-SAFE APPARATUS EXECUTING HIGHER ORDER FUNCTIONS IN CONJUNCTION WITH HARDWARE ERROR TOLERANCE

JONATHAN RICHARD ROBERT KIMMITT

COPYRIGHT

"All the calculations that would ever be needed in this country could be done on the three digital computers that were then being built – one in Cambridge, one in Teddington, and one in Manchester."

Professor Douglas Hartree

1 Introduction

Past research has led to several technologies that can improve confidence in the correctness of a system. These include formal methods for proving properties of programs, theorem checkers and provers, static type inference, garbage collection, fault tolerance, and fine-grained exception handling. Each of these technologies supports only a part of the full spectrum from a high level user application to implementation on real hardware. The eventual aim of the current research is to demonstrate a complete path from high level language to hardware, with correctness proved, where possible, at each stage, as well as to incorporate techniques to improve reliability in the presence of possible hardware faults. The emphasis is on identifying and preventing rejection of information flowing from one tool to another, in a loose analogy with the way a quarter-wave transformer prevents reflection when electromagnetic devices of different impedances are interfaced together.

1.1 Background

Originally, a 'computer' was a blue-collar worker with paper and a pencil, later to be augmented with mechanical assistance. The idea of a universal computing machine did not arise until the work of Alan Turing [Turing, 1936]. The first electro-mechanical computers were highly unreliable (then referred to by the general public as 'mechanical brains'). At this time it was still mandatory for the apparatus to be surrounded by a bevy of assistants and engineers. The computing pioneer John Von Neumann [Von Neumann, 1956] developed a systematic error correction approach based on majority gates. Von Neumann concluded that a sufficiently large network of carefully designed components could cope with a failure rate of around 1 in 100. His approach assumes random permutation of the wiring between stages, which is difficult to achieve systematically.

Von Neumann's work, rather than being characterised as a 'mechanical brain', was characterised by analogy with the human brain; there existed no adequate electronic technology to express his ideas at that time. His aim may be seen as replicating, in electronic circuitry, the desirable fault-tolerance properties of the human brain. Just as a healthy (euthymic) brain can cope with a certain proportion of faults without malfunctioning, so, ideally, its electronic equivalent should benefit from the same reliability.

Inspired by his work, this thesis uses the analogy of the human brain to expound an approach to fault tolerance. Just as in the euthymic brain, where only occasional faults occur due to cell death (which will be masked by redundancy), so the electronic circuitry needs to be designed to operate in a similarly redundant manner with respect to component failure. For a 'depressed brain' (perhaps characterised by lethargy, and difficulty in making decisions) which might be attributed to insufficient neurons firing, the analogy in electronic circuitry could be insufficient information feeding through. Likewise, for the 'manic brain' (perhaps characterised by too many inappropriate connections being made, without insight into the consequences: for example Van Gogh deciding it was a good idea to cut off his own ear [Runyan, 1981]), the faulty electronic circuit might be viewed as a network which delivers the incorrect result, without any error being detected.

Continuing the analogy with the 'euthymic brain', a higher-level exception mechanism, known as the 'inner critic' or the 'generalised other', is constantly weighing up the outputs of the decision-making procedure. In between the extremes of the 'depressed brain' and the 'manic brain', there exists a state of hyper-criticality, where works of genius are produced. Not surprisingly, these states of hyper-criticality are commonly associated with what might be perceived as flawed individuals. With the recent discovery of a realisable memristor [Williams, 2008a], it may be possible in the future to make a much closer analogue to the human brain in electronic form [Jo et al., 2010], which may show greater similarity in its fault model. At the time of writing, and for the next decade, the only economically viable model will be to make use of commodity hardware in new and imaginative ways. As electronic computers have advanced exponentially in performance and capacity, the general public expectation has been that human error would be eliminated altogether, however these expectations have not been met and are not likely to be met in the near future.

In the late twentieth century, vast improvements in quality and quantity of computer components were associated with a step change in computer reliability. However, recently the rate of shrinking semiconductor processing has augmented the influence of quantum mechanics as a significant effect in transistor operation (and not just in traditional areas like flash memory). The need for a new approach to fault-tolerance arises from commercial pressures, which force microchip manufacturers to migrate to ever-smaller geometries. Consequently, the products have reduced tolerance for noise, and increased opportunities for single event upsets [Asadi and Tahoori, 2005]. Full logic reliability can no longer be assured due to statistical variability and lower noise tolerances. Furthermore, as mass-production products consume more and more fabrication capacity, and new designs are forced onto smaller geometries, the up-front costs (such as mask making) can reach millions of pounds, which is prohibitive unless amortised against the largest production runs. Hence, the use of FPGA with very low geometries is inevitable, even in safety-critical projects such as medical and aerospace. The

concept of *approximate computing* [Han and Orshansky, 2013] is interesting from the point of view of making data-path algorithms that do not always deliver the correct result (or can fail gracefully). This is particularly relevant in areas where the consumer does not use all the information present in data playback, such as video and audio streams. Approximate processing would not work well for document typesetting, for example, which has very low redundancy.

An additional and very pressing issue is how to ensure quality in software, which is frequently orders of magnitude more complex than the hardware, and which usually executes relatively slowly, but with many latent transitions that are not visible at the interfaces. This presents a challenge for testing and proving that all states can be reached and have a suitable exit condition.

Viewed from the lowest level of abstraction, with no prior knowledge of intent, a computer program may be defined as an arbitrary sequence of bits. Consequently, an issue immediately arises whether that program is useful or not due to **the halting problem** [Boyer and Moore, 1984]. However, if the program is defined as an expression of λ-Calculus [Church, 1936], the meaning can be abstracted away from the implementation and it is possible to make assertions about the behaviour of particular programs (but not programs in general). The original λ-Calculus was defined only for numeric variables and was consequently untyped. For this reason it suffers from incompleteness in the sense of Gödel [Gödel, 1931], no distinction being made between free variables, which were natural numbers, and representations of other objects such as Gödelized theorems. For any Turing machine algorithm, there is no one correct implementation, but many incorrect implementations. The introduction of type inference gives a well defined, higher-order meaning to every expression, and makes it much more difficult to write a program which is not a meaningful expression of λ-Calculus, or is otherwise ill-formed. The use of types makes the program simultaneously less powerful but much safer, and the question of incompleteness does not arise, because the typing prevents improper questions from being asked. *The set of all sets that do not contain themselves* is not a question that can be asked in a theorem prover without causing a "Universe inconsistency" [Jacobs, 2013].

For historical and commercial reasons the imperative paradigm has been dominant, partly because of the ease of expressing relatively simple algorithms compactly and with fewer resources than using functional programs. However, just because the (functional) λ-Calculus is equivalent in expressivity to the (imperative) Turing machine [Dershowitz and Falkovich, 2012], it does not mean that it is easy or convenient to convert from one to the other. Unfortunately, at some stage, the importance of array bounds checking was abandoned, and a gradual diminution of the whole field of reliability resulted, leading to the current situation where critical bugs such as heart-bleed [Durumeric and Kasten, 2014], and shell-shock [CVE-2014-6271, 2014], involving tens of millions of computers are discovered on a regular basis. It is instructive to enquire whether the proposed apparatus, of this current work, would be vulnerable to these attacks. The first attack is a straightforward buffer overrun attack, and so is a direct result of choosing an unsafe imperative language without bounds checking. The second attack is subtler, but involves converting a function to a part of the program's environment and then back to a function. This coercion would not be allowed in a type-safe language. However, in this latter case, the implementation could be specified as a shallow embedding of an interpreter

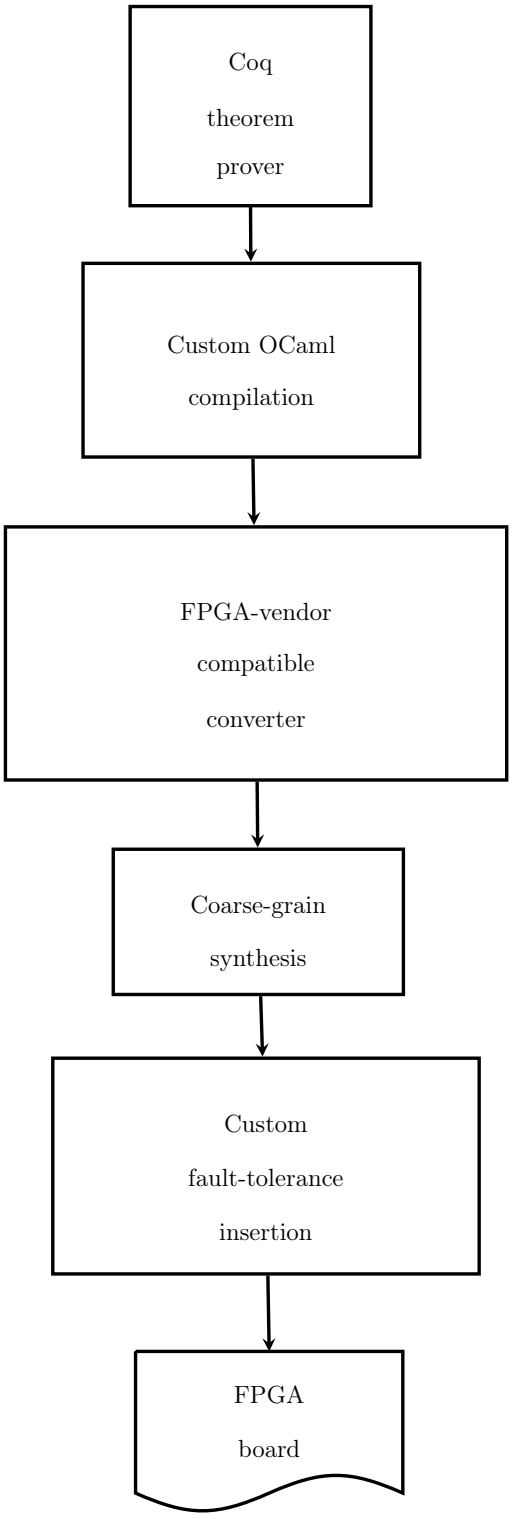
function within a native program. Consequently, the alternatives could be implemented as one polymorphic type, conversion of which could be allowed. However, this type of program is easier to analyse, and correct vulnerabilities, than an algorithm written in C or some other unsafe language. This type-safety strategy ensures that the user can only make use of data that is in scope of the λ-Calculus expressions that happen to be executing at the time. No lower-level access to the register machine will be possible.

1.2 Statement of the Hypothesis to be demonstrated

The majority of functional languages available now rely on type-unsafe foundations, principally to allow them to be bootstrapped in legacy operating system environments. Hence, though a majority of a program may be safe, there may be vulnerabilities that exist due to interactions with the operating system (for example a library routine returning a NULL pointer without generating an exception or being interpreted as an option value (None) or empty list([])). This situation has persisted for about three decades but recently various type-safe operating systems have started to appear. An early attempt was Snowflake [Hamilton, 2010], more recently the Mirage team have produced unikernels [Madhavapeddy et al., 2013] which are lightweight operating systems running under Xen [Barham et al., 2003] (and consequently Cloud compliant). High data integrity is assured with transport layer security [Mehnert and Meršinjak, 2014]. Since the Xen kernel is much smaller and simpler than Linux, Windows, OSX etc., it should be expected that vulnerabilities would be fewer.

In the present work, a novel approach is used to extend type-safety below the abstract language level, such that it applies not just to bare-metal operation but also raw FPGA fabric. This gives a great deal of freedom to the implementer. Given so much freedom, numerous methodologies could be chosen, without delivering the expected benefit. Consequently, *a type-safe apparatus executing higher order functions in conjunction with hardware error tolerance is a good approach* and is presented here along with a suitable methodology to ensure that an imperative layer of software is not necessary at the low level. The focus of this approach is on embedded environments, which typically have a greater risk profile compared to desktops, because of their varied applications, ranging from smart cards where there is a fraud risk at one end of the scale, to controlling heavy machinery where there is a danger to life at the other. However, the same ideas could be elevated to the cloud environment, with a suitable boost to performance and memory capacity.

The simplified overall flow may be seen by referring to Figure 1.1. It may be understood by reference to the following descriptions of the functions of the different boxes above:
The Calculus of constructions [Bertot, 2006], [Huet, Kahn and Paulin-Mohring, 2002] (Coq) theorem prover is a software tool that checks algorithms according to rigorous mathematical criteria. The technique is analogous to assertion writing in software, but the aim is to ensure that the assertion can never fail. The user decides what the assertions/proofs shall be according to an underlying mathematical formulation, invisible in the output code. The checked algorithm in executable semantics may be exported in various flavours of meta-language(ML). The custom OCaml backend allows ML to execute in a primitive hardware model by applying various optimisations such as closure conversion, register argument passing, and escape analysis.



See Chapter 4: Dependent types in Coq

See Chapter 5: An OCaml compiler for bare-FPGA

See Chapter 6: Conversion and compilation to FPGA

See Chapters 7 and 9

See Chapter 7: Comparison of TMR and Dual-rail logic

See Chapter 9: Discussion and analysis

Figure 1.1: Flow overview

It also provides for separate compilation using a type-safe intermediate object format.
The backend generates behavioural Verilog, which is automatically converted to an execution unit based state-machine. Based on the execution unit from the Amber project, the hardware engine provides a customisable subset of the ARM programmer's model, with the ability to add custom functional units. A simple FPGA state-machine can operate directly from the output of the modified OCaml code generator. Code is stored in static Random Access memory (RAM) and is automatically produced by the compiler. The compiled output is ready to feed FPGA tools directly and supports incremental changes at the bitstream level. Exception handling is generalised to allow hardware faults to be reported in software.
The standalone compiler output feeds into a type-safe coarse-grain synthesis program. This ensures that the underlying hardware operations are explicit in the Verilog netlist. The resulting netlist is compatible with various post-processors that implement a variety of fault-tolerance options.
Various hardware error detection schemes, such as triple-modular redundancy, as well as byte and word-level duplication, have been tested. Making a virtue of necessity, the FPGA allows the designer to break out of the straight-jacket of Von-Neumann embedded designs. Word-length choices are under control of the hardware designer and associated compiler writer. The methodology allows type-safety to be maintained down to FPGA primitive level because the main tools themselves are type safe.

1.3 Research questions, Scope and Objectives

Therefore, the ultimate goal of this original research is nothing less than the complete assertion of correctness, all the way from initial software development in a theorem prover, to fault-tolerant software operating on an hardware platform. Bearing in mind the remarks of Allan [Allan, 2010], it is unlikely all the required elements would all come together in one thesis. So, a reasonable subset of investigative research was framed as shown below. At the outset the following questions were identified as being key to the various problems of computer reliability:

(i) Is it meaningful to speak of functional programming in hardware as well as software?

(ii) Does the addition of hardware tagging to the virtual machine environment of a functional programming system boost the reliability of the result for safety critical systems [Benediktsson, Hunter and McGettrick, 2001]; [Hughes, 1989]; [Zeldovich et al., 2008]?

(iii) Does it help to run the entire operating system as well as the application in a functional programming language ([Yang and Hawblitzel, 2010]; [Maeda and Yonezawa, 2009])?

(iv) Can a direct hardware execution engine be made with an economic cost and respectable performance?

Within the overall scope of the thesis, these questions shall be investigated by means of a customised software methodology, with the aim of demonstrating:

a) that functional programming is very suitable to run directly on hardware;

b) that a separate operating system interface layer is unnecessary, due to the inherent safety of the approach;

c) that reliance on type-unsafe libraries is not required;

d) that fault checking at the gate level is the appropriate way to check for errors;

e) that, in principle, the technique can be scaled to an entire operating system; and finally

f) that an embedded implementation with no custom chips can run at 33MIPS.

By making this selection, the aim is to minimise the possibility of faults occurring due to hardware errors, software oversights, or deliberate malpractice by third parties.

1.4 Original contributions to knowledge

This section summarises the claimed original contributions to knowledge, which will be developed in the later chapters:

(i) An original methodology consisting of a toolchain and libraries to support extraction from dependent types in a theorem prover to reconfigurable hardware is considered to be novel. Techniques of proving programs using dependent types and theorems are already well established [Weirich, 2014]. However, in the present embodiment, a direct path is offered to hardware execution that does not depend on legacy techniques, that have previously been evaluated and subsequently deprecated.

(ii) An innovative extension of static type-safety to hardware description language and FPGA level is proposed. Static type safety has been well understood for about 20 years, but so far has not been incorporated as an essential aspect of mainstream embedded methodologies. Consequently, the one environment that has most to gain from improved safety (mathematically and practically), has yet to benefit from the theoretical advances of Computer Science.

(iii) The elimination of type punning, both from the target and toolchain, is claimed as a novel contribution. The necessity to forget about type information by accident or design is a deliberate feature of most functional languages running on type-unsafe workstations.

(iv) A novel hardware error detection scheme is proposed, and will be compared with conventional triple modular redundancy. In critical systems such as aerospace, it is usual to triplicate logic and vote on the correct answer. However, using this method it is not possible to cope with a situation where the majority is wrong. A novel method of detecting various kinds of errors, including single-event upsets, is introduced to complement the provable type safety and (optional) assertions that are provided by the rest of the toolchain.

1.5 Method Overview

Referring to Figure 1.2, it can be seen that the preferred form of input is an executable formal specification in the form of a Coq description. This is desirable for three reasons: firstly the Coq compilation guarantees termination, secondly it allows making use of existing proofs of correctness, and thirdly, it ensures that only a functional subset of Object-oriented categorical abstract meta-language [Remy and Vouillon, 1998] (OCaml) code will be output. The theorem prover will also output a series of axioms and assumptions (theorems that are required but not proven), which will need to be replaced with suitable OCaml equivalents if used. The pure functional code output does not assume any particular standard library implementation compatibility. One OCaml module per theorem file or library is the rule and so a number of separate compilation steps are required. These take place internally to the OCaml compiler; there is no need for a separate assembler and linker. Should it be necessary to inspect the internal format, it is available as an ASCII file (an example output is shown in Table 5.2 on page 38). Otherwise, marshalled data structures that are compiler version specific may be used as input to the linker. After linking, any necessary template code is appended, and the whole assembly with all symbols resolved is converted to a dual representation, in the form of behavioural Verilog, and control store contents for a generic execution unit. Subsequently, this description is suitable for behavioural simulation, cycle-by-cycle comparison with the control store representation, or conversion to the appropriate technology specific library. In this case, only Xilinx is targeted, but there is nothing preventing an alternative technology, that has sufficient on-chip memory, from being used instead. Once the first bitstream has been prepared, the control store contents can be repeatedly replaced as desired, in the same way that software is updated on a conventional system. The fixed Verilog libraries will typically have descriptions of peripherals such as a VGA display adaptor, or indeed any kind of hardware that suits the application.

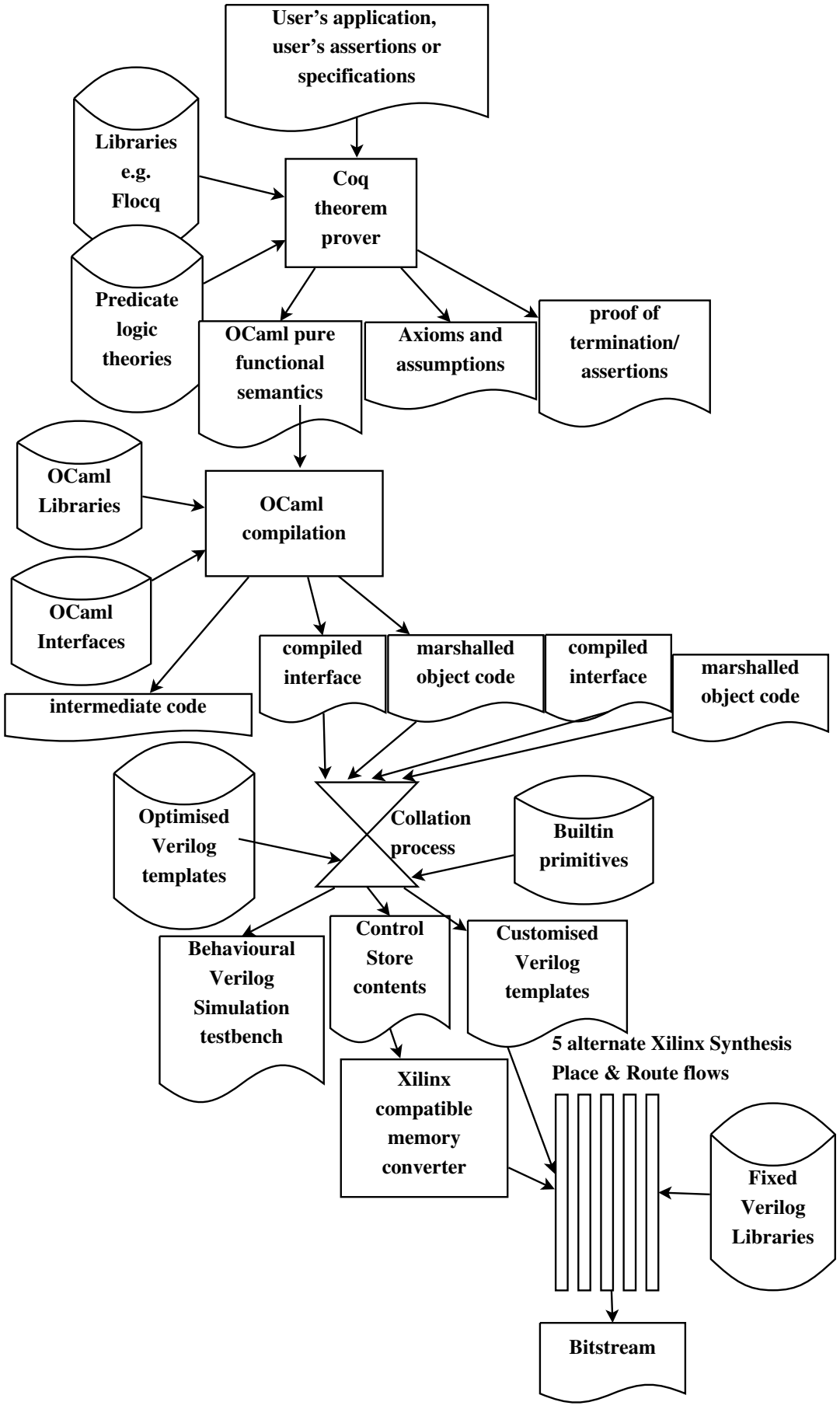


Figure 1.2: Software development flow

1.6 Thesis Structure

To describe the work, selected pertinent literature is reviewed in chapter 2, whilst the methodology to reproduce the apparatus is set out in chapter 3. The theorem proving and dependent type checking is explored in chapter 4, and the language and compilation issues are considered in chapter 5. The conversion to hardware is described in chapter 6. Chapter 7 investigates the reliability at the hardware level. The method of dynamic storage management is discussed in chapter 8, and the results of applying the methodology to a shallow embedding of a dependently-typed λ-Calculus, are given in chapter 9. Finally, the conclusions to support the thesis statement are summarised in chapter 10.

"Of making many books there is no end, and much study wearies the body."

Ecclesiastes 12:12

2

Literature review

As stated in the introduction and section 1.2 *Statement of the Hypothesis to be demonstrated*, the overall goal of the thesis is to identify a methodology for generating a robust electronic demonstrator using FPGA or application-specific integrated circuit [Chinnery and Keutzer, 2002] (ASIC) of a mathematically proven algorithm, and supporting tools for use in safety-critical environments. The starting point to reach this goal is a review of selected existing literature on the subject.

From the earliest days of computing, the issue of reliability has been of fundamental importance. In Charles Babbage's difference engine no. 2, special measures were used to prevent the computer from outputting a value that was metastable [Swade, 2005]. As a pioneer, he considered jamming the apparatus to be much more satisfactory than outputting an incorrect result.

Arguably, the foundations of modern predicate logic were formalised in the seminal work *Principia Mathematica*, first published in 1910, and considerably revised in 1925-27 [Whitehead and Russell, 1925], which was an attempt to address prevailing problems in mathematics, such as Russell's paradox [Grattan-Guinness, 1978]. To address the issue, a strict hierarchy of types was introduced, in the formalism that prevents the paradox. However, Gödel [Gödel, 1931] introduced a notation to represent theorems as numbers, which bypasses the type hierarchy, and demonstrates the existence of an unprovable theorem, despite the care taken in Principia to avoid self-reference.

After Gödel, it is known that any consistent constructive logical system, no matter how comprehensive, will have theorems that cannot be proven or disproven in any such scheme. According to da Costa [da Costa, 2012], a key assumption in the proof is the axiom of the excluded middle [Bourne, 2004]. If double-negation is allowed, it can be proved that there are propositions such that, while they cannot be proved false, they also cannot be proved

true. The fact that a proposition is undecidable does not invalidate the logic in itself, but it does show that care is needed to avoid asking inappropriate questions. A valid way to avoid introducing unprovable statements into the logic is to prevent the user from asking the question in the first place. This can be done by subsuming the unprovable theorems into a higher-order type system. This higher-order type system will still be incomplete, but capable of proving the theorems of the lower-order system. Therefore, by strict stratification of types, asking a question that cannot be answered (by the axioms and rules of inference of that system) is avoided. In other words, to reason about the equivalence of theorems, and the Gödel numbers representing those theorems, is not a proper question; it is a meta-question that must be answered by a higher-order reasoning.

2.1 Software correctness review

From the inception of high-level languages, there has been a desire to improve the quality, maintainability, and re-usability of software. Some of these improvements were in the hardware architecture, such as the use of index registers to make code more modular, and the use of virtual machines [Uhlig et al., 2005] to prevent the presumptions of older code from failing, when the range of meaningful addresses increased, or implicit assumptions about cache coherency were violated. Other improvements were concerned with isolation of the front-end language from considerations about how the underlying hardware worked. This became possible because computational cycles constantly became cheaper, as programmer hours became more expensive. It was relatively recently that hardware became so powerful, that it was conceivable that a computer could be produced, that ran code that was mathematically proven to be correct: the so-called Hoare challenge [Hoare, 2003]. It was estimated that this task would take around fifteen years [Allan, 2010].

For dynamically typed languages, the VLISP verified Scheme System [Guttman, Ramsdel and Swarup, 1995] was an early, non-automated piece of work to show rigorous proof of correctness. However, the advantages, and disadvantages, of dynamic typing are hotly debated [Meijer and Drayton, 2004]. In the present author's opinion, static typing is considered the superior methodology, for system critical systems such as are considered here, largely because the parameters and peripherals of the system will typically be known in advance. Of course, any algorithm that works in static typing, will also work in dynamic typing. The benefit, as mentioned earlier, was the reduction in the number of incorrect programs that would be allowed to compile.

Many popular mechanised proof systems exist based on higher-order types. Two of the most common are higher order logic [Slind and Norrish, 2008] (HOL) and Coq. As the name implies, the latter is based on constructive logic, and consequently does not assume the axiom of the excluded middle. However, this assumption can be added, if needed, as an additional axiom of the logic.

Both of these tools operate similarly, based on the Logic for computable functions (LCF) technique [Gordon, 2000]. The essentials of the system are, that all objects of type theorem have to be inferred directly from axioms or from theorems, which themselves have been derived

directly or indirectly from the axioms. Care has to be taken when introducing axioms, to prevent inconsistency. Because this type of proof is very laborious, it is customary to introduce a Graphical User Interface (GUI) such as Proof-general [Aspinall, 2000] and CoqIDE [Narboux, 2010] to allow the candidate theorems to be proven to be broken down, into simpler goals (similar to lemmas), that can be individually proved in order to infer the top-level proof. To economise on memory, the proof trace is not generally kept in memory after the proof is generated, because it can easily be re-generated from the prerequisites and the proof tactics. For this reason it is difficult to transfer proofs from one assistant to another, because the names of tactics, intermediate terms, and libraries vary considerably between tools and even different versions of the same tool.

Proving a hardware description can be carried out in either language (HOL or Coq); indeed HOL was originally constructed with hardware verification in mind [Iyoda, 2007]. By contrast, Coq was originally developed as a program prover, with the capabilities of hardware verification added on later. In the absence of the axiom of the excluded middle, the underlying logic is incompatible with Boolean/Aristotelian notions of truth/falsehood.

Partial proofs of correctness have previously been studied. For example, *Typed Closure Conversion* [Minamide, Morrisett and Harper, 1996] does consider polymorphic λ-Calculus, but it does not discuss the other phases of compilation, needed for total proof of correctness.

The Compiler Certified to be formally verified [Leroy, 2009a] (CompCert), provides a framework, based around the Coq theorem prover, which assures compiler semantic preservation from high-level language (C-light, a large subset of C), to assembly language. Although widely used, because of its efficiency in execution, C cannot be regarded as a rigorous platform for computer reliability. However, a verified compiler ensures that any bug in the source code will be faithfully reproduced in the executable. In addition, the CompCert developers recommend applying formal verification techniques (static analysis, program proof, model checking). At the time of writing, the compiler falls somewhat short of comprehensiveness [Leroy, 2014]; nevertheless, it is comprehensive relative to research languages put forward in other research papers ([Brady and Hammond, 2006], [Benton and Tabareau, 2009], [Benton and Hur, 2009], [Chlipala, 2010], [Dargaye and Leroy, 2010], [Benton and Hur, 2010], [Jaber, 2010]).

In Certified Type-Preserving Compiler from λ-Calculus to Assembly Language [Chlipala, 2007a], the author makes use of Coq to prove the compilation of simply-typed λ-Calculus, to a hypothetical assembly language. It specifies the garbage collection semantics, but not its implementation, as part of the axioms of the assembly code. In this case, the source language is raw λ-Calculus, and not particularly easy to follow, especially for more elaborate syntax examples such as curried functions. The proofs use an open-source library called *ltamer* [Chlipala, 2007b], which uses dependent types and denotational semantics, compared with CompCert's non-dependently-typed abstract syntax and operational semantics. The work was recently updated to use a subset of ML as the source language, in order to demonstrate the ease of adding new features, without upsetting the underlying proof.

Using CompCert as a foundation, the Verified software toolchain [Appel, 2011], goes one step further, and converts the C source code to a Coq proof, which can then be shown to satisfy the relevant characteristics of safety, termination, liveness, as needed. This is appropriate when

software is written from scratch, rather than converting legacy code. Significant attempts at more complicated parts of the Hoare challenge have been published, for example in the area of file system proof [Joshi and Holzmann, 2007].

Members of the same research team that produced CompCert attempted to extend the principle of compiler verification by semantic preservation to functional languages. In "Vérification formelle d'un compilateur optimisant pour langages fonctionnels" (Formal verification of an optimising compiler for functional languages) [Dargaye, 2009], a member of the CompCert team, extended the idea to enable compilation of a language called *mlcompcert*, a small subset of regular ML. Although not practical for real-world use, this language requires most of the modern facilities such as garbage collection to be tackled, albeit in a simplified form. For a more general, but admittedly incomplete treatment of garbage collection, see [McCreight, Chevalier and Tolmach, 2010; McCreight et al., 2007].

McCreight tackles the problem of creating a certified garbage collection runtime environment. The eventual aim is to allow lazy functional languages, such as Haskell, to be compiled, whilst the framework remains sufficiently general, to allow ML or Java to be used as well. The ongoing effort is subsumed within the High-Assurance Systems Programming (HASP) project, which eventually aims to produce a new high-security functional language, called Habit [HASP, 2010].

Another approach is to use Krivine's realizability [Krivine, 2009] to implement a Stack, Environment, Code and Dump [Landin, 1964] (SECD) machine [Jaber, 2010]. By this method, the semantics of transformations from abstract syntax tree to SECD interpreter can be certified. Input programs could be generated laboriously by proof checker, or else generated from a (non-verified) camlp4 front-end.

In "A Kripke logical relation between ML and assembly" [Hur and Dreyer, 2011], an equivalence is established between high-level code, and idealised assembly language that supports reference types.

One of the most ambitious research projects (still incomplete at time of writing) is the CakeML compiler [Kumar et al., 2014] and verification system, written in the theorem proving language HOL. The source language to be compiled is written in a subset of ML similar to Standard-ML/OCaml and supports proof of correctness from initial PEG parsing to x86 binary code, using a Read-Evaluate-Print-Loop [Findler et al., 1997] (REPL). Ideally, the algorithm to be compiled should first be written in HOL, and then converted to CakeML, which is then compiled to bytecode, or native x86 code. Because of the difficulty of formally proving compiler optimizations, many of the usual compiler passes are skipped. The intended application is bootstrapping a formally proved compiler, as well as providing a solid foundation to compile proof engines such as HOL-light [Myreen, Owens and Kumar, 2013].

As prior art, the closest implementation to the subject of this dissertation is the *Reduceron* [Naylor and Runciman, 2012]. Apart from the obvious difference that it is a lazy functional architecture, whereas OCaml is a strict evaluation architecture, the compilation flow is very different, and the optimisation for massive parallelism requires a complete rebuild of the hardware for every program change. There is no convenient build/debug/edit/build cycle. The initial synthesis alone takes more than 24 hours on a 32Gbyte workstation. By contrast,

with the present embodiment, a total iteration is less than half an hour, and a simple change of software is less than 5 minutes. Naturally, nothing comes for free, and the approach of this thesis requires manual intervention to adjust datapath or co-processor architecture. However, given the typical complexity of modern software engineering, even the best programmers require a few iterations to get things right (often due to ambiguity in the specification of what is to be achieved, as much as software bugs).

2.2 Logic synthesis review

This dissertation incorporates hardware synthesis technology, from RTL to fault-tolerant gate level and/or elaborated operator level. A number of other solutions offer similar functionality for FPGA technology. The market-leading commercial product is Synopsys Synplify_Premier_Pro [Sutherland and Mills, 2014], which has sophisticated facilities such as timing-driven operator sharing, targeting numerous FPGA technologies, as well as post-layout-driven critical path retiming. For proprietary commercial reasons, it does not provide access to the internal database, to allow technology independent post-processing such as fault-tolerance insertion. Another solution, targeting Xilinx only (now deprecated, in favour of the High-level synthesis [Cong et al., 2011], [Cong et al., 2012] (HLS) synthesis tool Vivado [Winterstein, Bayliss and Constantinides, 2013]), is Xilinx synthesis technology (XST). It can be made to output a gate-level netlist, but all higher-level structuring, such as fast-carry chains, are removed in the rendition.

From a research point of view, open source alternatives are more attractive. These tools mostly output structural Verilog, or Berkeley Logic Interchange Format [Berkeley, 1992] (BLIF), for later use with (AIG, BDD, and CNF) based synthesis [Brayton and Mishchenko, 2010] (ABC), the main open solution for gate-level optimisation.

(i) Icarus Verilog [Williams, 2008b] was an early attempt at simulation and synthesis. The synthesis function never reached maturity, and was dropped in 2008.

(ii) ODIN_II [Jamieson, Kent and Gharibian, 2010] was another early attempt, was made part of the larger VTR suite [Rose et al., 2012] that packages ODIN_II along with ABC and a non-technology specific place-and-route algorithm. It does not handle the Verilog architecture of Appendix-C.

(iii) HANA [Ahmad, 2011] uses a different approach, based on parameterised coarse-grain cells written in Verilog. Wordwidths of elaborated library models are limited to powers of two, which can be wasteful, especially for odd-sized busses.

(iv) QuteRTL [Yeh, Wu and Huang, 2012] accepts a subset of Verilog-2001. It can output elaborated Verilog at the operator level, or as a low level BLIF netlist. Because its sweet spot is high-level elaboration, it generates suitable netlists for fault-tolerance insertion, and consequently is one of the options discussed in chapter 9 (Discussion and analysis).

(v) Yosys [Wolf and Glaser, 2013] is the first open-source tool to offer Verilog 2005 support. Many unique features are offered, such as the ability to compare pre- and post- synthesis

netlists, using a mitre and associated SAT solver technique. It assumes the use of ABC within its own functionality. Synthesis proceeds via a series of simplification passes. However, there is no stage corresponding to, the high-level elaboration function of quteRTL. The nearest thing is the ability to import a coarse-grain library, via matching using a SAT solver. This technique is not very scalable, compared with mapping to the coarse-grain library in the first place. Hence, there is no obvious place to incorporate a fault-tolerance capability, with the characteristics required for this dissertation.

Each of these tools have restrictions on the particular subset of synthesisable Verilog they will accept; some have potential portability issues and none of them has direct support for fault-tolerance insertion, as far as the author is aware.

Since it outputs behavioural RTL, it is possible to regard the present embodiment as an HLS tool. As such, it has elements in common with other HLS tools, such as Kiwi [Singh and Greaves, 2008]. These two tools are complimentary, to the extent that the Kiwi I^2C controller is used as part of the display controller for the hardware demonstrator of the technology (chapter 9). However, they are also distinct, in that Kiwi targets optimisation and parallelization of C# code, including flattening of recursion and stack accesses into parallel registers, whereas the present work concentrates on implementing pure functional OCaml code, in the style of Coq.

2.3 Reliability review

The introduction of valves in place of mechanical devices provided a step change in reliability, primarily because each valve provides a gain that isolates previous stages from the consequences of subsequent stages (for example the arithmetic logic unit would not be disturbed by a faulty I/O device). As Alan Turing said, "It will be almost our most serious problem to make sure that the calculator is doing what it should." [Copeland and ..., 2005]. Notwithstanding, he was much more confident about the reliability of his proposed automatic calculating engine (ACE), and envisaged three categories of error: faulty components, unexpected noise or voltages, or program errors and misunderstandings about how the computer worked. The solutions proposed were extra monitoring components, reducing the voltage swings whilst running a known good program to identify components that are near failure, and for software diagnosis, running test vectors or comparing an optimised algorithm with a non-optimised alternative. Meanwhile, his more pragmatic Hungarian/American contemporary John Von Neumann ([Von Neumann, 1956]) was inventing his majority multiplexing system, the forerunner to N-modular redundancy.

The idea of using dual-rail signalling in logic is as old as computing itself. The original high-performance electronic computers used valves, operating as long tail pairs[Blumlein, 1936], to boost performance and compensate for any necessary delay, by using one signalling path to indicate logic one, and the other signalling path to indicate logic zero. Historically, asynchronous techniques failed to make the transition to integrated circuits in large numbers, due to the problem of isochronic forks ([Sretasereekul and Nanya, 2003]), which limit the applicability of automated CAD techniques ([Kondratyev and Lwin, 2002]). Static timing

analysis [Kondratyev and Lwin, 2002], which works effectively only on synchronous paths, rapidly loses effectiveness when any kind of manual checking of circuit topologies is needed. The advent of solid-state diodes and transistors enabled vastly larger assemblies to be made, whilst still being reliable. At first, these circuits were asynchronous [Vasyukevich, 2011] and used a three-state indication of the result, namely in progress, true or false. The introduction of global synchronous clocks allowed the assumption to be made, and largely proved, that after a certain number of clock cycles, the result could be assumed present and correct. To make an adequately reliable computer, a statistical probability of failure in the nine-sigma range might be acceptable, depending on complexity.

A second use of dual-rail signalling is to provide protection against digital power analysis (DPA), and other cryptographic attacks ([Razafindraibe et al., 2006]), ([Sokolov et al., 2005]). In this scheme, the use of invalid states (known as spacers), and other constraints, allows data to be processed, without its contents being visible in the power profile. The applications, in the case of smart cards, are obvious since an invidious card reader will always know exactly how much power the card is consuming.

Just as Turing classified his errors into hardware and software or conceptual errors, the same is true today, except that hardware reliability has improved dramatically. Much effort has been undertaken to mitigate the remaining weak-link, the human programmer factor.

Interest in rigorous proving of reliability is ramping up and the University of Cambridge and its partners recently began a project to use rigorous semantics to improve the quality of mainstream computer systems ([Chisnall et al., 2015]).

Having reviewed the relevant literature, a framework exists for reproducing the methodology of the new apparatus, and this is set out in chapter 3.

"Though this be madness, yet there is method in it"

Hamlet: Act 2, Scene 2, Page 9, Polonius

3 Methodology

In the previous two chapters, first the proposed apparatus was introduced, with its thesis statement 1.2, and then the supporting literature was reviewed. As previously covered in Figure 1.1, a summary of the methodology to derive executing FPGA code from software written in a theorem prover is given in this chapter. Of particular interest is the adaptation of techniques, not originally intended to work together, into an overall scheme. By analogy with impedance matching (which prevents unwanted reflections in electromagnetic communications), the steps in the process illustrated in this chart may be considered to be impedance matching that minimises rejection of information later on in the process.

Hence, the necessary framework is in place to present the methodology for developing a suitable apparatus. This methodology is largely a software methodology, although the final output is demonstrable in hardware, if a suitable FPGA platform is available, such as the ML605 [Xilinx, 2009a, 2012] (as shown in Figure 9.1). A condensed version of this methodology was presented at INDIN'15 [Kimmitt, Greaves and Cirstea, 2015]. By contrast with low-level imperative languages, such as C, the scope of the improvement that can be produced with a modern syntax and toolchain is as follows:

(i) OCaml compiler bugs will be reduced, due to the high level of abstraction in the compiled source code, as well as proof of preservation of type safety in compiler source code [Garrigue, 2010]. By contrast, most C compilers are also written in C, which provides machine independence, but also a lower level of abstraction, and consequently a greater probability of compiler writer error. The size of the language, and the number of legacy features, is also important in this regard.

(ii) Buffer overrun is eliminated by preventing type punning, when pointers to arrays are passed to library functions, and by incorporating bounds checking into the runtime code generation. By contrast, C uses a (void $*$) type in many library routines that operate on

buffers or arrays. In this context, it is impossible for the library routine to carry out the appropriate checks, and consequently an overflow will result in undefined behaviour, (such as giving control to a virus embedded in the overflowing buffer.

(iii) Failure to terminate is eliminated with no further precautions, provided the code compiles in Coq. (The glue code might intentionally loop repeatedly). Termination is difficult to verify in imperative code, typically because of the increased use of side effects.

(iv) Stack overflow will typically be detected by the memory management unit, in a workstation environment, regardless of the language. It is quite usual for a functional algorithm to require more stack space than a comparable imperative algorithm. Hardware checking of stack overflow is straightforward in a custom hardware environment, and considered essential in a traditional embedded environment if it lacks hardware for memory protection. Despite the differences, OCaml will have an advantage in memory protection because its exception mechanism is fine-grained.

(v) Incorrect algorithms are more easily detected in Coq because of the type-safety and the possibility of assertions, ranging from simple, general checks, to sophisticated proofs of equivalence to an abstract specification. By contrast, C algorithms are reliant on separate static analysis tools, such as Frama-C [Cuoq et al., 2012].

(vi) Hardware faults are approximately equally likely in OCaml and traditional imperative code. Again, the more sophisticated OCaml exception mechanism allows a more intelligent treatment of soft errors (for example, by returning to a safe point and retrying the operation). As shown in chapter 7, any synchronous network may be augmented with suitable fault tolerance.

3.1 Methodology introduction

Bearing in mind the research questions from section 1.3, it is necessary to develop a methodology to address the issues that arise, and although the questions are still valid when discussing ASIC hardware, it is essential to prototype any proposed apparatus using FPGA technology, before manufacture. The process begins with an optional step, involving a theorem prover. This is useful, because it allows the software to benefit from techniques that have already been proven from primitive axioms of set theory. The package *unified library for proving floating-point algorithms in Coq* [Boldo and Melquiond, 2011] (Flocq) is one such example, previously used in the floating-point component of CompCert. However, a proven algorithm is not the same as an efficient algorithm, so in applications where a real-time performance requirement on the floating-point arithmetic exists (usually the case), this type of component is best used for cross-checking an imperative hardware implementation. Since the theorem prover uses arbitrary-length arithmetic, there is no direct translation to hardware types. The semantics of program proof are analogous, but not identical to, operational semantics. Operational semantics can be converted to algorithms that actually execute and can derive a result. An operational statement can be thought of as the set of all possible inputs which satisfy that theorem, which in this case must be bounded to a reasonable number, roughly corresponding to

the graph of the function being enumerable in polynomial time, and finite memory. By contrast, statements of type proposition only need to have a hierarchy of proven statements, and axioms generated for them, in order to establish the truth of the statement from fundamental principles. This type of statement cannot readily be converted to an executable formal specification, but it is possible to have propositional theorems, that show the equivalence of the two forms.

3.2 Coq as a hardware verification language

The operational semantics of a Coq description, following proof acceptance, can be exported as meta-language in the form of OCaml, or Haskell code. This methodology is conventional, mature behaviour ([Filliâtre and Letouzey, 2012]). To answer the question whether type-safety can be maintained in hardware, there must exist a method to convert the output of the Coq extraction to executable FPGA code. This will have the following advantages, over conventional ML compilation and execution:

(i) The Coq functions are guaranteed to terminate.

(ii) User assertions are easy to incorporate.

(iii) Arbitrary precision arithmetic is assumed by default (subject to memory limitations).

(iv) Dependent types may be used, to help ensure correctness by construction.

(v) A mathematically proven reference implementation can run as a crosscheck to hardware floating point, or algorithms derived from integer arithmetic.

The subset of OCaml that is output by Coq is a strictly functional subset with deterministic termination. If functionality is required outside of this subset (such as I/O or indefinite looping), this can be done in the form of carefully chosen axioms. These axioms naturally will not be checked in Coq, and could be inconsistent with correct operation. In order for the computer to be a useful Turing machine, it needs to have parsing of source programs, as well as input and output of data. In view of the work of [Koprowski and Binsztok, 2010] and [Jourdan, Pottier and Leroy, 2012] it would be possible to construct a verified parser in Coq. However, for the purpose of this dissertation, the front-end can be treated as a convenience function, and not part of the core verified algorithm.

It should be noted, at this point, that the executable semantics can easily be compiled, and executed, using the workstation's own OCaml compiler. Compatibility exists between the subset supported by the FPGA conversion, and the conventional approach (hardware fault modelling excepted).

The details of theorem proving and extraction are covered in chapter 4. This process corresponds to the box marked 'Coq theorem prover' in Figure 1.2.

3.2.1 Limitations of Coq for hardware verification

It is entirely possible to create a Coq development without any significant assertions. However, some otherwise syntactically correct programming constructs are not allowed by the prover engine. Examples include inductive definitions that diverge, arbitrary recursive functions that do not terminate, imperative input/output, inconsistent axioms, and type-unsafe constructs. These limitations may be enough to rule out wholesale conversion of algorithms from many languages (for example imperative languages). This will be a learning curve for the typical software engineer, who might expect, for example, arbitrary recursion (a valid syntax) to be accepted. In terms of proof of correctness, the user's own assertions must also be added, and this requires a familiarity with the Coq language and the specific requirements of the application, in order to deduce which assertions should be added, and the mathematical steps (based on higher-order predicate logic and set theory), needed to prove those assertions.

3.3 Compiling OCaml for FPGA

This methodology is conventional, in that the source files are compatible with the version of OCaml that compiles native programs for the workstation. However, by substituting certain modified library files, developed for this methodology, degradation of the static type safety is avoided. The main backend development is available to download [Kimmitt, 2014]. The specifics of the backend customisation, to generate behavioural Verilog, are covered in detail in chapter 5.

3.4 Fault tolerance

To support the thesis statement, five alternative approaches to fault tolerance are compared. The zeroth method is plain, non-redundant, implementation. The first tolerance option is conventional triple modular redundancy, with suitable adjustments, to suit the peculiarities of FPGA architecture [Carrol, 2009]. A second alternative is the author's own technique [Kimmitt, Wilson and Greaves, 2012], described in detail in chapter 7. The final two alternatives are improvements based on redundancy insertion at the operator level, using coarse-grain logic synthesis. Standard Xilinx synthesis, place & route tools are used to create a bitstream that can be loaded by the Xilinx board (or other target).

3.5 Garbage Collection

In a type-safe language, to prevent the user from escaping type safety, it is not allowed to allocate or free memory directly. The lack of user-controlled memory is a key strength in program provability. Nevertheless, it places a burden on the run-time system, to decide how and when to perform garbage collection, which can be defined as scanning memory to find and remove objects that are not referenced (except possibly by themselves). The latter can never happen, unless mutable references are in use (a useful but potentially dangerous

feature). Nevertheless, there is a class of systems, that need not be concerned with garbage collection: those systems where the total memory is bounded by program design, and those systems, such as smart cards, where the device is powered off after every transaction (or after a few transactions), and therefore never has a chance to overflow. A further class of systems where a finite function is run under control of a server main loop, after which the heap pointer is returned to the value at the start of the main routine, are possible with care, if it is determined that no references to the said data remain in the top loop. The question arises, can the theorem prover determine whether these requirements are met, using a set of standing assertions that are portable over many applications. Naturally this is a higher-order question that needs to be answered as part of an assertion about a shallow-embedding language. Such a system would need to keep track of all allocations that could happen, within the context of a particular embedded algorithm. It is quite clear that it cannot be done for a general purpose embedded λ-Calculus, since this would be just another version of **the halting problem**. A more sophisticated parsing system could perform escape analysis [Blanchet, 1998] to prevent unsafe expressions from being entered into the system. The remaining systems, that do require garbage collection, require a trigger in order to begin the process. In the case of the present embodiment, which is no different from other systems in this respect, this could be a test whether the heap has reached 75%-100% of the available memory (being total memory less collector overhead - up to 50% for some algorithms). The collector should preferably be an algorithm that is suitable for hardware specialisation to take advantage of the greater performance of custom hardware operations. Garbage collection is discussed further in chapter 8.

3.6 Utility

From inception, it was always planned that all the results obtained would be proven on FPGA, for two reasons. Firstly, in view of the costs of mask-making and minimum wafer order quantities, it would be irresponsible to not verify as much as possible of the design on FPGA. Secondly, in some critical environments (such as aerospace), production volumes simply do not justify the cost of specially mask-programmed circuits. In this case, it is appropriate to make a virtue of necessity, and orient the behaviour towards making the most of FPGA-specific resources. This has numerous advantages, particularly being able to make use of the inherent parallelism of the FPGA, to operate several threads of execution in parallel. Current state-of the art static analysis techniques struggle to cope with the effect of concurrent interrupts on a software application. If the same architecture is replaced with multiple hardware independent threads of execution, and/or suitable hardware assistance, it must result in a dramatic simplification in verification difficulty.

3.7 Summary

To describe the methodology in more detail, individual flows have been separated into the following chapters as follows: firstly, the theorem proving and dependant type checking aspects are explored in chapter 4; next, the language and compilation issues are considered in chapter 5;

subsequently the conversion to hardware is described in chapter 6; following on from this the reliability at the hardware level is investigated in chapter 7; the method of dynamic storage management is discussed in chapter 8; finally, the results of applying the methodology to a shallow embedding of a dependently-typed λ-Calculus, are given in chapter 9.

"J'avais un peu le Cafard quand tu es parti mais ça va mieux car je suis sur que tu es comme un Coq en pâte. Il semble que je passe du Coq à l'Âne, donc revenons à nos Moutons, il ne me reste qu'a te faire pleins de bisous mon Canard."

[ThePopCase, 2015]

4

Dependent types in Coq

The computer program Coq (and subsidiary languages) is a theorem prover based on constructive logic; it has the property that a program can be derived automatically from a proof. For maximum assurance, all proofs are derived from as few axioms as possible. For the purposes of this thesis, a key foundation is that numeric arithmetic (integer and floating point) is proven from primitive notions of set theory. However, for performance reasons these primitive arithmetics will typically be mapped onto regular machine integers in the FPGA description of the generated logic.

In chapter 3, a summary of the methodology for converting a program, with proven characteristics, to FPGA was given. In this chapter more detail is given for the first phase of the process, namely producing a coding of an algorithm as an executable formal specification, together with various application-specific proof targets that have been selected by the user as the desirable characteristics. Concerning embedded environments, a number of properties might be important, such as: fairness (giving equal priority to different inputs to the system, and/or different internal algorithms, requiring access to shared resources); liveness (proving that the program can never end up in a state that is non-responsive); termination: (proving that the program can never get stuck in an infinite loop); safety (proving that transformations or transitions, can never lose their type).

4.1 Proving formalisations in Coq

The Coq theorem prover is based on constructive (also known as intuitive) logic. With the addition of the Axiom of the excluded middle, it can also behave in the manner of a classical, predicate logic, theorem prover. It is well established, from Coq's documentation, that the tool supports two styles of theorem proof, those of type proposition and those of type set. It

is less obvious, at first glance, that only proofs of type set, can be developed into executable semantics. Coq also supports the notion of dependent types ([Weirich, 2014]), which can be used to prevent certain kinds of program bugs from being introduced, so can be thought of, as a kind of compile-time proof by construction.

4.1.1 Executable proofs

All compilable formalisations of theorems in Coq are finite. In a mathematician's mind a proof by induction will continue to infinity, however this is inconvenient to represent in a computer, and it is necessary to desist as soon as sufficient terms are generated to satisfy the induction. Likewise, non-terminating recursions are not allowed. A subset of theorems, where the number of possibilities is not what Coq considers 'large', can be represented as type Set, meaning the set of all inputs that satisfy the theorem. In this case, executable semantics may be extracted as an OCaml, or Haskell program.

The operational semantics of a Coq description, following proof acceptance, can already be exported as meta-language in the form of OCaml, or Haskell code. Notwithstanding, what would be desirable is to be able to execute the verified code in hardware. This will have the advantage over conventional ML compilation and execution, in that the valid execution path of the process is proven. For example, if the formally proven floating-point arithmetic library for Coq, known as Flocq, is extracted to hardware, it can be used as a reference to establish whether the corresponding floating-point co-processor is correct. It can also be used as a reference implementation, to carry out certain tasks, in the absence of floating-point instructions, or other libraries. However, because of the overhead of arbitrary precision arithmetic, and the list representation, this kind of program would be too slow, and memory hungry, to completely replace conventional techniques.

4.2 Formalisation of the λ-Calculus

As a non-trivial extension of the work of Gribeiro [Gribeiro, 2012], which is in turn based on Software Foundations [Pierce et al., 2012], it is proposed to formalise signed integer arithmetic, that is closed under addition, subtraction, multiplication and division, as well as floating-point calculations, based on the aforementioned Flocq library for Coq. This is the same engine used in the CompCert verified compiler for the C language [Leroy, 2009b]. The special case of division by zero is considered in section 9.5.2. After Coq has proven some interesting properties of the formalisation, such as substitution preserving typing, context invariance, progress, correctness (only for specific examples!), it is possible to extract an executable program that meets these criteria, as well as termination (in the absence of contrary axioms) and static type safety.

The extracted, proven program needs to be compiled. In view of the work of Dargaye [Dargaye, 2009], a formally proven tool-chain from Coq to executable code is, in theory, possible. However, the subset of ML supported by Coq, is richer than the *EpsilonML* subset supported by Dargaye, consisting of (Var, Let(rec), Fun, App, Cons, Match) λ-expressions. Furthermore, it would be very laborious to modify hand-written code, such as a parser, to conform to this

subset. If it were desired to support a conversion written in Coq, then a better approach would be as an extraction routine that targets this language directly, together with further extensions to Dargaye's formalisation. Currently, although the principles remain valid, the work needs updating before it can be used again, because the theorem prover evolves along with its tactics.

4.3 The extraction process

As already mentioned, theorems of type Prop(osition) are skipped when generating OCaml output. In general, a theorem of type Prop, as in Table 4.1, (based on the formalisation due to Pierce [Pierce et al., 2012]) cannot be converted to one of type Set (the set of input conditions which satisfy the theorem), because the state space is too large. In addition, dependent types in OCaml are not allowed, so the alternative form of Table 4.2, that is statically typed, has been produced. As an example, the first line of Table 4.1, which can be read as *In order for the term to have dependent type ST_AppAbs, for all values of bound variables x, T11, t12, v2, where v2 is a value and the term tm is an application of an abstraction x on T11, and t12 to an argument v2, then the theorem will reduce to the substitution of v2 in x on t12.* In the executable semantics the following is instead: *If tm is typed as an application of an abstraction named a, applied to v2, and if v2 is a reducible value, then the same substitution occurs, otherwise (attempt to) reduce the abstraction known as a, and its argument v2.* The latter example can be readily extracted to OCaml, in the form of Table 4.3.

A computer is not useful if it cannot interact with the outside world, so the presence of an input/output axiom is necessary, operating by side effect of being called. This will not be represented as a theorem in Coq, because the side effect is transparent. To prevent spurious reliance on vacuous proofs that assert an axiom as part of the proof, Coq provides several mechanisms: firstly, the ability to print out what axioms a proof depends on (using the command Print Assumptions), secondly, by providing a warning message during extraction, and thirdly, providing an OCaml stub which fails immediately, not just when the axiom is invoked but during startup. This is not especially useful out of the box, so as a workaround an override such as *Extract Constant STLCExtended.axiom_io=>"Axiom_io.axiom_io"* should be used. This will redirect any calls to the axiom to a carefully prepared, hand-written OCaml file. This file would contain content similar to Table 4.4. At this point, compilation is possible to the native OCaml compiler, and any necessary debugging of axioms, and/or user interface issues, can be investigated. An introductory fragment of the corresponding (more verbose but equivalent) OCaml code is shown in Table 4.3.

Table 4.1: Dependently typed λ-Calculus formalisation

```
Inductive step : tm → tm → Prop :=
 | ST_AppAbs : ∀ x T11 t12 v2, value v2 → (tm_app (tm_abs x T11 t12) v2) ==> (subst x v2 t12)
 | ST_App1 : ∀ t1 t1' t2, t1 ==> t1' → (tm_app t1 t2) ==> (tm_app t1' t2)
 | ST_App2 : ∀ v1 t2 t2', value v1 → t2 ==> t2' → (tm_app v1 t2) ==> (tm_app v1 t2')
 | ST_Succn : ∀ n, tm_succ (tm_nat n) ==> tm_nat (Zsucc n)
 | ST_Succ : ∀ t t', t ==> t' → (tm_succ t) ==> (tm_succ t')
 | ST_Predn : ∀ n, tm_pred (tm_nat n) ==> tm_nat (Zpred n)
 | ST_Pred : ∀ t t', t ==> t' → (tm_pred t) ==> (tm_pred t')
 | ST_Add1 : ∀ t1 t1' t2, t1 ==> t1' → (tm_add t1 t2) ==> (tm_add t1' t2)
 | ST_Add2 : ∀ v1 t2 t2', value v1 → t2 ==> t2' → (tm_add v1 t2) ==> (tm_add v1 t2')
 | ST_AddV : ∀ n1 n2, tm_add (tm_nat n1) (tm_nat n2) ==> (tm_nat (n1 + n2))
 | ST_Sub1 : ∀ t1 t1' t2, t1 ==> t1' → (tm_sub t1 t2) ==> (tm_sub t1' t2)
 | ST_Sub2 : ∀ v1 t2 t2', value v1 → t2 ==> t2' → (tm_sub v1 t2) ==> (tm_sub v1 t2')
 | ST_SubV : ∀ n1 n2, tm_sub (tm_nat n1) (tm_nat n2) ==> (tm_nat (n1 - n2))
 | ST_Mult1 : ∀ t1 t1' t2, t1 ==> t1' → (tm_mult t1 t2) ==> (tm_mult t1' t2)
 | ST_Mult2 : ∀ v1 t2 t2', value v1 → t2 ==> t2' → (tm_mult v1 t2) ==> (tm_mult v1 t2')
 | ST_MultV : ∀ n1 n2, tm_mult (tm_nat n1) (tm_nat n2) ==> (tm_nat (n1 × n2))
 | ST_Div1 : ∀ t1 t1' t2, t1 ==> t1' → (tm_div t1 t2) ==> (tm_div t1' t2)
 | ST_Div2 : ∀ v1 t2 t2', value v1 → t2 ==> t2' → (tm_div v1 t2) ==> (tm_div v1 t2')
 | ST_DivV : ∀ n1 n2, tm_div (tm_nat n1) (tm_nat n2) ==> (tm_nat (n1 × n2))
 | ST_Mod1 : ∀ t1 t1' t2, t1 ==> t1' → (tm_mod t1 t2) ==> (tm_mod t1' t2)
 | ST_Mod2 : ∀ v1 t2 t2', value v1 → t2 ==> t2' → (tm_mod v1 t2) ==> (tm_mod v1 t2')
 | ST_ModV : ∀ n1 n2, tm_mod (tm_nat n1) (tm_nat n2) ==> (tm_nat (n1 × n2))
 | ST_IfZ : ∀ t2 t3, tm_if0 (tm_nat Z0) t2 t3 ==> t2
 | ST_IfS : ∀ n t2 t3, (Zeq_bool n Z0) = false → tm_if0 (tm_nat n) t2 t3 ==> t3
 | ST_If : ∀ t1 t1' t2 t3, t1 ==> t1' → (tm_if0 t1 t2 t3) ==> (tm_if0 t1' t2 t3)
 | ST_Let1 : ∀ x t1 t1' t2, t1 ==> t1' → tm_let x t1 t2 ==> tm_let x t1' t2
 | ST_LetValue : ∀ x v1 t2, value v1 → tm_let x v1 t2 ==> subst x v1 t2
 | ST_Pair1 : ∀ t1 t1' t2, t1 ==> t1' → (tm_pair t1 t2) ==> (tm_pair t1' t2)
 | ST_Pair2 : ∀ v1 t2 t2', value v1 → t2 ==> t2' → (tm_pair v1 t2) ==> (tm_pair v1 t2')
 | ST_Fst1 : ∀ t1 t1', t1 ==> t1' → tm_fst t1 ==> tm_fst t1'
 | ST_FstPair : ∀ v1 v2, value v1 → value v2 → tm_fst (tm_pair v1 v2) ==> v1
 | ST_Snd1 : ∀ t1 t1', t1 ==> t1' → tm_snd t1 ==> tm_snd t1'
 | ST_SndPair : ∀ v1 v2, value v1 → value v2 → tm_snd (tm_pair v1 v2) ==> v2
 | ST_Inl : ∀ t1 t1' T, t1 ==> t1' → tm_inl T t1 ==> tm_inl T t1'
 | ST_Inr : ∀ t1 t1' T, t1 ==> t1' → tm_inr T t1 ==> tm_inr T t1'
 | ST_Case : ∀ t0 t0' y1 y2 t1 t2, t0 ==> t0' → tm_case t0 y1 t1 y2 t2 ==> tm_case t0' y1 t1 y2 t2
 | ST_CaseInl : ∀ v0 x1 t1 x2 t2 T2, value v0 → tm_case (tm_inl T2 v0) x1 t1 x2 t2 ==> (subst x1 v0 t1)
 | ST_CaseInr : ∀ v0 x1 t1 x2 t2 T1, value v0 → tm_case (tm_inr T1 v0) x1 t1 x2 t2 ==> (subst x2 v0 t2)
 | ST_Cons1 : ∀ t1 t1' t2, t1 ==> t1' → tm_cons t1 t2 ==> tm_cons t1' t2
 | ST_Cons2 : ∀ v1 t2 t2', value v1 → t2 ==> t2' → tm_cons v1 t2 ==> tm_cons v1 t2'
 | ST_LCase1 : ∀ t1 t1' t2 x y t3, t1 ==> t1' → tm_lcase t1 t2 x y t3 ==> tm_lcase t1' t2 x y t3
 | ST_LCaseNil : ∀ T t2 x y t3, tm_lcase (tm_nil T) t2 x y t3 ==> t2
 | ST_LCaseCons : ∀ vh vt t2 x y t3, value vh → value vt →
                    tm_lcase (tm_cons vh vt) t2 x y t3 ==> (subst x vh (subst y vt t3))
 | ST_Fix1 : ∀ t1 t1', t1 ==> t1' → tm_fix t1 ==> tm_fix t1'
 | ST_FixAbs : ∀ x T1 t2, tm_fix (tm_abs x T1 t2) ==> (subst x (tm_fix (tm_abs x T1 t2)) t2)
 | ST_IO : ∀ x t t', t ==> t' → (tm_io x t) ==> (tm_io x t')
    where "t1 '==>' t2" := (step t1 t2).
```

Table 4.2: Statically typed executable semantics

```
Fixpoint reduce (t:tm) : tm :=
  match t with
  | tm_app ((tm_abs x ty t12) as a) v2 ⇒ if redvalue v2 then (subst x v2 t12)
    else tm_app (reduce a) (reduce v2)
  | tm_app a b ⇒ tm_app (reduce a) (reduce b)
  | tm_succ (tm_nat n) ⇒ tm_nat (Zsucc n)
  | tm_succ v ⇒ tm_succ (reduce v)
  | tm_pred (tm_nat n) ⇒ tm_nat (Zpred n)
  | tm_pred v ⇒ tm_pred (reduce v)
  | tm_add (tm_nat n1) (tm_nat n2) ⇒ (tm_nat (Zplus n1 n2))
  | tm_add a b ⇒ tm_add (reduce a) (reduce b)
  | tm_sub (tm_nat n1) (tm_nat n2) ⇒ (tm_nat (Zminus n1 n2))
  | tm_sub a b ⇒ tm_sub (reduce a) (reduce b)
  | tm_mult (tm_nat n1) (tm_nat n2) ⇒ (tm_nat (Zmult n1 n2))
  | tm_mult a b ⇒ tm_mult (reduce a) (reduce b)
  | tm_div (tm_nat n1) (tm_nat n2) ⇒ (tm_nat (Zdiv n1 n2))
  | tm_div a b ⇒ tm_div (reduce a) (reduce b)
  | tm_mod (tm_nat n1) (tm_nat n2) ⇒ (tm_nat (Zmod n1 n2))
  | tm_mod a b ⇒ tm_mod (reduce a) (reduce b)
  | tm_addf (tm_flt n1) (tm_flt n2) ⇒ (tm_flt (Fplus n1 n2))
  | tm_addf a b ⇒ tm_addf (reduce a) (reduce b)
  | tm_subf (tm_flt n1) (tm_flt n2) ⇒ (tm_flt (Fminus n1 n2))
  | tm_subf a b ⇒ tm_subf (reduce a) (reduce b)
  | tm_mulf (tm_flt n1) (tm_flt n2) ⇒ (tm_flt (Fmult n1 n2))
  | tm_mulf a b ⇒ tm_mulf (reduce a) (reduce b)
  | tm_divf (tm_flt n1) (tm_flt n2) ⇒ (tm_flt (Fdiv n1 n2))
  | tm_divf a b ⇒ tm_divf (reduce a) (reduce b)
  | tm_if0 (tm_nat Z0) t2 t3 ⇒ t2
  | tm_if0 (tm_nat _) t2 t3 ⇒ t3
  | tm_if0 a b c ⇒ tm_if0 (reduce a) b c
  | tm_iflt (tm_nat (Zneg _)) t2 t3 ⇒ t2
  | tm_iflt (tm_nat _) t2 t3 ⇒ t3
  | tm_iflt a b c ⇒ tm_iflt (reduce a) b c
  | tm_let x v1 t2 ⇒ if redvalue v1 then subst x v1 t2 else tm_let x (reduce v1) t2
  | tm_pair v1 t2 ⇒ if redvalue v1 then tm_pair v1 (reduce t2) else tm_pair (reduce v1) t2
  | tm_fst (tm_pair v1 v2) ⇒ if redvalue v1 && redvalue v2 then v1 else tm_fst (tm_pair v1 v2)
  | tm_fst v ⇒ tm_fst (reduce v)
  | tm_snd (tm_pair v1 v2) ⇒ if redvalue v1 && redvalue v2 then v2 else tm_snd (tm_pair v1 v2)
  | tm_snd v ⇒ tm_snd (reduce v)
  | tm_inl a b ⇒ tm_inl a (reduce b)
  | tm_inr a b ⇒ tm_inr a (reduce b)
  | tm_case ((tm_inl T2 v0) as v0') x1 t1 x2 t2 ⇒ if redvalue v0 then (subst x1 v0 t1)
    else tm_case (reduce v0') x1 t1 x2 t2
  | tm_case ((tm_inr T1 v0) as v0') x1 t1 x2 t2 ⇒ if redvalue v0 then (subst x2 v0 t2)
    else tm_case (reduce v0') x1 t1 x2 t2
  | tm_case a b c d e ⇒ tm_case (reduce a) b c d e
  | tm_cons v1 t2 ⇒ if redvalue v1 then tm_cons v1 (reduce t2) else tm_cons (reduce v1) t2
  | tm_lcase (tm_nil T) t2 x y t3 ⇒ t2
  | tm_lcase ((tm_cons vh vt) as v') t2 x y t3 ⇒ if redvalue vh && redvalue vt
    then (subst x vh (subst y vt t3)) else tm_lcase (reduce v') t2 x y t3
  | tm_lcase a b c d e ⇒ tm_lcase (reduce a) b c d e
  | tm_fix (tm_abs x T1 t2) ⇒ (subst x (tm_fix (tm_abs x T1 t2)) t2)
  | tm_fix v ⇒ tm_fix (reduce v)
  | tm_nat v ⇒ tm_nat v
  | tm_flt f ⇒ tm_flt f
  | tm_var v ⇒ tm_var v
  | tm_nil v ⇒ tm_nil v
  | tm_abs a b c ⇒ tm_abs a b (reduce c)
  | tm_io id (tm_nat n) ⇒ tm_nat (axiom_io id n)
  | tm_io id v ⇒ tm_io id (reduce v)
  end.
```

Table 4.3: Fragment of extracted code for λ-Calculus application

```
 let rec reduce = function
| Coq_tm_app (a, b) ->
   (match a with
    | Coq_tm_abs (x, ty0, t12) ->
      if redvalue b
      then subst x b t12
      else Coq_tm_app ((reduce a), (reduce b))
    | _ -> Coq_tm_app ((reduce a), (reduce b)))
    ... to be continued ...
```

Table 4.4: Replacement axioms in OCaml code

```
open BinNums
open Datatypes
open BinNat
open BinPos
open BinInt
open BinNums
open Mylib
open Ml_decls
open Fappli_IEEE
open To_from_Z

(** val axiom_io : nat -> coq_Z -> coq_Z **)

let axiom_io sel arg =
  let iosel = (init_codes()).io in
  if EqNat.beq_nat sel iosel.char_of_int then arg
  else if EqNat.beq_nat sel iosel.print_char then (print_char (char_of_int (fromZi arg)); arg)
  else if EqNat.beq_nat sel iosel.exn_failwith then failwith "failwith _"
  else failwith ("axiom_io: "^string_of_int (fromNat sel)^" - AXIOM TO BE REALIZED")

(** val big_endian : bool **)

let big_endian = false

type coq_float = float (* coq_float: AXIOM TO BE REALIZED *)

  (** val zero : float **)

  let zero = B754_zero false (* failwith "zero: AXIOM TO BE REALIZED" *)
```

4.4 OCaml for Hardware Description language use

The λ-Calculus execution environment envisaged by Church [Barendregt, 1997] requires a more complicated execution environment than the usual imperative register machine. The SECD arrangement of Landin [Danvy, 2005; Landin, 1964], and the Categorical Abstract Machine (CAM) implementation described by Cousineau ([Cousineau, Curien and Mauny, 1987]) are higher level, more abstract possibilities. The SECD design requires multiple storage areas, leading to fragmentation, and the CAM design requires external software library support for almost any non-trivial operation. Within the solution space, there is a continuum between totally application specific hardware, such as the *Reduceron* ([Naylor and Runciman, 2012]), and fixed datapath architectures where the instruction set cannot be expanded at all.

It should be immediately apparent, that the Coq output description is incompatible with direct execution as hardware. The possibility of an arbitrary recursion is not supported in VHDL/Verilog HDL, nor is dynamic memory allocation implicitly supported. However, if a custom backend to the optimising code generator is considered, it is apparent that all the necessary machine independent features, such as closure conversion and linearization, are shared between different machine implementations. This is beneficial, because the same code can be executed on a high-performance workstation, or on the embedded platform (subject to word-length differences). Therefore, any discrepancy will be isolated to a very few compiler source files. Furthermore, adopting the compiler backend from the Advanced *RISC* Machine architecture (ARM) variant as the template is attractive, because it has relatively many registers, which are cheap in hardware and reduce memory accesses that are always a bottleneck on FPGA designs. In addition, starting with a known-good port reduces the possibility of bugs. Given that the aim of validation in Coq is to improve reliability, some control over the program counter can be exercised, by making use of a hardware state machine. This has a second benefit, that all control delays, such as memory waits, can be isolated in one part of the state machine, associated with reads/writes.

There are advantages and disadvantages for register allocation in hardware. If global registers are used, then flip-flop count is reduced, but routing congestion will be increased. In modern FPGA architectures such as Xilinx, registers are plentiful, and associated with every logic operation. On the other hand, if local registers are used, these values need to be copied, to allow for procedure parameters, which are cheaper when executing from registers than on the stack.

At present OCaml has a second raft of machine dependencies, in the form of polymorphic comparison types, which are used in hashing and many library functions, such as testing for list membership. It also has an elaborate tagging system that can be used to distinguish, for example, between ints and floats being passed to a sorting routine, with the source code for these two cases apparently being generic. It would be difficult to provide equivalent replacements for many of these routines, and in particular, to provide a garbage collection routine that is in every way compatible with the OCaml runtime. Fortunately, the compiler does not itself require all these libraries to be present; instead it has a technology built in to it called Ulambda that provides inline replacements for common operations, which would otherwise have required a library routine. If access to any externals that don't have a '%' in the

name can be avoided, then no run-time library implications are assumed. Unfortunately, the 'Pervasives' library of OCaml contains many such dependencies, so it is necessary to compile using the -nopervasives option. This necessitates a replacement library for the new functions needed, otherwise type signatures will not be compatible between native and state machine modes.

4.5 Design and Proving Example

The formalisation due to Pierce [Pierce et al., 2012] makes an interesting demonstrator, because it offers the possibility of arbitrary λ-Calculus expression being entered into the machine, and executed in hardware, or in other words a Turing-complete interpreter. However, a simplified example is more convenient to illustrate the design and proving process. For this example, an embedding is defined which is just powerful enough to allow factorials to be calculated. Referring to the listing below, the following observations may be made. The language (of the shallow embedding) is a subset of that introduced in section 4.2, with arbitrary arithmetic, floating point operations, lists, sum and pair types removed. The resulting language is still powerful enough to represent a function such as factorial (n!). The λ-expression for factorial, using this notation is given as:

```
Definition fact :=
  tm_fix
    (tm_abs f (ty_arrow ty_Nat ty_Nat)
      (tm_abs a ty_Nat
        (tm_if0
           (tm_var a)
           (tm_nat 1)
           (tm_mult
              (tm_var a)
              (tm_app (tm_var f) (tm_pred (tm_var a)))))))).
```

and can be interpreted as: a function called fact contains a fixed-point, an abstraction of the name f of type natural integer -> natural integer, and a parameter a of type natural integer; its body is an if statement which evaluates to 1 if the argument is zero, otherwise to parameter a multiplied by the application of f to the predecessor of parameter a. This example is the same as given by [Gribeiro, 2012], except that the use of signed integers (also known as Z arithmetic in Coq), removes the need for special treatment of the case pred(0). A special feature of Coq is that it prevents the introduction of an incorrect proof into the system (unless the system is cheated by introducing an incorrect axiom), and this means that the entirety of the proof apparatus is removed further downstream, resulting in an ML program that is much shorter.

A second raft of axiom introduction can occur when the given program is extracted to the syntax of a different language, with more restrictive requirements. There are two kinds of mapping possible, 1:1 mappings such as lists and bools, and truncating internal Coq arithmetic types to machine integers of fixed precision. The listing below shows an extraction that does not perform any such truncation. The mappings are semantic equivalents that help with debugging and interfacing with existing OCaml code. Each *Extract Inductive* statement represents a different mapping.

```
Extraction Language Ocaml.
Require Import Bool.
```

```
Require Import List.

Extract Inductive bool => "bool" [ "true" "false" ].
Extract Inductive sumbool => "bool" [ "true" "false" ].
Extract Inductive unit => "unit" [ "()" ].
Extract Inductive list => "list" [ "[]" "(::)" ].

Require Import ZArith_base.
Require Import Zdiv.
Require Import Bool.
Require Import exampleZ.

Extraction Blacklist Lambda String List.

Recursive Extraction Library ZArith_base.
Recursive Extraction Library Zdiv.
Recursive Extraction Library Bool.
Recursive Extraction Library exampleZ.
```

To make the example executable, the contents of the library of Appendix-A and the example of Appendix-B need to be read in conjunction with this code. The explanation continues in section 5.5 (Extracted Compilation Example).

4.6 Summary

It has been seen that executable semantics that have been subjected to certain assertions can be dumped in a format that is a subset of OCaml code. In chapter 5, the compilation aspects are considered, as well as demonstrating that the chosen subset is compatible with ordinary execution on a workstation. An important class of embedded systems are reactive systems, such as engine management systems, whereby the software can only work in conjunction with a real engine, or simulation of some such system [Greaves and Gordon, 2006]. This could be a conventional simulation model, or it could include a formal model containing various assertions that could be exercised.

"My word ... shall not return to Me void"

Isaiah 55:11

5

An OCaml compiler for bare-FPGA

The purpose of the previous chapter was to explain facilities for proving programs with Coq, and for extracting the executable behaviour as an OCaml program. The purpose of the present chapter is to design and explain a program which takes specifications written in functional meta-language and translates them to an imperative register machine that can operate as an FPGA state-machine. The style of meta-language is OCaml. However there is no fundamental reason, that the present author is aware of, why a lazy language such as Haskell could not also be used.

The semantics of the translated executable will be the same, whether executed natively on the workstation that hosts the compilation suite, or on the eventual FPGA platform (subject to memory and performance limitations). In this way, the programs can be tested directly (once the axioms have been studied and replaced with suitable alternatives during extraction).

In an ideal world, all the stages from initial proof and/or assertions to direct execution in hardware should be fully verified for semantic equivalence. Bearing in mind *mlcompcert*, due to Dargaye [Dargaye, 2009], the possibility exists of a route from functional code, to assembly language, with equivalence of semantics. However, the Coq output is a superset of the syntax accepted by Dargaye. Indeed, the Coq type system is more advanced even than OCaml, and sometimes needs extra coercions, especially when outputting axioms. It would not be easy to alter the code generator to fit a simpler syntax, or indeed extend the *mlcompcert* source code in Coq, to cope with all the possible constructors that could be output. At some point in the future it might be possible to write a new backend for Coq, which targeted a lower level machine using an augmented version of *mlcompcert*. A related piece of work is found in [Jaber, 2010]. In this case, the target machine is SECD. This is a more abstract level than the CAM [Cousineau, Curien and Mauny, 1987] that the OCaml interpreter is based on. Although the optimised version of the CAM is sufficiently low level for hardware implementation, it

does not meet the requirement for maintaining type-safety, since almost every non-trivial operation requires recourse to a library of C-primitives. For the purposes of this dissertation, a more advanced solution based on the OCaml native code generator will be presented. As investigated by [Garrigue, 2010], it is not easy to demonstrate the correctness of a compilation solution that has many optimisations and specific features.

5.1 Comparison to High Level Synthesis

For a number of years, a dichotomy existed between hardware design techniques and software engineering, which failed to meet in the middle. Historically, the golden standard for hardware was schematics. Then the focus changed, to register transfer level [Crate, 1996], [Bhasker et al., 2002], [Meredith and Katelman, 2010]. Eventually HLS techniques were introduced ([Greaves, 2003], [Oliver, 2006], [Singh and Greaves, 2008], [Naylor and Runciman, 2012]), which involves the compiler in scheduling of accesses to RAM and registers, and possibly automatic register duplication or retiming. The majority of these methods, though interesting, have the disadvantage that any change of software is likely to result in a major recompilation of the entire FPGA. The biggest obstacle to overcome, in a functional environment, is how to deal with recursion in hardware. Although Coq always generates algorithms that terminate, there is no guarantee that unrolling an inductive theorem will terminate within the available logic capacity of an FPGA. Therefore, for this functional embodiment, the stack is implemented explicitly as internal block-RAM of the FPGA (as opposed to distributed RAM, that may be used for registers or small register files). For full robustness, given that objects and temporary results will also be allocated in RAM, hardware detection of a collision between heap pointer and stack pointer is highly desirable. This functionality is suitable for implementation as an exception, because exceptions in OCaml always reduce the size of the stack to a previous checkpoint.

5.2 Custom OCaml backend

Workstation compilers for OCaml typically output assembly language for the host machine. The behaviour of the host machine is usually axiomised below this point. However, what is actually required, for high reliability, is a procedure to transfer the extracted program to hardware in the form of programmable logic. If it is only desired to support Coq output, a smaller subset of the language is adequate. In practise, the necessary axioms and supporting libraries, together with parsing and printing code, result in a substantial subset being supported.

One of advantages of choosing OCaml initially is that it supports native code-generation, for typical processors such as x86, amd64, Sparc, ARM, and PowerPC etc. A downside is that the unusual task of porting to a new processor is not well documented. The ARM template was chosen as the starting point, as it has relatively few instructions to support, and is 32-bit. A naïve implementation (of the compiler) does not require a processor at all; the instructions can be output directly as behavioural Verilog. Such descriptions, though theoretically synthesisable, will surely fail eventually, due to resource utilisation, or congestion

of routing to shared registers. Since an FPGA has plenty of flip-flops available, the congestion could be relieved by using HLS techniques, such as register renaming, in strategic places in the code. However, the main bottlenecks, of call and return values, cannot be renamed, because there are insufficient degrees of freedom available, when a library item (such as currying or uncurrying), can be called from many different places. In addition, the application binary interface (ABI) would be violated, which causes difficulty in debugging. For these reasons, this methodology was not pursued.

The backend stages of OCaml, after closure conversion, use a C-like syntax known as C - - or cmm. It is straightforward to convert most of this language to an arbitrary processor, but there are some subtleties. Integers are stored unboxed by default, and always have their least-significant bit set. Any arithmetic operation on these objects must be more complicated, due to this restriction. Pointers, on the other hand, must be even, and in this case, the hardware is made significantly simpler, by adopting a strict alignment policy (pointers must be aligned on 4-byte boundaries). This choice is made based on a compromise between efficiency of memory usage and efficiency loss resulting from an unaligned access, resulting in multiple transactions on the memory bus (and associated loss of runtime predictability). Since most objects are pointers or 32-bit integers, the inefficiency of storing a few aligned characters here and there is acceptable. For small items, such as characters in strings, there are no such restrictions because only the entire string would be looked at as a whole by the garbage collection function.

In view of the custom nature of the ALU, it would be relatively trivial to provide instructions that directly manipulate built-in integers, to keep the LSB as logic one. At this time this has not been done, because the relevant manipulations are not concentrated in one place in the compiler, and it is necessary to carefully distinguish between pointer arithmetic and/or boxed integers, and built-in integers. A further complication is that programs that explicitly detect the word-length would potentially return a different answer with an optimised arithmetic, compared with a combination of conventional instructions.

Exceptions on OCaml make use of an efficient paradigm, whereby the innermost *try .. with* block will store its recovery vector in a fixed register. The process of setting up a trap is analogous to a function call into the current body; this can confuse tracing tools. Pushing and popping a trap is the process of replacing the current recovery vector by a new vector and saving the original on the stack. In the current implementation, pushing and popping traps is supported, but taking an exception in native mode is neither supported nor required (if reliance on native type-unsafe libraries is removed).

5.3 Library implementation

The standard pervasives library that comes with OCaml incorporates a variety of unrelated operations, including floating-point arithmetic and conversions. In theory, all of these functions could be implemented using Flocq based primitives, and library code, but it is inconvenient to force software to rely on Coq, even if it is not needed. As an alternative, a new library has been written, that only contains the chosen supported sub-set of OCaml. This includes all previously

encountered styles of Coq executable semantics, in addition to well-known axioms and I/O facilities. A summary is given in Table 5.1. Operation is similar to the standard library, with a compatible application programmer's interface (API), except that polymorphic comparison is not supported, hence library routines, such as association lists, that require comparison to be done, need to pass an explicit type-specific comparison function as a parameter. Further details are available in Appendix-D and Appendix-E.

In addition to Verilog output, two further options exist for code generation. The first one is intermediate code output, which consists largely of CMM, written in a form friendly to further OCaml compilation. This can be used to export startup sequences and other internal features. An example is shown in Table 5.2. It can be used to get internal implementations of currying, should it be needed (OCaml does not curry by default, unless functions are partially evaluated). The second option is C simulation, which outputs a simulation of what the Verilog would do, along with a suitable prelude.

Table 5.1: Embedded OCaml library features

Name	Function	Comment
raise	take an exception	Could happen in hardware at any point
arithmetic	+,-,*,/,mod,lsl,lsr,asr,land,lor,lxor	integer operations
boolean	&&,||	short-circuit evaluations
relational operators	<,<=,=,!=,>=,>	Polymorphic comparison not supported
references	ref, !, :=	complicates garbage collection
string functions	create,length,get,set,blit,fill,eqb,iter,make,sub,@	Uses built-in string type
list functions	length,hd,tl,nth,rev,iter,map,fold,mem',assoc	Explicit comparison needed
array functions	make,length,get,set,fold_left,iter,to_list	May need assembly support
I/O functions	input_byte, input_char, output_char, flush	Potentially implementation specific
conversions	string<->int, char<->int,	Type coercions and utility libraries

Table 5.2: Compilation of *let rec f n = if n = 0 then 1 else n*f(n-1) in f 6*

```
open Arch

let _camlFact__dump = ([(" camlFact", [Eint(0x00000000n);
Eglobal_symbol("_camlFact");
Edefine_symbol("_camlFact");
Eskip(0)]);
("_camlFact__1", [Eint(0x000008F7n);
Edefine_symbol(" camlFact__1");
Esymbol_address("_camlFact__f_1008");
Eint(0x0000003n)])]
([Elabdef(Elab("_camlFact__code_begin"));
Efundecl(8, _camlFact__f_1008, true);
Elabdef(Elab(" camlFact__f_1008", 2000001));
Econd(Ecompare (Esigned (Ene), Eregload(0), Eintconst32(11)), Egoto(Elabel("_camlFact__f_1008", 100)));
Ereloadretaddr(8, Eregstore(14));
Ecopy(Eshiftconst32(false, 31, 0), Eregstore(0));
Ereturn(8);
Elabdef(Elabel(" camlFact__f_1008", 100));
Ecopy(Eregload(0), Estackstoreref(0));
Ecopy(Earith(Esub, Eregload(0), Eintconst32(21)), Eregstore(0));
Ecall(Econstsym(true, "_camlFact__f_1008"), [Eregload(0)]);
Eresult("", [Eregload(0)]);
Ecopy(Estackloadref(0), Eregstore(8));
Ecopy(Earith(Easr, Eregload(0), Eintconst32(1)), Eregstore(3));
Ecopy(Earith(Esub, Eregload(8), Eintconst32(1)), Eregstore(4));
Ecopy(Earith(Emul, Eregload(4), Eregload(3)), Eregstore(5));
Ereloadretaddr(8, Eregstore(14));
Ecopy(Earith(Eadd, Eregload(5), Eintconst32(1)), Eregstore(0));
Ereturn(8);
Eadj(0);
Efundecl(8, _camlFact__entry, true);
Ecopy(Econstsym(false," _camlFact__1"), Eregstore(0));
Ecopy(Eshiftconst32(false, 131, 0), Eregstore(0));
Ecall(Econstsym(true, "_camlFact__f_1008"), [Eregload(0)]);
Eresult("" []);
Ecopy(Eshiftconst32(false, 11, 0), Eregstore(0));
Ereloadretaddr(8, Eregstore(14));
Ereturn(8);
Eadj(0);
Elabdef(Elab("_camlFact__code_end"));
Elabdef("camlFact__code_end");
Elabdef(Elab("_camlFact__data_begin"));
Elabdef(Elab("_camlFact__data_end"));
Elabdef(Elab("_camlFact__frametable")])]));;
```

5.4 Eliminating the type-unsafe layer

A major objective of the thesis, as presented here, is to avoid unnecessary reliance on type-unsafe library code (largely written in C in the conventional implementation). The major elements of support code are:

(i) operating system interfacing

(ii) file and stream I/O

(iii) garbage collection

(iv) polymorphic operator support.

The present embodiment offers only standard output streaming, but in principle can be expanded to offer:

(i) network I/O

(ii) making use of the dedicated Xilinx Ethernet media access controller

(iii) type-safe network stack ocamlnet/ocaml-tls [Mehnert and Meršinjak, 2014]

(iv) Mirage [Madhavapeddy et al., 2013].

The whole issue of garbage collection algorithms is more flexibly handled when custom hardware support is available, and will be discussed later. The polymorphic operator support requires further discussion, and some advantages and disadvantages are summarised in the Table 5.3.

In addition to the above, all the polymorphic techniques make garbage collection more complicated, because of the need to scan possibly variable sized fields. Efficiency is reduced, because the type of data item has to be determined before the correct comparison routine may be called. The requirement for both List.mem and List.memq, in the standard library, is perhaps a hint that not all is well. For this, and other reasons, it was decided not to support polymorphic primitives in this subset. If it is desired to make use of list membership, the alternative procedure is to pass a comparison function, along with the usual arguments, to determine membership or not. Likewise, with association lists, the comparison function needs to receive the same type as the list key. In an embedded environment, this gives more control over exactly which fields in a sorting key are significant, but without placing a huge burden on converting existing code to the subset. If an application makes use of hash tables exactly the same principles apply, with the addition of a user-supplied hash function being required. In

Table 5.3: Summary of polymorphic support in OCaml

Feature	Advantages	Disadvantages
Generic arrays	Sorting floats/ints in one routine	Runtime inspection of tag field needed to index array
Polymorphic equality	Enables generic hashing and matching	Possibility of cycles
Generic compare	Implicit in List.memq	Limited control of comparison keys
Generic hash function	Generic Hashtbl.t	Potentially implementation specific

Table 5.4: External functions in the embedded subset

Name	Purpose	Comment
caml_create_string	Create a new string on the heap	Frequently used in string manipulation
caml_ml_open_descriptor_in	Associate a stream with stdin	To facilitate native compilation
caml_ml_open_descriptor_out	Associate a stream with stdout	To facilitate native compilation
caml_ml_flush	Flush output stream	To facilitate native compilation
caml_ml_input_char	Read a character from stdin	Only keyboard input supported
caml_ml_output_char	Output a character to stdout	Only screen output supported
caml_make_vect	Create a new array on the heap	Allocates a contiguous chunk of objects
caml_sys_exit	Exit the program	To support native simulation

theory, the experienced embedded programmer should have a better idea what constitutes a good hash function for a given dataset, than a generic library support layer. Any such function is likely to violate type safety, since it takes an arbitrary function in, and returns an integer. However, from a code review point of view, it may be better to have such things in plain sight, rather than hidden away in a large, complicated run-time support library. After investigation of all the above, a few external functions remain which cannot be converted to native OCaml code (refer to Table 5.4). The majority of these are present just to maintain a level of compatibility with native compilation, and have little or no meaning in this embedded context. A more powerful embodiment would not necessarily require many more primitives. The use of type-safety allows traditional operating system functions to be integrated into the application, without compromising robustness [Madhavapeddy et al., 2010].

For embedded process control, general purpose I/O and analogue<->digital channels, would also be needed. These do not introduce further complexity; they can be made part of the I/O axiom in Coq, and can be considered no different from console stdin/stdout in the OCaml environment. Most embedded systems require timers that operate on a regular schedule. This is trivial to provide in hardware, to ensure that analogue channels receive regular, undistorted data. Out of the possible implementations, a busy wait loop is usually the worst option unless the time delay is very short. Bearing in mind the exception mechanism semantics, it is not a natural extension to provide a timer interrupt as a kind of hardware exception, largely because this involves discarding stack frames, and the desired behaviour is to return to the previous thread of execution, following the end of the timer interrupt. A messaging system would allow pending timer events to be removed from a queue, on a regular schedule, and this would be the preferred route, if the architecture can stand the latency, as it much easier to demonstrate correctness locally, rather than globally (in the event of every possible instruction being interrupted). An adaptation to the Amber core to provide multiple sets of registers, is an obvious alternative. It would still be necessary to provide safe FIFO structures that can be modified and read simultaneously.

5.5 Extracted Compilation Example

The Coq example, of section 4.5, may be extracted to the following code:

```
open BinInt
open BinNums
open Datatypes

type id =
  nat
```

```
let rec beq_id n m =
  match n with
  | O ->
    (match m with
     | O -> true
     | S n0 -> false)
  | S n1 ->
    (match m with
     | O -> false
     | S m1 -> beq_id n1 m1)

type ty =
| Coq_ty_arrow of ty * ty
| Coq_ty_Nat

type tm =
| Coq_tm_var of id
| Coq_tm_app of tm * tm
| Coq_tm_abs of id * ty * tm
| Coq_tm_nat of coq_Z
| Coq_tm_pred of tm
| Coq_tm_mult of tm * tm
| Coq_tm_if0 of tm * tm * tm
| Coq_tm_fix of tm

let rec subst x s t = match t with
| Coq_tm_var y -> if beq_id x y then s else t
| Coq_tm_app (t1, t2) -> Coq_tm_app ((subst x s t1), (subst x s t2))
| Coq_tm_abs (y, t0, t1) ->
  Coq_tm_abs (y, t0, (if beq_id x y then t1 else subst x s t1))
| Coq_tm_nat n -> Coq_tm_nat n
| Coq_tm_pred t0 -> Coq_tm_pred (subst x s t0)
| Coq_tm_mult (t1, t2) -> Coq_tm_mult ((subst x s t1), (subst x s t2))
| Coq_tm_if0 (t1, t2, t3) ->
  Coq_tm_if0 ((subst x s t1), (subst x s t2), (subst x s t3))
| Coq_tm_fix t0 -> Coq_tm_fix (subst x s t0)

let rec redvalue = function
| Coq_tm_abs (x, t11, t12) -> true
| Coq_tm_nat n -> true
| _ -> false

let rec reduce = function
| Coq_tm_app (a, b) ->
  (match a with
   | Coq_tm_abs (x, ty0, t12) ->
     if redvalue b
     then subst x b t12
     else Coq_tm_app ((reduce a), (reduce b))
   | _ -> Coq_tm_app ((reduce a), (reduce b)))
| Coq_tm_abs (a, b, c) -> Coq_tm_abs (a, b, (reduce c))
| Coq_tm_pred v ->
  (match v with
   | Coq_tm_nat n -> Coq_tm_nat (Z.pred n)
   | _ -> Coq_tm_pred (reduce v))
| Coq_tm_mult (a, b) ->
  (match a with
   | Coq_tm_nat n1 ->
     (match b with
      | Coq_tm_nat n2 -> Coq_tm_nat (Z.mul n1 n2)
      | _ -> Coq_tm_mult ((reduce a), (reduce b)))
   | _ -> Coq_tm_mult ((reduce a), (reduce b)))
| Coq_tm_if0 (a, b, c) ->
  (match a with
   | Coq_tm_nat z ->
     (match z with
      | Z0 -> b
      | _ -> c)
   | _ -> Coq_tm_if0 ((reduce a), b, c))
| Coq_tm_fix v ->
  (match v with
   | Coq_tm_abs (x, t1, t2) ->
     subst x (Coq_tm_fix (Coq_tm_abs (x, t1, t2))) t2
   | _ -> Coq_tm_fix (reduce v))
| x -> x
```

```
let fact =
 Coq_tm_fix(Coq_tm_abs ((S O), (Coq_ty_arrow(Coq_ty_Nat, Coq_ty_Nat)),
 (Coq_tm_abs(O, Coq_ty_Nat, (Coq_tm_if0((Coq_tm_var O), (Coq_tm_nat
 (Zpos Coq_xH)), (Coq_tm_mult((Coq_tm_var O), (Coq_tm_app((Coq_tm_var(S
  O)), (Coq_tm_pred (Coq_tm_var O))))))))))))

let fact_calc n =
  Coq_tm_app (fact, (Coq_tm_nat n))

let rec reduce_n n t =
  match n with
  | O -> t
  | S n0 -> reduce_n n0 (reduce t)
```

(Unused functions have been removed for clarity). Each Inductive statement compiles to a type declaration, Fixpoint functions are compiled to recursive functions, and Declare statements to let statements. The only statements not mentioned are the library implementations of nat (Peano arithmetic numbers) and BinInt (arbitrary precision binary numbers, implemented as lists). For convenience, an additional library of conversions, between OCaml native integers, and Coq arithmetic types is needed for human interaction. These functions are straightforward, but it must be borne in mind that as soon as this is done, the proof is no longer valid for large numbers, exceeding the wordlength of that datatype, in any intermediate step. As shown below, a small quantity of functions, hopefully correct by construction, may be used to convert between the user's concept of a number and Coq format. If performance and memory requirements are lax, the arbitrary precision arithmetic itself can be used for conversion between an arbitrary length type, such as a character string and the Coq data type. If machine integers are not used, the internal format needs to be converted for printing or input. In this simple case, the algorithm iterates a fixed number of times (11) and for each number, reduces the lambda expression to an integer in a variable number of steps, and eventually, prints it. In this case, for the sake of simplicity, no check is made, whether the expression is irreducible. Note the use of LSL/LSR (logical shift left/right) to assist with the conversion.

```
open Mylib
open Short
open BinNums

let rec asNat = function
  | O -> Datatypes.O
  | n -> Datatypes.S (asNat(n-1))

let rec asPi = function
  | 0 -> failwith "not positive"
  | 1 -> Coq_xH
  | n -> let asp = asPi(n lsr 1) in
    if n land 1 = 1 then Coq_xI asp else Coq_xO asp

let asZi = function
  | 0 -> Z0
  | 1 -> Zpos Coq_xH
  | n -> if n < 0 then Zneg (asPi(-n)) else Zpos (asPi n)

let rec fromPi = function
  | Coq_xH -> 1
  | Coq_xO num -> (fromPi num) lsl 1
  | Coq_xI num -> ((fromPi num) lsl 1) + 1

let fromZi = function
  | Z0 -> 0
  | Zpos num -> fromPi num
```

```
  | Zneg num -> - (fromPi num)

let rec reduce_all t =
  match t with
  | Coq_tm_nat num -> num
  | oth -> reduce_all (reduce t)

let _ = for i = 0 to 10 do
  print_int i; print_char '!'; print_char ' '; print_char '=';
  let num = reduce_all(fact_calc (asZi i)) in
  print_char ' '; print_int (fromZi num); print_newline()
done
```

The output of compilation is shown in Appendix-G. It has an equivalent overall function to Table 5.2, but will differ in that arbitrary precision arithmetic is used.

On the contrary, for the majority of applications, where memory and performance are limited, it may be convenient to axiomise the arithmetic and make use of built-in arithmetic hardware directly. Certain assumptions about the algorithm can then be proved, but these proofs will not have the generality of those based on axioms of set theory and predicate logic. For example, if the assumption that a square-root function is correct is made, and the hardware turns out to be wrong, necessitating an expensive recall, the theorem prover cannot be blamed if the arithmetic was axiomised. If desired to go down this route, for performance reasons for example, the extraction script of section 4.5 needs to be modified as follows:

```
Extraction Language Ocaml.

Require Import Bool.
Require Import List.
Require Import BinNums.
Require Import BinInt.
Require Import Zdiv.
Require Import Euclid.
Require Import EqNat.
Require Import Zops.
Require Import Bool.
Require Import exampleZ.

Extract Constant BinInt.Z.add => "(+)".
Extract Constant BinInt.Z.sub => "fun n m -> (n-m)".
Extract Constant BinInt.Z.pred => "fun n -> (n-1)".
Extract Constant BinInt.Z.mul => "( * )".

Extract Inductive BinNums.positive =>
 "int" [ "(fun n -> n*2+1)" "(fun n -> n*2)" "1" ]
 "(fun bI b0 b1 n -> if n=1 then b1() else if n land 1 = 1 then bI(n lsr 1) else b0(n lsr 1))".


Extract Inductive BinNums.N => "int" [ "0" "+" ]
 "(fun f0 fpos n -> if n=0 then f0 () else fpos n)".
Extract Inductive bool => "bool" [ "true" "false" ].
Extract Inductive sumbool => "bool" [ "true" "false" ].
Extract Inductive unit => "unit" [ "()" ].
Extract Inductive list => "list" [ "[]" "(::)" ].
Extract Inductive Z => int [ "0" "+" "-" ]
 "(fun f0 fpos fneg n -> if n=0 then f0 () else if n > 0 then fpos n else fneg (-n))".

Extract Inductive nat => int [ "0" "succ" ]
 "(fun f0 fS n -> if n=0 then f0 () else fS (n-1))".

Extract Constant plus => "(+)".
Extract Constant pred => "fun n -> max 0 (n-1)".
Extract Constant minus => "fun n m -> max 0 (n-m)".
Extract Constant mult => "( * )".
Extract Inlined Constant max => max.
Extract Inlined Constant min => min.
Extract Inlined Constant beq_nat => "(fun x y -> x - y = 0)".
Extract Inlined Constant EqNat.beq_nat => "(fun x y -> x - y = 0)".
Extract Inlined Constant EqNat.eq_nat_decide => "(fun x y -> x - y = 0)".
```

```
Extract Inlined Constant Peano_dec.eq_nat_dec => "(fun x y -> x - y = 0)".

Extract Constant Compare_dec.nat_compare =>
 "fun n m -> if n=m then Eq else if n<m then Lt else Gt".
Extract Inlined Constant Compare_dec.leb => "(<=)".
Extract Inlined Constant Compare_dec.le_lt_dec => "(<=)".
Extract Constant Compare_dec.lt_eq_lt_dec =>
 "fun n m -> if n>m then None else Some (n<m)".

Extract Constant Even.even_odd_dec => "fun n -> n mod 2 = 0".
Extract Constant Div2.div2 => "fun n -> n/2".

Extract Inductive Euclid.diveucl => "(int * int)" [ "" ].
Extract Constant Euclid.eucl_dev => "fun n m -> (m/n, m mod n)".
Extract Constant Euclid.quotient => "fun n m -> m/n".
Extract Constant Euclid.modulo => "fun n m -> m mod n".

Extraction Blacklist Lambda String List.

Recursive Extraction Library Zops.
Recursive Extraction Library Zdiv.
Recursive Extraction Library ZArith_base.
Recursive Extraction Library Zbool.
Recursive Extraction Library Zeven.
Recursive Extraction Library Bool.
Recursive Extraction Library exampleZ.
```

Each of these statements performs a translation between a certain Coq syntax, and the corresponding OCaml syntax. Constant translations always make the same substitution; this is suitable for mapping to an OCaml operator. Inductive translations make a selection, based on the inductive type, defined in the library. In this case, it is required to translate between Coq arbitrary precision arithmetic and OCaml native arithmetic. Use may be made of the fact that the most significant bit in a natural number is always one (true for Coq positive representation). These calculations are not valid if the number is greater than the wordlength of the machine. As an example, considering the following function (taking the successor of a number):

```
(** val succ : coq_N -> coq_N **)

let succ = function
| N0 -> Npos Coq_xH
| Npos p -> Npos (Pos.succ p)
```

under this extraction regime, the following code will be produced:

```
(** val succ : int -> int **)

let succ n =
  (fun f0 fpos n -> if n=0 then f0 () else fpos n)
    (fun _ -> +
    1)
    (fun p -> +
    (Pos.succ p))
    n
```

This extraction is not optimum, but it has the capability to convert arbitrary length numbers in coq_N format to OCaml machine integers. Note that this syntax would be used infrequently, only when Coq emits a constant in its internal format. The majority of operations under this regime would use the native OCaml succ function. This function refers to a definition of succ for positive numbers:

```
(** val succ : positive -> positive **)

let rec succ = function
| Coq_xI p -> Coq_x0 (succ p)
| Coq_x0 p -> Coq_xI p
```

Table 5.5: Influence of extraction algorithm on performance/size in bytes

Option	Code size	Global size	Run time	Heap Use	*Thumb code*
Coq arithmetic	182894	10220	4075985	28216	119957
machine arithmetic	81234	2232	3560849	27648	73313

```
| Coq_xH -> Coq_xO Coq_xH
```

which becomes, after translation:

```
(** val succ : int -> int **)

let rec succ x =
 (fun bI bO b1 n -> if n=1 then b1() else if n land 1 = 1 then bI(n lsr 1) else bO(n lsr 1))


    (fun p -> (fun n -> n*2)
    (succ p))
    (fun p -> (fun n -> n*2+1)
    p)
    (fun _ -> (fun n -> n*2)
    1)
    x
```

The intermediate code of Appendix-G represents the above algorithm translated into one of the intermediate languages of the OCaml compiler. At this point, closure conversion, and uncurrying has been done, as well as register allocation. As such, it is not straightforward to understand, partly because higher order function names and locations have been collapsed to a generic assembly language format, to avoid symbol name conflicts. This syntax has already been optimised for a 16-register machine, and a style of constant definition that favours short words, together with shifts, instead of arbitrary numeric operands.

The linked output of this intermediate code, discussed in the next chapter, is capable of being simulated as behavioural Verilog. It is also possible to convert directly to binary code that may be executed by an Amber [Santifort, 2013] derived architecture. At this time, no formal proof of equivalence exists. To produce such a proof, the large step semantics of the compiler would have to be embedded in Coq along with the processor model to the desired degree of detail, in a similar manner to the ARM-6 model in HOL [Parshin, 2004]. In the absence of any such proof, a useful alternative is parallel simulation that can quickly detect any divergence, because the simulations should remain in lockstep throughout. No temporal abstraction, or stuttering equivalence, is taken advantage of in this implementation. The behavioural simulation does not require this particular behaviour and would be capable of considerable optimisation at the Verilog level. However, any such changes would militate against operator sharing, and the intent is to support programs of arbitrary complexity, such as embedded controllers for machinery, where thousands of separate operations would be needed, if there were no processor present.

Simulatable output is, in general, voluminous, since all operations are explicit. A fragment of the simulatable output is shown in Appendix-H, corresponding to the functions *subst*, in the listing above.

The use of machine arithmetic, not surprisingly, has a dramatic impact on code size and, in this case, a smaller impact on performance, as can be seen in Table 5.5.

For comparison, the same code compiled for native Thumb on Linux, is shown. Surprisingly, the benefit of Thumb does not pull away significantly until the program gets quite large. This is due to the reduced amount of optimisation so far invested in OCaml code generation for Thumb/Linux systems (relative to the ARM realview developer suite [ARM, 2007]), and the large amount of library code that dominates the footprint up to a certain size, together with the run-time overheads of workstation OCaml. The redundantly large instruction word of this thesis would be expected to be typically four times larger for programs where the application size dominates. These figures are for dynamic executables; static executables would be considerably larger owing to the size of the operating system interface layer.

5.6 Summary

The executable semantics from Coq have been compiled into a format that is compatible with hardware execution. In particular, closures have been converted to regular function calls, and arbitrary recursion handled by the use of stack and heap pointers. In chapter 6, the method of conversion to FPGA is demonstrated (in a format compatible with suitable Xilinx tools).

"You are in a maze of twisty little passages, all alike"

Willie Crowther and Don Woods

6

Conversion and compilation to FPGA

The purpose of this chapter is to explain how separately compiled modules of functional code may be merged together in a type-safe manner and output as a Verilog program in behavioural or state-machine style. The majority of the modules would typically come from a theorem prover and consequently will have a guarantee of termination and some notion of correctness, as determined by the user's own assertions. Low-level libraries and top-level looping constructs cannot be achieved this way and must be written manually.

The previous chapter showed how a Coq description could eventually be converted to OCaml object code, with all necessary primitives supported. If mere simulation of the intended behaviour is required, it would be sufficient to use the workstation's own OCaml compiler, in conjunction with the simplified libraries mentioned above. However, the protected environment of a workstation does not lend itself to interfacing directly to hardware. Scheduling delays and multi-tasking will subtly modify the behaviour (which may or may not matter), but this is undesirable in an embedded apparatus. In this chapter, the question of how to format the compiler output, in such a way that it is acceptable to subsequent proprietary tools is considered, together with the eventual goal of correct operation on the hardware platform. As an embedded example, a VGA display makes a convenient debugging device, however in a real system, it would be more likely to be an optional extra that is plugged in by a field-application engineer. A second question to be answered is whether it is worthwhile to optimise the combined module, to take advantage of constant propagation for address constants, since the vast majority of higher-order functions take constant function parameters.

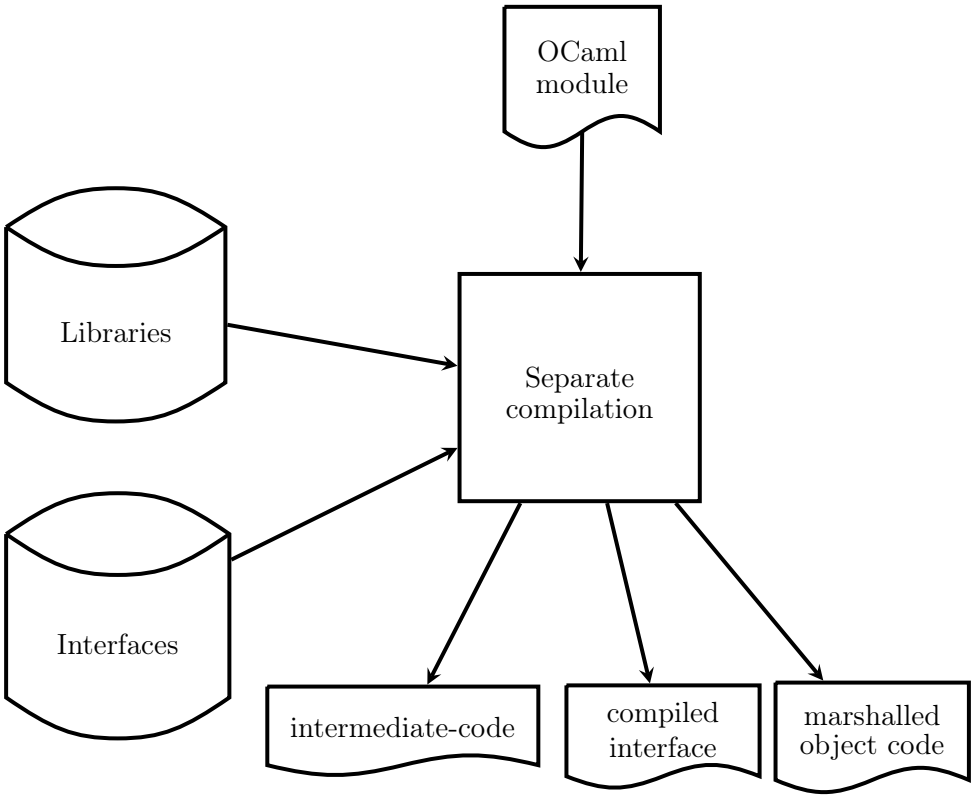


Figure 6.1: Separate compilation flow

6.1 The compilation and linking process

The compiler frontend proceeds conventionally, except that the assembly language phase is replaced by intermediate code that is OCaml-specific, as shown in Fig 6.1. The compiler backend described in the previous chapter generates behavioural Verilog that makes a suitable platform for simulation (as shown in Fig 6.2). Due to congestion and resource limitations, this needs to be converted to an explicit state machine to have any hope of passing through the resource allocation (MAP [Xilinx, 2013]), and routing (PAR) stages of the FPGA compilation process. The core of the hardware state-machine is based on the execution unit from the Amber project (a patent-free, cleanroom implementation of the ARMV2a architecture [Santifort, 2013]). The modified Verilog module is illustrated in Figure 9.3. No instruction set is necessary as such; instead the control store contents are dynamically generated from the compiler backend, in parallel with the behavioural Verilog. The resulting ROM will be far from optimal for code density, but is convenient for demonstration of the principle. A future refinement would be to encode the *don't care* space of the execution unit, and produce an instruction set that is optimally coded for the distribution of operations which occur in OCaml. This flexibility is available, because the entire process is captured within the compiler backend. It would not be possible if conventional tools were used for separate compilation, assembly, linking, and then conversion to ROM, followed by synthesis. An attractive benefit of the approach is that there is minimal reliance on type-unsafe tools in the toolchain. The FPGA vendor toolchain is opaque from this point of view: however once a valid bitstream has been obtained, it is a straightforward matter to replace the software content on-the-fly, and have a new program running on the FPGA platform within 5 minutes.

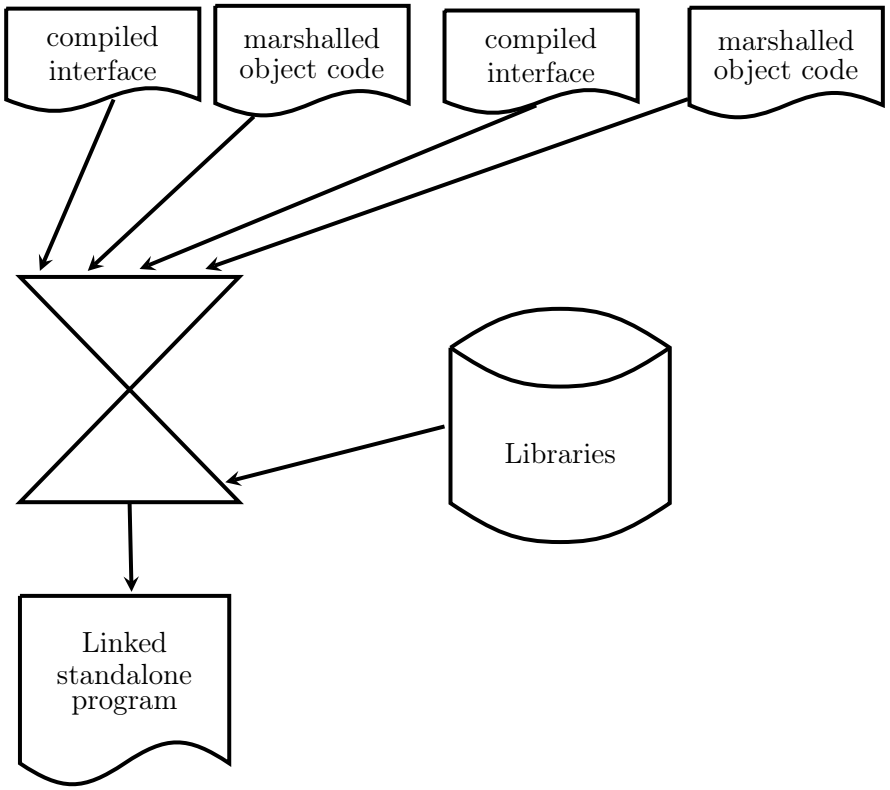


Figure 6.2: ROM linkage flow

6.2 Formatting the control store

It is highly desirable, for efficiency reasons, to allow an existing FPGA bitstream to be updated with new software. Clearly, the new software must be the same size *mutatis mutandis*, or smaller than the existing memory footprint. A limitation of the Xilinx tools is that the memory contents have to be formatted in hex, and consequently are not dynamically updateable, unless the width of the control store is a multiple of a nibble (4 bits). Accordingly, the following encoding is achieved, as shown in Table 6.1. All field widths are fixed, apart from the length of the program counter. However, in practice to allow software to be replaced dynamically, it is preferable to fix the program counter width at the maximum addressable storage (16 bits for internal ROM, up to 26-bits for external DDR memory).

Table 6.1: Control store format

Width	Name	Description
4	a23_ccode	OCaml condition code
1	read_enable	High if memory is to be read
8	imm8	An immediate value of up to 8-bits used in the instruction
5	imm_shift_amount	A left shift of up to 8 bits applied to imm8
4	rm_sel	The right-hand register for an ALU arithmetic operation
4	rds_sel	The source register for a data store or shift amount
4	rn_sel	The left-hand register for an ALU arithmetic operation and address arithmetic
2	barrel_shift_amount_sel	Choose a shift of zero, reg s, imm_shift_amount, or address low 2 bits
2	barrel_shift_data_sel	Choose the data to be shifted as imm8, read_data, reg m, or pc_nxt
2	barrel_shift_function	Select logical shift left, logical shift right, arithmetic shift right or rotate right
7	alu_function	Choose b input,add,zero ext 16/8, sext 16/8, xor, or, and. High bits are carry,not b,swap
3	address_sel	Choose pc+4,alu_out,vector,pc,reg n,addr+4,alu+4,reg n+4,reg rn-4
3	pc_sel	Choose next pc from pc+4,alu_out,vector,pc_nxt,r14,condition branch,reg m
pc_len	pc_nxt	Destination pc for branches
2	byte_enable_sel	Choose byte addressing instead of word
2	reg_write_sel	Choose to write alu_out, multiply, ext_input, or boolean comparison
1	write_data_wen	Memory write enable
1	copro_write_data_wen	Output port write enable
5	oreg_sel	Low nibble is output register select, high bit is write register enable

6.3 Embedded startup and Global memory

OCaml is an eager language, and as such, all modules linked into an application will be initialised at startup, whether used or not. In addition to the control store, OCaml requires a global memory to contain pointers to functions and static data, such as strings. Statically-allocated objects would also come into this category. For embedded use, it is useful to distinguish between objects which are deferred, because only the interface is known at compile time, whereas at link time the object is fully defined, and those objects that are dynamically generated at initialisation (or later), by the application of a function (for example List.map), to an existing constant object. Such initialisations may be of several types: the most common will be giving an initial value to static constructors such as ref types. Another common type would be a function such as a map to convert one type of constructor to another. A third type would be a structure initialisation that would require a number of nested initialisations, followed by a block copy. The final example would be launching the main function. Typically, this will be last in the link order and consequently latest to be called. This is important, because in an embedded application, the main program might never exit. The module system requires the order of linking to match the dependency order. Otherwise, an error occurs. Circular references (deprecated) may be handled using forward references provided the types are in scope.

After identifying the type of entry function, it will be possible to see if the function performs constant propagation for address constants, and if so, initialisations may be further sub-divided into mutable and immutable. Typically, the constant portion will be significantly larger than the non-constant portion. Immutable constants may be stored in ROM, providing a further sanity check against bad program behaviour, by providing the facility, if wanted, to prevent immutable storage from being overwritten. This also results in a faster startup time which is important for many embedded applications. This optimisation needs to be handled when all symbols are resolved. In the ML605 demonstrator, there is sufficient internal memory to allow a megabyte of stack/heap, including 64K of constant memory. In the present embodiment, the architecture would be described as Harvard since the data space is unified but program space is separate. This allows single-cycle operation for most instructions, but for reading data memory, it is desirable to make use of a delay slot, to allow both memories to operate on the same edge of the clock. This facilitates operation at the FPGA reference clock of 33MHz, a modest but respectable figure for FPGA technology. These functions include single cycle multiply capability, using parallel DSP48 dedicated modules in the FPGA. The delay slot is automatically inserted by the compiler back end, as necessary.

6.4 Parallel simulation and black-box testing

During development of the state machine architecture, it is important to have confidence that the behavioural Verilog output (considered the master, since it relates directly to the backend imperative register model), is always in lockstep with the state machine architecture. This is relatively easy to achieve since both formats come from the same tool. A dual testbench with cycle-by-cycle error checking of internal register contents demonstrates the equivalence to a

high degree of coverage, given the lack of combinational depth present in the control paths. Both formats are syntactically acceptable to a synthesis tool, but the behavioural output may not be used since its degree of sharing is not explicit in the architecture. Black box testing of known programs whose output has been captured in native compilation may also be used as a comparison point. The establishment of a good regression set for any backend maintenance will mean that regular users do not need to concern themselves with the behavioural output

6.5 Simulation Example

The factorial example intermediate code of section 5.5, following linking, as explained in this chapter, may be converted to behavioural Verilog, or directly to binary, for use with a modified Amber architecture. In the presence of detailed, arbitrary-precision numeric libraries, such programs will be large (refer to Table 5.5). A fragment of code just for the *subst* function is given in Appendix-H. This is a complete, operational example, that has the block diagram shown in Figure 6.3, and the detailed internals of the *standalone* block may be seen in Appendix-I.

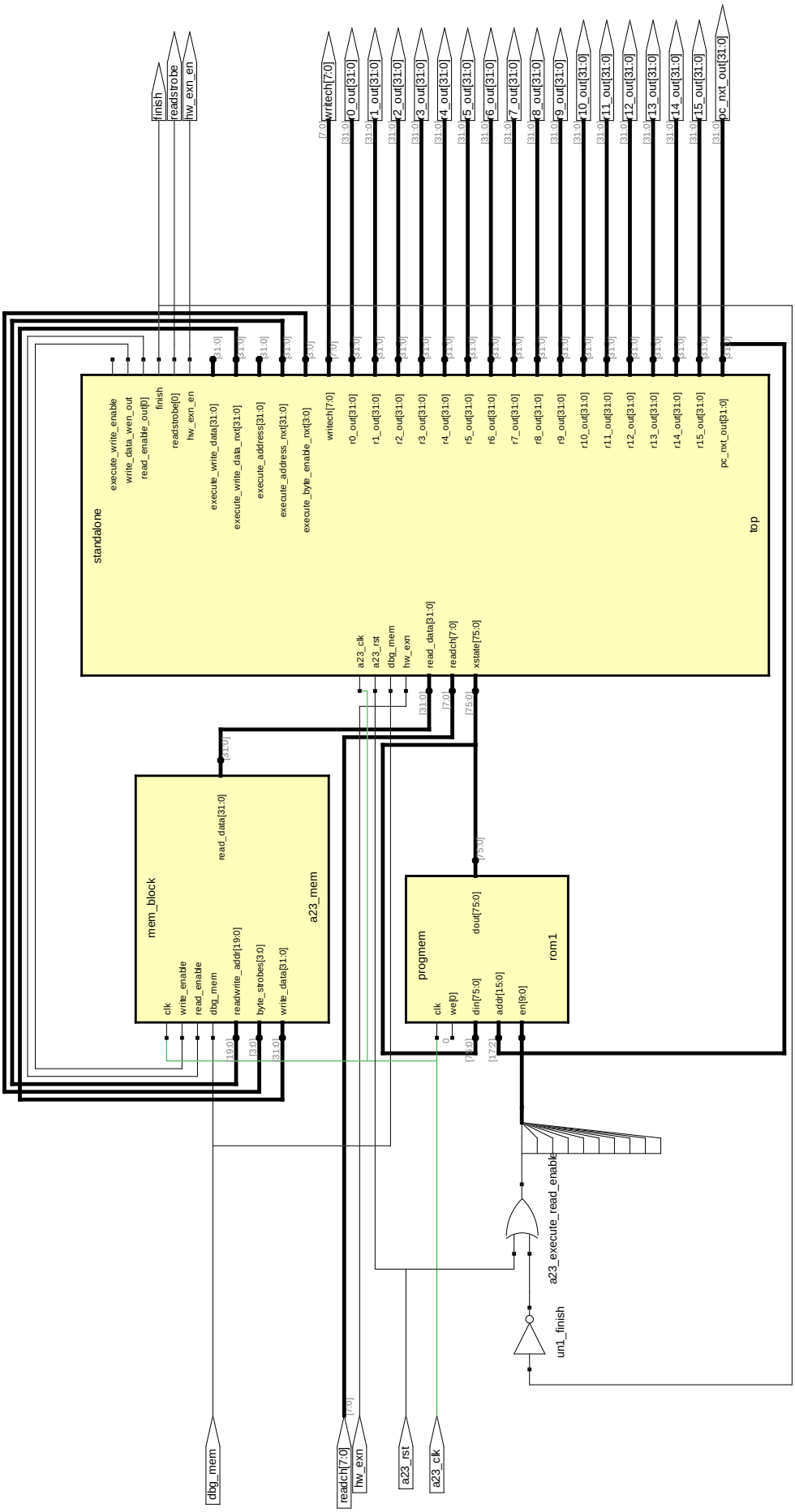


Figure 6.3: Block diagram of Amber-derived memory architecture

6.6 Summary

It has been seen that, the combination of fixed and variable elements makes an FPGA version of the executable semantics. In chapter 7, the appropriate methods of preventing the hardware from leaving its valid execution path are discussed and compared. If it is desired to obtain a guarantee of semantic equivalence between the RTL and the bitstream ready database, various proprietary tools (such as Cadence Conformal LEC [Seligman and Sokolover, 2006]) can be used, along with a register equivalence list, which requires certain optimisations to be suppressed. Clearly, the description presented here is relative to a simplified model of the true FPGA design flow. This is an over-simplification with respect to certain equivalence artefacts:

(i) Flip-flop duplication introduces logical redundancy that, in the event of a glitch, can cause different parts of the circuitry to be exercised, outside of the scope of the RTL model.

(ii) Using logic blocks as routing feed-throughs can introduce, or remove, timing hazards operating below the cycle level of the model.

(iii) Routing switch boxes incorporate pass transistors that do not provide gain, so can amplify signal integrity issues.

(iv) Xilinx chips are typically enormous, compared to the equivalent size of the circuitry that is being emulated. Therefore, some on-chip variation (for example in transistor strength) is inevitable.

"HAL: Let me put it this way, Mr. Amor. The 9000 series is the most reliable computer ever made. No 9000 computer has ever made a mistake or distorted information. We are all, by any practical definition of the words, foolproof and incapable of error."

Arthur C Clarke

7

Comparison of triple-modular redundancy and Dual-rail logic

This chapter is concerned with how to prevent a previously proven program from leaving its virtuous path and ending in an inconclusive, typically non-halting state. At this level of detail, no attempt is made to detect logically incorrect programs, the emphasis is on detecting situations where the electronics does not accurately model the underlying mathematical specification. With each method (apart from the plain or null method) the probability of detection is statistical rather than certain. Nevertheless in a harsh environment the benefits from any of the methods will be substantial over the plain or null method.

In chapter 6, the method of generating an FPGA version of an executable semantics was demonstrated. In the present chapter, the problem of the logic departing from the correct trajectory, into a mode that is not allowed by the appropriate assertions, is considered. Due to **the halting problem**, any such departure is likely to be fatal, and will invalidate the aforementioned theorems. Logical redundancy, introduced to aid with routing, or meeting electrical rules, or timing, will exacerbate any problems with single-event upsets, or other vulnerabilities.

7.1 Double-rail logic for fault-tolerance

7.1.1 Conventional Approaches

The conventional approach to the problem of statistical variability, as a cause of failure through timing hazards, is to simply scale the transistors and logic primitives of the previous generation, and develop more sophisticated Monte-Carlo Spice simulation [Maxim and Gheorghe, 2001],

and noise-aware static-timing analysis [Nitta et al., 2007]. This is a valid approach, but the downside is that very little of the speed benefit of the smaller transistors will feed into the final performance, because of the large variability. A consequence of this, is that, although prices have come down, maximum clock speeds have barely improved in the last half-decade.

7.1.2 Failure statistics

The problem of logic upsets is of great importance in FPGAs, used in aerospace applications. According to Swift [Swift, Carmichael and Allen, 2008], the worst-case (1200km, 65° inclination satellite orbit) results for an XQR4VLX200 (the largest Xilinx [Quinn et al., 2007] device in the available range of 90nm geometry process) are summarised in Table 7.1. The smaller RAM cell is easier to upset than the flip-flop, but techniques such as forward error correction and scrubbing, may be used. An upset in a critical flip-flop may require a full reset, which can result in service outage, or possible loss of synchronisation with associated systems. A treatment of the effect of heavy ions on single-event upsets is available from Edmunds [Edmonds, 2000]. At ground level, or inside buildings, the reliability is dominated by natural decay events such as lead isotopes (historically used in soldering). With modern processes, reliability with respect to natural decay is increased even though memory cell sizes are decreasing (see [Lesea et al., 2005], [Lesea, 2008]).

7.1.3 Asynchronous double-rail logic

A niche technique, popular in academic circles, but not particularly prevalent commercially, is the use of asynchronous (i.e. without requiring a clock) logic making use of double rail indication of success (see for example [LaFrieda, Hill and Manohar, 2010]). Any function, after a change in its inputs, will return a signal on either of its outputs to indicate completion. The rest of the circuit always waits for a definite outcome of an earlier calculation. One disadvantage of asynchronous techniques is that the correct timing of the circuit depends, in general, on the delays in local paths, relative to the delays in global paths. Hence, it is not robust against aggressive automatic timing optimisation, which is an essential requirement of ultra deep sub-micron design(UDSM), where digital cell libraries may have hundreds of cells, of various functions, and in particular, different drive strengths.

7.1.4 The chosen approach

This chapter demonstrates an improvement in fault tolerance of systems, by utilising the ever-increasing number of transistors on the latest CMOS processes, whilst at the same time, countering the effects of UDSM statistical variability, due to atomic and quantum effects. As downward price pressures mandate smaller, thinner, components year-on-year, the lifetime of electronic devices is decreasing, even though inherent defectivity is also decreasing. The occasional fault might be acceptable in a consumer product, assuming it can be detected, and put right, with little or no user intervention. It would not be allowed in high-cost, critical infrastructure, such as transport, medicine, aerospace, or military applications. Although double-rail asynchronous techniques can be directly used in FPGA, for the purposes of this

Table 7.1: XQR4VLX200 upsets

Event	Frequency
Configuration upset	65.1/day
block RAM upset	13.9/day
flip-flop upset	0.8/day
functional interruption	0.09/year

chapter, the meaning of the signals has been adapted, as shown in Table 7.3. Correct operation is indicated by complementary signals, incorrect operation by common-mode signals. The overall scheme is then incorporated into a timing-driven flow, and then demonstrated in FPGA. The method is illustrated in conjunction with the y86 educational processor [Bryant and Hallaron, 2011].

7.2 Methodology

7.2.1 Conventional Methodology

A typical FPGA flow, such as that provided by Xilinx [Xilinx, 2009b], consists of steps that are superficially similar to an application-specific integrated circuit(ASIC) flow, namely synthesis, mapping, placement, routing and design rule checking. In this context, the synthesis stage has an open API, the mapping stage is semi-proprietary, and the remaining steps are fully proprietary. If Xilinx synthesis technology(XST) is used, the input to the whole process will be VHDL [Delgado Kloos and Breuer, 1995] or Verilog HDL [Gordon, 1995], and the remaining steps are opaque. If third-party synthesis is used, then the input to the mapping process must be in electronic design interchange format [Stanford and Mancuso, 1989] (EDIF).

7.2.2 Description of the proposed new approach

In the proposed new approach, the fault-tolerance is introduced between synthesis and mapping. There are several reasons for this choice. Firstly, the format of the gate-level netlist is easier to process, because there are fewer alternatives to consider. Secondly, it is highly desirable to be able to debug the process of introduction of fault-tolerance, and a number of effective Verilog simulators are available. Thirdly, it is very easy to convert gate-level Verilog to EDIF as required by the mapping phase. Fourthly, the uniformity of the gate-level description reduces the number of fault-tolerant cells that will have to be specially designed.

7.2.3 Detailed discussion of the conversion to fault-tolerance

The custom flow sits in the overall flow as shown in Fig 7.1. The entirety of the custom flow, apart from certain libraries listed below, is written in OCaml, in order to provide type-safety in conjunction with efficiency and economy of expression. Code size statistics are mentioned in Table 7.2. Except where referenced, the code was created from scratch by the present author, apart from the Verilog grammar, which came from Verilator [Snyder, 2010] (originally written in *bison* [Donnelly and Stallman, 2006], using Backus-Naur formalism [Scowen, 1998]), and was

Table 7.2: Custom flow code statistics

Module	Line Count
Verilog Grammar	1950
Semantic Checks	2000
BDD package	600
Recognising Library	525
Flattening	450
Double Rail Conversion	100
Optimisation	1050
ABC wrapper	350
Edif Output	300

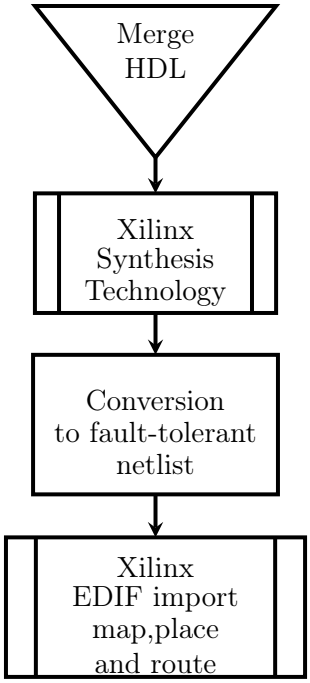


Figure 7.1: Modified FPGA flow

adapted to ocamlyacc format. As expected, the reduction in size and improved maintainability, due to the increased level of abstraction in the parser, was dramatic.

7.2.3.1 Optimistic and pessimistic gates

The internals of a dual-rail gate require clarification; referring to Table 7.3 again, it is apparent that the definition of logic gate is not unique. The purpose of stuck indications is not only to detect internal inconsistency, but also to forward a fault indication from a logic device's fan-in cone. If the gate forwards any fault at any input, the design will be optimum for fault detection, so can be described as a *pessimistic* gate. It can be shown by simple fault simulation that this logic is optimum for detecting faults. However, there is a problem with initialisation of the circuit, should it consist entirely of this kind of gate. Typically, it will power on in the unknown state, approximately half of the flip-flops will start in the fault state, which is not a useful model of the desired function. In the fan-out cone of the reset input to the design, the desired behaviour is that the reset indication will dominate over the unknown indication. Therefore, in order to reset to a known state, the idea of an *optimistic* gate is introduced, which during reset will convert any state into a valid state. This naïve version of the flow

Table 7.3: Double rail logic encoding

State	Encoding
00	stuck low
01	logic low
10	logic high
11	stuck high

Table 7.4: Library recognition and flattening intermediate code

```
library_env /home/arucad/Xilinx/13.1/ISE_DS/ISE/verilog/src/unisims/
scan_library
vparse /home/arucad/Xilinx/13.1/ISE_DS/ISE/verilog/hdlMacro/AND8.v
vparse /home/arucad/Xilinx/13.1/ISE_DS/ISE/verilog/hdlMacro/OR8.v
read_library
vparse processor_timesim.v
gen_flat_arch verilog processor
write_arch flat processor
quit
```

Table 7.5: Output from the library recognition stage

```
883 library cells detected
113 non-inverting buffers detected
Using library buffer BUF B1(.O(out),.I(in));
1 inverting buffers detected
Using library inverter INV N1(.O(out),.I(in));
1 power sources detected
Using library power source VCC (.P(out));
1 ground sources detected
Using library ground source GND (.G(out));
1 tri-state buffers detected
Using library tri-state buffer BUFE B1(.O(out),.E(in),.I(en));
```

does not have access to synthesis data structures so it is not easy to generate the reset fan-out cone. Therefore, the remainder of the discussion will consist of logic networks, made only with optimistic gates. It is necessary to have suitable reset structures in RTL, to force initialisation of critical flip-flops. This will be true anyway, for well-conditioned designs.

7.2.3.2 Recognition of the library primitives

It is necessary to recognise when the flow synthesis reaches a leaf cell (also known as a library primitive), in order to prevent over-flattening of the netlist. The Verilog syntax ‘celldefine is available for this purpose, however Xilinx libraries do not use it. Instead, the current approach is to use the intermediate code in Table 7.4. The operations are explained as follows. Firstly, the location of the libraries is defined. The library directory is then scanned to identify suitable leaf cells. Any special cells used in the design are then manually added in. Finally, the library is constructed, with the results shown in Table 7.5. At this stage, the special cells needed for flattening, and for EDIF output, are identified. The identification is made by an analysis of the underlying function of the Verilog. This final stage marks every cell successfully identified as a library primitive.

7.2.3.3 The Flattening process

The input to the flattening stage is the synthesised output of the synthesis, typically done by XST. In Table 7.4, the Verilog file processor_timesim.v would be the output from running XST on the module processor. Alternative front-ends, as discussed in section 2.2, could also be used, depending on the source syntax. In general, the output format will be hierarchical. It is conceptually easy to convert to a flat netlist but there are a number of subtle issues. Flattening the Verilog netlist (represented internally as a hash table), takes place by means of a standard recursive descent algorithm. It is likely that the synthesis process will have introduced feed-throughs, assigns or partial busses that cause problems for an optimum approach. Instead, a context-free algorithm is used, which places a buffer of appropriate direction on every hierarchical boundary. A later optimisation stage is relied on to remove the redundant elements. Flattening stops when a previously identified library cell is encountered, or a sub-circuit that is identified as behavioural (such as a block-RAM).

7.2.3.4 Conversion to double-rail logic

The input to the conversion to double-rail logic phase is the flat netlist internally generated by the flattening phase. Only a few primitives will be present. The fault-tolerant version of each primitive is assumed to have been hand-designed, or otherwise previously generated. Assuming there exists a primitive AND2(.O(Y), .I0(A), .I1(B)) (which trivially represents the boolean function Y = A.B). This example is then mechanically converted into F_AND2(.O({F_Y,Y}), .I0({F_A,A}), .I1({F_B,B})) where F_Y is the complement of Y, any other value represents a fault. It is apparent that the fault-tolerant netlist will be considerably larger than the assumed good netlist. However, the requirement for inversions is virtually eliminated, because AND,OR,NAND,NOR gates are freely available just by permuting the inputs and/or outputs. Clearly, only detection of errors has been discussed, not correction. Therefore, the new logic would have to be used inside a higher-level protocol, such as the exception logic of section 9.5.

By contrast, Triple-modular redundancy [Miller and Carmichael, 2008] (TMR) is good at masking faults, but not so good at testability, since the user is not made aware of faults when they happen.

7.2.3.5 Optimisation

After conversion and flattening, there will be a degree of unwanted logic, due to the naïve nature of the flattening process and due to chains of logic that have been individually converted. For efficiency of verification, it is essential to optimise the netlist before entering the opaque stage of the Xilinx flow, or the putative ASIC flow. Since gate-level optimisation algorithms are non-trivial, the ABC library of mapping routines is used. In this library, the optimisation flow itself consists of a number of stages, including conversion of the input to a network, transformation into logic, conversion into AIG form, and finally mapping. In the ABC flow, hierarchical netlists are deprecated, thus requiring another stage of flattening. Theoretically, the mapping library could be directly generated from the library primitives. At this stage, this has not been done because of the complex internal assumptions of the ABC library. The

Table 7.6: C code for smoke-test

```
int array[4] = {0xd, 0xc0, 0xb00, 0xa000};

int smoke_test(int *Start, int Count)
{
  int sum = 0;
  while (Count) {
    sum += *Start;
    Start++;
    Count–;
  }
  return sum;
}

int main(void)
{
  return smoke_test(array, 4);
}
```

Verilog netlist interface has a number of limitations, particularly in the area of flip-flops. A workaround is to bypass this stage, and go directly into ABC as a network. If the flip-flops are anything other than simple D-types with a global clock, the network does not represent this functionality directly, and so they are tracked separately in the main database. The result of this is that redundancies in clock, clock enables, and preset/clear networks will remain until the back-end flow. However, this is a small part of the total. The final network is restored to internal netlist format, as well as restoring the flip-flop name and control signals that will be needed later.

7.2.3.6 Output

The resulting netlist can be written as structural Verilog, for use in simulation, or as an EDIF netlist. The Verilog presents no difficulty as it is the internal format; the EDIF is another structural format, that differs chiefly in the sense that, it lists each net, with the cells that it connects to, instead of listing each cell, with the nets to which it connects. To make a valid EDIF netlist for subsequent Xilinx processing, appropriate input or output pads must be added to every primary pin. The possibility of a bidirectional pin is not catered for in this flow. If wanted, it could be added manually, using the ngdbuild feature of the Xilinx tools to merge EDIF netlists.

7.3 Results

The results of using the methodology on the y86 processor [Bryant and Hallaron, 2011] are shown below. This is a relatively low-level register-transfer level(RTL) description. This methodology lacks a way of optimising blocks such as RAMs, inside ABC. A workaround is to bring the RAM up a level of hierarchy, and implement it inside a top-level structural framework. This method allows the processor, with its dual-rail logic, to be optimised down to gate-level primitives, without having to worry about maintaining bus connections at the boundaries of black boxes.

7.3.1 Preparing the pre-synthesis simulation

The pre-synthesis simulation, also known as the RTL simulation, requires the processor RTL as mentioned above, a test bench, a dual-port RAM (either behavioural or Xilinx specific) and the software to be tested. For the smoke test (that is, the trivial program that attempts to prove some function, but if it fails provides ease of debugging) the C code shown in Table 7.6 is used. On the y86, instructions and addressing modes are drastically cut down from the x86 processor series, requiring manual modification of x86 compiler output for instructions or modes that are not valid. The alternative, adopted here, is to remove the invalid instructions from the compiler register-transfer expressions. However, only a subset of C will be supported (For example the only addressing mode is 32-bits). The output of the modified compiler is shown in Table 7.7. It is apparent that the C code needs to be topped and tailed with a machine-specific initialisation. If the code download feature of the y86 inside the testbench is utilised, the execution of this program will result in the 4-word sum ABCD(hex) being left in a register. The exercise can be repeated with the same result on the output of the flow.

7.3.2 Applying the tool-chain

Applying the tool-chain described above produces a report of the form Table 7.8. The corresponding result without the double-rail logic is in Table 7.9. As expected, the number of flip-flops is doubled, the number of combinational gates will be more than doubled. This overhead will vary by application, and could be substantially more than this, as shown in section 9.4. A naïve transistor count is available by multiplying the instances of a given cell in column 5 by the transistor count of a basic CMOS implementation in column 6, with a resulting ratio of 2.9:1. These results do not include the overhead of deciding what to do when an error is detected. However, as gate densities reach beyond millions of gates per square millimetre, this is unlikely to be a cost-prohibitive factor.

7.3.2.1 FPGA results

To obtain the FPGA overhead, the Xilinx EDIF flow is run, resulting in the report files as shown in Table 7.10 and Table 7.11. In this architecture, the slice usage ratio rises by about 5:1. This is not surprising since this is a generic, not an FPGA-specific flow, so assumptions about the mapping of the logic do not necessarily carry over to LUT based architectures. In particular the LUT architecture performs poorly when many signals fan-in to a flip-flop. This will inevitably be the case when trying to detect logic faults coming in from a wide fan-in cone and pass them on to the next stage via a flip-flop pair. A configurable logic block(CLB) normally has two flip-flops. In this technique, the CLB will only hold one bit of (4-state) information instead of the usual two. However, in most designs, the size is dominated by on-chip memory. The user can choose whether to replicate the double-rail feature in RAM or not. To reduce overhead, using the parity feature or error correcting code (a class of codes for parallel implementation in memories) (ECC) would be preferable. By contrast, the overhead reported by Miller [Miller and Carmichael, 2008] using TMR is 9:2 for LUTs and 3:1 for flip-flops (excluding the processor, which was a hard macro and therefore not comparable).

Table 7.7: Compiled assembly code for smoke-test

```
/* $begin code-yso */
/* $begin code-ysa */
# Execution begins at address 0
 .pos 0
start: irmovl cstack, %esp   # Set up stack pointer
 irmovl cstack, %ebp   # Set up base pointer
 call main  # Execute main program
 halt   # Terminate program
# gcc version 3.4.6 Wed Mar  7 14:30:18 2012

# global array
# data section
 .align 2
array:
 .long 13
 .long 192
 .long 2816
 .long 40960
# text section
# global smoke_test
smoke_test:
 rrmovl %ebx, %ecx #  88 *movsi_1/1 [length = 2]
 pushl %ecx #  89 *pushsi1/1 [length = 1]
 mrmovl 8(%esp), %eax #  3 *movsi_1/3 [length = 5]
 mrmovl 12(%esp), %edx #  4 *movsi_1/3 [length = 5]
 xorl %ebx, %ebx #  87 *movsi_xor [length = 2]
L7:
 andl %edx, %edx #  55 *cmpsi_ccno_1 [length = 3]
 je L6 #  56 *jcc_1 [length = 2]
 mrmovl (%eax), %ecx #  68 *movsi_1/3 [length = 3]
 addl %ecx, %ebx #  69 *addsi_1/1 [length = 2]
 irmovl $4, %ecx #  70 *movsi_1/2 [length = 5]
 addl %ecx, %eax #  71 *addsi_1/1 [length = 2]
 irmovl $-1, %ecx #  86 *movsi_or_else [length = 3]
 addl %ecx, %edx #  73 *addsi_1/1 [length = 2]
 jmp L7 #  84 jump [length = 2]
L6:
 rrmovl %ebx, %eax #  44 *movsi_1/1 [length = 2]
 popl %ebx #  78 popsi1 [length = 1]
 ret #  79 return_internal [length = 1]
# global main
main:
 irmovl $4, %ecx #  30 *movsi_1/2 [length = 5]
 pushl %ecx #  31 *pushsi1/1 [length = 1]
 irmovl $array, %ecx #  32 *movsi_1/2 [length = 5]
 pushl %ecx #  33 *pushsi1/1 [length = 1]
 call smoke_test #  12 *call_value_0 [length = 5]
 popl %edx #  39 popsi1 [length = 1]
 popl %edx #  40 popsi1 [length = 1]
 ret #  37 return_internal [length = 1]

# The stack starts here and grows to lower addresses
 .pos 0x100
cstack:
/* $end code-ysa */
/* $end code-yso */
```

7.3.2.2 Timing

The timing tables come from the place and route (PAR) results. The reports are shown in Tables 7.12. and 7.13. Degradation is approximately 2:1 which is acceptable. Again, the latency is expected to increase, once a higher-level protocol to deal with the consequences of an error has been added.

Table 7.8: Usage report for y86 (double-rail technology)

name	ios	primary	sequence	instances	transistors	total	truth table
XOR2	3	xor		152	4	608	O = I0 and not I1 or not I0 and I1
GND	1	binnum		784	2	1568	G = 0
RAMB16_S9_S9	18	empty		8			
NOR2	3	nor		12936	4	51744	O = not I0 or I1
NOR3	4	nor		5586	6	33516	O = not I0 or I1 or I2
NOR4	5	nor		859	8	6872	O = not I0 or I1 or I2 or I3
NAND2	3	nand		1064	4	4256	O = not I0 and I1
NAND3	4	nand		255	6	1530	O = not I0 and I1 and I2
NAND4	5	nand		87	8	696	O = not I0 and I1 and I2 and I3
FDPE	5	memory	posedge if	775	12	9300	Q = q_out
MUXF5	4	empty		6	4	24	O = 0
FDCE	5	memory	posedge if	775	12	9300	Q = q_out
AND2	3	and		9496	6	56976	O = I0 and I1
AND3	4	and		1730	8	13840	O = I0 and I1 and I2
AND4	5	and		337	10	3370	O = I0 and I1 and I2 and I3
AND5	6	and		90	12	1080	O = I0 and I1 and I2 and I3 and I4
AND8	9	and		8	18	144	O = I0 and I1 and I2 and I3 and I4 and I5 and I6 and I7
VCC	1	binnum		259	2	518	P = 1
FD	3	memory	posedge	2	6	12	Q = q_out
BUF	2	buf		518	4	2072	O = I
OR2	3	or		4137	6	24822	O = I0 or I1
OR3	4	or		2183	8	17464	O = I0 or I1 or I2
OR4	5	or		364	10	3640	O = I0 or I1 or I2 or I3
OR8	9	double		84	18	1512	O = I0 or I1 or I2 or I3 or I4 or I5 or I6 or I7
INV	2	not		3790	2	7580	O = not I
					Grand Total	252444	

Table 7.9: Usage report for y86 (normal/non double-rail technology)

name	ios	primary	sequence	instances	transistors	total	truth table
XOR2	3	xor		397	4	1588	O = I0 and not I1 or not I0 and I1
GND	1	binnum		50	2	100	G = 0
RAMB16_S9_S9	18	empty		8			
FDCE	5	memory	posedge if	775	12	9300	Q = q_out
AND2	3	and		6434	6	38604	O = I0 and I1
AND3	4	and		141	8	1128	O = I0 and I1 and I2
AND4	5	and		28	10	280	O = I0 and I1 and I2 and I3
AND5	6	and		4	12	48	O = I0 and I1 and I2 and I3 and I4
AND8	9	and		15	18	270	O = I0 and I1 and I2 and I3 and I4 and I5 and I6 and I7
VCC	1	binnum		17	2	34	P = 1
FD	3	memory	posedge	1	6	6	Q = q_out
BUF	2	buf		3754	4	15016	O = I
OR2	3	or		1743	6	10458	O = I0 or I1
OR3	4	or		11	8	88	O = I0 or I1 or I2
OR4	5	or		49	10	490	O = I0 or I1 or I2 or I3
OR8	9	double		150	18	2700	O = I0 or I1 or I2 or I3 or I4 or I5 or I6 or I7
INV	2	not		3665	2	7330	O = not I
					Grand Total	87440	

Table 7.10: Y86 device summary (double-rail logic)

```
Device Utilization Summary:

Number of External IOBs                539 out of 640    84%
   Number of LOCed IOBs                539 out of 539   100%

Number of RAMB18X2s                      6 out of 148     4%
Number of Slices                      8055 out of 17280  46%
Number of Slice Registers             1552 out of 69120   2%
   Number used as Flip Flops          1552
   Number used as Latches                0
   Number used as LatchThrus             0

Number of Slice LUTS                 18506 out of 69120  26%
Number of Slice LUT-Flip Flop pairs  18506 out of 69120  26%
```

Table 7.11: Y86 device summary (normal logic)

```
Device Utilization Summary:

Number of External IOBs                539 out of 640    84%
   Number of LOCed IOBs                539 out of 539   100%

Number of RAMB18X2s                      6 out of 148     4%
Number of Slices                      1527 out of 17280   8%
Number of Slice Registers              776 out of 69120   1%
   Number used as Flip Flops          776
   Number used as Latches               0
   Number used as LatchThrus            0

Number of Slice LUTS                  3895 out of 69120   5%
Number of Slice LUT-Flip Flop pairs   3895 out of 69120   5%
```

Table 7.12: Y86 timing (double-rail logic)

Constraint	Check	Worst Case Slack	Best Case Achievable	Timing Errors	Timing Score
Autotimespec constraint for clock net qq_clock	SETUP	N/A	72.181ns	N/A	0
	HOLD	0.056ns		0	0

Table 7.13: Y86 timing (normal logic)

Constraint	Check	Worst Case Slack	Best Case Achievable	Timing Errors	Timing Score
Autotimespec constraint for clock net qq_clock	SETUP	N/A	33.598ns	N/A	0
	HOLD	0.042ns		0	0

7.4 Benefits

It can be seen that the introduction of the current methodology carries a substantial area penalty and some speed penalty. Nevertheless, there are benefits:

7.4.1 Security

7.4.1.1 Confidence in the computation

In the early days of computing, it was assumed a computer would either produce the correct result (as defined by its programming), or crash visibly, such as the well-known blue screen of death. The intermediate scenario, where nothing visible goes wrong, but the output is not correct either, will become increasingly common, as delay faults [Chmelaf, 2003] become more difficult to detect. Using dual-rail logic, the computer positively asserts that all is well on every clock cycle, and any faults will become apparent, after a latency only determined by the sequential depth between input and output. In most designs, this value is rather low in order to achieve high throughput. The recovery mechanism would be application dependent. In an engineering or fast-moving consumer goods situation, it might be enough to just hit the reset button and start again. In an engine management system, it would make sense to switch to the backup system automatically based on the good/fault output status. In more sophisticated applications it makes more sense to work at the word level. The method of section 9.4 does work at the word level, and that is how it gets its relative efficiency.

7.4.1.2 Inscrutability

In systems such as smart cards, the security of the secret keys depends on the lack of a suitable cryptographic attack. Where access to the power line is available, as is the case for most smart cards in use, the power drain will depend on the internal logic state. If the device can be made to execute the same algorithm repeatedly, it is possible to infer the secret keys, or dramatically reduce the search space, by measuring the power consumption profile. This situation naturally arises because of the different rise and fall times of NOR and NAND gates. However, with dual-rail logic, an equal number of flip-flops are high and low, and NOR and NAND structures can be made, from the same transistor topology. Other software-based obfuscation measures are still required.

7.4.1.3 Harsh environments

In ionising radiation intensive environments, a gradual shift in the transistor threshold will occur, which will affect timing and symmetry and eventually lead to failure. To alleviate this problem, a dose of radiation can be pre-applied, under controlled conditions, to try to ensure all transistors will shift by the same amount. However, the effects of radiation dose are unpredictable in real operation, and having positive confirmation of correct operation is valuable.

7.4.2 Adaptive clocking

Since there is a definite go/no-go status at all times, the clock speed/voltage scenario can be adjusted to match the temperature and process. A more expensive system could also control temperature. All that is needed is a worst-case program to execute, which utilises the deepest combinational paths, and a model of how the delay varies with environment. Because of difficulties with global skew, clock gating is usually replaced by flip-flop enables in FPGAs, unless there are overriding power consumption reasons. In this case, extra control modes might be needed to bring the worst-case logic into operation. Calibration may be done by varying the clock rate until a failure occurs. With a good environmental model and a suitable safety factor, a high level of confidence combined with optimum relative power consumption may be obtained.

7.5 Limitations

The flow thus described only handles flat netlists. Consequently, it is unwieldy and memory intensive, when large designs are employed. The support for a full hierarchical flow is a topic for chapter 9. More investigation is needed to determine the optimum design, to minimise area overhead and maximise fault coverage. A new tool option is needed to mix double rail gates of different truth tables, to ensure proper initialisation without unnecessary masking of faults. Further research can simulate and refine the low-level architecture, for maximum fault coverage.

7.6 Comparison of different approaches

It has been seen that an OCaml bare-metal flow with or without proof of correctness in Coq will be a satisfactory platform for functional programming. In view of the research done at Brigham Young University [Carrol, 2009], it is appropriate to compare the TMR approach described there, with Duplication with compare [Johnson et al., 2008] (DWC) and the present author's approach above (previously published in [Kimmitt, Wilson and Greaves, 2012]). A hybrid approach based on DWC with smart detection was presented in [Anderson, 2010], which shall be referred to here as DWC+. Some similarities and differences are summarised in Table 7.14. Based on this summary, it can be said immediately that TMR has good robustness, especially when cascaded (Cascaded triple-modular redundancy (CTMR)) or associated with regular scrubbing. The other methods save area but potentially sacrifice throughput. If the philosophy of operation is maximum confidence in the delivered result (for example in fundamental research calculations), it is preferable to fail to give a result than to give the wrong result without realising. When pure functional programming is to be used, as recommended in this dissertation, any failure can be linked to an exception mechanism of section 9.5, and used as a way of retrying or reporting the problem. A retry could consist of running the same functions again, or retrying following scrubbing. If scrubbing is used in the middle of a live program, it will be important to identify the mask of the bit stream (that part which identifies the configuration rather than the flip-flop state or block RAM contents) and only restore the

Table 7.14: Comparison of fault-tolerance techniques

Aspect	TMR	DWC	DWC+	Author's method
Area overhead	high	medium	medium	medium
Masking	high	low	medium/high	low
Detection	unlikely	likely	unlikely	very likely
Robustness	high	low	medium	low

known good values. The block RAM contents, if ECC protected, could be error-corrected separately. The easiest way to restore the processor state would be to save and restore any registers in software. By this means, the full benefit of any masking in the hardware would be preserved.

7.7 Comparing the redundancy approaches

Using the data from Swift [Swift, Carmichael and Allen, 2008] for a worst case (1200km, 65° inclination satellite orbit), less than one single-event upset (SEU) might be expected per day in flip-flops (this includes configuration memory), and around 14 SEU per day in block memory cells. This assumes no coronal mass ejection (CME) is in progress. Such an extreme circumstance is likely to cause power supply upsets, or perhaps total destruction of a satellite, as well as considerable disruption on Earth. Otherwise, the chance of a double-upset could be modelled as a Poisson distribution, and assuming the probability of a second upset will be negligible, scrubbing will re-instate the incorrect bit. The vulnerability of the configuration memory is more concerning, since this has control of, amongst other things, the routing between configurable logic blocks. If the voting between TMR blocks occurs at a very high level (for example outside the chip), this would not matter so much. However, with today's dense designs, and the critical requirement for satellites to be as light as possible, to save rocket fuel and associated infrastructure, it is preferable for the voting logic to be contained within the device. The Brigham Young University TMR tool [Carrol, 2009] has various options, including voting on feedback paths, to prevent error propagation, but since the technique is concerned with masking errors it would not be possible, with this method alone, to decide when to trigger a re-configuration.

By contrast, the DWC method has explicit self-checking checkers (as described by [Wakerly, 1974]), which can feed forward an error, to cause eventual reconfiguration. Under normal circumstances, an external management processor would perform this function. However, this would not be necessary for a device with on-board processor, such as the Zync [Dobai and Sekanina, 2013] series. Alternatively, the redundant parts of the design could be arranged in a form where partial reconfiguration was possible, via the on-chip configuration interface. Since TMR has only single fault tolerance, and no way of distinguishing a valid from an invalid result, it is very important that the redundant parts of the design are not assigned to similar areas of the chip, such as the same CLBs or routing switch boxes. This may be done using placement constraints in conjunction with verification of the database at the XDL (Xilinx design language) level [Beckhoff, Koch and Torresen, 2011]. A further complication with TMR is that error masking would have to be detected via a separate readback mechanism after

burn-in to approve a critical system for operation. The recommended method of chapter 9 has the desirable property that all input code words map onto output code words which are derived from independent logic structures, making it robust against common mode error, consequently no single fault can silently convert between valid codes. A fault tolerance system designed purely for detection, and not correction, can make use of self-checking checkers [Smith, 1978], and redundant operand coding, to provide positive assurance of correctness, independent of confidence in the reliability of routing. A generic approach to redundancy will not be a panacea: the computer that prints the pay cheques for a large institution, would most definitely be required to stop immediately in the event of a fault. Likewise, a computerised system for printing tape measures would not be appreciated for printing out a bonus foot for random customers, contrary to trading standards. However, the autopilot system for an airliner with 200 passengers would be a very different matter. Arguably, a sudden change in the relationship between aircraft, computer and disorientation of the pilots let directly to the loss of flight AF 447 on 1st June 2009 [BEA, 2012].

7.8 Chapter Summary

It has been seen that the preferred method of error handling will depend on the application. The content of chapter 3 - chapter 7, taken as a whole, is claimed to be a novel methodology. In the next chapter the issue of garbage collection in hardware is developed. In chapter 9, the methodology is applied to various examples, in order to evaluate the solutions to the research questions.

"I have to say that in 1981, making those decisions, I felt like I was providing enough freedom for 10 years. That is, a move from 64k to 640k felt like something that would last a great deal of time. Well, it didn't - it took about only 6 years before people started to see that as a real problem."

Bill Gates [Gates, 1989]

8

Memory Management

This chapter is concerned with how to prevent the user from having to worry about the legacy and error-prone process of manually managing memory storage. It hardly needs stating that it would be a violation of type-safety to give the user explicit control over memory allocation and destruction. Nevertheless, it might still be appropriate for the user to be involved in the timing of the initiation of garbage collection, at some suitable stage in the algorithm. In a server application this could be every time a connection is closed, for example.

An inherent property of implicit memory allocation, a requirement for type-safety, is that it will not be immediately obvious when a result or intermediate calculation goes out of scope. Static analysis can prove that a result is always statically in the scope of its declaration (otherwise the program would not compile), but for efficient memory management, some sort of global notion of liveness is required. In traditional OCaml, generational garbage collection is used, naturally enough written in software, since no other option is available on a general purpose workstation. Bearing in mind the aim of the methodology, to avoid losing type-safety, it would not be appropriate to use a similar algorithm in the FPGA environment, when techniques that are more flexible are available, taking advantage of hardware assistance. Therefore, the garbage collection (which should ideally be incremental, to maintain, as far as possible, real-time performance) together with hardware support for virtualisation of garbage collection, is together grouped under the title of memory management.

8.1 Basic garbage collection techniques

A thorough treatment of garbage collection is available in [McCreight, 2008], in particular the problem of verifying the correctness of garbage collection. Not all garbage collection algorithms are suitable for hardware implementation. Typically, a layered approach, whereby

the software operates as long as it can with the assumption that memory is plentiful, and hardware carves up and serves memory as required, but without detailed knowledge of what is going on in software (such as offering a buffer of a certain size, without being told it contains a string) are more suitable for hardware implementation. To track usage, a simple follow/no-follow flag is sufficient. Those algorithms that need to store a recursive state, of arbitrary size, would lead to complex hardware implementations. The non-recursive algorithm, due to Cheney [Cheney, 1970], subsequently improved by Walden [Walden, 1972], is suitable for hardware implementation. However, it is not incremental and assumes a copying operation that will consume extra memory, up to the total size of the existing heap. The solution due to Bacon [Bacon, Cheng and Shukla, 2013] is different, in that it is specifically designed for incremental collection, on reconfigurable hardware. Since the goal in Bacon's usage is massive parallelism, the fundamental figure-of-merit that this solution aims for, is that it should be stall free, a laudable objective. By contrast, the biggest limitation of his system is that all the blocks have to be uniform, an unreasonably wasteful restriction for the purposes of this apparatus. Bacon's solution is based on an algorithm due to Yuasa [Yuasa, 1990], subsequently adapted to hardware implementation.

8.2 Garbage collection as part of memory management

The rôle of garbage collection is identifying memory that is not being used. In addition, the complete hardware memory management problem embraces:

(i) Protection of areas of memory, identified as instructions, from accidental overwrite due to a transient error.

(ii) Dividing global memory into suitable sized chunks, to be allocated as heap objects, or stack area as appropriate.

(iii) Setting up shared areas of memory to communicate with parallel threads or DMA units.

(iv) Checking for out-of-memory or stack overflow conditions.

(v) Dealing with array bounds checking exceptions.

(vi) Mapping logical addresses, given out by the object allocator, to physical locations in RAM.

(vii) Returning the space occupied, by objects no longer needed, to the global pool, in a manner immune to race hazards.

8.3 In-place garbage collection algorithm

In a typical OCaml system running on a workstation, the garbage collector will scan the entire memory looking for objects that are in use, then perform a compaction phase looking for objects in memory and registers, then relocating the objects to save memory. After compaction, it may be possible to free a certain number of pages from the process virtual size, which

will then result in a speed and efficiency boost. However, in an incremental scheme, it is inconvenient to move objects around in memory, because this would result in a certain number of references being invalid, at a particular time. Therefore, it is desirable to keep an object once allocated, always at the same address. In a naïve scheme, this would result in freed objects occupying gaps in the memory map, defeating the aim. However, the introduction of a fine-grained memory management unit (MMU) changes this picture, because every access will be translated from a logical to a physical memory domain. In a workstation environment the MMU is designed for relocation and protection between processes, and as such will have a relatively course granularity, typically 4096 bytes or more. This is far too large for the apparatus proposed here: a figure of 4 or perhaps 8 words would suit the application better, if the overheads can stand it. If access is available to page tables at this level, it becomes quite easy to free objects that are no longer needed, returning them to the pool for future use.

What is the relationship between object size and overhead? In a traditional system, mediated by translation look-aside buffers, the overhead is determined by the number of levels of page-table lookup, and this also determines the delay, when a page-table cache miss occurs. In a heterogenous apparatus, such as the one considered here, a low overhead is required, but the requirements are simpler:

(i) Objects to be allocated need to be of variable size. In the absence of any profiling, of a particular application, it makes sense to allocate in powers of two, since this makes the hardware decision-making process faster.

(ii) Objects returned can have any address (on 32-bit boundaries); objects of the same size (to the nearest power of 2) can be allocated sequentially from a pool.

(iii) The type-safety aspect means that there is no concern over data overflowing from one object to the next object, because bounds checking will already have been carried out, for eligible objects.

(iv) To reduce the overhead, a standard technique is to put the pointers to the free list inside the free objects themselves.

(v) Newly allocated objects do not need to be demand-zero initialised. Since OCaml does not allow constructors be created without a defined valid initialiser, it does not matter what the previous contents were. The one exception is character strings which are always valid regardless of contents.

(vi) Because typical embedded software footprints are compact, relative to the virtual address size, it does not matter if there are some gaps in the memory map.

If garbage collection determines that an object is free, it needs to be returned to the list of objects, of that particular size. This is usually different from the size that OCaml thinks that it has allocated. Strings and arrays, for example, must be a particular size for the run-time system to work; rounding up is not allowed. The first simplification that can be made is due to the relatively small size of most embedded products. To keep cost and power consumption down, these systems are usually optimised for a smaller memory footprint, of the order of

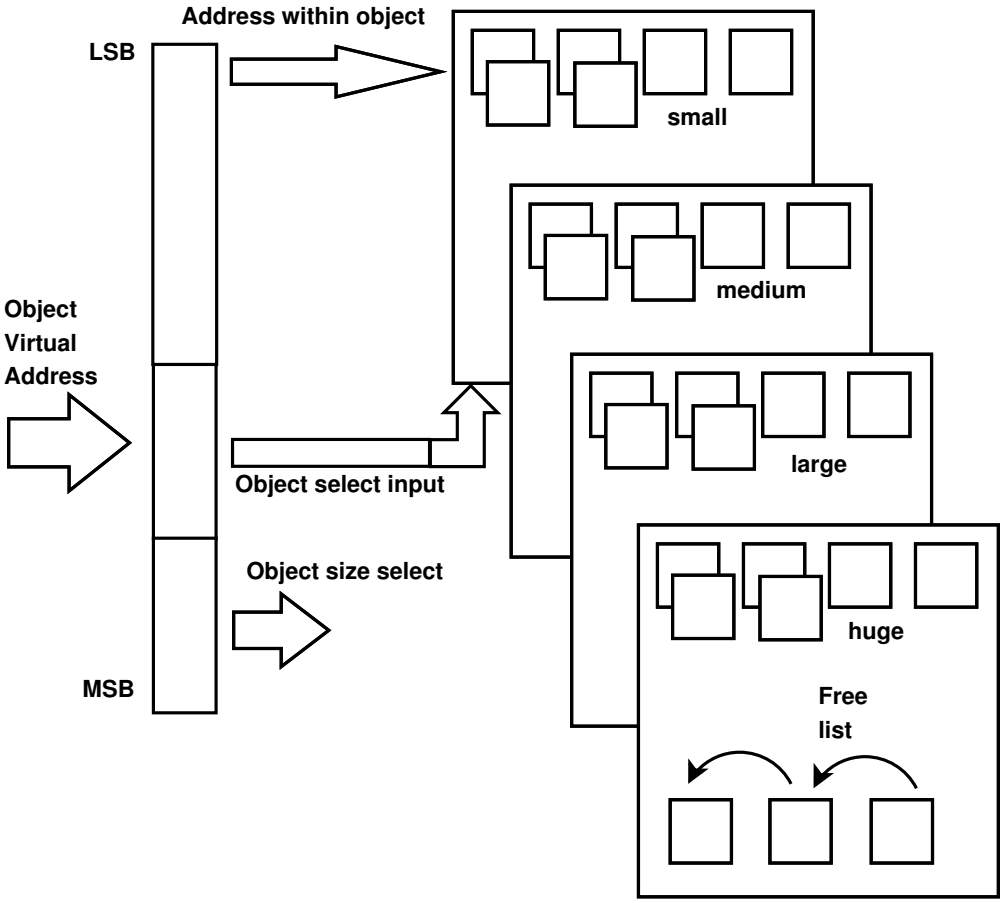


Figure 8.1: Organisation of object memory

megabytes, rather than gigabytes. This flexibility may be made use of by chunking objects of the same size in the same place in the memory map. When an object is freed, the address in memory may be decoded to find out to which pool it belongs. It is then a simple matter to add the block to the head of the free list. This needs to be an atomic operation, since it could happen simultaneously with the processor requesting more memory for new objects.

The proposed solution is illustrated in Fig 8.1. Referring to the diagram, it is clear there will be gaps in the address map because a constant number of address bits are used to refer to objects of various sizes; the least-significant bit of the address maps to the address into the object, as expected. There should be sufficient bits in the low third of the address for the largest supported object size, for example 15-bits. The most-significant bit of the address maps to the object size select (as a power of 2). A size select of 5 bits in the top third will be sufficient to cover a 32-bit address space, with some inefficiency for larger sizes. The middle third of the bits choose which object within a certain size range is chosen, from the available objects: this part of the address should have sufficient bits to allow for growth of the number of objects, for example 12-bits.

8.4 Summary

Historically embedded processors typically had no MMU. Processors that do have this facility generally allocate large pages, which then have to be sub-divided in software. The simpler

variant, the memory protection unit (MPU) protects but does not map. Verification of garbage collection algorithms, written in software, is a difficult problem [McCreight et al., 2007], [Hawblitzel et al., 2007], [Bacon, Cheng and Shukla, 2013]. If an implementation can allocate entirely in hardware (even if freeing is under software control), it helps to prevent race conditions. In the algorithm presented here, the relatively large address space, relative to the amount of memory typically installed in an embedded device, allows holes and/or aliasing in the address map not to matter, which then simplifies the hardware algorithm, so that it only needs to work on powers of two. It is a simple matter to add logic on the address bus, seen by the software, but complicated algorithms will lead to an unacceptable performance loss. The demonstrator of Figure 9.1 does not yet implement a garbage collection algorithm as presented here. However, no fundamental problems with implementation are anticipated.

"The theoretician is forced, ever more, to allow himself to be directed by purely mathematical, formal points of view in the search for theories, because the physical experience of the experimenter is not capable of leading us up to the regions of the highest abstraction."

Albert Einstein

9

Discussion and comparative analysis

This chapter is primarily concerned with the results of executing the methodology of this thesis under various trivial and not-so-trivial situations. It also gives more details of the apparatus which was constructed to illustrate the methodology, and compares and contrasts the merits of different fault-tolerance techniques.

In this chapter, various examples of Verilog state machines, using the methodology of chapter 3 - chapter 7 are presented. The first two examples take their input from the console, using the parser of section 9.1 to generate an internal abstract syntax tree (AST). This tree is then transformed by pattern matching to the associated λ-Calculus expression, which is then reduced by the rules of Table 4.2, to a final value (which may have the side effect of producing output). Table 9.1 gives the results for the traditional recursive factorial algorithm (which has already been demonstrated to reduce to the correct value in Coq). Table 9.2 shows the same process of executing the parser on the deeply recursive Fibonacci series. It is apparent that the technique of parsing **parsing expression grammar** (PEG) is greedy in terms of memory, although it is compact in source code, and expressive. Consequently, these two examples have only been run in simulation, not on the actual hardware platform (which is limited to a megabyte). Table 9.3 gives some results for a simple division calculation (1/0.81 - which lacks an exact binary representation) using the Flocq floating point library. This particular example avoids the need for a floating point co-processor in the hardware, at the expense of extra memory consumption. However, it does offer proof of IEEE-754 conformance [Zuras et al., 2008], which is worthwhile for such a simple technique. The conformance proof only applies to the core functionality. The conversion to and from decimal is the author's own prototype implementation in OCaml, with limited accuracy (refer to Table 9.4). As is usual with binary to decimal conversion, there is no exact representation of the vast majority of rational approximations in floating point arithmetic.

9.1 Selection of the parser

The choice of technology for the parser was largely dictated by the convenience of adaptation of the source code, to the chosen subset. A conservative choice would be a table based parser, along the lines of ocamlyacc. This would have the advantage of relatively low dynamic memory consumption. However, in addition to generating tables that are quite large (for example the AST parser for OCaml is about 85K of code and 140K of data), some low-level coercions are used which would undermine the goal of front-to-back type-safety. A camlp4 parser, which generates objects that are very compact, was another possibility considered, but it requires a library of over 900K. A hand-written, recursive descent, parser would be the ultimate in efficiency, but not maintainability, so for a demonstrator a compact Left to Right Tree Traversal (LRTT) parser (a kind of PEG parser) design was chosen. The ML grammar, and associated functor, is derived from Delatre [Delatre, 2012]. Necessary parsing expressions are generated on the fly, using a table-based approach, and backtracking when matching fails. This small parser functor is easily portable to the chosen subset, given the simplifications described in chapter 5 (primarily removing references to polymorphic equality, and printf). An extract from the grammar is shown below. The extended listing is available, in Appendix-J.

```
...

let keywords = [ "let"; "rec"; "in"; "if"; "then"; "else"; "mod" ] in

let grammar : grammar = [
  "start", [
    "", "", [NT "osp"; NT "expr"];
    "", "", [NT "osp"; NT "top"];
  ];

  "top", [
    "", "", [NT "osp"; NT "topexp"; NT "expr"; S (fun beg pos pval ->
      print_endline ("top: "^impl.dump pval);
      match popvalue() with List lst -> LET (List.rev lst, pval) | oth ->
 failwith (String.concat ", " (List.map impl.dump [oth;pval])))];
  ];

...

  "expr", [
  "fundecl", "", [NT "osp"; A(tsymbol, "fun"); NT "sp"; NT "patt"; NT "osp";
    A(tsymbol, "->"); NT "osp"; NT "expr";
    S (fun beg pos pval -> Abstraction (popvalue (), pval))];
  "fadd", "", [NT "subexp"; NT "osp"; A(tsymbol, "+."); NT "osp"; NT "subexp";
 S (fun beg pos pval -> FAdd (popvalue (), pval))];
  "fsub", "", [NT "subexp"; NT "osp"; A(tsymbol, "-."); NT "osp"; NT "subexp";
 S (fun beg pos pval -> FSub (popvalue (), pval))];
  "addition", "", [NT "subexp"; NT "osp"; A(tsymbol, "+"); NT "osp"; NT "subexp";
     S (fun beg pos pval -> Add (popvalue (), pval))];
  "subtraction", "", [NT "subexp"; NT "osp"; A(tsymbol, "-"); NT "osp"; NT "subexp";
       S (fun beg pos pval -> Sub (popvalue (), pval))];
  "if", "", [NT "osp"; A (tsymbol, "if") ; NT "osp"; NT "boolean"; NT "osp";
       A (tsymbol, "then") ; NT "expr"; NT "osp"; A (tsymbol, "else");
       NT "osp"; NT "expr"; S (fun beg pos pval -> let mid = popvalue () in
       If (popvalue (), mid, pval))];
  "subexp", "", [NT "subexp"];
  "eof", "", [A (testeof, "EOF")];
  ];

...
```

9.2 Example of a simply-typed λ-Calculus

Figure 9.1 shows a photograph of the screen of the ML605 during execution of the simply-typed λ-Calculus example. This evidence is significant because it demonstrates that the techniques developed in the methodology chapters lead to a result that can actually operate on a real FPGA, communicating with a real SVGA monitor, driven by the previously discussed hardware description language. For the avoidance of doubt, no other computer is required once the appropriate binary format (on memory card) has been downloaded. For clarity of capturing the text, this version is limited to 32 columns/rows. Scrolling (by means of buttons) is needed to see the remaining output - as shown in the montage of Figure 9.2. As previously mentioned the same software may be compiled and run for a workstation, and the corresponding output is shown in Table 9.5.

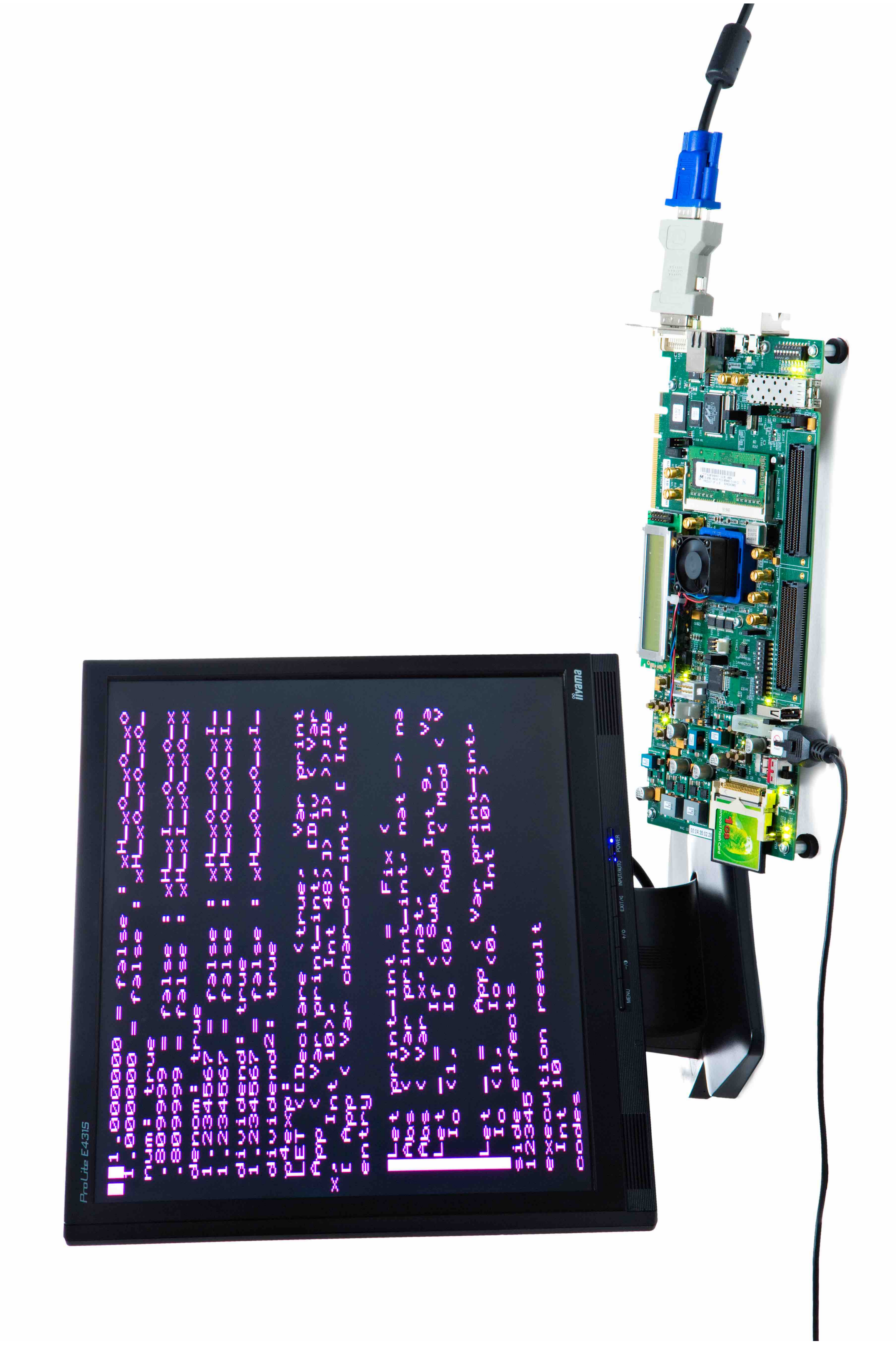


Figure 9.1: Execution on ML605 apparatus of λ-Calculus example of Table 9.5

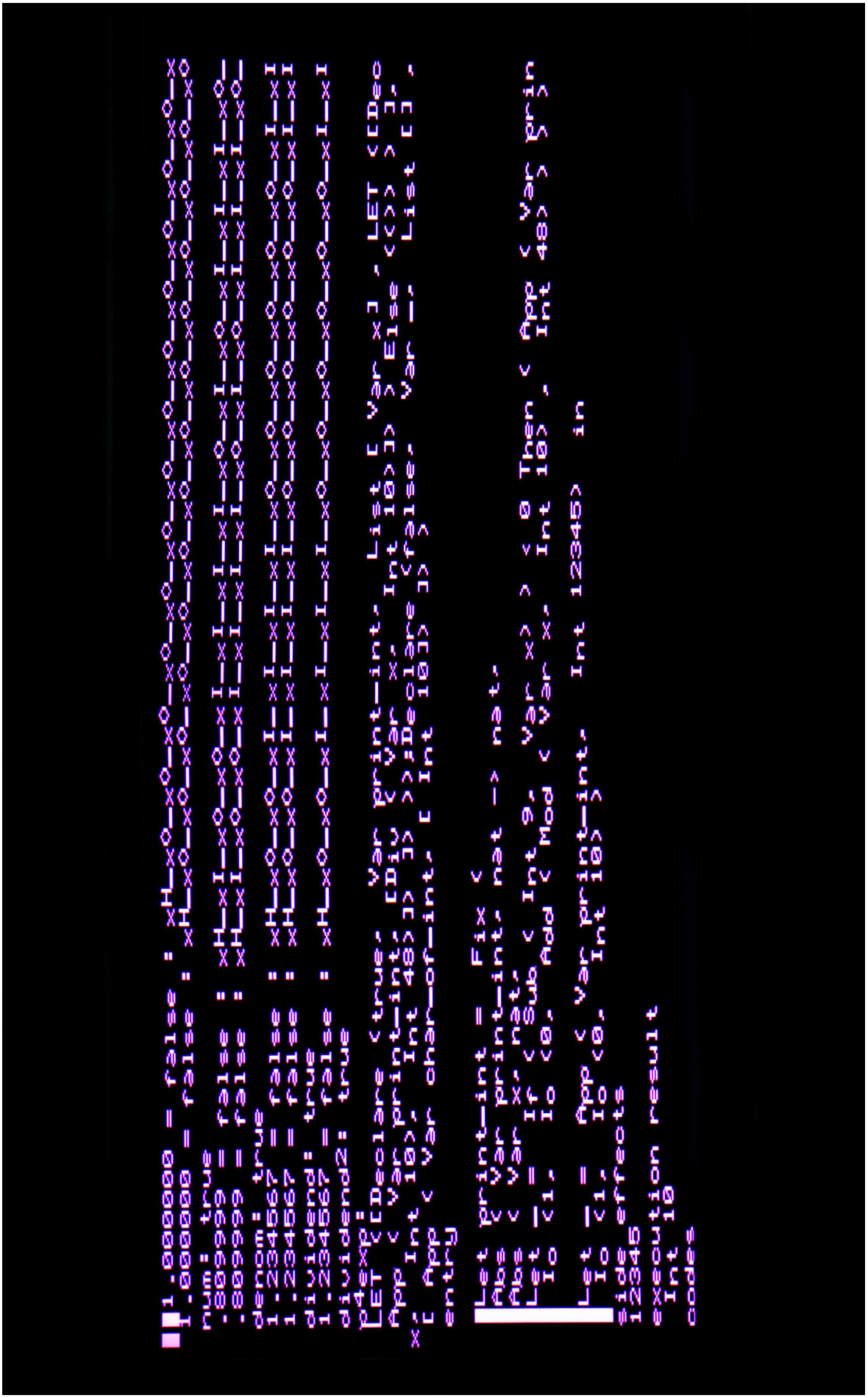


Figure 9.2: Close-up display of λ-Calculus example of Table 9.5 (SVGA montage)

Table 9.1: Traditional factorial test results

```
Expression entered:
let rec f x = if x = 0 then 1 else x * (f (x-1)) in f 6
Expression:
LET ([Declare (true,  Var f,  Var x,  If (Eq ( Var x,  Int 0))  Then ( Int 1) Else
    (Mul ( Var x, Application ( Var f, Sub ( Var x,  Int 1)))) ) ], Application ( Var f,  Int 6))
Added: f with encoding S(S(S(O)))
Added: x with encoding S(S(S(S(O))))
Compiled to:

 Let f =  Fix (
 Abs ( Var f, nat -> nat,
 Abs ( Var x, nat,  If ( Var x) = 0 Then ( Int 1) Else ( Mul (  Var x,  App ( Var f,
    Pred ( Var x) ) ) ) ) ) )  in
  App ( Var f,  Int 6)
Reduced to:
 Int 720
Static used alloc = 45712
Dynamic used alloc = 1756224
```

Table 9.2: Traditional fib test results

```
Expression entered:
let rec fib n = if n < 3 then 1 else fib (n-1) + fib (n-2) in fib 10
Expression:
LET ([Declare (true,  Var fib,  Var n,  If (Lt ( Var n,  Int  3)) Then ( Int 1) Else
    (Add (Application ( Var fib, Sub ( Var n,  Int 1)),  Application ( Var fib,
         Sub ( Var n,  Int 2)))) ) ], Application ( Var fib,  Int 10))
Added: fib with encoding S(S(S(O)))
Added: n with encoding S(S(S(O)))
Compiled to:

 Let fib =  Fix (
 Abs ( Var fib, nat -> nat,
 Abs ( Var n, nat,  If ( Sub ( Var n,  Int 3) ) < 0 Then ( Int  1) Else
     ( Add ( App ( Var fib,  Pred ( Var n) ) ) ,  App ( Var fib,  Sub ( Var n,  Int 2) ) ) ) ) ) ) in
   App ( Var fib,  Int 10)
Reduced to:
 Int 55
Static used alloc = 45712
Dynamic used alloc = 1433476
```

Table 9.3: Flocq demonstration

```
1.000000 = false : xH_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0 : neg(xH_x0_xI_xI_xI)
.809999 = false : xH_xI_x0_x0_xI_xI_xI_xI_x0_xI_x0_xI_xI_xI_x0_x0_x0_x0_xI_x0_xI_x0_x0_x0 : neg(xH_xI_x0_x0_x0)
1.234567 = false : xH_x0_x0_xI_xI_xI_xI_x0_x0_x0_x0_x0_x0_xI_xI_x0_x0_xI_x0_xI_x0_x0_xI_x0 : neg(xH_x0_xI_xI_xI)
p4exp:
FDiv (Float 1.000000, Float .809999)
entry
 FDiv ( Float 1.000000,  Float .809999)
execution result
 Float 1.234567
    1.234567 = false :  xH_x0_x0_xI_xI_xI_xI_x0_x0_x0_x0_x0_x0_xI_xI_x0_x0_xI_x0_xI_x0_x0_xI_x0_xI_..continued..
      xI_xI_x0_x0_xI_xI_xI_xI_x0_x0_x0_xI_x0_x0_xI_x0_x0_x0_xI_x0_x0_xI_xI_x0_xI_x0_x0_x0 : neg(xH_xI_x0_xI_x0_x0)
Static used alloc = 38424
Dynamic used alloc = 258420
```

Table 9.4: Algorithm for decimal conversion

```
let string_of_float_b754' dec top = function
  | Fappli_IEEE.B754_finite (sgn,mant,expon) ->
    let five = asPi 5 in
    let rec adjust binscale dec mant =
      if dec > 0 then adjust binscale (dec-1) (Pos.shiftr_nat (Pos.mul five mant) binscale) else mant in
    let binscale = (top-dec)/dec in
    let adj' = adjust (asNat binscale) dec mant in
    let shft = -(binscale+1)*dec-(fromZi expon) in
    let adj = fromZ (Zpos (if shft < 0 then Pos.shiftl_nat adj'  (asNat(-shft)) else
        Pos.shiftr_nat adj' (asNat shft))) in
    let len' = dec - String.length adj in
    let pad = (if len' > 0 then String.make len' '0' else "")^adj in
    let len = String.length pad in
    let split1 = String.sub pad 0 (len - dec) in
    let split2 = String.sub pad (len - dec) dec in
    let split = split1^"."^split2 in
    if sgn then "-"^split else split
  | B754_zero _ -> "0.0"
  | B754_infinity _ -> "inf"
  | B754_nan (_, _) -> "nan"
```

Table 9.5: Workstation output corresponding to Figure 9.1

```
1.000000 = false : xH_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0 : neg(xH_x0_xI_xI_xI)
1.000000 = false : xH_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0_x0 : neg(xH_x0_xI_xI_xI)
num: true
.809999 = false : xH_xI_x0_x0_xI_xI_xI_xI_x0_xI_x0_xI_xI_xI_x0_x0_x0_x0_xI_x0_xI_x0_x0_x0 : neg(xH_xI_x0_x0_x0)
.809999 = false : xH_xI_x0_x0_xI_xI_xI_xI_x0_xI_x0_xI_xI_xI_x0_x0_x0_x0_xI_x0_xI_x0_x0_x0 : neg(xH_xI_x0_x0_x0)
denom: true
1.234567 = false : xH_x0_x0_xI_xI_xI_xI_x0_x0_x0_x0_x0_x0_xI_xI_x0_x0_xI_x0_xI_x0_x0_xI_x0 : neg(xH_x0_xI_xI_xI)
1.234567 = false : xH_x0_x0_xI_xI_xI_xI_x0_x0_x0_x0_x0_x0_xI_xI_x0_x0_xI_x0_xI_x0_x0_xI_x0 : neg(xH_x0_xI_xI_xI)
dividend: true
1.234567 = false : xH_x0_x0_xI_xI_xI_xI_x0_x0_x0_x0_x0_x0_xI_xI_x0_x0_xI_x0_xI_x0_x0_xI_x0 : neg(xH_x0_xI_xI_xI)
dividend2: true
p4exp:
LET ([Declare (true,  Var print_int,  List [ Var x] , LET  ([Declare (false,  Var _,  List [] ,  If (Gt ( Var x,  Int 9))
     Then  ( App ( Var print_int, [Div ( Var x,  Int 10)]) )  Else  (()) ) ],  App ( Var print_char, [ App ( Var  char_of_int,
   [Add  (Mod ( Var x,  Int 10),  Int 48)]) ]) ));Declare (false,  Var _,  List [] ,  App ( Var print_int, [ Int 12345]) ) ],
  App ( Var print_char, [ App ( Var char_of_int, [ Int 10]) ]) )
entry

 Let print_int =  Fix (
 Abs ( Var print_int, nat -> nat,
 Abs ( Var x, nat,
 Let _ =  If ( Sub ( Int 9,  Var x) ) < 0 Then ( App ( Var print_int,  Div ( Var x,  Int 10) ) ) Else ( Int 0)  in
  Io (1,  Io (0,  Add ( Mod ( Var x,  Int 10) ,  Int 48) ) ) ) ) )  in

 Let _ =  App ( Var print_int,  Int 12345)  in
  Io (1,  Io (0,  Int 10) )
side effects
12345
execution result
 Int 10
codes
x -> 5
_ -> 4
print_int -> 3
failwith -> 2
print_char -> 1
char_of_int -> 0
```

9.3 Built-in fault mitigation circuitry

More recent Xilinx devices [Xilinx, 2011] have the capability to detect and correct single-event upsets, by means of Soft Error Mitigation control. [Xilinx, 2010]. This is an important step forward since there is a concern about the configuration circuits that will not be addressed by logic redundancy alone. Although the Cyclic redundancy-check (CRC) check on configuration codes is not sufficient alone, to correct possible errors, it may be used in conjunction with an external error categorisation device (such as an SPI-FLASH memory), to detect and reload any compromised bit-streams. The same device may be used to simulate a soft error, by means of a built-in configuration interface. There is a small probability that the mitigation control device itself could be compromised by a soft error. This feature could be complemented by an external watchdog, that triggers a full re-configuration, if the mitigation detects a fatal error, or otherwise ceases to function. Since active logic state is not a configuration feature, no protection against malfunction of the user's logic behaviour (such as incorrect values in flip-flops) is available, so all the aforementioned arguments in favour of redundancy are still needed.

9.4 Comparison of implementation by five methods

The ARM derived architecture, shown in block diagram form in Figure 9.3, and schematically in Figure 9.6, was encapsulated in the FPGA top-level, and then implemented by five different methods (the preferred source format is shown in Appendix-C):

(i) Plain(no redundancy) with results shown in Table 9.6.

(ii) TMR with the method due to Carrol [Carrol, 2009] with results shown in Table 9.7.

(iii) The method of chapter 7 with results shown in Table 9.8.

(iv) Coarse-grain elaboration with quteRTL with results shown in Table 9.9.

(v) Combined coarse-grain logic synthesis and redundancy insertion with the author's flow summarised in figure 9.7 and with results shown in Table 9.10.

Not surprisingly, the plain technique is the smallest and fastest. In terms of performance and gate count, the technique of chapter 7 is inferior to TMR, furthermore the area overhead is greater than extrapolating from chapter 7 would suggest, and fails to meet timing by a larger amount. This is disappointing and the reason is to be found in the specific architecture of the Xilinx implementation of adders and multiplexers, of which there are a considerable number inferred by the design of Figure 9.3. Consequently, by pre-converting to a gate-level description (by using Xilinx synthesis in a naïve way), even though it makes for straightforward replacement (of every candidate logic gate by a suitable self-checking gate), the end result is FPGA-unfriendly. Unfortunately, in a large block such as a wide multiplexer, there will be many opportunities for fault-detection to feed forward a fault indication. Furthermore, the special Xilinx structures, that provide the carry look-ahead function, will not be recognised

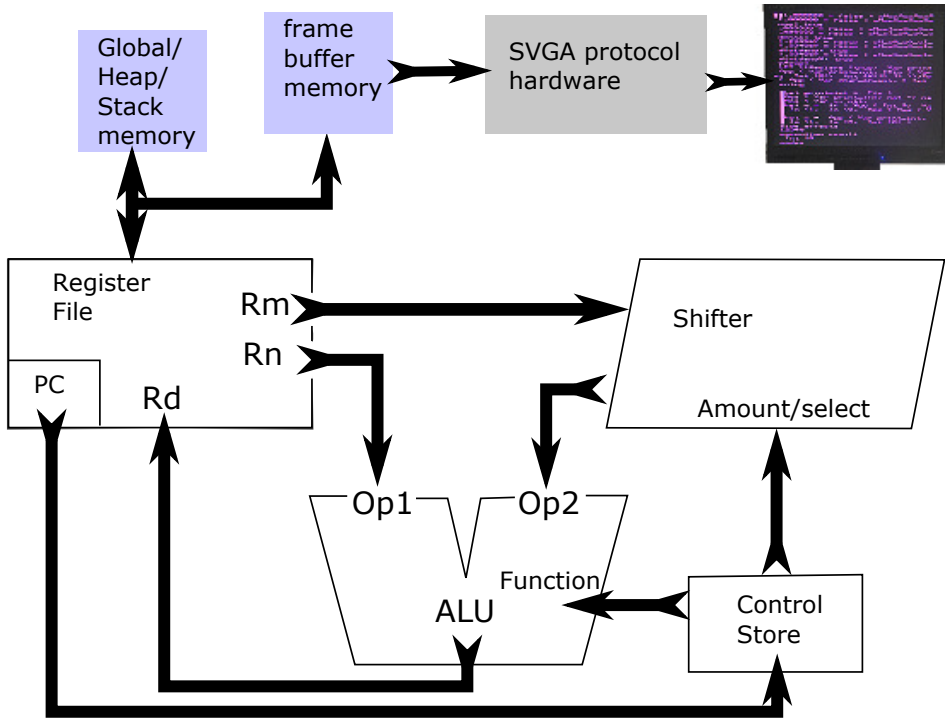


Figure 9.3: Top-level (block diagram) of FPGA processor

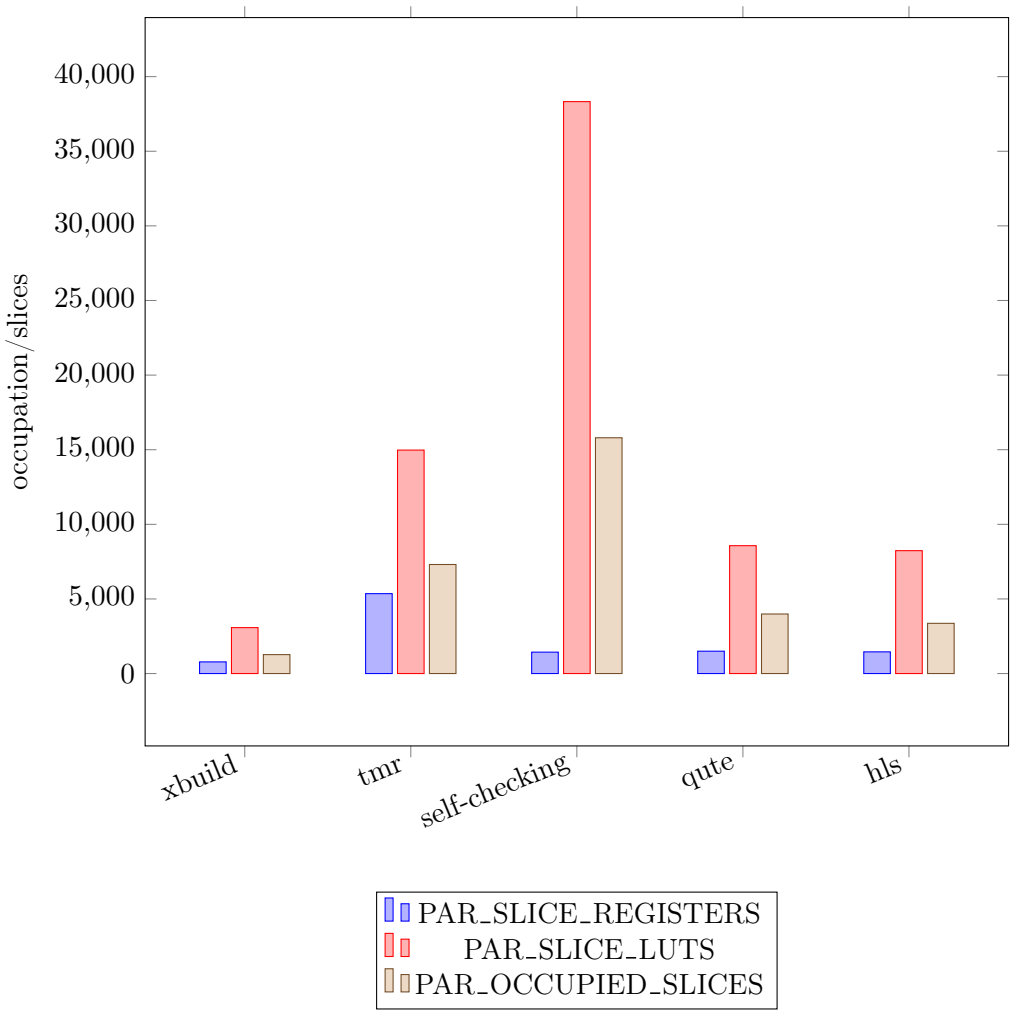


Figure 9.4: Four redundancy techniques

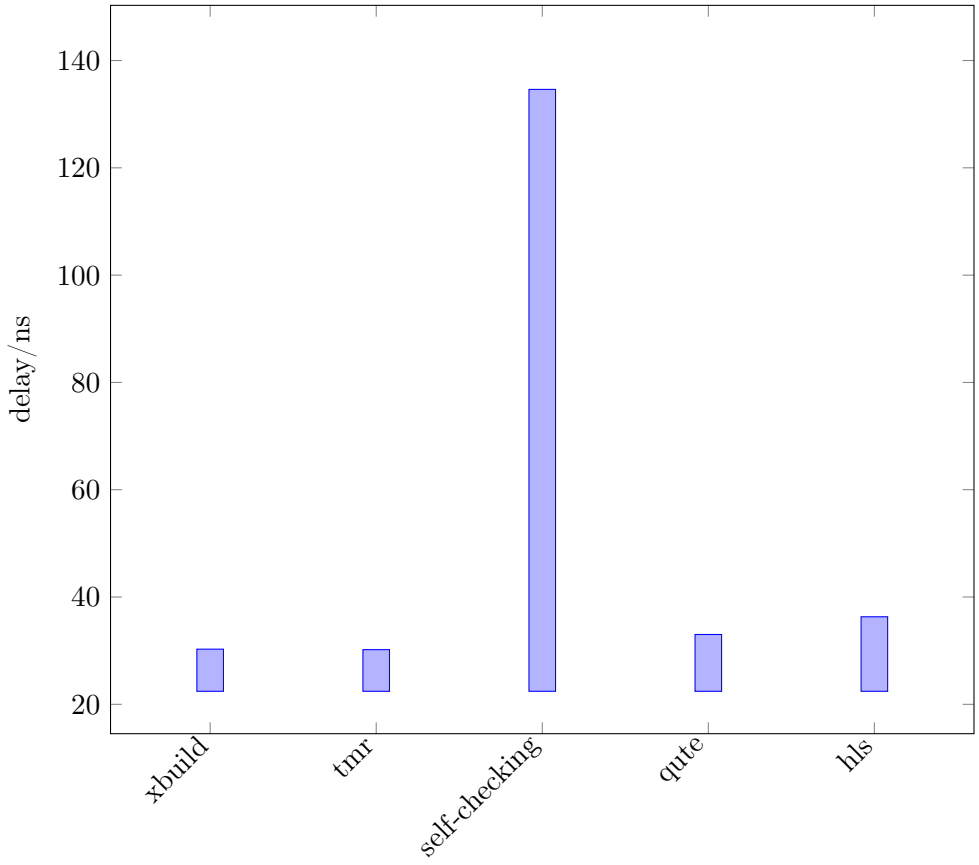


Figure 9.5: Four timing results

since at the low level the gate-level structure will be dual-rail. This will have a knock on effect on the timing, resulting in a large effort of duplication and reduction of logic depth, to try to meet timing. This will ultimately be unsuccessful because the majority of the timing budget will be lost in elaborate routing structures. The simple inter-CLB low-latency routing (for carry chains and similar purposes) will be less relevant for complex designs.

A solution to the challenge of making use of specialised structures, requires a change to methodology in that the high-level operators need to be kept abstract, at the point where the fault-tolerance is inserted. This methodology allows conventional carry look-ahead chains to be used, as well as error handling at the macro level (such as array multipliers).

It is not trivial to define such a methodology. The HDL-Compiler front-end from Synopsys Design-Compiler [Bhatnagar, 2001] could do it (but its output is not valid Verilog until after mapping). Alternatively, Xilinx XST synthesis would be an obvious choice, but it only allows CLBs or basic gates as output options. Similarly, the cross-platform synthesis product Synplify-Premier [Sutherland and Mills, 2014] has a mode to allow previewing of the technology independent synthesis output, in the form of a schematic, but (ridiculously) does not have a way to extract or simulate at this level of abstraction. A number of open-source alternatives might be considered, such as the previously reviewed tools in section 2.2. At an earlier stage, it was thought, the easiest solution would be to enhance the fault-tolerance tool itself to handle synthesis. While this option was being investigated, the *quteRTL* solution became available, which incorporates the desired functionality.

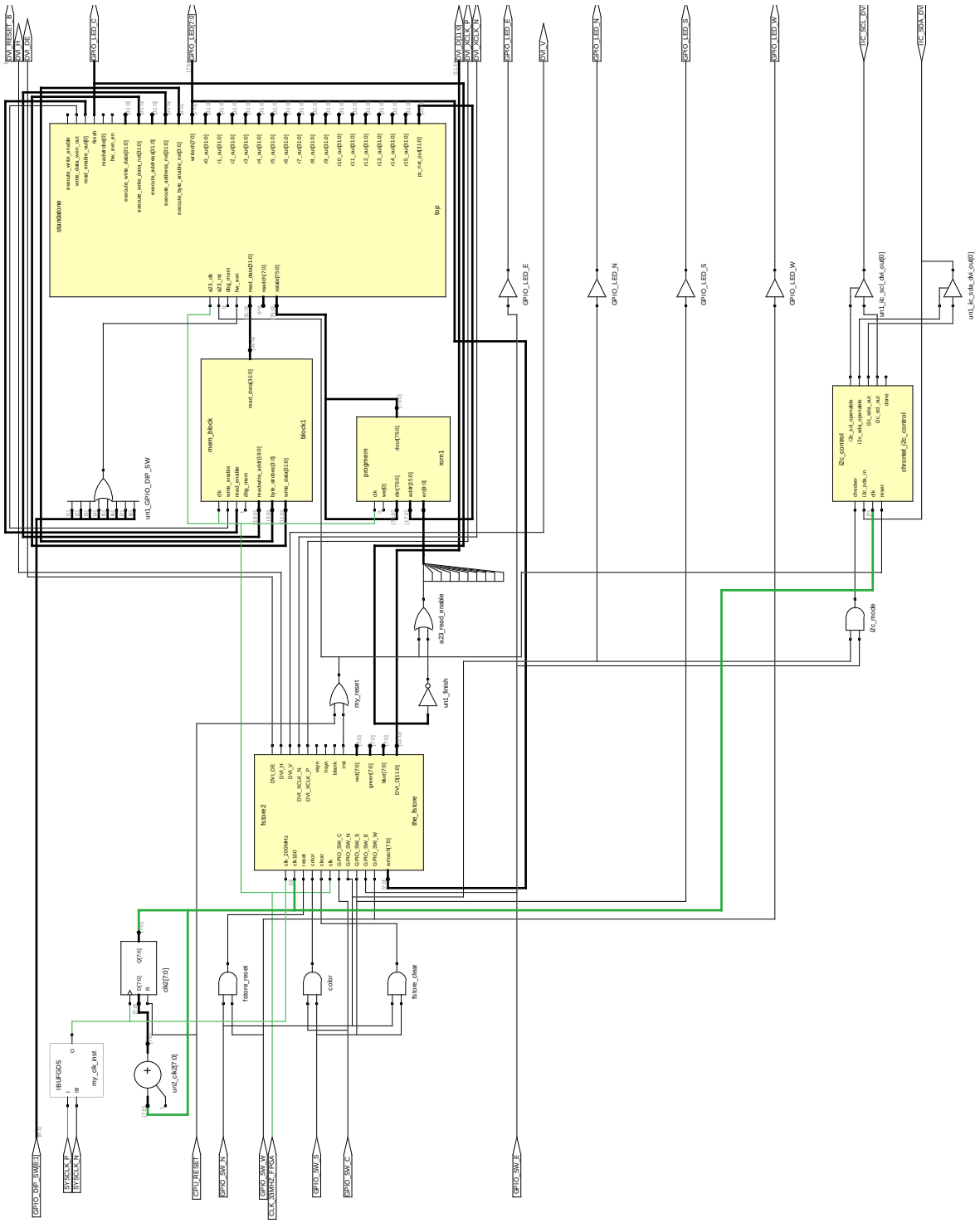


Figure 9.6: Top-level (schematic) of FPGA processor

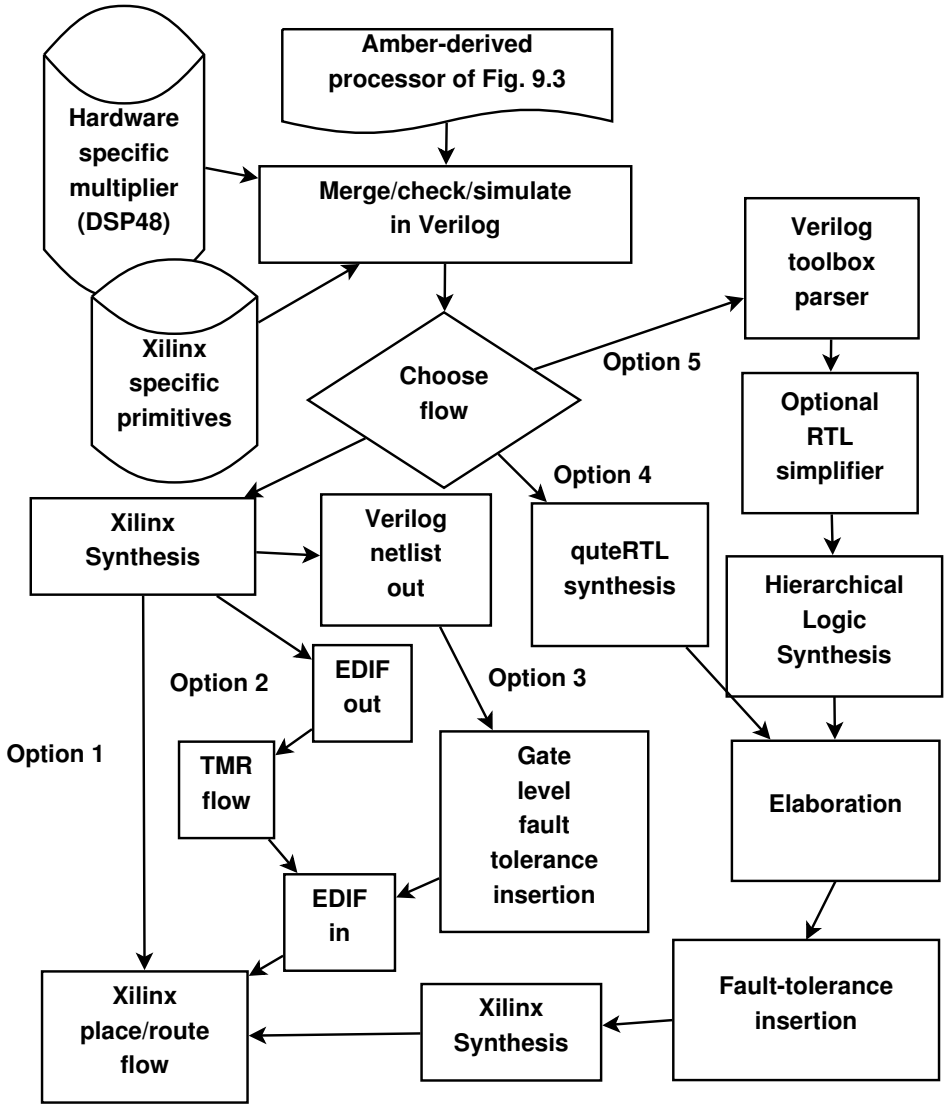


Figure 9.7: Hardware logic synthesis redundancy flow choices

By abstracting fault-tolerance at the operator level, a much greater control over the complexity/detection compromise is possible, and importantly, it does not prevent the use of dedicated arithmetic structures in FPGAs. The overall results are shown in Table 9.9, and will be seen there is a doubling of the quantity of flip-flops in the core processor, and arithmetic functions are duplicated. Timing is only slightly worse than the TMR solution, and is within the tolerance of chip voltage/process variation. The solution requires a library (as shown in Appendix-F). Subsequently, the necessary functionality was incorporated into the fault-tolerance infrastructure of chapter 7, producing the results shown in Table 9.10. This tool (available to download at [Kimmitt, 2015]) now provides an integrated environment for Verilog parsing, simplification, coarse-grain logic synthesis, elaboration, mapping, and various transformations, as shown in Figure 9.7. All five methods are summarised in terms of area in Figure 9.4, and in terms of timing in Figure 9.5. It will be seen that the fault detection coverage available, is architecture and logic dependent. Specifically AND/OR operations will have a tendency to mask any unknown or fault states. As a partial workaround for this problem, the bitwise AND/OR library functions are rewritten to use the truth tables based on the algorithm of chapter 7.

Table 9.6: Standard Amber processor place & route results

```
Slice Logic Utilization:
  Number of Slice Registers:                762 out of 301,440    1%
    Number used as Flip Flops:              761
    Number used as Latches:                   0
  Number of Slice LUTs:                   3,011 out of 150,720    1%
    Number used as logic:                 3,001 out of 150,720    1%

  Number of occupied Slices:              1,256 out of  37,680    3%
  Number of bonded IOBs:                     42 out of     600    7%

Specific Feature Utilization:
  Number of RAMB36E1/FIFO36E1s:             256 out of     416   61%
  Number of RAMB18E1/FIFO18E1s:             113 out of     832   13%
  Number of DSP48E1s:                         3 out of     768    1%
```

Constraint	Check	Worst Case Slack	Best Case Achievable	Timing Errors	Timing Score
TS_CLK_33MHZ_FPGA = PERIOD TIMEGRP "CLK_3 3MHZ_FPGA" 33 MHz HIGH 50%	SETUP	0.367ns	29.936ns	0	0
	HOLD	0.114ns		0	0
TS_sysclk = PERIOD TIMEGRP "sysclk_grp" 5 ns HIGH 50%	SETUP	3.871ns	1.129ns	0	0
	HOLD	0.085ns		0	0

```
All constraints were met.
```

Table 9.7: Triple modular redundancy Amber processor place & route results

```
Slice Logic Utilization:
  Number of Slice Registers:                 5,519 out of 301,440    1%
    Number used as Flip Flops:               3,932
    Number used as Latches:                  1,584
  Number of Slice LUTs:                     14,750 out of 150,720    9%
    Number used as logic:                   14,644 out of 150,720    9%

  Number of occupied Slices:                 7,347 out of  37,680   19%
  Number of bonded IOBs:                        42 out of     600    7%

Specific Feature Utilization:
  Number of RAMB36E1/FIFO36E1s:                  0 out of     416    0%
  Number of RAMB18E1/FIFO18E1s:                832 out of     832  100%
  Number of DSP48E1s:                            9 out of     768    1%
--------------------------------------------------------------------------------------------------
  Constraint                                |    Check    | Worst Case |  Best Case | Timing |   Timing
                                            |             |    Slack   | Achievable | Errors |    Score
--------------------------------------------------------------------------------------------------
* TS_CLK_33MHZ_FPGA = PERIOD TIMEGRP "CLK_3 | SETUP       |    -0.979ns|    31.282ns|     368|       94835
  3MHZ_FPGA" 33 MHz HIGH 50%                | HOLD        |     0.108ns|            |       0|           0
--------------------------------------------------------------------------------------------------
  TS_sysclk = PERIOD TIMEGRP "sysclk_grp" 5 | SETUP       |     1.703ns|     3.297ns|       0|           0
   ns HIGH 50%                              | HOLD        |     0.139ns|            |       0|           0
--------------------------------------------------------------------------------------------------


1 constraint not met.
```

Table 9.8: Self-checking Amber processor place & route results

```
Slice Logic Utilization:
  Number of Slice Registers:                 1,502 out of 301,440    1%
    Number used as Flip Flops:               1,273
    Number used as Latches:                    229
  Number of Slice LUTs:                     34,329 out of 150,720   22%
    Number used as logic:                   34,282 out of 150,720   22%

  Number of occupied Slices:                13,987 out of  37,680   37%
  Number of bonded IOBs:                        42 out of     600    7%

Specific Feature Utilization:
  Number of RAMB36E1/FIFO36E1s:                  0 out of     416    0%
  Number of RAMB18E1/FIFO18E1s:                821 out of     832   98%
  Number of DSP48E1s:                            4 out of     768    1%

--------------------------------------------------------------------------------------------------------
  Constraint                                |    Check    | Worst Case |  Best Case | Timing |   Timing
                                            |             |    Slack   | Achievable | Errors |    Score
--------------------------------------------------------------------------------------------------------
* TS_CLK_33MHZ_FPGA = PERIOD TIMEGRP "CLK_3 | SETUP       |   -57.533ns|    87.836ns|    6236|   207516561
  3MHZ_FPGA" 33 MHz HIGH 50%                | HOLD        |     0.153ns|            |       0|           0
--------------------------------------------------------------------------------------------------------
  TS_sysclk = PERIOD TIMEGRP "sysclk_grp" 5 | SETUP       |     3.516ns|     1.484ns|       0|           0
   ns HIGH 50%                              | HOLD        |     0.112ns|            |       0|           0
--------------------------------------------------------------------------------------------------------


1 constraint not met.
```

Table 9.9: QuteRTL fault-tolerant Amber processor place & route results

```
Slice Logic Utilization:
  Number of Slice Registers:          1,499 out of 301,440    1%
    Number used as Flip Flops:        1,413
    Number used as Latches:               0
  Number of Slice LUTs:               8,343 out of 150,720    5%
    Number used as logic:             8,337 out of 150,720    5%

  Number of occupied Slices:          3,297 out of  37,680    8%
  Number of bonded IOBs:                 50 out of     600    8%

Specific Feature Utilization:
  Number of RAMB36E1/FIFO36E1s:           0 out of     416    0%
  Number of RAMB18E1/FIFO18E1s:         821 out of     832   98%
  Number of DSP48E1s:                     6 out of     768    1%
---------------------------------------------------------------------------
  Constraint                    |    Check    | Worst Case |  Best Case | Timing |   Timing
                                |             |    Slack   | Achievable | Errors |    Score
---------------------------------------------------------------------------
* TS_CLK_33MHZ_FPGA = PERIOD TIMEGRP "CLK_3 | SETUP       |    -1.739ns|    32.042ns|     171|       73572
  3MHZ_FPGA" 33 MHz HIGH 50%                | HOLD        |     0.108ns|            |       0|           0
---------------------------------------------------------------------------
  TS_sysclk = PERIOD TIMEGRP "sysclk_grp" 5 | SETUP       |     3.804ns|     1.196ns|       0|           0
   ns HIGH 50%                              | HOLD        |     0.132ns|            |       0|           0
---------------------------------------------------------------------------
```

Table 9.10: Amber processor place & route results for coarse-grain logic synthesis

```
Slice Logic Utilization:
  Number of Slice Registers:          1,454 out of 301,440    1%
    Number used as Flip Flops:        1,410
    Number used as Latches:               0
  Number of Slice LUTs:               8,235 out of 150,720    5%
    Number used as logic:             8,229 out of 150,720    5%

  Number of occupied Slices:          3,366 out of  37,680    8%
  Number of bonded IOBs:                 50 out of     600    8%

Specific Feature Utilization:
  Number of RAMB36E1/FIFO36E1s:           0 out of     416    0%
  Number of RAMB18E1/FIFO18E1s:         821 out of     832   98%
  Number of DSP48E1s:                     6 out of     768    1%
---------------------------------------------------------------------------
  Constraint                    |    Check    | Worst Case |  Best Case | Timing |   Timing
                                |             |    Slack   | Achievable | Errors |    Score
---------------------------------------------------------------------------
* TS_CLK_33MHZ_FPGA = PERIOD TIMEGRP "CLK_3 | SETUP       |    -6.012ns|    36.315ns|    1850|    1651034
  3MHZ_FPGA" 33 MHz HIGH 50%                | HOLD        |     0.108ns|            |       0|          0
---------------------------------------------------------------------------
  TS_sysclk = PERIOD TIMEGRP "sysclk_grp" 5 | SETUP       |     3.651ns|     1.349ns|       0|          0
   ns HIGH 50%                              | HOLD        |     0.143ns|            |       0|          0
---------------------------------------------------------------------------
```

9.5 Error recovery in a fault-tolerant processor

It has been seen that the TMR method, by assuming a majority vote, does not have a mechanism to detect errors. This is no problem, in the event of a single error, but if errors are more likely the solution can be improved, by generalisation to N-modular redundancy (or by cascading). Undetectable errors are also a problem, because masking an error may reduce the capability to handle potential future errors. In some applications, it is desirable to report an error to the software level. This may be achieved in the following manner, with this processor. The fault tolerant outputs (dual-rail) may be externally compared and reduced to an overall error signal. Two further components are required, a method of translating this error signal to an exception in software (complete with an appropriate *try .. fail* block to handle the exception), and a method of provoking the error to occur. This may be achieved in the prototype flow, by adding a special vector (associated with the flip-flop module F_DVL_DFF_SYNC of Appendix-F) into the fault tolerant netlist, which will have the effect of injecting a common mode error. This will, in turn, be connected to the external DIP-switch inputs of the apparatus, to prevent netlist optimisation from eliminating the complement signals. The operation may be seen in Figure 9.8, which shows the idealised behaviour in parallel with behavioural simulation.

9.5.1 Linking hardware exceptions to the OCaml exception mechanism

Hardware faults need to be linked into the OCaml runtime exception mechanism to allow the software to respond appropriately (such as by retrying the failing operation). The introduction of a new built-in Sys.hw_exn() captures the program counter in *register 11* (a register which is conveniently not allocated to code generation), and then returns boolean false. In the event of a real hardware exception, built in behaviour of the logic of Appendix-C copies the value of *register 11* into the program counter, and copies three (representing boolean true in OCaml unboxed integer representation) into *register 0*, which holds the function return value. This behaviour will cause the current function to be abandoned, without needing to read any memory, and control will be transferred to the *if clause* following the most recent call to Sys.hw_exn. This clause is expected to be a raise statement that will recover the stack pointer, and appropriate program counter, from the most recent *try .. fail* block (not necessarily the one in static scope). The type of exception raised should be one of global scope, since the type is predetermined in advance of the point when the exception will occur, and to raise an exception that is out of scope, would be an error in type safety. It frequently happens that an inner *try .. fail* block will not be equipped to handle a hardware error. This will cause further unwinding, until finally the global scope is reached. As an incompatible feature, if hardware exceptions are used, they will destroy semantic compatibility with the workstation version of OCaml, which ordinarily does not have any such feature. A workaround for this problem would be to add a dummy hardware error handler, which always returns false, into the native run-time library.

Table 9.11: Example hardware exception handler

```
open Mylib
let _ = if Sys.hw_exn() then raise (Invalid_argument "hw_err")
let _ = try
  for i = 0 to 13 do
  print_int i; print_char ' '; print_int (fact i); print_newline()
  done;
  with _ -> print_int 42
```

Table 9.12: Software to demonstrate Hardware exceptions

```
open Mylib

let mydiv denom = try
  print_int (355000000 / denom); print_newline()
   with
   | Division_by_zero -> print_endline "Divide by zero trap"
   | Invalid_argument err -> print_endline (err^": exception caught")

let mymod num denom = try
  print_int (num mod denom); print_newline()
   with
   | Division_by_zero -> print_endline "Divide by zero trap"
   | Invalid_argument err -> print_endline (err^": exception caught")

let _ =
(* *)
  mydiv 113;
  mydiv 1;
(* *)
  mydiv 0;
(* *)
  mymod 12345 10;
  mymod 12345 0
```

9.5.2 Hardware exceptions example

The basic Amber architecture does not include a hardware implementation of the divide algorithm. Instead, these operators are mapped onto library functions in the basic implementation. An alternative technique is to implement the divider as a co-processor (due to [Studboy-ga, 2002]), which will quite naturally need to signal an exception if the user tries to divide by zero. The upgraded design has already been implemented into the listing in Appendix-C. The same additional hardware calculates both divide and remainder every time, which is wasteful in applications such as printing numbers where both results are wanted. To use this feature optimally, compiler support for a four operand div_mod instruction would have to be added. In a larger application, where it is desired to distinguish between different types of hardware exception, the *hw_exn* signalling needs to be enhanced to return a variety of results, depending on the type of hardware exception, and this can quite naturally map onto OCaml builtins of type *exn*. In order to test this functionality the program of 9.12 may be used:

The usual output from this program would be as follows:

```
3141592
355000000
Divide by zero trap
5
Divide by zero trap
Exited
```

The reason is that OCaml does not like to trigger a hardware exception since on some operating systems it upsets the internal state. By recompiling the program with the -unsafe option we get the required behaviour:

```
3141592
355000000
hw_err: exception caught
5
hw_err: exception caught
Exited
```

A simulator trace of relevant internal signals for this software usage is shown in Figure 9.8. The vertical line (when viewed landscape) indicates the point at which the exception is triggered in hardware. At this point *register 0* get the value 3 representing false in OCaml and *register 15* get the previous value of *register 11*. The process is delayed from the start of the divide algorithm, because the entire core is stalled, during operation of the divide unit.

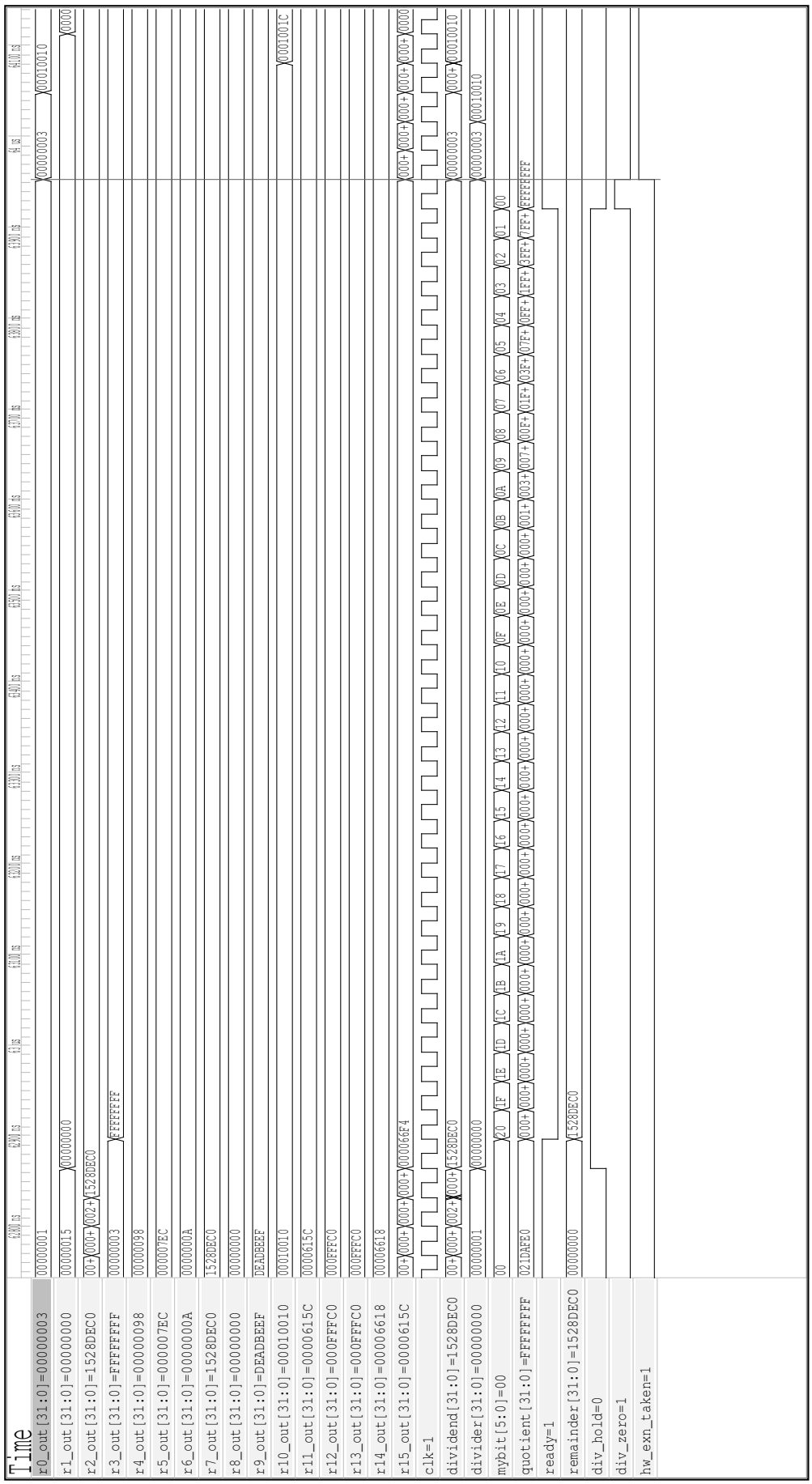


Figure 9.8: Simulation of hardware error handling in fault-tolerant processor

9.6 Summary

A robust implementation of fault-tolerance is the last milestone on this quest for improved computer reliability. By necessity, some solutions will be hardware dependent, but the use of a generic toolchain with a variety of mechanisms, all encapsulated within a type-safe environment, becomes a key enabler to round off the overall methodology. Furthermore, the performance of the improved algorithm is competitive with N-modular redundancy, whilst at the same time, providing a mechanism to detect, recover from, and retry any kind of soft error.

"We end this introduction by telling what seems to be the story how the letter λ was chosen to denote function abstraction. In Principia Mathematica [Whitehead and Russell, 1925] the notation for the function f with f(x)=2x+1 is 2x̂+1. Church originally intended to use the notation x̂.2x+1. The typesetter could not position the hat on top of the x and placed it in front of it, resulting in Λx.2x+1. Then another typesetter changed it into λx.2x+1."

[Barendregt, 1997]

10 Conclusions

It is time to summarise what has been learned in the course of the investigations.

Recapping the original contributions to knowledge, a key aspect of the methodology is a flow to extract algorithms, verified in a theorem prover (by the user's own assertions to prove certain desirable properties) to FPGA technology. This aspect is covered in detail in section 4.5. Continuity of type-safety throughout the process is a very important aspect for end-to-end verification of the user's algorithm, and, separately, ensuring robustness of the tool chain. It falls short of complete formal verification as reported by Kumar [Kumar et al., 2014], but has been demonstrated practically in dual simulation (between behavioural Verilog and state-machine). The simulation gives identical results on an FPGA platform as well as workstation operation. Importantly, the methodology uses verified memory bounds throughout, thus defeating the most important weapon of the third-party attacker. In addition, the toolchain does not itself make use of legacy tools that rest on unsafe foundations.

The use of a custom library to avoid the requirement for low-level type-unsafe algorithms is considered a contribution. In this case, the maintenance of type-safety is clear from inspection, along with the appropriate demonstrator working in hardware. As a part of this methodology, the customisation of an OCaml backend, to target behavioural and state-machine based Verilog, is considered a contribution. Although primarily an engineering task, this backend is a necessary step to demonstrate the overall goal. The concentration on embedded computing is deliberate, since it is limited resource machines which make up the vast majority of global computing, and which have the greatest benefit in greater operational and design safety.

To take advantage of theoretical advances, a computer must be a robust model of the underlying algorithms, hence the contribution of the fault-tolerance chapter and associated handling of soft errors and other exceptions. The exact formulation will be application dependent. Again, the existence of a type-safe toolchain, that can take in Verilog, and convert in a customisable way to a robust fault-tolerant netlist, is demonstrated as a contribution.

It has been shown that it is meaningful to talk about static type-safety as a technique

that is relevant right down to FPGA fabric level. This work demonstrates that functional programming is a suitable paradigm for embedded use, where memory usually is limited, and that a large quantity of type-unsafe imperative code is not necessary to support bare-metal operation, which in turn enables easier analysis of reliability issues. It has also been shown that the functional flow is compatible with FPGA N-modular redundancy and, in addition to this, the reliability flow presented here can provide explicit notification of faults arising in the low-level logic.

At the end of the research journey, the correspondence to the original research questions is still close. It can be said that there is no need for an imperative layer, addressing section 1.3, question (i). Section 1.3, question (iii) is addressed by the strict type-safety maintained between user and system processes. It would be grandiose to describe the current facilities as an operating system; a hardware abstraction layer might be a better label. Nevertheless, a framework is available to implement a more elaborate system such as *unikernels* [Madhavapeddy et al., 2013]. The prevalence of FPGA technology makes it easy to make elaborate hardware designs, at relatively low cost. By ASIC standards, the performance is low and the power consumption is high. However, once the design is finalised, a choice of technologies, such as hard-wiring [Hur et al., 2012] or application-specific standard product (ASSP) [Takahashi and Goetz, 2004], are available to meet a certain volume/price/power consumption compromise, thus addressing section 1.3 question (iv).

The one question that is not addressed is section 1.3 question (ii) the possibility of hardware tagging, though the use of dual-rail logic could be considered a step in that direction. After investigation, it was discovered that most systems discard type-safety information as redundant, once type-inference has completed. Special treatment would also be required for situations where a non-ground type is specified [Pottier and Rémy, 2005]. For example, a function that takes the head of a list does not know in advance what kind of list will be passed to it, hence run-time checking adds little in this situation. A hardware scheme that only applies in some situations would be complicated and less worthwhile.

Of necessity, a practical apparatus of this kind must be slower, and have a smaller memory capacity, than a typical computer of the day. The usual stress test, of making the computer compile its own toolchain, would not be possible at present, partly due to a lack of operating system infrastructure. Many of the components are available, however, they would just need modifying to avoid reliance on type-unsafe infrastructure. This is not strictly a limitation of the methodology, merely a limitation of convenient hardware emulation mechanisms. None of the techniques presented are limited to FPGA targets. With sufficient justification, an ASIC version could be produced. The requirement that software programs meet the syntax requirements of Coq is more of a practical than a theoretical one. Nevertheless, this would practically limit the technique to being used solely for new projects, as opposed to adapting legacy projects.

10.1 Limitations and Further work

A fully worked out implementation of the hardware garbage collection algorithm presented in chapter 8 would significantly increase the power of the demonstrator, and with the help of a keyboard, enable a fully-interactive REPL.

Likewise, an extension to implement program and data operation in external dynamic memory would be welcome, to allow much larger programs to be compiled. Eventually the operation of converting the linked executable in one piece would become too inefficient, and a separate compilation, operating on the individually compiled modules, would be required.

Eventually it should be possible to fully mechanically verify the flow, from Coq to FPGA for semantic equivalence. This would be a big step forward, and many of the components are already in place.

The apparatus developed in this work incorporates heterogenous parallel processing, in the sense that, the video frame buffer is initialised from HLS software, and operates asynchronously to the functional programming. As a further exercise and demonstration, it would be fruitful to offer networking, perhaps using the built-in Xilinx media access controller, and appropriate network stack handling, as prototyped in the style of Mirage [Madhavapeddy et al., 2013]. A significant difference would be the replacement of interrupts by messages, an optimisation that is generally not possible in conventional architectures, due to hardware limitations. The improvement in verifiability that would result from this change, would be a worthwhile step-forward in itself.

References

Ahmad, P. [2011]. HDL Analyzer and Netlist Architect. URL http://sourceforge.net/projects/sim-sim/. 15

Allan, R. [2010]. *Computing Grand Challenges*. Science and Technology Facilities Council. URL https://epubs.stfc.ac.uk/work/50400. 6, 12

Anderson, J. [2010]. Using statistical models with duplication and compare for reduced cost FPGA reliability. *In: IEEE Aerospace Conference*, pp. 1–8. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5446660. 68

Appel, A. [2011]. Verified software toolchain. *Programming Languages and Systems*, volume 1(March), 1–17. URL http://www.springerlink.com/index/T30254025X453R32.pdf. 13

ARM [2007]. RealView Development Suite. URL http://infocenter.arm.com/help/topic/com.arm.doc.dui0255l/DUI0255L_getting_started.pdf. 46

Asadi, G. and Tahoori, M.B. [2005]. Soft Error Rate Estimation and Mitigation for SRAM-Based FPGAs. *In: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pp. 149–160. Association for Computing Machinery, New York, NY, USA. URL http://portal.acm.org/citation.cfm?id=1046212. 2

Aspinall, D. [2000]. Proof General : A Generic Tool for Proof Development. *In:* S. Graf and M. Schwartzbach, eds., *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785, pp. 38–43. Springer Berlin Heidelberg. URL http://link.springer.com/chapter/10.1007/3-540-46419-0_3. 13

Bacon, D.F.; Cheng, P. and Shukla, S. [2013]. And then there were none: a stall-free real-time garbage collector for reconfigurable hardware. *Communications of the ACM*, volume 56(December), 101–109. URL http://dl.acm.org/citation.cfm?id=2534726. 72, 75

Barendregt, H. [1997]. The Impact of the Lambda Calculus in Logic and Computer Science. *The Bulletin of Symbolic Logic*, volume 3(2), 181–215. URL http://www.jstor.org/stable/421013. 31, 101

Barham, P.; Dragovic, B.; Fraser, K.; Hand, S.; Harris, T.; Ho, A.; Neugebaur, R.; Pratt, I. and Warfield, A. [2003]. Xen and the art of virtualization. *SIGOPS Operating Systems Review*, volume 37(5), 164–177. URL http://www.cs.ubc.ca/~andy/papers/2003-sosp-xen.pdf. 4

BEA [2012]. Final Report - AF447. Technical Report June 2009, Bureau d'Enquetes et D'Analyses. URL http://www.bea.aero/docspa/2009/f-cp090601.en/pdf/f-cp090601.en.pdf. 70

Beckhoff, C.; Koch, D. and Torresen, J. [2011]. The Xilinx Design Language (XDL): Tutorial and use cases. *In: 6th International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC)*, pp. 1–8. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5981545. 69

Benediktsson, O.; Hunter, R.B. and McGettrick, A.D. [2001]. Processes for software in safety critical systems. *Software Process: Improvement and Practice*, volume 6(1), 47–62. URL http://www3.interscience.wiley.com/journal/78002446/abstract. 6

Benton, N. and Hur, C. [2009]. Biorthogonality, step-indexing and compiler correctness. *In: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pp. 97–108. ACM. URL http://portal.acm.org/citation.cfm?id=1596567. 13

Benton, N. and Hur, C. [2010]. Realizability and compositional compiler correctness for a polymorphic language. Technical report, MSR-TR-2010-62, Microsoft Research. URL http://www.mpi-sws.org/~gil/publications/cccmsrtr.pdf. 13

Benton, N. and Tabareau, N. [2009]. Compiling functional types to relational specifications for low level imperative code. *In: Proceedings of the 4th international workshop on Types in language design and implementation*, pp. 3–14. URL http://portal.acm.org/citation.cfm?doid=1481861.1481864. 13

Berkeley, U. [1992]. Berkeley logic interchange format (BLIF). *Oct Tools Distribution*, pp. 1–11. URL https://www.ece.cmu.edu/~ee760/760docs/blif.pdf. xii, 15

Bertot, Y. [2006]. Coq in a Hurry. *arXiv preprint cs/0603118*, volume 1(April), 1–43. URL http://arxiv.org/abs/cs/0603118. xii, 4

Bhasker, J.; Berman, V.; Bishop, D.; Gerousis, V.; Hejna, D.; Quayle, M.; Sarkar, A.; Smith, D.; Trivedi, Y. and Vora, R. [2002]. IEEE Standard for Verilog Register Transfer Level Synthesis. URL http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1146718. xi, xiii, 35

Bhatnagar, H. [2001]. *Advanced ASIC Chip Synthesis Using Synopsys(R) Design Compiler(TM) Physical Compiler(TM) and PrimeTime(R)*. Kluwer Academic Publishers. URL https://books.google.co.uk/books?isbn=0792376447. 88

Blanchet, B. [1998]. Escape analysis: correctness proof, implementation and experimental results. *In: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 25–37. URL http://citeseer.ist.psu.edu/viewdoc/download;jsessionid=93429F6C5BB57607E1A73776DC7A9039?doi=10.1.1.47.386&rep=rep1&type=pdf. 22

Blumlein, A.D. [1936]. Patent 482740: Long Tailed Pair. URL http://worldwide.espacenet.com/publicationDetails/biblio?DB=worldwide.espacenet.com&II=0&ND=3&adjacent=true&locale=en_EP&FT=D&date=19380404&CC=GB&NR=482740A&KC=A. 16

Boldo, S. and Melquiond, G. [2011]. Flocq: A unified library for proving floating-point algorithms in Coq. *In: Proceedings - Symposium on Computer Arithmetic*, pp. 243–252. URL https://www.lri.fr/~melquion/doc/11-arith20-article.pdf. xiii, 19

Bourne, C. [2004]. Future Contingents, Non-Contradiction and the Law of Excluded Middle Muddle. *Analysis*, volume 64(2), 1–11. URL http://onlinelibrary.wiley.com/doi/10.1111/j.1467-8284.2004.00471.x/abstract. 11

Boyer, R.S. and Moore, J.S. [1984]. A mechanical proof of the unsolvability of the halting problem. *Journal of the ACM*, volume 31(July), 441–458. URL http://dl.acm.org/citation.cfm?id=1882. x, 3

Brady, E. and Hammond, K. [2006]. A verified staged interpreter is a verified compiler. *In: Proceedings of the 5th international conference on Generative programming and component engineering*, pp. 111–120. ACM. URL http://portal.acm.org/citation.cfm?id=1173724. 13

Brayton, R. and Mishchenko, A. [2010]. ABC: An academic industrial-strength verification tool. *In: Computer Aided Verification*, pp. 24–40. Springer. URL http://www.springerlink.com/index/R03371760P68202U.pdf. xii, 15

Bryant, R.E. and Hallaron, D.R.O. [2011]. Verilog Implementation of a Pipelined Y86 Processor. Technical report, Carnegie Mellon University. URL http://csapp.cs.cmu.edu/public/waside/waside-verilog.pdf. 57, 61

Carrol, J.F. [2009]. BYU-LANL Triple Modular Redundancy Usage Guide. Published Online at sourceforge by byuediftools. URL http://sourceforge.net/projects/byuediftools/files/byuediftools. 21, 68, 69, 86

Cheney, C. [1970]. A nonrecursive list compacting algorithm. *Communications of the ACM*, volume 13, 677–678. URL http://dl.acm.org/citation.cfm?id=362798. 72

Chinnery, D. and Keutzer, K. [2002]. *Closing the Gap between ASIC & Custom: Tools and Techniques for High-Performance ASIC Design*. Kluwer Academic Publishers. URL http://www.springer.com/gp/book/9781402071133. xii, 11

Chisnall, D.; Watson, R.N.M.; Moore, S.W.; Davis, B. and Neumann, P.G. [2015]. Beyond the PDP-11 : Architectural support for a memory-safe C abstract machine. *In: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 117–130. 17

Chlipala, A. [2007a]. A certified type-preserving compiler from lambda calculus to assembly language. *ACM Sigplan Notices*, volume 42(6), 54. URL http://portal.acm.org/citation.cfm?doid=1273442.1250742. 13

Chlipala, A. [2007b]. Lambda Tamer. URL http://ltamer.sourceforge.net/. 13

Chlipala, A. [2010]. A verified compiler for an impure functional language. *ACM Sigplan Notices*, volume 45(1), 93. URL http://portal.acm.org/citation.cfm?doid=1707801.1706312. 13

Chmelaf, E. [2003]. Fpga interconnect delay fault testing. *International Test Conference 2003 Proceedings ITC 2003*, pp. 1239–1247. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1271113. 67

Church, A. [1936]. An Unsolvable Problem of Elementary Number Theory. *American journal of mathematics*, volume 58(2), 345–363. URL http://www.jstor.org/stable/2371045. 3

Cong, J.; Liu, B.; Luo, G. and Prabhakar, R. [2012]. Towards layout-friendly high-level synthesis. *In: Proceedings of the 2012 ACM international symposium on International Symposium on Physical Design*, pp. 165–172. URL http://dl.acm.org/citation.cfm?doid=2160916.2160952. xiii, 15

Cong, J.; Liu, B.; Neuendorffer, S.; Noguera, J.; Vissers, K. and Zhang, Z. [2011]. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 30(4), 473–491. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5737854. xiii, 15

Copeland, B. and ... [2005]. Proposed Electronic Calculator (1945). *In: Alan Turings' Electronic Brain*. OUP. URL https://global.oup.com/academic/product/alan-turings-electronic-brain-9780199609154?cc=gb&lang=en&. 16

da Costa, N.C.A. [2012]. Gödel's Incompleteness Theorems and Physics. *Principia: an international journal of epistemology*, volume 15(3). URL http://dialnet.unirioja.es/descarga/articulo/3974057.pdf. 11

Cousineau, G.; Curien, P.L. and Mauny, M. [1987]. The Categorical Abstract Machine. *Science of Computer Programming*, volume 8(2), 173–202. URL http://linkinghub.elsevier.com/retrieve/pii/0167642387900207. 31, 34

Crate, D. [1996]. On the use of Verilog HDL in the conversion of existing hardware designs to newer technology. *In: Proceedings. IEEE International Verilog HDL Conference*, 1, pp. 39–44. IEEE Comput. Soc. Press. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=496016. 35

Cuoq, P.; Kirchner, F.; Yakobowski, B.; Labbé, S.; Thuy, N. and Hilsenkopf, P. [2012]. Formal verification of software important to safety using the FRAMA-C tool suite. *In: 8th International Topical Meeting on Nuclear Plant Instrumentation, Control, and Human-Machine Interface Technologies 2012, NPIC and HMIT 2012: Enabling the Future of Nuclear Energy*, volume 1, pp. 40–51. URL http://www.scopus.com/inward/record.url?eid=2-s2.0-84880459602&partnerID=40&md5=1d3c00808fa08050a839db3d55c40138. 19

CVE-2014-6271 [2014]. CVE-2014-6271. URL http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-6271. 3

Danvy, O. [2005]. A Rational Deconstruction of Landin's SECD Machine. *In: Implementation and Application of Functional Languages*, October, pp. 52–71. Springer. URL http://www.springerlink.com/index/ydtj3tvyerqam4dk.pdf. 31

Dargaye, Z. [2009]. *Verification formelle d'un compilateur optimisant pour langages fonctionnels*. Ph.D. thesis, Ecole doctorale Sciences Mathematiques de Paris Centre. URL http://gallium.inria.fr/~dargaye/pub/main.pdf. 14, 25, 34

Dargaye, Z. and Leroy, X. [2010]. A verified framework for higher-order uncurrying optimizations. *Higher-Order and Symbolic Computation*, volume 22(3), 199–231. URL http://link.springer.com/10.1007/s10990-010-9050-z. 13

Delatre, J.L. [2012]. The LRTT parser. Published Online at kevembuangga. URL http://www.kevembuangga.com/lrtt/. 77

Delgado Kloos, C. and Breuer, P.T. [1995]. Formal Semantics for VHDL. *Journal of Computer and System Sciences*, volume 76, 663–685. 57

Dershowitz, N. and Falkovich, E. [2012]. A Formalization and Proof of the Extended Church-Turing Thesis -Extended Abstract-. 1207.7148, URL http://arxiv.org/pdf/1207.7148. 3

Dobai, R. and Sekanina, L. [2013]. Towards evolvable systems based on the Xilinx Zynq platform. *In: Proceedings of the 2013 IEEE International Conference on Evolvable Systems, ICES 2013 - 2013 IEEE Symposium Series on Computational Intelligence, SSCI 2013*, pp. 89–95. URL http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6613287. 69

Donnelly, C. and Stallman, R. [2006]. Bison The Yacc-compatible Parser Generator. URL http://www.gnu.org/software/bison/manual/bison.pdf. 57

Durumeric, Z. and Kasten, J. [2014]. The matter of Heartbleed. *ACM Internet ....* URL https://nebelwelt.net/publications/14IMC/heartbleed-imc14.pdf. 3

Edmonds, L. [2000]. Proton SEU cross sections derived from heavy-ion test data. *IEEE Transactions on Nuclear Science*, volume 47(5), 1713–1728. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=890997. 56

Filliatre, J.C. [2010]. Binary decision diagrams (BDDs). URL http://www.lri.fr/~filliatr/ftp/ocaml/bdd/. xii

Filliâtre, J.C. and Letouzey, P. [2012]. The Coq Proof Assistant Reference Manual. Published Online by INRIA. URL http://coq.inria.fr/distrib/current/files/refman.tar.gz. 20

Findler, R.B.; Flanagan, C.; Flatt, M.; Krishnamurthi, S. and Felleisen, M. [1997]. DrScheme: A pedagogic programming environment for Scheme. *In:* H. Glaser; P. Hartel and H. Kuchen, eds., *Programming Languages: Implementations, Logics, and Programs*, pp. 369–388. Springer Berlin Heidelberg. URL http://link.springer.com/chapter/10.1007/BFb0033856. xiii, 14

Garrigue, J. [2010]. A certified implementation of ML with structural polymorphism. *Programming Languages and Systems*, pp. 360–375. URL http://www.springerlink.com/index/V51V4M674313840V.pdf. 18, 35

Gates, B. [1989]. University of Waterloo Computer Science Club. URL http://csclub.uwaterloo.ca/media/1989BillGatesTalkonMicrosoft.html. 71

Gödel, K. [1931]. On formally undecidable propositions of Principia Mathematica and related systems. *Monatshefte fur Mathematik und Physik,*, pp. 1–75. URL http://people.ucalgary.ca/~rzach/static/godel1931.pdf. 3, 11

Gordon, M. [1995]. The Semantic Challenge of Verilog HDL. *The Proceedings of Tenth Annual IEEE Symposium on Logic in Computer Science*, pp. 136–145. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=523251. 57

Gordon, M. [2000]. From LCF to HOL: a short history. *Proof, Language, and Interaction*, pp. 1–16. URL http://books.google.com/books?hl=en&lr=&id=g8DEO9DwmZoC&oi=fnd&pg=PA169&dq=From+LCF+to+HOL+:+a+short+history&ots=_k41copy4N&sig=lbIk3NTnqhq8-2Pjipjy_q11Yk8. 12

Grattan-Guinness, I. [1978]. How Bertrand Russell discovered his paradox. *Historia mathematica*, volume 5(2), 127–137. URL http://www.sciencedirect.com/science/article/pii/0315086078900460. 11

Greaves, D. [2003]. Hardware and Embedded Software Synthesis from Executable Specifications. URL http://www.cl.cam.ac.uk/~djg11/wwwhpr. 35

Greaves, D. and Gordon, D. [2006]. Using Simple Pushlogic. *In: WEBIST (1)*. URL http://www.cl.cam.ac.uk/~djg11/ao/heating.pdf. 33

Gribeiro, R. [2012]. sf-solutions. Published Online at github by rodrigogribeiro. URL https://github.com/rodrigogribeiro/sf-solutions. 25, 32

Guttman, J.D.; Ramsdel, J.D. and Swarup, V. [1995]. The VLISP verified Scheme System. *Lisp and Symbolic Computation*, volume 8, 33–110. URL http://link.springer.com/article/10.1007/BF01128407. 12

Hamilton, J. [2010]. Snowflake-OS. URL http://code.google.com/p/snowflake-os/. 4

Han, J. and Orshansky, M. [2013]. Approximate computing: An emerging paradigm for energy-efficient design. *In: 18th IEEE European Test Symposium*, pp. 1–6. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6569370. 3

HASP [2010]. The Habit Programming Language : The Revised Preliminary Report. Technical report, Portland State University. URL http://hasp.cs.pdx.edu/habit-report-Nov2010.pdf. 14

Hawblitzel, C.; Huang, H.; Wittie, L. and Chen, J. [2007]. A garbage-collecting typed assembly language. *In: Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, p. 41. URL http://portal.acm.org/citation.cfm?doid=1190315.1190323. 75

Hoare, T. [2003]. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, volume 50(1), 63–69. URL http://portal.acm.org/citation.cfm?doid=602382.602403. 12

Huet, G.; Kahn, G. and Paulin-Mohring, C. [2002]. The Coq Proof Assistant A Tutorial. Technical report, INRIA. URL https://hal.inria.fr/inria-00069918. xii, 4

Hughes, J. [1989]. Why Functional Programming Matters. *The Computer Journal*, volume 32(2), 98–107. URL http://comjnl.oupjournals.org/cgi/doi/10.1093/comjnl/32.2.98. 6

Hur, C. and Dreyer, D. [2011]. A Kripke logical relation between ML and assembly. *ACM SIGPLAN Notices*, volume 46(1), 133–146. URL http://portal.acm.org/citation.cfm?id=1926402. 14

Hur, J.Y.; Goossens, K.; Mhamdi, L. and Wahlah, M.A. [2012]. Comparative analysis of soft and hard on-chip interconnects for field-programmable gate arrays. *IET Computers & Digital Techniques*, volume 6(July), 396–405. URL http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6403643. 102

Iyoda, J. [2007]. Translating HOL functions to hardware. Technical report, University of Cambridge. URL https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-682.pdf. 13

Jaber, G. [2010]. Krivine realizability for compiler correctness. Technical report, dumas.ccsd.cnrs.fr. URL http://dumas.ccsd.cnrs.fr/dumas-00530710/. 13, 14, 34

Jacobs, B. [2013]. The Essence of Coq as a Formal System. Technical report, Stellenbosch University. URL http://people.cs.kuleuven.be/~bart.jacobs/coq-essence.pdf. 3

Jamieson, P.; Kent, K. and Gharibian, F. [2010]. Odin II - An Open-source Verilog HDL Synthesis Tool for CAD Research. *In: 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 149 – 156. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5474055. 15

Jo, S.H.; Chang, T.; Ebong, I.; Bhadviya, B.B.; Mazumder, P. and Lu, W. [2010]. Nanoscale memristor device as synapse in neuromorphic systems. *Nano letters*, volume 10(4), 1297–301. URL http://www.ncbi.nlm.nih.gov/pubmed/20192230. 2

Johnson, J.; Howes, W.; Wirthlin, M.; McMurtrey, D.L.; Caffrey, M.; Graham, P. and Morgan, K. [2008]. Using Duplication with Compare for On-line Error Detection in FPGA-based Designs. *In: 2008 IEEE Aerospace Conference*, pp. 1–11. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4526470. xii, 68

Joshi, R. and Holzmann, G.J. [2007]. A mini challenge: build a verifiable filesystem. *Formal Aspects of Computing*, volume 19(2), 269–272. URL http://www.springerlink.com/index/10.1007/s00165-006-0022-3. 14

Jourdan, J.; Pottier, F. and Leroy, X. [2012]. Validating LR (1) parsers. *In: Proceedings of the 21st European conference on Programming Languages and Systems*, pp. 397–416. URL http://link.springer.com/chapter/10.1007/978-3-642-28869-2_20. 20

Kimmitt, J.R.R. [2014]. kimmitt/ocaml-experiment.git. URL https://bitbucket.org/kimmitt/ocaml-experiment.git. 21

Kimmitt, J.R.R. [2015]. gnusynthesis. URL https://bitbucket.org/jrrk/gnusynthesis. 90

Kimmitt, J.R.R.; Greaves, D.J. and Cirstea, M. [2015]. A toolchain for safety-critical embedded processor programming using FPGAs. *In: Proceedings of the 2015 IEEE International Conference on Industrial Informatics*. Cambridge. URL http://ieeexplore.ieee.org/servlet/opac?punumber=1001443. 18

Kimmitt, J.R.R.; Wilson, G. and Greaves, D. [2012]. A novel design flow for fault-tolerant computing. *2012 4th Computer Science and Electronic Engineering Conference (CEEC)*, pp. 35–40. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6375375. 21, 68

Kondratyev, A. and Lwin, K. [2002]. Design of Asynchronous Circuits Using Synchronous CAD Tools. *IEEE Design & Test of Computers*, volume 19(4), 107–117. URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1018139. 16, 17

Koprowski, A. and Binsztok, H. [2010]. TRX: A formally verified parser interpreter. *Programming Languages and Systems*, volume 7, 345–365. URL http://arxiv.org/pdf/1105.2576.pdf. 20

Krivine, J. [2009]. Realizability in classical logic. *Panoramas et synthèses*, volume 27, 197–229. URL http://hal.inria.fr/hal-00154500/. 14

Kumar, R.; Myreen, M.; Norrish, M. and Owens, S. [2014]. CakeML: A verified implementation of ML. *In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, volume 1, pp. 179–191. URL http://dl.acm.org/citation.cfm?id=2535841. 14, 101

LaFrieda, C.; Hill, B. and Manohar, R. [2010]. An Asynchronous FPGA with Two-Phase Enable-Scaled Routing. *2010 IEEE Symposium on Asynchronous Circuits and Systems*, pp. 141–150. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5476972. 56

Landin, P. [1964]. The mechanical evaluation of expressions. *The Computer Journal*, volume 6(4), 308. URL http://comjnl.oxfordjournals.org/content/6/4/308.short. xiii, 14, 31

Leroy, X. [2009a]. A Formally Verified Compiler Back-end. *Journal of Automated Reasoning*, volume 43(4), 363–446. URL http://www.springerlink.com/index/10.1007/s10817-009-9155-4. xii, 13

Leroy, X. [2009b]. Formal verification of a realistic compiler. *Communications of the ACM*, volume 52(7), 107. URL http://portal.acm.org/citation.cfm?doid=1538788.1538814. 25

Leroy, X. [2014]. Compiler Verification for fun and profit. *In: Formal Methods in Computer-Aided Design (FMCAD 2014)*, p. 9. URL http://repositories.lib.utexas.edu/bitstream/handle/2152/26151/FMCAD_2014.pdf?sequence=2#page=21. 13

Lesea, A. [2008]. Continuing Experiments of Atmospheric Neutron Effects on Deep Submicron Integrated Circuits. *WP286 (v1. 0), Xilinx Inc*, volume 2. URL http://www.xilinx.com/support/documentation/white_papers/wp286.pdf. 56

Lesea, A.; Drimer, S.; Fabula, J.; Carmichael, C. and Alfke, P. [2005]. The rosetta experiment: atmospheric soft error rate testing in differing technology FPGAs. *IEEE Transactions on Device and Materials Reliability*, volume 5(3), 317–328. URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1545892. 56

Madhavapeddy, A.; Mortier, R.; Rotsos, C.; Scott, D.; Singh, B.; Gazagnaire, T.; Smith, S.; Hand, S. and Crowcroft, J. [2013]. Unikernels: library operating systems for the cloud. *In: Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, pp. 461–472. URL http://anil.recoil.org/papers/2013-asplos-mirage.pdfhttp://dl.acm.org/citation.cfm?id=2499368.2451167. 4, 39, 102, 103

Madhavapeddy, A.; Mortier, R.; Sohan, R.; Gazagnaire, T.; Hand, S.; Deegan, T.; Mcauley, D. and Crowcroft, J. [2010]. Turning Down the LAMP : Software Specialisation for the Cloud. *In: Proceedings of the 2nd USENIX conference on Hot topics in cloud computing, HotCloud*, p. 11. URL http://anil.recoil.org/papers/2010-hotcloud-lamp.pdf. 40

Maeda, T. and Yonezawa, A. [2009]. Writing an OS Kernel in a Strictly and Statically Typed Language. *Formal to Practical Security*, volume 5458, 181–197. URL http://www.springerlink.com/content/347631g5h2448178/. 6

Maxim, A. and Gheorghe, M. [2001]. A novel physical based model of deep-submicron CMOS transistors mismatch for Monte Carlo SPICE simulation. *In: The 2001 IEEE International Symposium on Circuits and Systems*, volume 5, pp. 511–514. URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=922097. 55

McCreight, A.E. [2008]. *The mechanized verification of garbage collector implementations*. Ph.D. thesis, Yale University. URL http://portal.acm.org/citation.cfm?id=1627134. 71

McCreight, A.E.; Chevalier, T. and Tolmach, A. [2010]. A certified framework for compiling and executing garbage-collected languages. *In: Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, volume 45, pp. 273–284. ACM. URL http://portal.acm.org/citation.cfm?id=1863584. 14

McCreight, A.E.; Shao, Z.; Lin, C. and Li, L. [2007]. A general framework for certifying garbage collectors and their mutators. *In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 468–479. URL http://portal.acm.org/citation.cfm?id=1250788. 14, 75

Mehnert, H. and Meršinjak, D. [2014]. Transport Layer Security purely in OCaml. *In: The OCaml Users and Developers Workshop*. URL https://ocaml.org/meetings/ocaml/2014/ocaml2014_4.pdf. 4, 39

Meijer, E. and Drayton, P. [2004]. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. *In: Workshop on Revival of Dynamic Languages*. URL https://www.ics.uci.edu/~lopes/teaching/inf212W12/readings/rdl04meijer.pdf. 12

Meredith, P. and Katelman, M. [2010]. A formal executable semantics of Verilog. *In: 8th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, pp. 179–188. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5558634. 35

Miller, G. and Carmichael, C. [2008]. Single-Event Upset Mitigation Design Flow for Xilinx FPGA PowerPC Systems. Technical report, Xilinx. URL http://application-notes.digchip.com/077/77-43116.pdf. xiii, 60, 62

Minamide, Y.; Morrisett, G. and Harper, R. [1996]. Typed Closure Conversion. *In: POPL 96 Proceedings of the 23rd ACM SIGPLANSIGACT symposium on Principles of programming languages*, pp. 271–283. ACM. URL http://portal.acm.org/citation.cfm?id=237721.237791&coll=Portal&dl=ACM&CFID=85333289&CFTOKEN=95953107. 13

Myreen, M.; Owens, S. and Kumar, R. [2013]. Steps Towards Verified Implementations of HOL Light. *In: 4th International Conference on Interactive Theorem Proving*, pp. 490–495. URL http://www.cl.cam.ac.uk/~mom22/itp13.pdf. 14

Narboux, J. [2010]. CoqIDE. Technical report, INRIA. URL http://coq.inria.fr/V8.1/refman/Reference-Manual016.html. 13

Naylor, M. and Runciman, C. [2012]. The Reduceron reconfigured and re-evaluated. *Journal of Functional Programming*, volume 22(4-5), 574–613. URL http://www.journals.cambridge.org/abstract_S0956796812000214. 14, 31, 35

Nitta, I.; Shibuya, T.; Homma, K.; Laboratories, F.; Limited, F. and Limited, F.V. [2007]. Statistical Static Timing Analysis Technology. *Fujitsu Science Technical Journal*, volume 43(4), 516–523. URL http://www.fujitsu.com/downloads/MAG/vol43-4/paper18.pdf?q=statistical-static-timing-analysis-how-simple-can-we-get. 56

Oliver, I. [2006]. A Demonstration of Specifying and Synthesising Hardware using B and Bluespec. *In: Forum on Specification and Design Languages*. URL http://rodin.cs.ncl.ac.uk/Publications/. 35

Parshin, O. [2004]. Specification and verification of the ARM6 microprocessor in HOL. *In: State of the Art of Formal Hardware Verification*. URL http://www-wjp.cs.uni-saarland.de/lehre/seminar/ss03/reports/pereZ.pdf. 45

Pierce, B.C.; Casinghino, C.; Gaboardi, M.; Greenberg, M.; Hritcu, C.; Sjöberg, V. and Yorgey, B. [2012]. Software Foundations. Published Online at upenn by bcpierce. URL http://www.cis.upenn.edu/~bcpierce/sf/current/toc.html. 25, 26, 32, 118

Pottier, F. and Rémy, D. [2005]. The Essence of ML Type Inference. *Advanced Topics in Types and Programming Languages*, pp. 389—-489. URL http://gallium.inria.fr/~fpottier/publis/emlti-final.pdf. 102

Quinn, H.; Morgan, K.; Graham, P.; Krone, J. and Caffrey, M. [2007]. A review of Xilinx FPGA architectural reliability concerns from Virtex to Virtex-5. *In: 9th European Conference on Radiation and Its Effects on Components and Systems*, pp. 1–8. URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5205533. 56

Razafindraibe, A.; Maurine, P.; Robert, M. and Renaudin, M. [2006]. Security evaluation of dual rail logic against DPA attacks. *In: IFIP International Conference on Very Large Scale Integration*, pp. 181 – 186. URL http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4107626. 17

Remy, D. and Vouillon, J. [1998]. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, volume 4(1), 27–50. URL http://doi.wiley.com/10.1002/%28SICI%291096-9942%281998%294%3A1%3C27%3A%3AAID-TAPO3%3E3.0.CO%3B2-4. xiii, 8

Rose, J.; Luu, J.; Yu, C.W.; Densmore, O.; Goeders, J.; Somerville, A.; Kent, K.B.; Jamieson, P. and Anderson, J. [2012]. The VTR project: Architecture and CAD for FPGAs from Verilog to Routing. *In: Proceedings of the ACMSIGDA international symposium on Field Programmable Gate Arrays*, pp. 77–86. ACM Press. URL http://dl.acm.org/citation.cfm?id=2145694.2145708. 15

Runyan, W.M. [1981]. Why did Van Gogh cut off his ear? The problem of alternative explanations in psychobiography. *Journal of personality and social psychology*, volume 40(6), 1070–7. URL http://www.ncbi.nlm.nih.gov/pubmed/7021798. 2

Santifort, C. [2013]. Amber Open Source Project Amber 2 Core Specification. Technical report, Opencores.org. URL http://opencores.org/websvn,filedetails?repname=amber&path=/amber/trunk/doc/amber-core.pdf. 45, 48, 134

Scowen, R. [1998]. Extended BNF-a generic base standard. *Software Engineering Standards Symposium*, volume 3(1), 6–2. URL http://www.cl.cam.ac.uk/~mgk25/iso-14977-paper.pdf. 57

Seligman, E. and Sokolover, A. [2006]. Cadence Conformal LEC The Intel Experience Moore's Law and Us. Technical report, Cadence. URL https://www.cadence.com/rl/Resources/conference_papers/dtp_cdnliveemea2006_itayarom.pdf. 54

Singh, S. and Greaves, D.J. [2008]. Kiwi: Synthesis of FPGA Circuits from Parallel Programs. *In: 16th International Symposium on Field-Programmable Custom Computing Machines*, pp. 3–12. URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4724884. 16, 35

Slind, K. and Norrish, M. [2008]. A brief overview of HOL4. *Theorem Proving in Higher Order Logics*, pp. 28–32. URL http://link.springer.com/chapter/10.1007/978-3-540-71067-7_6. xiii, 12

Smith, J.E. [1978]. Strongly Fault Secure Logic Networks. *IEEE Transactions on Computers*, volume C-27(6), 491–499. URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1675139. 70

Snyder, W. [2010]. Verilator-3.805. Technical report, Veripool.org. URL http://www.veripool.org/ftp/verilator_doc.pdf. 57

Sokolov, D.; Murphy, J.; Bystrov, A. and Yakovlev, A. [2005]. Design and Analysis of Dual-Rail Circuits for Security Applications. *IEEE Transactions on Computers*, volume 54(4), 449–460. URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1401864. 17

Sretasereekul, N. and Nanya, T. [2003]. Eliminating isochronic-fork constraints in quasi-delay-insensitive circuits. *IIEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, volume 86(4), 900–907. URL http://search.ieice.org/bin/summary.php?id=e86-a_4_900. 16

Stanford, P. and Mancuso, P. [1989]. EDIF Electronic Design Interchange Format, Reference Manual for Version 2 0 0. Technical report, Electronic Industries Association. URL http://www.amazon.com/Electronic-Design-Interchange-Format-Version/dp/0790800004. xiii, 57

Studboy-ga [2002]. Verilog Multiplier/Divider. URL http://answers.google.com/answers/threadview/id/109219.html. 97

Sutherland, S. and Mills, D. [2014]. Can My Synthesis Compiler Do That? *In: Design and Verification Conference*. URL http://www.sutherland-hdl.com/papers/2014-DVCon_ASIC-FPGA_SV_Synthesis_paper.pdf. 15, 88

Swade, D. [2005]. The Construction of Charles Babbage's Difference Engine No. 2. *IEEE Annals of the History of Computing*, volume 27(3), 70–78. URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1498720. 11

Swift, G.; Carmichael, C. and Allen, G. [2008]. Virtex-4QV static SEU characterization summary. *JPL Publication 08-16*. 56, 69

Takahashi, T. and Goetz, J. [2004]. Implementation of complete AC servo control in a low cost FPGA and subsequent ASSP conversion. *In: Nineteenth Annual IEEE Applied Power Electronics Conference and Exposition*, volume 1, pp. 565–570. URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1295863. 102

ThePopCase [2015]. Idées pour envoyer des lettres amusantes à ses enfants. URL http://www.thepopcase.com/blog/courrier-amusant/. 24

Turing, A. [1936]. On computable numbers, with an application to the Entscheidungsproblem. *London Mathematical Society*, volume 42(2). URL http://plms.oxfordjournals.org/content/s2-42/1/230.full.pdf. 1

Uhlig, R.; Neiger, G.; Rodgers, D.; Santoni, A.L.; Martins, F.C.M.; Anderson, A.V.; Bennett, S.M.; Kagi, A.; Leung, F.H. and Smith, L. [2005]. Intel virtualization technology. *Computer*, volume 38(5), 48–56. URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1430631. 12

Vasyukevich, V.O. [2011]. *Asynchronous Operators of Sequential Logic: Venjunction & Sequention : Digital Circuit Analysis and Design*. Springer. URL http://asynlog.balticom.lv/Content/Files/en.pdf. 17

Von Neumann, J. [1956]. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata studies*, volume 34, 43–98. URL http://books.google.com/books?hl=en&lr=&id=oL57iECEeEwC&oi=fnd&pg=PA43&dq=Probabilistic+Logics+and+the+Synthesis+of+Reliable+Organisms+from+Unreliable+Components&ots=xvC_nfIU9-&sig=1PGkT7iEoihnXy2ucKzPDf4XabA. 1, 16

Wakerly, J. [1974]. Partially Self-Checking Circuits and Their Use in Performing Logical Operations. *IEEE Transactions on Computers*, volume C-23(7), 658–666. URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1672607. 69

Walden, D.C. [1972]. A note on Cheney's nonrecursive list-compacting algorithm. *Communications of the ACM*, volume 15(4), 275. URL http://portal.acm.org/citation.cfm?doid=361284.361300. 72

Weirich, S. [2014]. Depending on types. *In: Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pp. 241–241. URL http://dl.acm.org/citation.cfm?doid=2628136.2631168. 7, 25

Whitehead, A. and Russell, B. [1925]. *Principia Mathematica*. Cambridge University Press, 2nd edition. 11, 101

Williams, R. [2008a]. How We Found The Missing Memristor. *Spectrum, IEEE*, volume 1(december). URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4687366. 2

Williams, S. [2008b]. Icarus Verilog. URL http://sourceforge.net/projects/iverilog/. 15

Winterstein, F.; Bayliss, S. and Constantinides, G.A. [2013]. High-level synthesis of dynamic data structures: A case study using Vivado HLS. *In: International Conference on Field-Programmable Technology*, pp. 362–365. URL http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6718388. 15

Wolf, C. and Glaser, J. [2013]. Yosys - A Free Verilog Synthesis Suite. *In: Proceedings of Austrochip 2013*. URL http://www.clifford.at/yosys/. 15

Xilinx [2009a]. ML605 HARDWARE SETUP GUIDE. Technical report, Xilinx Inc. URL http://www.xilinx.com/support/documentation/boards_and_kits/xtp084.pdf. 18

Xilinx [2009b]. Xilinx UG190 Virtex-5 FPGA User Guide. Technical report, Xilinx Inc. URL http://www.xilinx.com/support/documentation/user_guides/ug190.pdf. xiii, 57

Xilinx [2010]. LogiCORE(TM) IP Soft Error Mitigation Controller. Technical report, Xilinx Inc. URL http://www.xilinx.com/support/documentation/ip_documentation/sem_ug764.pdf. 86

Xilinx [2011]. Virtex-6 Family Overview. Technical report, Xilinx Inc. URL http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf. 86

Xilinx [2012]. ML605 Hardware User Guide. Technical report, Xilinx Inc. URL http://www.xilinx.com/support/documentation/boards_and_kits/ug534.pdf. 18

Xilinx [2013]. Xilinx Command Line Tools. Technical report, Xilinx Inc. URL http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/devref.pdf. 48

Yang, J. and Hawblitzel, C. [2010]. Safe to the last instruction: automated verification of a type-safe operating system. *ACM Sigplan Notices*, volume 45(6), 99–110. URL http://portal.acm.org/citation.cfm?id=1806610. 6

Yeh, H.; Wu, C. and Huang, C. [2012]. Qutertl: towards an open source framework for rtl design synthesis and verification. *In:* Flanagan; Cormac and Konig, eds., *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7214, pp. 377–391. Springer Berlin Heidelberg, ebook edition. URL http://dvlab.ee.ntu.edu.tw/~publication/QuteRTL/QuteRTL.pdf. x, 15

Yuasa, T. [1990]. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, volume 11(3), 181–198. URL http://www.sciencedirect.com/science/article/pii/016412129090084Y. 72

Zeldovich, N.; Kannan, H.; Dalton, M. and Kozyrakis, C. [2008]. Hardware Enforcement of Application Security Policies Using Tagged Memory. *In: 8th USENIX Symposium on Operating Systems Design and Implementation*, pp. 225–240. URL http://www.usenix.org/events/osdi08/tech/full_papers/zeldovich/zeldovich_html/index.html. 6

Zuras, D.; Cowlishaw, M.; Aiken, A.; Applegate, M.; Bailey, D.; Bass, S.; Bhandarkar, D.; Bhat, M.; Bindel, D.; Boldo, S.; Canon, S.; Carlough, S.R.; Cornea, M.; Crawford, J.H.; Darcy, J.D.; Sarma, D.D.; Daumas, M.; Davis, B.; Davis, M.; Delp, D.; Demmel, J.; Erle, M.A.; H., H.A.F.; Fasano, J.; Fateman, R.; Feng, E.; Ferguson, W.E.; Fit-Florea, A.; Fournier, L.; Freitag, C.; Godard, I.; Golliver, R.A.; Gustafson, D.; Hack, M.; Harrison, J.R.; Hauser, J.; Hida, Y.; Hinds, C.N.; Hoare, G.; Hough, D.G.; Huck, J.; Hull, J.; Ingrassia, M.; James, D.V.; James, R.; Kahan, W.; Kapernick, J.; Karpinski, R.; Kidder, J.; Koev, P.; Li, R.C.; Liu, Z.A.; Mak, R.; Markstein, P.; Matula, D.; Melquiond, G.; Mori, N.; Morin, R.; Nedialkov, N.; Nelson, C.; Oberman, S.; Zimmermann, P.; Ollmann, I.; Parks, M.; Pittman, T.; Postpischil, E.; Riedy, J.; Schwarz, E.M.; Scott, D.; Senzig, D.; Sharapov, I.; Shearer, J.; Siu, M.; Smith, R.; Stevens, C.; Tang, P.; Taylor, P.J.; Thomas, J.W.; Thompson, B.; Thrash, W.; Toda, N.; Trong, S.D.; Tsai, L.; Tsen, C.; Tydeman, F.; Wang, L.K.; Westbrook, S.; Winkler, S.; Wood, A.; Yalcinalp, U. and Zemke, F. [2008]. IEEE Std 754(TM)-2008 (Revision of IEEE Std 754-1985), IEEE Standard for Floating-Point Arithmetic. URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5976968. 76

Appendix-A

This appendix contains a library of tactics and hints to support the example of section 4.5. Referring to the listing below and the example of Appendix-B, it is apparent that statements of various classes are in use. Statements of the form:

(i) Require... to import Coq libraries.

(ii) Tactic/Hint... to guide the proof assistant.

(iii) Fixpoint... roughly corresponding to ML recursive functions.

(iv) Theorem/Lemma... to assert the truth of a proposition.

(v) Proof... roughly corresponding to traditional mathematical proofs.

(vi) Notation... to introduce a more compact or readable notation.

(vii) Inductive... similar to ML declarations suitable for inductive proof.

(viii) Definition... similar to ML let statements

(ix) Ltac... Coq's domain-specific language for proof search.

A much more comprehensive explanation is available at [Pierce et al., 2012].

```
Require Omega. Require Export Bool.
Require Export List.
Require Export Arith.
Require Export Arith.EqNat.

Require String. Open Scope string_scope.

Ltac move_to_top x :=
  match reverse goal with
  | H : _ ⊢ _ ⇒ try move x after H
  end.

Tactic Notation "assert_eq" ident(x) constr(v) :=
  let H := fresh in
  assert (x = v) as H by reflexivity;
  clear H.

Tactic Notation "Case_aux" ident(x) constr(name) :=
  first [
    set (x := name); move_to_top x
  | assert_eq x name; move_to_top x
  | fail 1 "because we are working on a different case" ].

Tactic Notation "Case" constr(name) := Case_aux Case name.
Tactic Notation "SCase" constr(name) := Case_aux SCase name.
Tactic Notation "SSCase" constr(name) := Case_aux SSCase name.
```

```
Fixpoint ble_nat (n m : nat) : bool :=
  match n with
  | O ⇒ true
  | S n' ⇒
      match m with
      | O ⇒ false
      | S m' ⇒ ble_nat n' m'
      end
  end.

Theorem andb_true_elim1 : ∀ b c,
  andb b c = true → b = true.
Proof.
  intros b c H.
  destruct b.
  Case "b = true".
    reflexivity.
  Case "b = false".
    rewrite ← H. reflexivity. Qed.

Theorem andb_true_elim2 : ∀ b c,
  andb b c = true → c = true.
Proof.
  intros b c.
  destruct b.
  Case "b = true". auto.
  Case "b = false".
    destruct c ; auto.
Qed.

Theorem beq_nat_sym : ∀ (n m : nat),
  beq_nat n m = beq_nat m n.
Proof.
  intros n.
  induction n ; destruct m ; auto ; simpl ; apply IHn.
Qed.

Notation "[ ]" := nil.
Notation "[ x , .. , y ]" := (cons x .. (cons y []) ..).
Notation "x ++ y" := (app x y)
                     (at level 60, right associativity).

Inductive ev : nat → Prop :=
  | ev_0 : ev O
  | ev_SS : ∀ n:nat, ev n → ev (S (S n)).

Theorem andb_true : ∀ b c,
  andb b c = true → b = true ∧ c = true.
Proof.
  intros b c H.
  destruct b.
    destruct c.
      apply conj. reflexivity. reflexivity.
      inversion H.
    inversion H. Qed.

Theorem ex_falso_quodlibet : ∀ (P:Prop),
  False → P.
```

```
Proof.
  intros P contra.
  inversion contra. Qed.

Theorem not_eq_beq_false : ∀ n n' : nat,
      n ≠ n' →
      beq_nat n n' = false.
Proof.
  intros n.
  induction n as [| n'']. 
  destruct n' as [| n''']. 
  Case "n = 0".
    SCase "n' = 0".
      intros H.
      simpl.
      apply ex_falso_quodlibet.
      apply H.
      reflexivity.
    SCase "n' = S n'''".
      intros H.
      reflexivity.
 destruct n' as [| n''']. 
 Case "n = S n''".
    SCase "n' = 0".
      intros H.
      reflexivity.
    SCase "n' = S n'''".
      intros H.
      simpl.
      apply IHn''.
      unfold not.
      unfold not in H.
      intros H1.
      rewrite → H1 in H.
      apply H.
      reflexivity.
Qed.

Theorem ev_not_ev_S : ∀ n,
  ev n → ¬ ev (S n).
Proof.
  unfold not. intros n H. induction H.
  Case "ev 1".
    intros H. inversion H.
  Case "ev 2".
    intros H1.
    inversion H1.
    apply IHev in H2. assumption.
Qed.

Theorem O_le_n : ∀ n,
  0 ≤ n.
Proof.
  induction n as [| n'] ;
    apply le_n || apply le_S ;
    apply IHn'.
```

```
Qed.

Theorem n_le_m__Sn_le_Sm : ∀ n m,
  n ≤ m → S n ≤ S m.
Proof.
  intros n m.
  destruct n as [| n'].
  Case "n = 0".
    induction m as [| m'].
    SCase "m = 0".
      intros H. apply le_n.
    SCase "m = S m'".
      intros H.
      apply le_S.
      apply IHm'.
      apply O_le_n.
  Case "n = S n'".
    induction m as [| m'].
    intros H.
    SCase "m = 0".
      inversion H.
    SCase "m = S m'".
      intros H.
      inversion H ; subst.
      apply le_n.
      apply le_S.
      apply IHm'.
      apply H1.
Qed.

Theorem ble_nat_true : ∀ n m,
  ble_nat n m = true → n ≤ m.
Proof.
  induction n as [| n'].
  Case "n = 0".
    intros m Heq.
    apply O_le_n.
  Case "n = S n'".
    intros m.
    destruct m as [| m'].
    SCase "m = 0".
      simpl. intros H. inversion H.
    SCase "m = S m'".
      simpl. intros H. apply IHn' in H.
      apply n_le_m__Sn_le_Sm ; assumption.
Qed.

Theorem Sn_le_Sm__n_le_m : ∀ n m,
  S n ≤ S m → n ≤ m.
Proof.
  intros n m. generalize dependent n. induction m as [| m'].
  Case "m = 0".
    intros n.
    destruct n as [| n'].
    SCase "n = 0".
     intros H. apply le_n.
```

```
    SCase "n = S n'".
     intros H. inversion H. inversion H1.
  Case "m = S m'".
    intros n H.
    inversion H ; subst.
    apply le_n.
    apply le_S. apply IHm'.
    assumption.
Qed.

Theorem ble_nat_false : ∀ n m,
  ble_nat n m = false → ~(n ≤ m).
Proof.
  induction n as [| n'].
  Case "n = 0".
    intros m.
    destruct m as [| m'].
    SCase "m = 0".
      intros H. inversion H.
    SCase "m = S m'".
      simpl.
      intros H. inversion H.
  Case "n = S n'".
    intros m.
    destruct m as [| m'].
    SCase "m = 0".
      simpl.
      intros H contra. clear H.
      inversion contra.
    SCase "m = S m'".
      simpl.
      intros H. apply IHn' in H.
      intros H1.
      apply Sn_le_Sm__n_le_m in H1.
      contradiction.
Qed.

Inductive appears_in (n : nat) : list nat → Prop :=
| ai_here : ∀ l, appears_in n (n::l)
| ai_later : ∀ m l, appears_in n l → appears_in n (m::l).

Definition relation (X:Type) := X → X → Prop.

Definition partial_function {X: Type} (R: relation X) :=
  ∀ x y1 y2 : X, R x y1 → R x y2 → y1 = y2.

Inductive next_nat (n:nat) : nat → Prop :=
  | nn : next_nat n (S n).

Inductive total_relation : nat → nat → Prop :=
  tot : ∀ n m : nat, total_relation n m.

Inductive empty_relation : nat → nat → Prop := .

Inductive refl_step_closure (X:Type) (R: relation X)
                                    : X → X → Prop :=
  | rsc_refl : ∀ (x : X),
                    refl_step_closure X R x x
  | rsc_step : ∀ (x y z : X),
```

```
                    R x y →
                    refl_step_closure X R y z →
                    refl_step_closure X R x z.
Implicit Arguments refl_step_closure [[X]].

Tactic Notation "rsc_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "rsc_refl" | Case_aux c "rsc_step" ].

Theorem rsc_R : ∀ (X:Type) (R:relation X) (x y : X),
       R x y → refl_step_closure R x y.
Proof.
  intros X R x y r.
  apply rsc_step with y. apply r. apply rsc_refl. Qed.

Theorem rsc_trans :
  ∀ (X:Type) (R: relation X) (x y z : X),
      refl_step_closure R x y →
      refl_step_closure R y z →
      refl_step_closure R x z.
Proof.
  intros X R x y z H1.
  induction H1. trivial.
  intros H2.
  apply IHrefl_step_closure in H2.
  eapply rsc_step.
  apply H.
  assumption.
Qed.

Inductive id : Type :=
  Id : nat → id.

Definition beq_id id1 id2 :=
  match (id1 , id2) with
    (Id n1 , Id n2) ⇒ beq_nat n1 n2
  end.

Theorem beq_id_refl : ∀ i,
  true = beq_id i i.
Proof.
  intros. destruct i.
  apply beq_nat_refl. Qed.

Theorem beq_id_eq : ∀ i1 i2,
  true = beq_id i1 i2 → i1 = i2.
Proof.
  intros i1 i2 H.
  destruct i1. destruct i2.
  apply beq_nat_eq in H. subst.
  reflexivity. Qed.

Theorem beq_id_false_not_eq : ∀ i1 i2,
  beq_id i1 i2 = false → i1 ≠ i2.
Proof.
  intros i1 i2 H.
  destruct i1. destruct i2.
  apply beq_nat_false in H.
  intros C. apply H. inversion C. reflexivity. Qed.
```

```
Theorem not_eq_beq_id_false : ∀ i1 i2,
  i1 ≠ i2 → beq_id i1 i2 = false.
Proof.
  intros i1 i2 H.
  destruct i1. destruct i2.
  assert (n ≠ n0).
    intros C. subst. apply H. reflexivity.
  apply not_eq_beq_false. assumption. Qed.

Theorem beq_id_sym: ∀ i1 i2,
  beq_id i1 i2 = beq_id i2 i1.
Proof.
  intros i1 i2. destruct i1. destruct i2. apply beq_nat_sym. Qed.

Definition partial_map (A:Type) := id → option A.

Definition empty {A:Type} : partial_map A := (fun _ ⇒ None).

Definition extend {A:Type} (Gamma : partial_map A) (x:id) (T : A) :=
  fun x' ⇒ if beq_id x x' then Some T else Gamma x'.

Lemma extend_eq : ∀ A (ctxt: partial_map A) x T,
  (extend ctxt x T) x = Some T.
Proof.
  intros. unfold extend. rewrite ← beq_id_refl. auto.
Qed.

Lemma extend_neq : ∀ A (ctxt: partial_map A) x1 T x2,
  beq_id x2 x1 = false →
  (extend ctxt x2 T) x1 = ctxt x1.
Proof.
  intros. unfold extend. rewrite H. auto.
Qed.

Lemma extend_shadow : ∀ A (ctxt: partial_map A) t1 t2 x1 x2,
  extend (extend ctxt x2 t1) x2 t2 x1 = extend ctxt x2 t2 x1.
Proof with auto.
  intros. unfold extend. destruct (beq_id x2 x1)...
Qed.

Tactic Notation "solve_by_inversion_step" tactic(t) :=
  match goal with
  | H : _ ⊢ _ ⇒ solve [ inversion H; subst; t ]
  end
  || fail "because the goal is not solvable by inversion.".

Tactic Notation "solve" "by" "inversion" "1" :=
  solve_by_inversion_step idtac.
Tactic Notation "solve" "by" "inversion" "2" :=
  solve_by_inversion_step (solve by inversion 1).
Tactic Notation "solve" "by" "inversion" "3" :=
  solve_by_inversion_step (solve by inversion 2).
Tactic Notation "solve" "by" "inversion" :=
  solve by inversion 1.

Definition state := id → nat.

Definition empty_state : state :=
  fun _ ⇒ 0.

Definition update (st : state) (X:id) (n : nat) : state :=
  fun X' ⇒ if beq_id X X' then n else st X'.
```

```
Require Import Relations.

Definition normal_form {X:Type} (R:relation X) (t:X) : Prop :=
  ¬ ∃ t', R t t'.

Definition normalizing {X:Type} (R:relation X) :=
  ∀ t, ∃ t',
    (refl_step_closure R) t t' ∧ normal_form R t'.

Hint Constructors refl_step_closure.
Hint Resolve beq_id_eq beq_id_false_not_eq.

Tactic Notation "print_goal" := match goal with ⊢ ?x ⇒ idtac x end.
Tactic Notation "normalize" :=
   repeat (eapply rsc_step ;
             [ (eauto 10; fail) | (instantiate; simpl)]);
   apply rsc_refl.
```

Appendix-B

This appendix contains the complete example of shallow embedding of a simple λ-Calculus in Coq, described in section 4.5. It needs to be read in conjunction with the library of Appendix-A. The λ-expressions available are:

(i) tm_var: a term containing a variable

(ii) tm_app: a term containing the application of one term to another

(iii) tm_abs: a term containing an abstraction of a term into a variable (used to define functions)

(iv) tm_nat: a term containing an integer

(v) tm_pred: a term which returns the predecessor of another term (an integer expression)

(vi) tm_mult: a term which multiplies two terms (which are integer expressions)

(vii) tm_if0: a term which chooses an expression according to whether its argument is zero

(viii) tm_fix: a term which creates a fix point (a limited form of recursive function)

This routine also contains definitions and proofs of substitution (of one λ-term into another), testing if a λ-expression is a value, stepping from one λ-expression to a potentially simpler one, proving type preservation, reducing an expression with executable semantics, and proving that the algorithm implements a factorial for a specimen argument.

```
Require Import ZArith_base.
Require Export SfLib.

Inductive ty : Type :=
  | ty_arrow : ty → ty → ty
  | ty_Nat : ty.

Tactic Notation "ty_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "ty_arrow"
  | Case_aux c "ty_Nat"
  ].

Inductive tm : Type :=

  | tm_var : id → tm
  | tm_app : tm → tm → tm
  | tm_abs : id → ty → tm → tm

  | tm_nat : Z → tm
  | tm_pred : tm → tm
  | tm_mult : tm → tm → tm
  | tm_if0 : tm → tm → tm → tm
```

```
  | tm_fix : tm → tm.

Tactic Notation "tm_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "tm_var"
  | Case_aux c "tm_app"
  | Case_aux c "tm_abs"
  | Case_aux c "tm_nat"
  | Case_aux c "tm_pred"
  | Case_aux c "tm_mult"
  | Case_aux c "tm_if0"
  | Case_aux c "tm_fix"
  ].

Fixpoint subst (x:id) (s:tm) (t:tm) : tm :=
  match t with
  | tm_var y ⇒
      if beq_id x y then s else t
  | tm_abs y T t1 ⇒
      tm_abs y T (if beq_id x y then t1 else (subst x s t1))
  | tm_app t1 t2 ⇒
      tm_app (subst x s t1) (subst x s t2)

  | tm_nat n ⇒ tm_nat n
  | tm_pred t ⇒ tm_pred (subst x s t)
  | tm_mult t1 t2 ⇒ tm_mult (subst x s t1) (subst x s t2)
  | tm_if0 t1 t2 t3 ⇒ tm_if0 (subst x s t1) (subst x s t2) (subst x s t3)
  | tm_fix t ⇒ tm_fix (subst x s t)
  end.

Inductive value : tm → Prop :=
  | v_abs : ∀ x T11 t12,
      value (tm_abs x T11 t12)
  | v_nat : ∀ n, value (tm_nat n)
  .

Hint Constructors value.

Reserved Notation "t1 '==>' t2" (at level 40).

Inductive step : tm → tm → Prop :=
  | ST_AppAbs : ∀ x T11 t12 v2,
         value v2 →
         (tm_app (tm_abs x T11 t12) v2) ==> (subst x v2 t12)
  | ST_App1 : ∀ t1 t1' t2,
         t1 ==> t1' →
         (tm_app t1 t2) ==> (tm_app t1' t2)
  | ST_App2 : ∀ v1 t2 t2',
         value v1 →
         t2 ==> t2' →
         (tm_app v1 t2) ==> (tm_app v1 t2')
    numbers      | ST_Predn : ∀ n, tm_pred (tm_nat n) ==> tm_nat (Zpred n)
   | ST_Pred : ∀ t t', t ==> t' → (tm_pred t) ==> (tm_pred t')
   | ST_Mult1 : ∀ t1 t1' t2, t1 ==> t1' → (tm_mult t1 t2) ==> (tm_mult t1' t2)
   | ST_Mult2 : ∀ v1 t2 t2', value v1 → t2 ==> t2' → (tm_mult v1 t2) ==> (tm_mult v1 t2')
   | ST_MultV : ∀ n1 n2, tm_mult (tm_nat n1) (tm_nat n2) ==> (tm_nat (n1 × n2))
   | ST_IfZ : ∀ t2 t3, tm_if0 (tm_nat Z0) t2 t3 ==> t2
```

```
    | ST_IfS : ∀ n t2 t3, (Zeq_bool n Z0) = false → tm_if0 (tm_nat n) t2 t3 ==> t3
    | ST_If : ∀ t1 t1' t2 t3, t1 ==> t1' → (tm_if0 t1 t2 t3) ==> (tm_if0 t1' t2 t3)

    | ST_Fix1 : ∀ t1 t1', t1 ==> t1' → tm_fix t1 ==> tm_fix t1'
    | ST_FixAbs : ∀ x T1 t2, tm_fix (tm_abs x T1 t2) ==> (subst x (tm_fix (tm_abs x T1 t2))
t2)
      where "t1 '==>' t2" := (step t1 t2).
Tactic Notation "step_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "ST_AppAbs" | Case_aux c "ST_App1" | Case_aux c "ST_App2" |

    Case_aux c "ST_Succn" | Case_aux c "ST_Succ" | Case_aux c "ST_Predn" |
    Case_aux c "ST_Pred" | Case_aux c "ST_Mult1" |
    Case_aux c "ST_Pred" |
    Case_aux c "ST_Mult1" | Case_aux c "ST_Mult2" | Case_aux c "ST_MultV" |
    Case_aux c "ST_IfZ" | Case_aux c "ST_IfS" | Case_aux c "ST_If" |
  ].
Notation stepmany := (refl_step_closure step).
Notation "t1 '==>*' t2" := (stepmany t1 t2) (at level 40).
Hint Constructors step.
Definition context := partial_map ty.
Inductive has_type : context → tm → ty → Prop :=

  | T_Var : ∀ Gamma x T,
      Gamma x = Some T →
      has_type Gamma (tm_var x) T
  | T_Abs : ∀ Gamma x T11 T12 t12,
      has_type (extend Gamma x T11) t12 T12 →
      has_type Gamma (tm_abs x T11 t12) (ty_arrow T11 T12)
  | T_App : ∀ T1 T2 Gamma t1 t2,
      has_type Gamma t1 (ty_arrow T1 T2) →
      has_type Gamma t2 T1 →
      has_type Gamma (tm_app t1 t2) T2

  | T_Nat : ∀ n Gamma, has_type Gamma (tm_nat n) ty_Nat
  | T_Pred : ∀ t Gamma, has_type Gamma t ty_Nat → has_type Gamma (tm_pred t) ty_Nat
  | T_Mult : ∀ t1 t2 Gamma, has_type Gamma t1 ty_Nat → has_type Gamma t2 ty_Nat →
                                  has_type Gamma (tm_mult t1 t2) ty_Nat
  | T_If0 : ∀ t1 t2 t3 ty Gamma, has_type Gamma t1 ty_Nat → has_type Gamma t2 ty →
                                      has_type Gamma t3 ty → has_type Gamma (tm_if0
t1 t2 t3) ty

  | T_Fix : ∀ t1 T1 Gamma, has_type Gamma t1 (ty_arrow T1 T1) →
                                  has_type Gamma (tm_fix t1) T1
  .
Hint Constructors has_type.
Tactic Notation "has_type_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "T_Var" | Case_aux c "T_Abs" | Case_aux c "T_App" |

    Case_aux c "T_Nat" |
    Case_aux c "T_Pred" |
```

```
    Case_aux c "T_Mult" |
    Case_aux c "T_If0" |
    Case_aux c "T_Fix"
].

Fixpoint redvalue (t:tm) : bool :=
  match t with
  | tm_abs x T11 t12 ⇒ true
  | tm_nat n ⇒ true
  | _ ⇒ false
  end.

Fixpoint reduce (t:tm) : tm :=
  match t with
  | tm_app ((tm_abs x ty t12) as a) v2 ⇒ if redvalue v2 then (subst x v2 t12) else tm_app
(reduce a) (reduce v2)
  | tm_app a b ⇒ tm_app (reduce a) (reduce b)
  | tm_pred (tm_nat n) ⇒ tm_nat (Zpred n)
  | tm_pred v ⇒ tm_pred (reduce v)
  | tm_mult (tm_nat n1) (tm_nat n2) ⇒ (tm_nat (Zmult n1 n2))
  | tm_mult a b ⇒ tm_mult (reduce a) (reduce b)
  | tm_if0 (tm_nat Z0) t2 t3 ⇒ t2
  | tm_if0 (tm_nat _) t2 t3 ⇒ t3
  | tm_if0 a b c ⇒ tm_if0 (reduce a) b c
  | tm_fix (tm_abs x T1 t2) ⇒ (subst x (tm_fix (tm_abs x T1 t2)) t2)
  | tm_fix v ⇒ tm_fix (reduce v)
  | tm_nat v ⇒ tm_nat v
  | tm_var v ⇒ tm_var v
  | tm_abs a b c ⇒ tm_abs a b (reduce c)
  end.

Ltac inverts H := inversion H ; subst ; clear H ; auto.

Lemma types_unique : ∀ Gamma t T1, has_type Gamma t T1 → ∀ T2, has_type Gamma t
T2 → T1 = T2.
Proof.
  intros Gamma t T1 H1 ; has_type_cases (induction H1) Case ; intros T2' H2 ; inverts
H2.
  Case "T_Var".
    rewrite H in H3 ; inverts H3.
  Case "T_Abs".
    f_equal. apply IHhas_type ; auto.
  Case "T_App".
    apply IHhas_type1 in H3.
    injection H3 ; auto.
  Case "T_Fix".
    eapply IHhas_type in H3 ; inverts H3.
Qed.

Theorem progress : ∀ t T,
     has_type empty t T →
     value t ∨ ∃ t', t ==> t'.
Proof with eauto.
  intros t T Ht.
  remember (@empty ty) as Gamma.
  generalize dependent HeqGamma.
  has_type_cases (induction Ht) Case; intros HeqGamma; subst.
```

```
  Case "T_Var".
    inversion H.
  Case "T_Abs".
    left...
  Case "T_App".
    right.
    destruct IHHt1; subst...
    SCase "t1 is a value".
      destruct IHHt2; subst...
      SSCase "t2 is a value".
        inversion H; subst; try (solve by inversion).
        ∃ (subst x t2 t12)...
      SSCase "t2 steps".
        destruct H0 as [t2' Hstp]. ∃ (tm_app t1 t2')...
    SCase "t1 steps".
      destruct H as [t1' Hstp]. ∃ (tm_app t1' t2)...
  Case "T_Nat".
    left ...
  Case "T_Pred".
    destruct IHHt ; auto ; destruct H ; try solve by inversion.
    destruct n ; right ; eexists ; eauto.
    right ; eexists ; eauto.
  Case "T_Mult".
    destruct IHHt1 ; destruct IHHt2 ; auto ;
    destruct H ; destruct H0 ; try solve by inversion.
    right ; ∃ (tm_nat (n × n0)) ; auto.
    right ; ∃ (tm_mult (tm_nat n) x) ; auto.
    right ; ∃ (tm_mult x (tm_nat n)) ; auto.
    right ; eexists ; eauto.
  Case "T_If0".
    destruct IHHt1 ; auto ; right.
    destruct H ; try solve by inversion.
    destruct n.
    ∃ t2 ; auto.
    ∃ t3; auto.
    ∃ t3; auto.

    destruct H.
    ∃ (tm_if0 x t2 t3) ; auto.
  Case "T_Fix".
    destruct (IHHt (eq_refl empty)).
    inverts H ; try solve by inversion.
    right ; eexists ...
    destruct H as [t' H'].
    right ; eexists ...
Qed.

Inductive appears_free_in : id → tm → Prop :=
  | afi_var : ∀ x,
      appears_free_in x (tm_var x)
  | afi_app1 : ∀ x t1 t2,
      appears_free_in x t1 → appears_free_in x (tm_app t1 t2)
  | afi_app2 : ∀ x t1 t2,
      appears_free_in x t2 → appears_free_in x (tm_app t1 t2)
  | afi_abs : ∀ x y T11 t12,
```

```
        y ≠ x →
        appears_free_in x t12 →
        appears_free_in x (tm_abs y T11 t12)

  | afi_pred : ∀ x t1, appears_free_in x t1 →
                        appears_free_in x (tm_pred t1)
  | afi_mult1 : ∀ x t1 t2, appears_free_in x t1 →
                          appears_free_in x (tm_mult t1 t2)
  | afi_mult2 : ∀ x t1 t2, appears_free_in x t2 →
                          appears_free_in x (tm_mult t1 t2)
  | afi_if01 : ∀ x t1 t2 t3, appears_free_in x t1 →
                            appears_free_in x (tm_if0 t1 t2 t3)
  | afi_if02 : ∀ x t1 t2 t3, appears_free_in x t2 →
                            appears_free_in x (tm_if0 t1 t2 t3)
  | afi_if03 : ∀ x t1 t2 t3, appears_free_in x t3 →
                            appears_free_in x (tm_if0 t1 t2 t3)

  | afi_fix : ∀ x t, appears_free_in x t → appears_free_in x (tm_fix t)
  .
Hint Constructors appears_free_in.
Lemma context_invariance : ∀ Gamma Gamma' t S,
     has_type Gamma t S →
     (∀ x, appears_free_in x t → Gamma x = Gamma' x) →
     has_type Gamma' t S.
Proof with eauto.
  intros. generalize dependent Gamma'.
  has_type_cases (induction H) Case;
    intros Gamma' Heqv...
  Case "T_Var".
    apply T_Var... rewrite ← Heqv...
  Case "T_Abs".
    apply T_Abs... apply IHhas_type. intros y Hafi.
    unfold extend. remember (beq_id x y) as e.
    destruct e...
  Case "T_App".
    eapply T_App; auto.
  Case "T_Mult".
    apply T_Mult...
  Case "T_If0".
    apply T_If0 ...
Qed.
Lemma free_in_context : ∀ x t T Gamma,
   appears_free_in x t →
   has_type Gamma t T →
   ∃ T', Gamma x = Some T'.
Proof with eauto.
  intros x t T Gamma Hafi Htyp.
  has_type_cases (induction Htyp) Case; inversion Hafi; subst...
  Case "T_Abs".
    destruct IHHtyp as [T' Hctx]... ∃ T'.
    unfold extend in Hctx.
    apply not_eq_beq_id_false in H2. rewrite H2 in Hctx...
Qed.
```

```
Lemma substitution_preserves_typing : ∀ Gamma x U v t S,
      has_type (extend Gamma x U) t S →
      has_type empty v U →
      has_type Gamma (subst x v t) S.
Proof with eauto.
  intros Gamma x U v t S Htypt Htypv.
  generalize dependent Gamma. generalize dependent S.
  tm_cases (induction t) Case;
    intros S Gamma Htypt; simpl; inversion Htypt; subst...
  Case "tm_var".
    simpl. rename i into y.
    remember (beq_id x y) as e. destruct e.
    SCase "x=y".
      apply beq_id_eq in Heqe. subst.
      unfold extend in H1. rewrite ← beq_id_refl in H1.
      inversion H1; subst. clear H1.
      eapply context_invariance...
      intros x Hcontra.
      destruct (free_in_context _ _ S empty Hcontra) as [T' HT']...
      inversion HT'.
    SCase "x<>y".
      apply T_Var... unfold extend in H1. rewrite ← Heqe in H1...
  Case "tm_abs".
    rename i into y. rename t into T11.
    apply T_Abs...
    remember (beq_id x y) as e. destruct e.
    SCase "x=y".
      eapply context_invariance...
      apply beq_id_eq in Heqe. subst.
      intros x Hafi. unfold extend.
      destruct (beq_id y x)...
    SCase "x<>y".
      apply IHt. eapply context_invariance...
      intros z Hafi. unfold extend.
      remember (beq_id y z) as e0. destruct e0...
      apply beq_id_eq in Heqe0. subst.
      rewrite ← Heqe...
Qed.

Theorem preservation : ∀ t t' T,
      has_type empty t T →
      t ==> t' →
      has_type empty t' T.
Proof with eauto.
  intros t t' T HT.
  remember (@empty ty) as Gamma. generalize dependent HeqGamma.
  generalize dependent t'.
  has_type_cases (induction HT) Case;
    intros t' HeqGamma HE; subst; inversion HE; subst...
  Case "T_App".
    inversion HE; subst...
    SCase "ST_AppAbs".
      apply substitution_preserves_typing with T1...
      inversion HT1...
```

```
  Case "T_Fix".
    inverts HT ...
    eapply substitution_preserves_typing...
Qed.

Hint Extern 2 (has_type _ (tm_app _ _) _) ⇒
  eapply T_App; auto.

Hint Extern 2 (_ = _) ⇒ compute; reflexivity.

Notation a := (Id 0).
Notation f := (Id 1).

Definition fact :=
  tm_fix
    (tm_abs f (ty_arrow ty_Nat ty_Nat)
      (tm_abs a ty_Nat
        (tm_if0
            (tm_var a)
            (tm_nat 1)
            (tm_mult
                (tm_var a)
                (tm_app (tm_var f) (tm_pred (tm_var a))))))).

Theorem fact_typechecks :
  has_type (@empty ty) fact (ty_arrow ty_Nat ty_Nat).
Proof. unfold fact. auto 10.
Qed.

Definition fact_calc n := (tm_app fact (tm_nat n)).

Theorem fact_example:
  refl_step_closure step (fact_calc 4) (tm_nat 24).

Proof.
unfold fact_calc.
unfold fact.
normalize.
Qed.

Fixpoint reduce_n (n:nat) (t:tm) : tm :=
  match n with
    | 0 ⇒ t
    | S n ⇒ reduce_n n (reduce t)
  end.

Example FixTest1_fact_example2:

reduce_n 11 (fact_calc 2) = (tm_nat 2).

Proof.
unfold reduce_n.
simpl.
intuition.
Qed.
```

Appendix-C

This appendix contains the complete Verilog code for a standalone synthesisable processor based on the Amber architecture of [Santifort, 2013]. This code is output by the OCaml compiler in the form of a template with customisable memory sizes and program counter length. A set of schematics for this module may be seen in Appendix-I. The content of this file would be more compact and clearer by making use of arrays of registers, but this feature is not yet available in the fault-tolerant synthesis flow that is being used.

```
////////////////////////////////////////////////////////////////////
//                                                                //
//  Standalone Execute module based on Amber 2 Core               //
//                                                                //
//  This file is based on the Amber project                       //
//  http://www.opencores.org/project,amber                        //
//                                                                //
//  Description                                                   //
//  Executes control store contents in a state machine            //
//                                                                //
//  Author(s):                                                    //
//      - Conor Santifort, csantifort.amber@gmail.com             //
//      - Jonathan Kimmitt, jonathan@kimmitt.co.uk                //
//                                                                //
////////////////////////////////////////////////////////////////////
//                                                                //
// Copyright (C) 2010 Authors and OPENCORES.ORG                   //
//                                                                //
// This source file may be used and distributed without           //
// restriction provided that this copyright statement is not      //
// removed from the file and that any derivative work contains    //
// the original copyright notice and the associated disclaimer.   //
//                                                                //
// This source file is free software; you can redistribute it     //
// and/or modify it under the terms of the GNU Lesser General     //
// Public License as published by the Free Software Foundation;   //
// either version 2.1 of the License, or (at your option) any     //
// later version.                                                 //
//                                                                //
// This source is distributed in the hope that it will be         //
// useful, but WITHOUT ANY WARRANTY; without even the implied     //
// warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR        //
// PURPOSE.  See the GNU Lesser General Public License for more   //
// details.                                                       //
//                                                                //
// You should have received a copy of the GNU Lesser General      //
// Public License along with this source; if not, download it     //
// from http://www.opencores.org/lgpl.shtml                       //
//                                                                //
////////////////////////////////////////////////////////////////////


module standalone(
a23_clk, a23_rst, dbg_mem,
read_data,
execute_write_data,
execute_write_data_nxt,
execute_address,      // registered version of execute_address to the ram
execute_address_nxt,  // un-registered version of execute_address to the ram
execute_byte_enable_nxt,
execute_write_enable,
write_data_wen_out,
read_enable_out,
finish,
readstrobe,
readch,
writech,
```

```
r0_out,
r1_out,
r2_out,
r3_out,
r4_out,
r5_out,
r6_out,
r7_out,
r8_out,
r9_out,
r10_out,
r11_out,
r12_out,
r13_out,
r14_out,
r15_out,
pc_nxt_out,
xstate,
hw_exn,
hw_exn_en);

input           a23_clk, a23_rst, dbg_mem, hw_exn;
input   [31:0]  read_data;

output  [31:0]  execute_write_data_nxt;
output  [31:0]  execute_address_nxt;  // un-registered version of execute_address to the ram
output  [3:0]   execute_byte_enable_nxt;
output          write_data_wen_out;
output  [0:0]   read_enable_out;
output          finish, hw_exn_en;
output  [0:0]   readstrobe;
input   [7:0]   readch;
output  [7:0]   writech;
output  [31:0]  r15_out;
output  [31:0]  pc_nxt_out;
input   [75:0]  xstate;

output  [31:0]  execute_write_data;
output  [31:0]  execute_address;// registered version of execute_address to the ram
output          execute_write_enable;
output  [31:0]  r0_out;
output  [31:0]  r1_out;
output  [31:0]  r2_out;
output  [31:0]  r3_out;
output  [31:0]  r4_out;
output  [31:0]  r5_out;
output  [31:0]  r6_out;
output  [31:0]  r7_out;
output  [31:0]  r8_out;
output  [31:0]  r9_out;
output  [31:0]  r10_out;
output  [31:0]  r11_out;
output  [31:0]  r12_out;
output  [31:0]  r13_out;
output  [31:0]  r14_out;

reg     [31:0]  execute_write_data;
reg     [31:0]  execute_address;      // registered version of execute_address to the ram
reg             execute_write_enable, execute_copro_write_strb;
reg     [31:0]  r0_out;
reg     [31:0]  r1_out;
reg     [31:0]  r2_out;
reg     [31:0]  r3_out;
reg     [31:0]  r4_out;
reg     [31:0]  r5_out;
reg     [31:0]  r6_out;
reg     [31:0]  r7_out;
reg     [31:0]  r8_out;
reg     [31:0]  r9_out;
reg     [31:0]  r10_out;
reg     [31:0]  r11_out;
reg     [31:0]  r12_out;
reg     [31:0]  r13_out;
reg     [31:0]  r14_out;
reg     [31:0]  r15_out;
reg     [3:0]   prev_alu_function;

wire    [4:0]   oreg_sel;
wire    [0:0]   copro_write_data_wen;
wire    [0:0]   write_data_wen;
wire    [1:0]   reg_write_sel;
wire    [1:0]   byte_enable_sel;
wire    [17:2]  pc_nxt;
wire    [2:0]   pc_sel;
```

```
wire    [2:0]   address_sel;
wire    [6:0]   alu_function;
wire    [1:0]   barrel_shift_function;
wire    [1:0]   barrel_shift_data_sel;
wire    [1:0]   barrel_shift_amount_sel;
wire    [3:0]   rn_sel;
wire    [3:0]   rds_sel;
wire    [3:0]   rm_sel;
wire    [4:0]   imm_shift_amount;
wire    [7:0]   imm8;
wire    [0:0]   read_enable;
wire    [3:0]   a23_ccode;

reg     [4:0]   hw_oreg_sel;

reg     [31:0]  pc_nxt_out;
reg     [31:0]  execute_copro_write_data;
assign          writech = {execute_copro_write_strb,execute_copro_write_data[7:1]};

wire            system_rdy = !a23_rst;

wire    [31:0]  copro_read_data = 0;

reg     [3:0]   execute_byte_enable;
wire    ready;
wire div_sel = alu_function[3:0] == alu_Ediv;
wire mod_sel = alu_function[3:0] == alu_Emod;
wire div_rst = a23_rst || ((div_sel|mod_sel) && (prev_alu_function != alu_function[3:0]));
wire div_hold = (div_sel|mod_sel) && (div_rst || !ready);
wire div_flag_zero = |rm == 1'd0;
wire div_zero = (div_sel|mod_sel) && div_flag_zero && ready && !div_rst;
wire            fetch_stall = finish || div_hold;



assign          finish = &r15_out[17:2];

parameter
        _caml_int32_ops = -1,
        ASR = 2'd2,
        LSL = 2'd0,
        LSR = 2'd1,
        alu_inb = 4'd0,
        alu_not_sel = 7'd32,
        alu_swap_sel = 7'd64,
        alu_Eadd = 4'd1,
        alu_zex16 = 4'd2,
        alu_zex8 = 4'd3,
        alu_sex16 = 4'd4,
        alu_sex8 = 4'd5,
        alu_Exor = 4'd6,
        alu_Eor = 4'd7,
        alu_Eand = 4'd8,
        alu_Esub = 7'd49,
        alu_Enot = 7'd32,
        alu_rsub = 7'd113,
        alu_Easr = 4'd0,
        alu_Elsl = 4'd0,
        alu_Elsr = 4'd0,
        alu_Emul = 4'd12,
        alu_Ediv = 4'd13,
        alu_Emod = 4'd14,
        barrel_Eadd = 2'd0,
        barrel_Eand = 2'd0,
        barrel_Easr = 2'd2,
        barrel_Elsl = 2'd0,
        barrel_Elsr = 2'd1,
        barrel_Emul = 2'd0,
        barrel_Ediv = 2'd0,
        barrel_Emod = 2'd0,
        barrel_Eor = 2'd0,
        barrel_Exor = 2'd0,
        barrel_Esub = 2'd0,
        Eunsigned_Eeq = 4'd1,
        Esigned_Eeq = 4'd2,
        Eunsigned_Ene = 4'd3,
        Esigned_Ene = 4'd4,
        Esigned_Ele = 4'd5,
        Esigned_Ege = 4'd6,
        Esigned_Elt = 4'd7,
        Esigned_Egt = 4'd8,
        Eunsigned_Ele = 4'd9,
        Eunsigned_Ege = 4'd10,
        Eunsigned_Elt = 4'd11,
```

```
        Eunsigned_Egt = 4'd12;

// ========================================================
// Internal signals
// ========================================================
wire    [31:0]  pc_plus4;
wire    [31:0]  pc_minus4;
wire    [31:0]  address_plus4;
wire    [31:0]  alu_plus4;
wire    [31:0]  rn_plus4;
wire    [31:0]  rn_minus4;
wire    [7:0]   shift_amount;
wire    [31:0]  barrel_shift_in;
wire            alu_flag_neg, alu_flag_zero, alu_flag_cout, alu_flag_ov;

wire    [31:0]  reg_write_nxt;
wire    [31:0]  multiply_out;
wire    [1:0]   multiply_flags;

wire            address_update;
wire            write_data_update;
wire            copro_write_data_update;
wire            byte_enable_update;
wire            write_enable_update;

wire    [31:0]  alu_out_pc_filtered;

reg     [32:0]  barrel_shift_out;
reg     [31:0]  rd, rm, rn, rs, alu_out;
reg             compare_a23;

// ========================================================
// Rds Selector
// ========================================================
always @*
    case (rds_sel)
       4'd0 :  rd = r0_out;
       4'd1 :  rd = r1_out;
       4'd2 :  rd = r2_out;
       4'd3 :  rd = r3_out;
       4'd4 :  rd = r4_out;
       4'd5 :  rd = r5_out;
       4'd6 :  rd = r6_out;
       4'd7 :  rd = r7_out;
       4'd8 :  rd = r8_out;
       4'd9 :  rd = r9_out;
       4'd10 :  rd = r10_out;
       4'd11 :  rd = r11_out;
       4'd12 :  rd = r12_out;
       4'd13 :  rd = r13_out;
       4'd14 :  rd = r14_out;
       default: rd = {6'd0, pc_nxt_out[25:2], 2'b0};
    endcase
// ========================================================
// Rn Selector
// ========================================================
always @*
    case (rn_sel)
       4'd0 :  rn = r0_out;
       4'd1 :  rn = r1_out;
       4'd2 :  rn = r2_out;
       4'd3 :  rn = r3_out;
       4'd4 :  rn = r4_out;
       4'd5 :  rn = r5_out;
       4'd6 :  rn = r6_out;
       4'd7 :  rn = r7_out;
       4'd8 :  rn = r8_out;
       4'd9 :  rn = r9_out;
       4'd10 :  rn = r10_out;
       4'd11 :  rn = r11_out;
       4'd12 :  rn = r12_out;
       4'd13 :  rn = r13_out;
       4'd14 :  rn = r14_out;
       default: rn = r15_out;
    endcase
// ========================================================
// Rm Selector
// ========================================================
always @*
    case (rm_sel)
       4'd0 :  rm = r0_out;
       4'd1 :  rm = r1_out;
       4'd2 :  rm = r2_out;
       4'd3 :  rm = r3_out;
```

```
       4'd4 :  rm = r4_out;
       4'd5 :  rm = r5_out;
       4'd6 :  rm = r6_out;
       4'd7 :  rm = r7_out;
       4'd8 :  rm = r8_out;
       4'd9 :  rm = r9_out;
       4'd10 :  rm = r10_out;
       4'd11 :  rm = r11_out;
       4'd12 :  rm = r12_out;
       4'd13 :  rm = r13_out;
       4'd14 :  rm = r14_out;
       default: rm = r15_out;
    endcase
// ========================================================
// Rds Selector
// ========================================================
always @*
    case (rds_sel)
       4'd0 :  rs = r0_out;
       4'd1 :  rs = r1_out;
       4'd2 :  rs = r2_out;
       4'd3 :  rs = r3_out;
       4'd4 :  rs = r4_out;
       4'd5 :  rs = r5_out;
       4'd6 :  rs = r6_out;
       4'd7 :  rs = r7_out;
       4'd8 :  rs = r8_out;
       4'd9 :  rs = r9_out;
       4'd10 :  rs = r10_out;
       4'd11 :  rs = r11_out;
       4'd12 :  rs = r12_out;
       4'd13 :  rs = r13_out;
       4'd14 :  rs = r14_out;
       default: rs = r15_out;
    endcase

// ========================================================
// Adders
// ========================================================
assign pc_plus4      = r15_out         + 32'd4;
assign pc_minus4     = r15_out         - 32'd4;
assign address_plus4 = execute_address + 32'd4;
assign alu_plus4     = alu_out         + 32'd4;
assign rn_plus4      = rn              + 32'd4;
assign rn_minus4     = rn              - 32'd4;

// ========================================================
// Barrel Shift Amount Select
// ========================================================
// An immediate shift value of 0 is translated into 32
assign shift_amount = barrel_shift_amount_sel == 2'd0 ? 8'd0                          :
                      barrel_shift_amount_sel == 2'd1 ? rs[7:0]                       :
                      barrel_shift_amount_sel == 2'd2 ? {3'd0, imm_shift_amount    } :
                                                        {3'd0, execute_address[1:0], 3'b0 };

// ========================================================
// Barrel Shift Data Select
// ========================================================
assign barrel_shift_in = barrel_shift_data_sel == 2'd0 ? {24'b0,imm8}:
                         barrel_shift_data_sel == 2'd1 ? read_data   :
                         barrel_shift_data_sel == 2'd2 ? rm          :
                                                       {14'b0,pc_nxt,2'b0};

// ========================================================
// Interrupt vector Select
// ========================================================


// ========================================================
// Address Select
// ========================================================

// If rd is the pc, then separate the address bits from the status bits for
// generating the next address to fetch
assign alu_out_pc_filtered = pc_sel == 3'd1 ? {6'd0, alu_out[25:2], 2'd0} : alu_out;

assign execute_address_nxt = (address_sel == 3'd0) ? alu_out_pc_filtered   :
                       (address_sel == 3'd1) ? rn                  :
                       (address_sel == 3'd2) ? rn_plus4            :
                       (address_sel == 3'd3) ? rn_minus4           :
                                               32'hDEAD_BEEF         ;

// ========================================================
// Program Counter Select
```

```
// ====================================================

always @*
  case (a23_ccode)
      Eunsigned_Eeq : compare_a23 = alu_flag_zero;
      Esigned_Eeq : compare_a23 = alu_flag_zero;
      Eunsigned_Ene : compare_a23 = !alu_flag_zero;
      Esigned_Ene : compare_a23 = !alu_flag_zero;
      Esigned_Ele : compare_a23 = alu_flag_neg | alu_flag_zero;
      Esigned_Ege : compare_a23 = alu_flag_zero | !alu_flag_neg;
      Esigned_Elt : compare_a23 = alu_flag_neg;
      Esigned_Egt : compare_a23 = !(alu_flag_neg | alu_flag_zero);
      Eunsigned_Ele : compare_a23 = alu_flag_zero | !alu_flag_cout;
      Eunsigned_Ege : compare_a23 = alu_flag_cout;
      Eunsigned_Elt : compare_a23 = !alu_flag_cout;
      Eunsigned_Egt : compare_a23 = alu_flag_cout & !alu_flag_zero;
      default: compare_a23 = 1'b1;
  endcase

reg hw_exn_en, hw_exn_taken, hw_exn_raise, dummy;
wire hw_exn_raise2 = hw_exn_raise | div_zero;

always @*
  if (!system_rdy) pc_nxt_out = 32'b0;
  else if (hw_exn_raise2) pc_nxt_out = r11_out;
  else if (fetch_stall) pc_nxt_out = r15_out;
  else case (pc_sel)
      3'b000 : pc_nxt_out = pc_plus4              ;
      3'b001 : pc_nxt_out = alu_out               ;
      3'b010 : pc_nxt_out = 32'h0                 ;
      3'b011 : pc_nxt_out = {14'b0,pc_nxt,2'b0}   ;
      3'b100 : pc_nxt_out = r14_out               ;
      3'b101 : pc_nxt_out = (compare_a23 ? {14'b0,pc_nxt,2'b0} : pc_plus4) ;
      3'b110 : pc_nxt_out = rm                    ;
      3'b111 : pc_nxt_out = r11_out;
      default: pc_nxt_out = 32'hDEAD_BEEF       ;
  endcase // case (pc_sel)

// ====================================================
// Register Write Select
// ====================================================

assign reg_write_nxt = hw_exn_raise2 ? 32'd3 :
                       reg_write_sel == 2'd0 ? alu_out               :
                       reg_write_sel == 2'd1 ? multiply_out          :
                       reg_write_sel == 2'd2 ? copro_read_data     :  // mrc
                       reg_write_sel == 2'd3 ? {31'b0,compare_a23}   : 0; // Boolean comparison

// ====================================================
// Byte Enable Select
// ====================================================
assign execute_byte_enable_nxt = byte_enable_sel == 2'd0  ? 4'b1111 :  // word write
                         byte_enable_sel == 2'd2  ?            // halfword write
                       ( execute_address_nxt[1] == 1'd0 ? 4'b0011 :
                                                   4'b1100  ) :

                         execute_address_nxt[1:0] == 2'd0 ? 4'b0001 :  // byte write
                         execute_address_nxt[1:0] == 2'd1 ? 4'b0010 :
                         execute_address_nxt[1:0] == 2'd2 ? 4'b0100 :
                                                   4'b1000;


// ====================================================
// Write Data Select
// ====================================================

assign execute_write_data_nxt = byte_enable_sel == 2'd0 ? rd : {rd[ 7:0],rd[ 7:0],rd[ 7:0],rd[ 7:0]};

// ====================================================
// Register Update
// ====================================================

assign write_enable_update     = !fetch_stall;
assign write_data_update       = !fetch_stall && write_data_wen;
assign address_update          = !fetch_stall;
assign byte_enable_update      = !fetch_stall && write_data_wen;
assign copro_write_data_update = !fetch_stall && copro_write_data_wen;

always @( posedge a23_clk )
if (!system_rdy)
        begin
        execute_copro_write_data <= 'd0;
        execute_copro_write_strb <= 'd0;
        execute_write_data <= 'd0;
```

```
        execute_address <= 32'hdead_dead;
        execute_write_enable <= 'd0;
        execute_byte_enable <= 'd0;
        prev_alu_function <= 'd0;
        end
else
    begin
    if (write_enable_update)     execute_write_enable     <= write_data_wen;
    if (write_data_update)       execute_write_data       <= execute_write_data_nxt;
    if (address_update)          execute_address          <= execute_address_nxt;
    if (byte_enable_update)      execute_byte_enable      <= execute_byte_enable_nxt;
    if (copro_write_data_update) execute_copro_write_data <= execute_write_data_nxt;
    execute_copro_write_strb <= copro_write_data_update;
    prev_alu_function <= alu_function[3:0];
    end


// ======================================================
// Instantiate Barrel Shift
// ======================================================

wire    [32:0] lsl_out = {1'b0,barrel_shift_in} << shift_amount;
wire    [32:0] lsr_out = {1'b0,barrel_shift_in} >> shift_amount;
wire    [63:0] asr_out = {{32{barrel_shift_in[31]}},barrel_shift_in} >>
            (shift_amount[7:5] != 0 ? 31 : shift_amount[4:0]);

always @* case (barrel_shift_function)
        LSL : barrel_shift_out = lsl_out ;
        LSR : barrel_shift_out = lsr_out ;
        ASR : barrel_shift_out = asr_out[32:0] ;
        default: barrel_shift_out = 33'hDEAD_BEEF ;
        endcase // case (barrel_shift_function)

// ======================================================
// Instantiate ALU
// ======================================================
wire    [31:0]  a       = (alu_function[6] ) ? barrel_shift_out[31:0] : rn ;
wire    [31:0]  b       = (alu_function[6] ) ? rn : barrel_shift_out[31:0] ;
wire    [31:0]  b_not   = (alu_function[5] ) ? ~b : b ;
wire    [32:0]  fadder_out      = { 1'd0,a} + {1'd0,b_not} + {32'd0,alu_function[4]};

wire    [31:0]  and_out        = a & b_not;
wire    [31:0]  or_out         = a | b_not;
wire    [31:0]  xor_out        = a ^ b_not;
wire    [31:0]  zero_ex8_out   = {24'd0,  b_not[7:0]};
wire    [31:0]  zero_ex_16_out = {16'd0,  b_not[15:0]};
wire    [31:0]  sign_ex8_out   = {{24{b_not[7]}}, b_not[7:0]};
wire    [31:0]  sign_ex_16_out = {{16{b_not[15]}}, b_not[15:0]};

wire [31:0] quotient, remainder;

divide divider1(
         .ready(ready),
         .quotient(quotient),
         .remainder(remainder),
         .dividend(rn),
         .divider(rm),
         .sign(1'b1),
         .clk(a23_clk),
         .rst(div_rst)
       );

always @* case(alu_function[3:0])
        alu_inb : alu_out = b_not            ;
        alu_Eadd : alu_out = fadder_out[31:0];
        alu_zex16 : alu_out = zero_ex_16_out ;
        alu_zex8 : alu_out = zero_ex8_out    ;
        alu_sex16 : alu_out = sign_ex_16_out ;
        alu_sex8 : alu_out = sign_ex8_out    ;
        alu_Exor : alu_out = xor_out         ;
        alu_Eor : alu_out = or_out           ;
        alu_Eand : alu_out = and_out         ;
        alu_Emul : alu_out = multiply_out    ;
        alu_Ediv : alu_out = div_hold ? -1 : quotient        ;
        alu_Emod : alu_out = div_hold ? -1 : remainder       ;
        default: alu_out = 32'hDEAD_BEEF     ;
        endcase

   assign alu_flag_neg = alu_out[31];
   assign alu_flag_zero = |alu_out == 1'd0;
   assign alu_flag_cout = fadder_out[32];
   assign alu_flag_ov = alu_function[3:0] == alu_Eadd &&
                          ((!a[31] && !b_not[31] && fadder_out[31]) ||
                           (a[31] && b_not[31] && !fadder_out[31]));
```

```
// Parallel multiply based on DSP48E

   wire [35:0] absa = a[31] ? 36'b0-{4'b0,rn} : {4'b0,rn};
   wire [35:0] absb = b[31] ? 36'b0-{4'b0,rm} : {4'b0,rm};

`ifdef state_mem
   wire [31:0]   absmult = absa * absb;
`else
   wire [35:0]   p0, p1, p2;

   wire [17:0] pada0 = {1'b0,absa[16:0]};
   wire [17:0] padb0 = {1'b0,absb[16:0]};
   wire [17:0] pada1 = {1'b0,absa[33:17]};
   wire [17:0] padb1 = {1'b0,absb[33:17]};

MULT18X18 mult0 (
  .P(p0),
  .A(pada0),
  .B(padb0)
    );

MULT18X18 mult1 (
  .P(p1),
  .A(pada0),
  .B(padb1)
    );

MULT18X18 mult2 (
  .P(p2),
  .A(pada1),
  .B(padb0)
    );

   wire [52:0]   absmult = {{17'b0,p0[35:17]}+p1+p2,p0[16:0]};
`endif

   assign multiply_out = a[31]^b[31] ? 32'b0-absmult[31:0] : absmult[31:0];

// ========================================================
// Register Update
// ========================================================

always @*
  if (hw_exn_raise2)
    hw_oreg_sel = 5'h10;
  else
    hw_oreg_sel = oreg_sel;

always @ ( posedge a23_clk )
if (!system_rdy)
  begin
    r0_out = 32'hDEAD_BEEF;
    r1_out = 32'hDEAD_BEEF;
    r2_out = 32'hDEAD_BEEF;
    r3_out = 32'hDEAD_BEEF;
    r4_out = 32'hDEAD_BEEF;
    r5_out = 32'hDEAD_BEEF;
    r6_out = 32'hDEAD_BEEF;
    r7_out = 32'hDEAD_BEEF;
    r8_out = 32'hDEAD_BEEF;
    r9_out = 32'hDEAD_BEEF;
    r10_out = 32'hDEAD_BEEF;
    r11_out = 32'hDEAD_BEEF;
    r12_out = 32'hDEAD_BEEF;
    r13_out = 32'hDEAD_BEEF;
    r14_out = 32'hDEAD_BEEF;
    r15_out <= 32'h0;
    hw_exn_en = 1'b0;
    hw_exn_taken <= 1'b0;
    hw_exn_raise <= 1'b0;
  end
else
    begin
    if (!fetch_stall)
        begin
        hw_exn_taken <= hw_exn_taken | hw_exn_raise2;
        hw_exn_raise <= hw_exn & hw_exn_en & !hw_exn_taken & !hw_exn_raise2;
        r15_out <= {6'd0, pc_nxt_out[25:2], 2'd0};
        end

    if (!finish) case(hw_oreg_sel)
         5'h10  :  r0_out  = reg_write_nxt;
         5'h11  :  r1_out  = reg_write_nxt;
```

```
         5'h12  :  r2_out  = reg_write_nxt;
         5'h13  :  r3_out  = reg_write_nxt;
         5'h14  :  r4_out  = reg_write_nxt;
         5'h15  :  r5_out  = reg_write_nxt;
         5'h16  :  r6_out  = reg_write_nxt;
         5'h17  :  r7_out  = reg_write_nxt;
         5'h18  :  r8_out  = reg_write_nxt;
         5'h19  :  r9_out  = reg_write_nxt;
         5'h1a  :  r10_out  = reg_write_nxt;
         5'h1b  :  begin r11_out = reg_write_nxt; hw_exn_en = 1'b1; hw_exn_taken <= 1'b0; end
         5'h1c  :  r12_out  = reg_write_nxt;
         5'h1d  :  r13_out  = reg_write_nxt;
         5'h1e  :  r14_out  = reg_write_nxt;
        default :  dummy = 1'b0;
       endcase
       end

assign readstrobe = 0;
assign write_data_wen_out = write_data_wen;
assign read_enable_out = read_enable;


assign
        {
        oreg_sel,
        copro_write_data_wen,
        write_data_wen,
        reg_write_sel,
        byte_enable_sel,
        pc_nxt,
        pc_sel,
        address_sel,
        alu_function,
        barrel_shift_function,
        barrel_shift_data_sel,
        barrel_shift_amount_sel,
        rn_sel,
        rds_sel,
        rm_sel,
        imm_shift_amount,
        imm8,
        read_enable,
        a23_ccode} = xstate;

endmodule
// Unsigned/Signed division based on Patterson and Hennessy's algorithm.
// Copyrighted 2002 by studboy-ga / Google Answers.  All rights reserved.
// Description: Calculates quotient.  The "sign" input determines whether
//              signs (two's complement) should be taken into consideration.

module divide(ready,quotient,remainder,dividend,divider,sign,clk,rst);

   input         clk, rst;
   input         sign;
   input [31:0]  dividend, divider;
   output [31:0] quotient, remainder;
   output        ready;

   reg [31:0]    quotient, quotient_temp;
   reg [63:0]    dividend_copy, divider_copy, diff;
   reg           negative_output;

   wire [31:0]   remainder = (!negative_output) ?
                             dividend_copy[31:0] :
                             -dividend_copy[31:0];

   reg [5:0]     mybit;
   wire         ready = mybit == 0;

   always @( posedge clk )

     if( rst ) begin

        mybit = 6'd32;
        quotient = 0;
        quotient_temp = 0;
        dividend_copy = (!sign || !dividend[31]) ?
                        {32'd0,dividend} :
                        {32'd0,-dividend};
        divider_copy = (!sign || !divider[31]) ?
                       {1'b0,divider,31'd0} :
                       {1'b0,-divider,31'd0};

        negative_output = sign &&
```

```
                        ((divider[31] && !dividend[31])
                       ||(!divider[31] && dividend[31]));

      end
      else if ( mybit > 0 ) begin

         diff = dividend_copy - divider_copy;

         quotient_temp = quotient_temp << 1;

         if( !diff[63] ) begin

            dividend_copy = diff;
            quotient_temp[0] = 1'd1;

         end

         quotient = (!negative_output) ?
                    quotient_temp :
                    -quotient_temp;

         divider_copy = divider_copy >> 1;
         mybit = mybit - 1'b1;

      end
endmodule
```

Appendix-D

This appendix contains the interface file needed to compile OCaml code which avoids the loss of type-safety in embedded programs. It should be read in conjunction with Appendix-E which gives the corresponding code ready for compilation. This file should be used instead of pervasives.mli, and it provides largely compatible replacements except where comparison is to be done on polymorphic types such as association lists.

```
type 'a ref = { mutable contents : 'a; }
external raise : exn -> 'a = "%raise"
external ignore : 'a -> unit = "%ignore"
external string_length : string -> int = "%string_length"
external input_byte : Std.in_channel -> int = "caml_ml_input_char"
external output_char : Std.out_channel -> char -> unit
  = "caml_ml_output_char"
external flush : Std.out_channel -> unit = "caml_ml_flush"
external int_of_char : char -> int = "%identity"
external sys_exit : int -> 'a = "caml_sys_exit"
external ( := ) : 'a ref -> 'a -> unit = "%setfield0"
external ( ! ) : 'a ref -> 'a = "%field0"
external incr : int ref -> unit = "%incr"
external ( ~- ) : int -> int = "%negint"
external ( = ) : 'a -> 'a -> bool = "%equal"
external ( > ) : 'a -> 'a -> bool = "%greaterthan"
external ( >= ) : 'a -> 'a -> bool = "%greaterequal"
external ( < ) : 'a -> 'a -> bool = "%lessthan"
external ( <= ) : 'a -> 'a -> bool = "%lessequal"
external ( <> ) : 'a -> 'a -> bool = "%notequal"
external ( == ) : 'a -> 'a -> bool = "%eq"
external ( != ) : 'a -> 'a -> bool = "%noteq"
external compare : 'a -> 'a -> int = "%compare"
external ( + ) : int -> int -> int = "%addint"
external ( - ) : int -> int -> int = "%subint"
external ( * ) : int -> int -> int = "%mulint"
external ( lsl ) : int -> int -> int = "%lslint"
external ( lsr ) : int -> int -> int = "%lsrint"
external ( asr ) : int -> int -> int = "%asrint"
external ( land ) : int -> int -> int = "%andint"
external ( lor ) : int -> int -> int = "%orint"
external ( lxor ) : int -> int -> int = "%xorint"
external ( && ) : bool -> bool -> bool = "%sequand"
external ( || ) : bool -> bool -> bool = "%sequor"
external not : bool -> bool = "%boolnot"
external ref : 'a -> 'a ref = "%makemutable"
external fst : 'a * 'b -> 'a = "%field0"
external snd : 'a * 'b -> 'b = "%field1"
external succ : int -> int = "%succint"

module Char :
  sig
    external unsafe_chr : int -> char = "%identity"
    external code : char -> int = "%identity"
    val lowercase : char -> char
    val uppercase : char -> char
  end
module Sys :
  sig
    external hw_exn : unit -> bool = "caml_hw_exn"
    val get_config : unit -> string * int * bool
    val get_argv : unit -> string * string array
  end

external ( / ) : int -> int -> int = "%divint"
external ( mod ) : int -> int -> int = "%modint"

module Array :
```

```
  sig
    external make : int -> 'a -> 'a array = "caml_make_vect"
    external length : 'a array -> int = "%array_length"
    external get : 'a array -> int -> 'a = "%array_safe_get"
    external unsafe_get : 'a array -> int -> 'a = "%array_unsafe_get"
    external set : 'a array -> int -> 'a -> unit = "%array_safe_set"
    val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b array -> 'a
    val iter : ('a -> 'b) -> 'a array -> unit
    val to_list : 'a array -> 'a list
  end
val failwith : string -> 'a
val invalid_arg : string -> 'a
val string_of_bool : bool -> string
val stdin : Std.in_channel
val stdout : Std.out_channel
val print_char : char -> unit
val print_int : int -> unit
val print_int_nl : int -> unit
val print_newline : unit -> unit
module Int32 :
  sig
    type forbidden
    type t = int32
    external neg : int32 -> int32 = "%int32_neg"
    external add : int32 -> int32 -> int32 = "%int32_add"
    external sub : int32 -> int32 -> int32 = "%int32_sub"
    external mul : int32 -> int32 -> int32 = "%int32_mul"
    external div : int32 -> int32 -> int32 = "%int32_div"
    external rem : int32 -> int32 -> int32 = "%int32_mod"
    external logand : int32 -> int32 -> int32 = "%int32_and"
    external logor : int32 -> int32 -> int32 = "%int32_or"
    external logxor : int32 -> int32 -> int32 = "%int32_xor"
    external shift_left : int32 -> int -> int32 = "%int32_lsl"
    external shift_right : int32 -> int -> int32 = "%int32_asr"
    external shift_right_logical : int32 -> int -> int32 = "%int32_lsr"
    external of_int : int -> int32 = "%int32_of_int"
    external to_int : int32 -> int = "%int32_to_int"
    external compare : t -> t -> int = "%compare"
  end
module Int64 :
  sig
    type forbidden
    type t = int64
    external neg : int64 -> int64 = "%int64_neg"
    external add : int64 -> int64 -> int64 = "%int64_add"
    external sub : int64 -> int64 -> int64 = "%int64_sub"
    external mul : int64 -> int64 -> int64 = "%int64_mul"
    external div : int64 -> int64 -> int64 = "%int64_div"
    external rem : int64 -> int64 -> int64 = "%int64_mod"
    external logand : int64 -> int64 -> int64 = "%int64_and"
    external logor : int64 -> int64 -> int64 = "%int64_or"
    external logxor : int64 -> int64 -> int64 = "%int64_xor"
    external shift_left : int64 -> int -> int64 = "%int64_lsl"
    external shift_right : int64 -> int -> int64 = "%int64_asr"
    external shift_right_logical : int64 -> int -> int64 = "%int64_lsr"
    external of_int : int -> int64 = "%int64_of_int"
    external to_int : int64 -> int = "%int64_to_int"
    external of_int32 : int32 -> int64 = "%int64_of_int32"
    external to_int32 : int64 -> int32 = "%int64_to_int32"
    external of_nativeint : nativeint -> int64 = "%int64_of_nativeint"
    external to_nativeint : int64 -> nativeint = "%int64_to_nativeint"
    external compare : t -> t -> int = "%compare"
  end
module List :
  sig
    val length_aux : int -> 'a list -> int
    val length : 'a list -> int
    val hd : 'a list -> 'a
    val tl : 'a list -> 'a list
    val nth : 'a list -> int -> 'a
    val rev_append : 'a list -> 'a list -> 'a list
    val rev : 'a list -> 'a list
    val iter : ('a -> 'b) -> 'a list -> unit
    val map : ('a -> 'b) -> 'a list -> 'b list
    val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
    val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
    val mem' : ('a -> 'b -> bool) -> 'b -> 'a list -> bool
    val assoc' : ('a -> 'b -> bool) -> 'b -> ('a * 'c) list -> 'c
    val mem_assoc' : ('a -> 'b -> bool) -> 'b -> ('a * 'c) list -> bool
    val remove_assoc' :
      ('a -> 'b -> bool) -> 'b -> ('a * 'c) list -> ('a * 'c) list
  end
module String :
  sig
```

```
    external length : string -> int = "%string_length"
    external create : int -> string = "caml_create_string"
    external get : string -> int -> char = "%string_safe_get"
    external set : string -> int -> char -> unit = "%string_safe_set"
    external unsafe_get : string -> int -> char = "%string_unsafe_get"
    external unsafe_set : string -> int -> char -> unit
      = "%string_unsafe_set"
    val blit : string -> int -> string -> int -> int -> unit
    val unsafe_blit : string -> int -> string -> int -> int -> unit
    val unsafe_fill : string -> int -> int -> char -> unit
    val eqb : string -> string -> bool
    val output : Std.out_channel -> string -> unit
    val print : string -> unit
    val endline : string -> unit
    val iter : (char -> 'a) -> string -> unit
    val make : int -> char -> string
    val sub : string -> int -> int -> string
    val concat : string -> string list -> string
  end
val output_string : Std.out_channel -> string -> unit
val print_string : string -> unit
val print_endline : string -> unit
val string_of_int : int -> string
val ( @ ) : 'a list -> 'a list -> 'a list
val char_of_int : int -> char
val input_char : Std.in_channel -> char
val ( ^ ) : string -> string -> string
val int_of_string : string -> int
```

Appendix-E

This appendix contains the compilable code needed to compile OCaml code that avoids the loss of type-safety in embedded programs. It should be read in conjunction with Appendix-D which gives the corresponding interface file. The principle idea is to replace calls to library code (other than inlined) to a λ-Calculus routine in type-safe land. The API is not completely compatible with the workstation version of OCaml, largely because of the avoidance of polymorphic comparison, a convenient but potentially dangerous feature.

```
type 'a ref = { mutable contents : 'a }

external raise : exn -> 'a = "%raise"
external ignore : 'a -> unit = "%ignore"
external string_length : string -> int = "%string_length"
external input_byte : Std.in_channel -> int = "caml_ml_input_char"
external output_char : Std.out_channel -> char -> unit = "caml_ml_output_char"
external flush : Std.out_channel -> unit = "caml_ml_flush"
external int_of_char : char -> int = "%identity"
external sys_exit : int -> 'a = "caml_sys_exit"
external ( := ) : 'a ref -> 'a -> unit = "%setfield0"
external ( ! ) : 'a ref -> 'a = "%field0"
external incr : int ref -> unit = "%incr"
external ( ~- ) : int -> int = "%negint"
external ( = ) : 'a -> 'a -> bool = "%equal"
external ( > ) : 'a -> 'a -> bool = "%greaterthan"
external ( >= ) : 'a -> 'a -> bool = "%greaterequal"
external ( < ) : 'a -> 'a -> bool = "%lessthan"
external ( <= ) : 'a -> 'a -> bool = "%lessequal"
external ( <> ) : 'a -> 'a -> bool = "%notequal"
external ( == ) : 'a -> 'a -> bool = "%eq"
external ( != ) : 'a -> 'a -> bool = "%noteq"
external compare : 'a -> 'a -> int = "%compare"
external ( + ) : int -> int -> int = "%addint"
external ( - ) : int -> int -> int = "%subint"
external ( *  ) : int -> int -> int = "%mulint"
external ( lsl ) : int -> int -> int = "%lslint"
external ( lsr ) : int -> int -> int = "%lsrint"
external ( asr ) : int -> int -> int = "%asrint"
external ( land ) : int -> int -> int = "%andint"
external ( lor ) : int -> int -> int = "%orint"
external ( lxor ) : int -> int -> int = "%xorint"
external ( && ) : bool -> bool -> bool = "%sequand"
external ( || ) : bool -> bool -> bool = "%sequor"
external not : bool -> bool = "%boolnot"
external ref : 'a -> 'a ref = "%makemutable"
external fst : 'a * 'b -> 'a = "%field0"
external snd : 'a * 'b -> 'b = "%field1"
external succ : int -> int = "%succint"

module Char = struct
  external unsafe_chr : int -> char = "%identity"
  external code : char -> int = "%identity"
  let of_int n = unsafe_chr (n land 255)

  let lowercase c =
    if (c >= 'A' && c <= 'Z')
    || (c >= '\192' && c <= '\214')
    || (c >= '\216' && c <= '\222')
    then unsafe_chr(code c + 32)
    else c

  let uppercase c =
    if (c >= 'a' && c <= 'z')
    || (c >= '\224' && c <= '\246')
    || (c >= '\248' && c <= '\254')
```

```
    then unsafe_chr(code c - 32)
    else c

end

module Sys = struct
  external hw_exn : unit -> bool = "caml_hw_exn"

  let get_config () = ("standalone", 32, false)
  let get_argv () = ("a.out", [|"a.out"|])

end

external ( / ) : int -> int -> int = "%divint"
external ( mod ) : int -> int -> int = "%modint"

module Array = struct
  external get: 'a array -> int -> 'a = "%array_safe_get"
  external make: int -> 'a -> 'a array = "caml_make_vect"
  external length : 'a array -> int = "%array_length"
  external get: 'a array -> int -> 'a = "%array_safe_get"
  external unsafe_get: 'a array -> int -> 'a = "%array_unsafe_get"
  external set: 'a array -> int -> 'a -> unit = "%array_safe_set"

  let fold_left f x a =
    let r = ref x in
    for i = 0 to length a - 1 do
      r := f !r (unsafe_get a i)
    done;
    !r

  let iter f a = for i = 0 to length a - 1 do f(unsafe_get a i) done

  let to_list a =
    let rec tolist i res =
      if i < 0 then res else tolist (i - 1) (unsafe_get a i :: res) in
    tolist (length a - 1) []
end

let failwith s = raise(Failure s)
let invalid_arg s = raise(Invalid_argument s)
let string_of_bool b = if b then "true" else "false"
let stdin = Std.in_
let stdout = Std.out_
let print_char c = output_char (stdout) c
let rec print_int n = if n > 9 then
  print_int (n/10); print_char (Char.unsafe_chr (n mod 10 + Char.code '0'))
let print_int n = if n < 0 then (print_char '-'; print_int (0-n)) else print_int n
let print_int_nl n = print_int n; print_char '\n'; flush (stdout)
let print_newline () = output_char (stdout) '\n'; flush (stdout)

module Int32 = struct
  type forbidden
  type t = int32
  external neg : int32 -> int32 = "%int32_neg"
  external add : int32 -> int32 -> int32 = "%int32_add"
  external sub : int32 -> int32 -> int32 = "%int32_sub"
  external mul : int32 -> int32 -> int32 = "%int32_mul"
  external div : int32 -> int32 -> int32 = "%int32_div"
  external rem : int32 -> int32 -> int32 = "%int32_mod"
  external logand : int32 -> int32 -> int32 = "%int32_and"
  external logor : int32 -> int32 -> int32 = "%int32_or"
  external logxor : int32 -> int32 -> int32 = "%int32_xor"
  external shift_left : int32 -> int -> int32 = "%int32_lsl"
  external shift_right : int32 -> int -> int32 = "%int32_asr"
  external shift_right_logical : int32 -> int -> int32 = "%int32_lsr"
  external of_int : int -> int32 = "%int32_of_int"
  external to_int : int32 -> int = "%int32_to_int"
  external compare : t -> t -> int = "%compare"
end

module Int64 = struct
  type forbidden
  type t = int64
  external neg : int64 -> int64 = "%int64_neg"
  external add : int64 -> int64 -> int64 = "%int64_add"
  external sub : int64 -> int64 -> int64 = "%int64_sub"
  external mul : int64 -> int64 -> int64 = "%int64_mul"
  external div : int64 -> int64 -> int64 = "%int64_div"
  external rem : int64 -> int64 -> int64 = "%int64_mod"
  external logand : int64 -> int64 -> int64 = "%int64_and"
  external logor : int64 -> int64 -> int64 = "%int64_or"
  external logxor : int64 -> int64 -> int64 = "%int64_xor"
  external shift_left : int64 -> int -> int64 = "%int64_lsl"
```

```
  external shift_right : int64 -> int -> int64 = "%int64_asr"
  external shift_right_logical : int64 -> int -> int64 = "%int64_lsr"
  external of_int : int -> int64 = "%int64_of_int"
  external to_int : int64 -> int = "%int64_to_int"
  external of_int32 : int32 -> int64 = "%int64_of_int32"
  external to_int32 : int64 -> int32 = "%int64_to_int32"
  external of_nativeint : nativeint -> int64 = "%int64_of_nativeint"
  external to_nativeint : int64 -> nativeint = "%int64_to_nativeint"
  external compare : t -> t -> int = "%compare"
end

module List = struct
  let rec length_aux len = function
      [] -> len
    | a::l -> length_aux (len + 1) l

  let length l = length_aux 0 l

  let hd = function
      [] -> failwith "hd"
    | a::l -> a

  let tl = function
      [] -> failwith "tl"
    | a::l -> l

  let nth l n =
    if n < 0 then invalid_arg "List.nth" else
    let rec nth_aux l n =
      match l with
      | [] -> failwith "nth"
      | a::l -> if n = 0 then a else nth_aux l (n-1)
    in nth_aux l n

  let rec rev_append l1 l2 =
    match l1 with
      [] -> l2
    | a :: l -> rev_append l (a :: l2)

  let rev l = rev_append l []

  let rec iter f = function
      [] -> ()
    | a::l -> f a; iter f l

  let rec map f = function
      [] -> []
    | a::l -> let r = f a in r :: map f l

  let rec fold_left f accu l =
    match l with
      [] -> accu
    | a::l -> fold_left f (f accu a) l

  let rec fold_right f l accu =
    match l with
      [] -> accu
    | a::l -> f a (fold_right f l accu)

  let rec mem' cmp x = function
      [] -> false
    | a::l -> cmp a x || mem' cmp x l

  let rec assoc' cmp x = function
      [] -> raise Not_found
    | (a,b)::l -> if cmp a x then b else assoc' cmp x l

  let rec mem_assoc' cmp x = function
    | [] -> false
    | (a, b) :: l -> cmp a x || mem_assoc' cmp x l

  let rec remove_assoc' cmp x = function
    | [] -> []
    | (a, b as pair) :: l ->
        if cmp a x then l else pair :: remove_assoc' cmp x l
end

module String = struct
  external length : string -> int = "%string_length"
  external create : int -> string = "caml_create_string"
  external get : string -> int -> char = "%string_safe_get"
  external set : string -> int -> char -> unit = "%string_safe_set"
  external unsafe_get : string -> int -> char = "%string_unsafe_get"
  external unsafe_set : string -> int -> char -> unit = "%string_unsafe_set"
```

```
  let blit (str1: string) (int1: int) (str2: string) (int2: int) (int3: int): unit =
   for i = 0 to int3-1 do
     set str2 (int2+i) (get str1 (int1+i))
   done

  let unsafe_blit (str1: string) (int1: int) (str2: string) (int2: int) (int3: int): unit =
   for i = 0 to int3-1 do
     unsafe_set str2 (int2+i) (unsafe_get str1 (int1+i))
   done

  let unsafe_fill (str1: string) (int1: int) (int2: int) (char1: char): unit =
    assert(int1+int2 <= length str1);
    for i = int1 to int1+int2-1 do
      unsafe_set str1 i char1
    done

  let eqb (a:string) (b:string) =
    let len1 = length a and len2 = length b in
    if len1 = len2 then
      (try
        for i = 0 to len1 - 1 do
          if unsafe_get a i <> unsafe_get b i then raise (Failure "eqb");
        done;
        true
      with
        Failure _ -> false)
    else
      false

  let output oc s = for i = 0 to (string_length s) - 1 do output_char oc (unsafe_get s i) done
  let print s = output (stdout) s
  let endline s = output (stdout) s; print_newline()

  let iter f a =
    for i = 0 to length a - 1 do f(unsafe_get a i) done

  let make n c =
    let s = create n in
    unsafe_fill s 0 n c;
    s

  let sub s ofs len =
    if ofs < 0 || len < 0 || ofs > length s - len
    then invalid_arg "String.sub"
    else begin
      let r = create len in
      unsafe_blit s ofs r 0 len;
      r
    end

  let concat sep l =
    match l with
      [] -> ""
    | hd :: tl ->
        let num = ref 0 and len = ref 0 in
        List.iter (fun s -> incr num; len := !len + length s) l;
        let r = create (!len + length sep * (!num - 1)) in
        unsafe_blit hd 0 r 0 (length hd);
        let pos = ref(length hd) in
        List.iter
          (fun s ->
            unsafe_blit sep 0 r !pos (length sep);
            pos := !pos + length sep;
            unsafe_blit s 0 r !pos (length s);
            pos := !pos + length s)
          tl;
        r

end

let output_string = String.output
let print_string = String.print
let print_endline = String.endline

let string_of_int (arg1: int): string =
  let rec format_int_d ostr ptr n len =
    if n > 9 && len > 0 then
      format_int_d ostr ptr (n/10) (len-1);
    let ch = Char.unsafe_chr (n mod 10 + Char.code '0') in
    ostr.[!ptr] <- ch;
    incr ptr in
  let ostr = String.create 20 and ptr = ref 0 in
  if arg1 < 0 then
```

```
    (ostr.[!ptr] <- '-'; incr ptr; format_int_d ostr ptr (0-arg1) (String.length ostr))
  else
    format_int_d ostr ptr arg1 (String.length ostr);
  String.sub ostr 0 !ptr

let rec ( @ ) l1 l2 =
  match l1 with
    [] -> l2
  | hd :: tl -> hd :: (tl @ l2)

let input_char in_channel = match input_byte in_channel with
| (-1) -> raise End_of_file
| oth -> Char.of_int oth

let ( ^ ) s1 s2 =
  let l1 = String.length s1 and l2 = String.length s2 in
  if l1 < 0 || l2 < 0 then
    begin
      print_endline "string create with negative length";
      raise(Failure "concat")
    end;
  let s = String.create (l1 + l2) in
  String.blit s1 0 s 0 l1;
  String.blit s2 0 s l1 l2;
  s

let int_of_string str =
  let rslt = ref 0 in
  let sign = str.[0] = '-' in
  let len = String.length str in
  for i = if sign then 1 else 0 to len - 1 do
    match str.[i] with
    | '0'..'9' as dig -> let dig' = Char.code dig - Char.code '0' in
                         rslt := !rslt * 10 + (if sign then -dig' else dig')
    | _ -> invalid_arg "int_of_string"
  done;
  !rslt

let char_of_int = Char.of_int
```

Appendix-F

This appendix contains the fault-tolerant library of high-level Verilog primitives, suitable for substitution into the net list produced by the tool of chapter 7 (Comparison of TMR and Dual-rail logic). Each module is parameterised by the word width, after elaboration each unique instance is elaborated to a different module name corresponding to the set of parameters chosen.

```
module F_DVL_BUF(Y, A);
   parameter width = 1;
   input [width*2:1] A;
   output [width*2:1] Y;

   assign Y = A;
endmodule // DVL_BUF

module F_DVL_ARI_MUL(Y, A, B);
   parameter width1 = 1;
   parameter width2 = 1;
   parameter width3 = 1;
   output [width1*2:1] Y;
   input [width2*2:1] A;
   input [width3*2:1] B;

   assign Y = A*B;
endmodule // DVL_ARI_MUL

module F_DVL_EQ(Y, A, B);
   parameter width1 = 1;
   parameter width2 = 1;
   output [2:1] Y;
   input [width1*2:1] A;
   input [width2*2:1] B;

   assign Y = !(|(A[width1:1]^B[width2:1]));

endmodule // DVL_EQ

module F_DVL_BW_NOT(Y, A);
   parameter width = 1;
   input [width*2:1] A;
   output [width*2:1] Y;

   assign Y = ~A;

endmodule // DVL_BW_NOT

module F_DVL_RED_AND(Y, A);
   parameter width = 1;
   output [2:1] Y;
   input [width*2:1] A;

   assign Y = & A[width:1];

endmodule // DVL_RED_AND

module F_DVL_ARI_ADD(Y, A, B);
   parameter width1 = 1;
   parameter width2 = 1;
   parameter width3 = 1;
   output [width1*2:1] Y;
   input [width2*2:1] A;
   input [width3*2:1] B;

   assign Y = A+B;
```

```
endmodule // DVL_ARI_ADD

module F_DVL_ARI_SUB(Y, A, B);
   parameter width1 = 1;
   parameter width2 = 1;
   parameter width3 = 1;
   output [width1*2:1] Y;
   input [width2*2:1] A;
   input [width3*2:1] B;

   assign Y = A-B;
endmodule // DVL_ARI_SUB

module F_DVL_MUX(Y, A, B, S);
   parameter width = 1;
   output [width*2:1] Y;
   input [width*2:1] A, B;
   input [2:1]     S;


   assign Y = S[1] ? B : A;

endmodule // DVL_MUX

module F_DVL_LOG_OR(Y, A, B);
   parameter width = 1;
   input [width*2:1] A, B;
   output [width*2:1] Y;

   assign Y = A[width:1] | B[width:1];
endmodule // DVL_LOG_OR

module F_DVL_LOG_AND(Y, A, B);
   parameter width = 1;
   input [width*2:1] A, B;
   output [width*2:1] Y;

   assign Y = A[width:1] & B[width:1];
endmodule // DVL_LOG_AND

module F_DVL_X_CELL(Y);
   parameter width = 1;
   output [width*2:1] Y;

   assign Y = -1;
endmodule // DVL_X_CELL

module F_DVL_SH_L(Y, A, B);
   parameter width1 = 1;
   parameter width2 = 1;
   parameter width3 = 1;
   output [width1*2:1] Y;
   input [width2*2:1] A;
   input [width3*2:1] B;

   assign Y = A << B[width3:1];

endmodule // DVL_SH_L

module F_DVL_SH_R(Y, A, B);
   parameter width1 = 1;
   parameter width2 = 1;
   parameter width3 = 1;
   output [width1*2:1] Y;
   input [width2*2:1] A;
   input [width3*2:1] B;

   assign Y = A[width2:1] >> B[width3:1];

endmodule // DVL_SH_R

module F_DVL_BW_XOR(Y, A, B);
   parameter width = 1;
   output [width*2:1] Y;
   input [width*2:1] A, B;

   assign Y = A^B;
endmodule // DVL_BW_XOR

module F_DVL_BW_AND(Y, A, B);
   parameter width = 1;
   output [width*2:1] Y;
   input [width*2:1] A, B;
```

```
   assign Y = A&B;
endmodule // DVL_BW_AND

module F_DVL_BW_OR(Y, A, B);
   parameter width = 1;
   input [width*2:1] A, B;
   output [width*2:1] Y;

   assign Y = A|B;
endmodule // DVL_BW_OR

module F_DVL_DFF_SYNC(Q, D, CK);
   parameter width = 1;
   output reg [width*2:1] Q;
   input [width*2:1] D;
   input [2:1]     CK;

   always @(posedge CK[1]) Q = D;
endmodule // DVL_DFF_SYNC

module F_DVL_RED_OR(Y, A);
   parameter width = 1;
   output [2:1] Y;
   input [width*2:1] A;

   assign Y = | A[width:1];

endmodule // DVL_RED_OR
```

Appendix-G

This appendix contains the compiled output from OCaml for the factorial example of section 5.5 (Extracted Compilation Example). It takes the form of an OCaml data structure which can be serialised to/deserialised from disk, and subsequently merged with other modules such as libraries. It sits in the toolchain after register allocation but before linking and final conversion to a memory which will ultimately be compiled to ROM in Xilinx land. The names of the constructors are intended to be self-explanatory and correspond to a context-free version of the second intermediate language of the compiler as follows:

```
EDouble - 64-bit-aligned 64-bit float
EDouble_u - word-aligned 64-bit float
ESingle - 32-bit float
Eabort reason - prematurely terminate program
Eabsf - modulus of a float
Eadd - integer addition
Eaddf - floating point addition
Eadj (n) - stack adjustment
Ealign n - align memory layout
Ealloc (int1) - allocate a fixed amount of heap
Eand - bitwise logical and
Earith(op,arg0,arg1) - calculate result of dyadic arithmetic
Easr - arithmetic shift left
Ebased(s, d) - represent a symbolic base address with an integer offset
Eblockcopy(str,off,dest,length) - block copy of a data structure
Ebool str - represent result of a relational/logic operation as a truth value
Ebyte_signed - represent a signed byte
Ebyte_unsigned - represent an unsigned byte
Ecall(arg, arglst) - execution a function call (with argument list for debugging)
Ecall_imm (str) - machine representation of function with immediate argument
Ecall_ind - machine representation of function indirect
Ecaml_ml_array_bound_error() - raise an array bounds exception
Ecaml_sys_abort () - abort the program
Ecaml_sys_exit () - exit the program
Echeckbound - check the bounds of an operation
Ecomment s - include comment in listing
Ecomp (integer_comparison) - represent integer comparison as an expression
Ecompare(op, left, right) - relational comparison
Ecompare_flt(op, negate, left, right) - floating-point relational comparison
Econd(cond, dest) - conditional branch
Econst_float (str) - represent a constant floating point value in global memory
Econst_int (nativeint) - represent a constant native integer in global memory
Econst_symbol (str) - represent a constant symbol in global memory
Econstsym(force, s) - refer to a constant symbol in global memory
Ecopy(src, dest) - copy an expression between registers and/or memory
Edefaults - represent default control store contents
Edefine_label(func, label) - define a label location inside an enclosing function
Edefine_symbol s - define a symbol location inside global memory
Edelayslot(reg,byte) - wait for word/byte to arrive from global memory
Ediv - integer divide
Edivf - floating point divide
Edouble f - 64-bit floating point representation as an immediate value
Eeq - relational equality
Eeventest - test for pointer to object
Eextcall (str, bool1) - represent external call as a machine operation
Eexternalcall(dest, alloc, arglst) - call external assembly language stub
Eextref dest - reference to an external function
Efalsetest - boolean truth testing
Efloatconst flt - use a floating-point constant inline
Efloatfunc(oper,arg0) - represent a floating-point monadic function
Efloatofint - convert integer to float
Efloatop(oper,arg0,arg1) - represent a floating-point dyadic function
Efloattest (cmm_comparison, bool) - represent relational floating-point operation
Efundecl(num_slots,name,calls) - identify entry point of a function, and whether leaf
Ege - relational greater than or equal
Eglobal_symbol str - mark a global symbol in global memory
Egoto dest - unconditional branch
Egt - relational greater than
Einit(array,dest) - optimised function entry initialisation
Eint num- represent native integer in global memory
Eint16 num - represent 16-bit integer in global memory
Eint32 num - represent 32-bit integer in global memory
Eint8 num - represent 8-bit integer in global memory
Eintconst32 num - represent 32-bit integer in immediate mode
Eintoffloat - truncate floating-point value to an integer
Eintop (integer_operation) - internal representation of integer operation
Eintop_imm (oper, int1) - internal representation of integer immediate operation
Einttest (compare) - internal representation of integer relational testing
Einttest_imm (compare, int1) - internal representation of relational immediate testing
Elab s - represent a global label in code memory
Elabdef dest - represent a branch target in code memory
Elabel(func, dest) - represent a function and string as a label
Elabel_address(func,lbl) - reference to a label in global memory
Elabelref dest - reference to a label
Ele - relational less than or equal
```

```
Eload (cmm_memory_chunk, arch_addressing_mode) - represent a load from memory
Elsl - logical shift left
Elsr - logical shift right
Elt - relational less than
Emisc misc - represent unresolved symbols etc in global memory
Emod - integer modulo operation
Emove - represent a move operation
Emul - integer multiplication
Emulf - floating-point multiplication
Ene - relational not equal
Enegf - floating-point negation
Enop - no operation
Eoddtest - tests for unboxed integer
Eoffset(r,d) - allow offset addressing from a register
Eor - logical or
Epcstore() - save pc for try operation
Epoptrap() - successful completion of try operation
Epushtrap() - begin try operation
Eraise_exn() - raise exception during try operation
Ereadaddrb s - read byte from global memory
Ereadaddrd s - read 64-bit double from global memory
Ereadaddrw s - read 32-bit word from global memory
Eregload n - refer to a register source in an expression
Eregstore n - refer to a register destination in an expression
Ereload - represent a reload as an internal machine operation
Ereloadretaddr(int1,r) - reload return address at end of a function
Eresult(label,reslst) - refer to function result (for debugging)
Ereturn int1 - return to calling function
Esetuptrap(func,lbl,lab) - setup destination for trapping subsequent exceptions
Eshiftconst32(negate, n,s) - represent a non-standard constant
Esigned (cmm_comparison) - represent a signed comparison
Esingle f - represent single-precision float constant
Esixteen_signed - represent 16-bit signed integer
Esixteen_unsigned - represent 16-bit unsigned integer
Eskip n - skip some bytes
Especific(op,arg,res) - represent processor-specific operation
Especific_op (arch_specific_operation) - container for architecture specific extensions
Espill - represent a spill as an internal machine operation
Espload() - load stack pointer to recover from exception
Espstore() - store stack pointer at start of try block
Estackloadref ofs - represent local variables as a source in an expression
Estackoffset (int1) - represent address of a local variable
Estackstoreref ofs - represent local variables as a destination in an expression
Estatecase(r, r2, d) - placeholder for instruction scheduling
Estore (cmm_memory_chunk, arch_addressing_mode) - represent a store to global memory
Estring s - represent constant string in global memory
Esub - integer subtraction
Esubf - floating-point subtraction
Eswitch(func,arg0,argvec) - represent cases in a match expression
Esymbol_address s - represent an object address in global memory
Esymbol_address_off(s,off) - represent object address with constant offset in global memory
Etailcall_imm str - represent tail call with immediate argument
Etailcall_ind - represent tail call with indirect argument
Etailcallimm(calls,int1,dst, arglst) - execute tail call with immediate argument
Etailcallind(calls,int1,dst, arglst) - execute tail call with indirect argument
Etask nam - placeholder for a named Verilog task
Etask2(nam,arg) - placeholder for a named Verilog task with argument
Ethirtytwo_signed - represent 32-bit signed integer
Ethirtytwo_unsigned - represent 32-bit unsigned integer
Etrap str - placeholder for an operating system trap
Etruetest - convert relational test to a boolean
Eunknownload - emulate unsupported load
Eunknownstore - emulate unsupported store
Eunsigned (cmm_comparison) - represent an unsigned comparison (such as pointer comparison)
Eword - represent a word in global memory
Ewriteaddrb s - write byte to global memory
Ewriteaddrd s - write 64-bit double to global memory
Ewriteaddrw s - write 32-bit word to global memory
Exor - integer exclusive-or
```

The corresponding listing is shown below:

```
open Arch

let _camlShort__dump = ([("_camlShort", [Eint(0x00001C00n);
Eglobal_symbol("_camlShort");
Edefine_symbol("_camlShort");
Esymbol_address("_camlShort__7");
Esymbol_address("_camlShort__6");
Esymbol_address("_camlShort__5");
Esymbol_address("_camlShort__4");
Esymbol_address("_camlShort__1");
Esymbol_address("_camlShort__3");
Esymbol_address("_camlShort__2")]);

("_camlShort__1", [Eglobal_symbol("_camlShort__1");
Eint(0x00000407n);
Edefine_symbol("_camlShort__1");
Elabel_address("_camlShort__data_prefix_", 8);
Eint(0x00000C02n);
Edefine_label("_camlShort__data_prefix_", 8);
Elabel_address("_camlShort__data_prefix_", 9);
Elabel_address("_camlShort__data_prefix_", 10);
Elabel_address("_camlShort__data_prefix_", 11);
Eint(0x00000C02n);
Edefine_label("_camlShort__data_prefix_", 11);
Eint(0x00000001n);
Eint(0x00000001n);
Elabel_address("_camlShort__data_prefix_", 12);
Eint(0x00000C06n);
Edefine_label("_camlShort__data_prefix_", 12);
Elabel_address("_camlShort__data_prefix_", 13);
Elabel_address("_camlShort__data_prefix_", 14);
Elabel_address("_camlShort__data_prefix_", 16);
Eint(0x00000805n);
Edefine_label("_camlShort__data_prefix_", 16);
Elabel_address("_camlShort__data_prefix_", 17);
Elabel_address("_camlShort__data_prefix_", 18);
Eint(0x00000801n);
Edefine_label("_camlShort__data_prefix_", 18);
Elabel_address("_camlShort__data_prefix_", 19);
Elabel_address("_camlShort__data_prefix_", 21);
Eint(0x00000404n);
Edefine_label("_camlShort__data_prefix_", 21);
Elabel_address("_camlShort__data_prefix_", 22);
Eint(0x00000400n);
Edefine_label("_camlShort__data_prefix_", 22);
Eint(0x00000001n);
Eint(0x00000400n);
Edefine_label("_camlShort__data_prefix_", 19);
Elabel_address("_camlShort__data_prefix_", 20);
Eint(0x00000400n);
Edefine_label("_camlShort__data_prefix_", 20);
Eint(0x00000001n);
Eint(0x00000400n);
Edefine_label("_camlShort__data_prefix_", 17);
Eint(0x00000001n);
Eint(0x00000403n);
Edefine_label("_camlShort__data_prefix_", 14);
Elabel_address("_camlShort__data_prefix_", 15);
Eint(0x00000400n);
Edefine_label("_camlShort__data_prefix_", 15);
Eint(0x00000001n);
Eint(0x00000400n);
Edefine_label("_camlShort__data_prefix_", 13);
Eint(0x00000001n);
Eint(0x00000800n);
Edefine_label("_camlShort__data_prefix_", 10);
Eint(0x00000001n);
Eint(0x00000001n);
Eint(0x00000400n);
Edefine_label("_camlShort__data_prefix_", 9);
Eint(0x00000001n)]);

("_camlShort__2", [Eint(0x00000CF7n);
Edefine_symbol("_camlShort__2");
Esymbol_address("_caml_curry2");
Eint(0x00000005n);
Esymbol_address("_camlShort__reduce_n_1165")]);

("_camlShort__3", [Eint(0x000008F7n);
Edefine_symbol("_camlShort__3");
Esymbol_address("_camlShort__fact_calc_1163");
Eint(0x00000003n)]);

("_camlShort__4", [Eint(0x000008F7n);
Edefine_symbol("_camlShort__4");
Esymbol_address("_camlShort__reduce_1138");
Eint(0x00000003n)]);

("_camlShort__5", [Eint(0x000008F7n);
Edefine_symbol("_camlShort__5");
Esymbol_address("_camlShort__redvalue_1133");
Eint(0x00000003n)]);

("_camlShort__6", [Eint(0x00000CF7n);
Edefine_symbol("_camlShort__6");
Esymbol_address("_caml_curry3");
Eint(0x00000007n);
Esymbol_address("_camlShort__subst_1115")]);

("_camlShort__7", [Eint(0x00000CF7n);
Edefine_symbol("_camlShort__7");
Esymbol_address("_caml_curry2");
Eint(0x00000005n);
Esymbol_address("_camlShort__beq_id_1097")]);

],
([Elabdef(Elab("_camlShort__code_begin"));
Efundecl(0, _camlShort__beq_id_1097, false);
```

```
Elabdef(Elabel("_camlShort__beq_id_1097", 2000001));
Econd(Ecompare (Esigned (Eeq), Eregload(0), Eintconst32(1l)), Egoto(Elabel("_camlShort__beq_id_1097", 101)));
Econd(Ecompare (Esigned (Eeq), Eregload(1), Eintconst32(1l)), Egoto(Elabel("_camlShort__beq_id_1097", 102)));
Ecopy(Ereadaddrw(Eoffset(Eregload(0), 0)), Eregstore(0));
Ecopy(Ereadaddrw(Eoffset(Eregload(1), 0)), Eregstore(1));
Econd(Enop, Egoto(Elabel("_camlShort__beq_id_1097", 2000001)));
Elabdef(Elabel("_camlShort__beq_id_1097", 102));
Ecopy(Eshiftconst32(false, 1l, 0), Eregstore(0));
Ereturn(0);
Elabdef(Elabel("_camlShort__beq_id_1097", 101));
Econd(Ecompare (Esigned (Eeq), Eregload(1), Eintconst32(1l)), Egoto(Elabel("_camlShort__beq_id_1097", 100)));
Ecopy(Eshiftconst32(false, 1l, 0), Eregstore(0));
Ereturn(0);
Elabdef(Elabel("_camlShort__beq_id_1097", 100));
Ecopy(Eshiftconst32(false, 31, 0), Eregstore(0));
Ereturn(0);
Eadj(0);
Efundecl(24, _camlShort__subst_1115, true);
Elabdef(Elabel("_camlShort__subst_1115", 2000001));
Ecopy(Ereadaddrb(Eoffset(Eregload(2), -4)), Eregstore(3));
Ecopy(Eregload(2), Estackstoreref(12));
Ecopy(Eregload(1), Estackstoreref(16));
Ecopy(Eregload(0), Estackstoreref(8));
Econd(Ecompare (Esigned (Eeq), Eregload(3), Eintconst32(01)), Egoto(Elabel("_camlShort__subst_1115", 113)));
Econd(Ecompare (Esigned (Eeq), Eregload(3), Eintconst32(1l)), Egoto(Elabel("_camlShort__subst_1115", 111)));
Econd(Ecompare (Esigned (Eeq), Eregload(3), Eintconst32(2l)), Egoto(Elabel("_camlShort__subst_1115", 110)));
Econd(Ecompare (Esigned (Eeq), Eregload(3), Eintconst32(3l)), Egoto(Elabel("_camlShort__subst_1115", 107)));
Econd(Ecompare (Esigned (Eeq), Eregload(3), Eintconst32(4l)), Egoto(Elabel("_camlShort__subst_1115", 106)));
Econd(Ecompare (Esigned (Eeq), Eregload(3), Eintconst32(5l)), Egoto(Elabel("_camlShort__subst_1115", 105)));
Econd(Ecompare (Esigned (Eeq), Eregload(3), Eintconst32(6l)), Egoto(Elabel("_camlShort__subst_1115", 104)));
Econd(Ecompare (Esigned (Eeq), Eregload(3), Eintconst32(7l)), Egoto(Elabel("_camlShort__subst_1115", 103)));
Elabdef(Elabel("_camlShort__subst_1115", 113));
Ecopy(Ereadaddrw(Eoffset(Eregload(2), 0)), Eregstore(1));
Ecall(Econstsym(true, "_camlShort__beq_id_1097"), [Eregload(0);Eregload(1)]);
Eresult("", [Eregload(0)]);
Econd(Ecompare (Esigned (Eeq), Eregload(0), Eintconst32(1l)), Egoto(Elabel("_camlShort__subst_1115", 112)));
Ereloadretaddr(24, Eregstore(14));
Ecopy(Estackloadref(16), Eregstore(0));
Ereturn(24);
Elabdef(Elabel("_camlShort__subst_1115", 112));
Ereloadretaddr(24, Eregstore(14));
Ecopy(Estackloadref(12), Eregstore(0));
Ereturn(24);
Elabdef(Elabel("_camlShort__subst_1115", 111));
Ecopy(Ereadaddrw(Eoffset(Eregload(2), 4)), Eregstore(2));
Ecall(Econstsym(true, "_camlShort__subst_1115"), [Eregload(0);Eregload(1);Eregload(2)]);
Eresult("", [Eregload(0)]);
Ecopy(Estackloadref(12), Eregstore(1));
Ecopy(Eregload(0), Estackstoreref(0));
Ecopy(Estackloadref(8), Eregstore(0));
Ecopy(Ereadaddrw(Eoffset(Eregload(1), 0)), Eregstore(2));
Ecopy(Estackloadref(16), Eregstore(1));
Ecall(Econstsym(true, "_camlShort__subst_1115"), [Eregload(0);Eregload(1);Eregload(2)]);
Eresult("", [Eregload(0)]);
Ecopy(Eregload(10), Eregstore(1));
Ecopy(Earith(Eadd, Eregload(10), Eintconst32(12l)), Eregstore(10));
Ereloadretaddr(24, Eregstore(14));
Ecopy(Estackloadref(0), Eregstore(4));
Ecopy(Eshiftconst32(false, 1l, 0), Eregstore(2));
Ecopy(Earith(Eadd, Eregload(2), Eshiftconst32(false, 1l, 11)), Eregstore(2));
Ecopy(Eregload(2), Ewriteaddrw(Eoffset(Eregload(1), -4)));
Ecopy(Eregload(0), Ewriteaddrw(Eoffset(Eregload(1), 0)));
Ecopy(Eregload(4), Ewriteaddrw(Eoffset(Eregload(1), 4)));
Ecopy(Eregload(1), Eregstore(0));
Ereturn(24);
Elabdef(Elabel("_camlShort__subst_1115", 110));
Ecopy(Ereadaddrw(Eoffset(Eregload(2), 0)), Eregstore(1));
Ecopy(Ereadaddrw(Eoffset(Eregload(2), 8)), Eregstore(3));
Ecopy(Eregload(1), Estackstoreref(0));
Ecopy(Eregload(3), Estackstoreref(4));
Ecall(Econstsym(true, "_camlShort__beq_id_1097"), [Eregload(0);Eregload(1)]);
Eresult("", [Eregload(0)]);
Econd(Ecompare (Esigned (Eeq), Eregload(0), Eintconst32(1l)), Egoto(Elabel("_camlShort__subst_1115", 109)));
Ecopy(Estackloadref(4), Eregstore(6));
Econd(Enop, Egoto(Elabel("_camlShort__subst_1115", 108)));
Elabdef(Elabel("_camlShort__subst_1115", 109));
Ecopy(Estackloadref(4), Eregstore(2));
Ecopy(Estackloadref(16), Eregstore(1));
Ecopy(Estackloadref(8), Eregstore(0));
Ecall(Econstsym(true, "_camlShort__subst_1115"), [Eregload(0);Eregload(1);Eregload(2)]);
Eresult("", [Eregload(0)]);
Ecopy(Eregload(0), Eregstore(6));
Elabdef(Elabel("_camlShort__subst_1115", 108));
Ecopy(Eregload(10), Eregstore(0));
Ecopy(Earith(Eadd, Eregload(10), Eintconst32(16l)), Eregstore(10));
Ecopy(Estackloadref(0), Eregstore(1));
Ecopy(Eshiftconst32(false, 1l, 1), Eregstore(8));
Ecopy(Earith(Eadd, Eregload(8), Eshiftconst32(false, 31, 10)), Eregstore(8));
Ecopy(Eregload(8), Ewriteaddrw(Eoffset(Eregload(0), -4)));
Ecopy(Eregload(1), Ewriteaddrw(Eoffset(Eregload(0), 0)));
Ecopy(Estackloadref(12), Eregstore(1));
Ereloadretaddr(24, Eregstore(14));
Ecopy(Ereadaddrw(Eoffset(Eregload(1), 4)), Eregstore(1));
Ecopy(Eregload(1), Ewriteaddrw(Eoffset(Eregload(0), 4)));
Ecopy(Eregload(6), Ewriteaddrw(Eoffset(Eregload(0), 8)));
Ereturn(24);
Elabdef(Elabel("_camlShort__subst_1115", 107));
Ecopy(Eregload(10), Eregstore(0));
Ecopy(Earith(Eadd, Eregload(10), Eintconst32(8l)), Eregstore(10));
Ecopy(Eshiftconst32(false, 3l, 0), Eregstore(3));
Ecopy(Earith(Eadd, Eregload(3), Eshiftconst32(false, 1l, 10)), Eregstore(3));
Ecopy(Eregload(3), Ewriteaddrw(Eoffset(Eregload(0), -4)));
Ecopy(Ereadaddrw(Eoffset(Eregload(2), 0)), Eregstore(3));
Ereloadretaddr(24, Eregstore(14));
Ecopy(Eregload(3), Ewriteaddrw(Eoffset(Eregload(0), 0)));
Ereturn(24);
Elabdef(Elabel("_camlShort__subst_1115", 106));
Ecopy(Ereadaddrw(Eoffset(Eregload(2), 0)), Eregstore(2));
Ecall(Econstsym(true, "_camlShort__subst_1115"), [Eregload(0);Eregload(1);Eregload(2)]);
```

```
Eresult("", [Eregload(0)]);
Ecopy(Eregload(10), Eregstore(6));
Ecopy(Earith(Eadd, Eregload(10), Eintconst32(81)), Eregstore(10));
Ereloadretaddr(24, Eregstore(14));
Ecopy(Eshiftconst32(false, 1l, 2), Eregstore(7));
Ecopy(Earith(Eadd, Eregload(7), Eshiftconst32(false, 1l, 10)), Eregstore(7));
Ecopy(Eregload(7), Ewriteaddrw(Eoffset(Eregload(6), -4)));
Ecopy(Eregload(0), Ewriteaddrw(Eoffset(Eregload(6), 0)));
Ecopy(Eregload(6), Eregstore(0));
Ereturn(24);
Elabdef(Elabel("_camlShort__subst_1115", 105));
Ecopy(Ereadaddrw(Eoffset(Eregload(2), 4)), Eregstore(2));
Ecall(Econstsym(true, "_camlShort__subst_1115"), [Eregload(0);Eregload(1);Eregload(2)]);
Eresult("", [Eregload(0)]);
Ecopy(Estackloadref(12), Eregstore(2));
Ecopy(Estackloadref(16), Eregstore(1));
Ecopy(Ereadaddrw(Eoffset(Eregload(2), 0)), Eregstore(2));
Ecopy(Eregload(0), Estackstoreref(0));
Ecopy(Estackloadref(8), Eregstore(0));
Ecall(Econstsym(true, "_camlShort__subst_1115"), [Eregload(0);Eregload(1);Eregload(2)]);
Eresult("", [Eregload(0)]);
Ecopy(Eregload(10), Eregstore(3));
Ecopy(Earith(Eadd, Eregload(10), Eintconst32(12l)), Eregstore(10));
Ereloadretaddr(24, Eregstore(14));
Ecopy(Estackloadref(0), Eregstore(5));
Ecopy(Eshiftconst32(false, 5l, 0), Eregstore(4));
Ecopy(Earith(Eadd, Eregload(4), Eshiftconst32(false, 1l, 11)), Eregstore(4));
Ecopy(Eregload(4), Ewriteaddrw(Eoffset(Eregload(3), -4)));
Ecopy(Eregload(0), Ewriteaddrw(Eoffset(Eregload(3), 0)));
Ecopy(Eregload(5), Ewriteaddrw(Eoffset(Eregload(3), 4)));
Ecopy(Eregload(3), Eregstore(0));
Ereturn(24);
Elabdef(Elabel("_camlShort__subst_1115", 104));
Ecopy(Ereadaddrw(Eoffset(Eregload(2), 8)), Eregstore(2));
Ecall(Econstsym(true, "_camlShort__subst_1115"), [Eregload(0);Eregload(1);Eregload(2)]);
Eresult("", [Eregload(0)]);
Ecopy(Estackloadref(12), Eregstore(6));
Ecopy(Estackloadref(16), Eregstore(1));
Ecopy(Ereadaddrw(Eoffset(Eregload(6), 4)), Eregstore(2));
Ecopy(Eregload(0), Estackstoreref(4));
Ecopy(Estackloadref(8), Eregstore(0));
Ecall(Econstsym(true, "_camlShort__subst_1115"), [Eregload(0);Eregload(1);Eregload(2)]);
Eresult("", [Eregload(0)]);
Ecopy(Eregload(0), Estackstoreref(0));
Ecopy(Estackloadref(12), Eregstore(0));
Ecopy(Estackloadref(16), Eregstore(1));
Ecopy(Ereadaddrw(Eoffset(Eregload(0), 0)), Eregstore(2));
Ecopy(Estackloadref(8), Eregstore(0));
Ecall(Econstsym(true, "_camlShort__subst_1115"), [Eregload(0);Eregload(1);Eregload(2)]);
Eresult("", [Eregload(0)]);
Ecopy(Eregload(10), Eregstore(2));
Ecopy(Earith(Eadd, Eregload(10), Eintconst32(16l)), Eregstore(10));
Ecopy(Eshiftconst32(false, 3l, 1), Eregstore(3));
Ecopy(Earith(Eadd, Eregload(3), Eshiftconst32(false, 31, 10)), Eregstore(3));
Ecopy(Eregload(3), Ewriteaddrw(Eoffset(Eregload(2), -4)));
Ecopy(Estackloadref(0), Eregstore(3));
Ereloadretaddr(24, Eregstore(14));
Ecopy(Estackloadref(4), Eregstore(4));
Ecopy(Eregload(0), Ewriteaddrw(Eoffset(Eregload(2), 0)));
Ecopy(Eregload(3), Ewriteaddrw(Eoffset(Eregload(2), 4)));
Ecopy(Eregload(4), Ewriteaddrw(Eoffset(Eregload(2), 8)));
Ecopy(Eregload(2), Eregstore(0));
Ereturn(24);
Elabdef(Elabel("_camlShort__subst_1115", 103));
Ecopy(Ereadaddrw(Eoffset(Eregload(2), 0)), Eregstore(2));
Ecall(Econstsym(true, "_camlShort__subst_1115"), [Eregload(0);Eregload(1);Eregload(2)]);
Eresult("", [Eregload(0)]);
Ecopy(Eregload(10), Eregstore(6));
Ecopy(Earith(Eadd, Eregload(10), Eintconst32(81)), Eregstore(10));
Ereloadretaddr(24, Eregstore(14));
Ecopy(Eshiftconst32(false, 7l, 0), Eregstore(7));
Ecopy(Earith(Eadd, Eregload(7), Eshiftconst32(false, 1l, 10)), Eregstore(7));
Ecopy(Eregload(7), Ewriteaddrw(Eoffset(Eregload(6), -4)));
Ecopy(Eregload(0), Ewriteaddrw(Eoffset(Eregload(6), 0)));
Ecopy(Eregload(6), Eregstore(0));
Ereturn(24);
Eadj(0);
Efundecl(0, _camlShort__redvalue_1133, false);
Elabdef(Elabel("_camlShort__redvalue_1133", 2000001));
Ecopy(Ereadaddrb(Eoffset(Eregload(0), -4)), Eregstore(1));
Ecopy(Earith(Esub, Eregload(1), Eintconst32(21)), Eregstore(2));
Econd(Ecompare (Eunsigned (Ele), Eregload(2), Eintconst32(11)), Egoto(Elabel("_camlShort__redvalue_1133", 114)));
Ecopy(Eshiftconst32(false, 1l, 0), Eregstore(0));
Ereturn(0);
Elabdef(Elabel("_camlShort__redvalue_1133", 114));
Ecopy(Eshiftconst32(false, 3l, 0), Eregstore(0));
Ereturn(0);
Eadj(0);
Efundecl(16, _camlShort__reduce_1138, true);
Elabdef(Elabel("_camlShort__reduce_1138", 2000001));
Ecopy(Ereadaddrb(Eoffset(Eregload(0), -4)), Eregstore(3));
Ecopy(Eregload(0), Estackstoreref(0));
Econd(Ecompare (Esigned (Eeq), Eregload(3), Eintconst32(0l)), Egoto(Elabel("_camlShort__reduce_1138", 129)));
Econd(Ecompare (Esigned (Eeq), Eregload(3), Eintconst32(1l)), Egoto(Elabel("_camlShort__reduce_1138", 128)));
Econd(Ecompare (Esigned (Eeq), Eregload(3), Eintconst32(2l)), Egoto(Elabel("_camlShort__reduce_1138", 125)));
Econd(Ecompare (Esigned (Eeq), Eregload(3), Eintconst32(3l)), Egoto(Elabel("_camlShort__reduce_1138", 129)));
Econd(Ecompare (Esigned (Eeq), Eregload(3), Eintconst32(4l)), Egoto(Elabel("_camlShort__reduce_1138", 124)));
Econd(Ecompare (Esigned (Eeq), Eregload(3), Eintconst32(5l)), Egoto(Elabel("_camlShort__reduce_1138", 122)));
Econd(Ecompare (Esigned (Eeq), Eregload(3), Eintconst32(6l)), Egoto(Elabel("_camlShort__reduce_1138", 119)));
Econd(Ecompare (Esigned (Eeq), Eregload(3), Eintconst32(7l)), Egoto(Elabel("_camlShort__reduce_1138", 116)));
Elabdef(Elabel("_camlShort__reduce_1138", 129));
Ereloadretaddr(16, Eregstore(14));
Ereturn(16);
Elabdef(Elabel("_camlShort__reduce_1138", 128));
Ecopy(Ereadaddrw(Eoffset(Eregload(0), 0)), Eregstore(5));
Ecopy(Ereadaddrw(Eoffset(Eregload(0), 4)), Eregstore(4));
Ecopy(Ereadaddrb(Eoffset(Eregload(5), -4)), Eregstore(6));
Ecopy(Eregload(4), Estackstoreref(0));
Ecopy(Eregload(5), Estackstoreref(4));
```

```
Econd(Ecompare (Esigned (Eeq), Eregload(6), Eintconst32(21)), Egoto(Elabel("_camlShort__reduce_1138", 127)));
Ecopy(Eregload(4), Eregstore(0));
Ecall(Econstsym(true, "_camlShort__reduce_1138"), [Eregload(0)]);
Eresult("", [Eregload(0)]);
Ecopy(Eregload(0), Estackstoreref(0));
Ecopy(Estackloadref(4), Eregstore(0));
Ecall(Econstsym(true, "_camlShort__reduce_1138"), [Eregload(0)]);
Eresult("", [Eregload(0)]);
Ecopy(Eregload(10), Eregstore(7));
Ecopy(Earith(Eadd, Eregload(10), Eintconst32(121)), Eregstore(10));
Ecopy(Eshiftconst32(false, 1l, 0), Eregstore(8));
Ecopy(Earith(Eadd, Eregload(8), Eshiftconst32(false, 1l, 11)), Eregstore(8));
Ecopy(Eregload(8), Ewriteaddrw(Eoffset(Eregload(7), -4)));
Ecopy(Estackloadref(0), Eregstore(8));
Ereloadretaddr(16, Eregstore(14));
Ecopy(Eregload(0), Ewriteaddrw(Eoffset(Eregload(7), 0)));
Ecopy(Eregload(8), Ewriteaddrw(Eoffset(Eregload(7), 4)));
Ecopy(Eregload(7), Eregstore(0));
Ereturn(16);
Elabdef(Elabel("_camlShort__reduce_1138", 127));
Ecopy(Eregload(4), Eregstore(0));
Ecall(Econstsym(true, "_camlShort__redvalue_1133"), [Eregload(0)]);
Eresult("", [Eregload(0)]);
Econd(Ecompare (Esigned (Eeq), Eregload(0), Eintconst32(11)), Egoto(Elabel("_camlShort__reduce_1138", 126)));
Ecopy(Estackloadref(4), Eregstore(8));
Ecopy(Estackloadref(0), Eregstore(1));
Ecopy(Ereadaddrw(Eoffset(Eregload(8), 8)), Eregstore(2));
Ecopy(Ereadaddrw(Eoffset(Eregload(8), 0)), Eregstore(0));
Eadj(16);
Ecopy(Estackloadref(-4), Eregstore(14));
Econd(Enop, Egoto(Econstsym(false, "_camlShort__subst_1115")));
Elabdef(Elabel("_camlShort__reduce_1138", 126));
Ecopy(Estackloadref(0), Eregstore(0));
Ecall(Econstsym(true, "_camlShort__reduce_1138"), [Eregload(0)]);
Eresult("", [Eregload(0)]);
Ecopy(Eregload(0), Estackstoreref(0));
Ecopy(Estackloadref(4), Eregstore(0));
Ecall(Econstsym(true, "_camlShort__reduce_1138"), [Eregload(0)]);
Eresult("", [Eregload(0)]);
Ecopy(Eregload(10), Eregstore(1));
Ecopy(Earith(Eadd, Eregload(10), Eintconst32(121)), Eregstore(10));
Ereloadretaddr(16, Eregstore(14));
Ecopy(Estackloadref(0), Eregstore(3));
Ecopy(Eshiftconst32(false, 1l, 0), Eregstore(2));
Ecopy(Earith(Eadd, Eregload(2), Eshiftconst32(false, 1l, 11)), Eregstore(2));
Ecopy(Eregload(2), Ewriteaddrw(Eoffset(Eregload(1), -4)));
Ecopy(Eregload(0), Ewriteaddrw(Eoffset(Eregload(1), 0)));
Ecopy(Eregload(3), Ewriteaddrw(Eoffset(Eregload(1), 4)));
Ecopy(Eregload(1), Eregstore(0));
Ereturn(16);
Elabdef(Elabel("_camlShort__reduce_1138", 125));
Ecopy(Ereadaddrw(Eoffset(Eregload(0), 8)), Eregstore(0));
Ecall(Econstsym(true, "_camlShort__reduce_1138"), [Eregload(0)]);
Eresult("", [Eregload(0)]);
Ecopy(Eregload(10), Eregstore(2));
Ecopy(Earith(Eadd, Eregload(10), Eintconst32(161)), Eregstore(10));
Ecopy(Estackloadref(0), Eregstore(5));
Ecopy(Eshiftconst32(false, 1l, 1), Eregstore(3));
Ecopy(Earith(Eadd, Eregload(3), Eshiftconst32(false, 31, 10)), Eregstore(3));
Ecopy(Eregload(3), Ewriteaddrw(Eoffset(Eregload(2), -4)));
Ecopy(Ereadaddrw(Eoffset(Eregload(5), 0)), Eregstore(4));
Ereloadretaddr(16, Eregstore(14));
Ecopy(Eregload(4), Ewriteaddrw(Eoffset(Eregload(2), 0)));
Ecopy(Ereadaddrw(Eoffset(Eregload(5), 4)), Eregstore(5));
Ecopy(Eregload(5), Ewriteaddrw(Eoffset(Eregload(2), 4)));
Ecopy(Eregload(0), Ewriteaddrw(Eoffset(Eregload(2), 8)));
Ecopy(Eregload(2), Eregstore(0));
Ereturn(16);
Elabdef(Elabel("_camlShort__reduce_1138", 124));
Ecopy(Ereadaddrw(Eoffset(Eregload(0), 0)), Eregstore(0));
Ecopy(Ereadaddrb(Eoffset(Eregload(0), -4)), Eregstore(7));
Econd(Ecompare (Esigned (Eeq), Eregload(7), Eintconst32(31)), Egoto(Elabel("_camlShort__reduce_1138", 123)));
Ecall(Econstsym(true, "_camlShort__reduce_1138"), [Eregload(0)]);
Eresult("", [Eregload(0)]);
Ecopy(Eregload(10), Eregstore(5));
Ecopy(Earith(Eadd, Eregload(10), Eintconst32(81)), Eregstore(10));
Ereloadretaddr(16, Eregstore(14));
Ecopy(Eshiftconst32(false, 1l, 2), Eregstore(6));
Ecopy(Earith(Eadd, Eregload(6), Eshiftconst32(false, 1l, 10)), Eregstore(6));
Ecopy(Eregload(6), Ewriteaddrw(Eoffset(Eregload(5), -4)));
Ecopy(Eregload(0), Ewriteaddrw(Eoffset(Eregload(5), 0)));
Ecopy(Eregload(5), Eregstore(0));
Ereturn(16);
Elabdef(Elabel("_camlShort__reduce_1138", 123));
Ecopy(Econstsym(false, "_camlBinInt__6"), Eregstore(1));
Ecopy(Ereadaddrw(Eoffset(Eregload(0), 0)), Eregstore(0));
Ecall(Econstsym(true, "_camlBinInt__add_1199"), [Eregload(0);Eregload(1)]);
Eresult("", [Eregload(0)]);
Ecopy(Eregload(10), Eregstore(2));
Ecopy(Earith(Eadd, Eregload(10), Eintconst32(81)), Eregstore(10));
Ereloadretaddr(16, Eregstore(14));
Ecopy(Eshiftconst32(false, 31, 0), Eregstore(3));
Ecopy(Earith(Eadd, Eregload(3), Eshiftconst32(false, 1l, 10)), Eregstore(3));
Ecopy(Eregload(3), Ewriteaddrw(Eoffset(Eregload(2), -4)));
Ecopy(Eregload(0), Ewriteaddrw(Eoffset(Eregload(2), 0)));
Ecopy(Eregload(2), Eregstore(0));
Ereturn(16);
Elabdef(Elabel("_camlShort__reduce_1138", 122));
Ecopy(Ereadaddrw(Eoffset(Eregload(0), 0)), Eregstore(8));
Ecopy(Ereadaddrw(Eoffset(Eregload(0), 4)), Eregstore(7));
Ecopy(Ereadaddrb(Eoffset(Eregload(8), -4)), Eregstore(0));
Ecopy(Eregload(8), Estackstoreref(4));
Econd(Ecompare (Esigned (Eeq), Eregload(0), Eintconst32(31)), Egoto(Elabel("_camlShort__reduce_1138", 121)));
Ecopy(Eregload(7), Eregstore(0));
Ecall(Econstsym(true, "_camlShort__reduce_1138"), [Eregload(0)]);
Eresult("", [Eregload(0)]);
Ecopy(Eregload(0), Estackstoreref(0));
Ecopy(Estackloadref(4), Eregstore(0));
Ecall(Econstsym(true, "_camlShort__reduce_1138"), [Eregload(0)]);
```

```
Eresult("", [Eregload(0)]);
Ecopy(Eregload(10), Eregstore(4));
Ecopy(Earith(Eadd, Eregload(10), Eintconst32(121)), Eregstore(10));
Ereloadretaddr(16, Eregstore(14));
Ecopy(Estackloadref(0), Eregstore(6));
Ecopy(Eshiftconst32(false, 5l, 0), Eregstore(5));
Ecopy(Earith(Eadd, Eregload(5), Eshiftconst32(false, 1l, 11)), Eregstore(5));
Ecopy(Eregload(5), Ewriteaddrw(Eoffset(Eregload(4), -4)));
Ecopy(Eregload(0), Ewriteaddrw(Eoffset(Eregload(4), 0)));
Ecopy(Eregload(6), Ewriteaddrw(Eoffset(Eregload(4), 4)));
Ecopy(Eregload(4), Eregstore(0));
Ereturn(16);
Elabdef(Elabel("_camlShort__reduce_1138", 121));
Ecopy(Ereadaddrb(Eoffset(Eregload(7), -4)), Eregstore(1));
Econd(Ecompare (Esigned (Eeq), Eregload(1), Eintconst32(31)), Egoto(Elabel("_camlShort__reduce_1138", 120)));
Ecopy(Eregload(7), Eregstore(0));
Ecall(Econstsym(true, "_camlShort__reduce_1138"), [Eregload(0)]);
Eresult("", [Eregload(0)]);
Ecopy(Eregload(0), Estackstoreref(0));
Ecopy(Estackloadref(4), Eregstore(0));
Ecall(Econstsym(true, "_camlShort__reduce_1138"), [Eregload(0)]);
Eresult("", [Eregload(0)]);
Ecopy(Eregload(10), Eregstore(1));
Ecopy(Earith(Eadd, Eregload(10), Eintconst32(121)), Eregstore(10));
Ereloadretaddr(16, Eregstore(14));
Ecopy(Estackloadref(0), Eregstore(8));
Ecopy(Eshiftconst32(false, 5l, 0), Eregstore(2));
Ecopy(Earith(Eadd, Eregload(2), Eshiftconst32(false, 1l, 11)), Eregstore(2));
Ecopy(Eregload(2), Ewriteaddrw(Eoffset(Eregload(1), -4)));
Ecopy(Eregload(0), Ewriteaddrw(Eoffset(Eregload(1), 0)));
Ecopy(Eregload(8), Ewriteaddrw(Eoffset(Eregload(1), 4)));
Ecopy(Eregload(1), Eregstore(0));
Ereturn(16);
Elabdef(Elabel("_camlShort__reduce_1138", 120));
Ecopy(Ereadaddrw(Eoffset(Eregload(8), 0)), Eregstore(0));
Ecopy(Ereadaddrw(Eoffset(Eregload(7), 0)), Eregstore(1));
Ecall(Econstsym(true, "_camlBinInt__mul_1218"), [Eregload(0);Eregload(1)]);
Eresult("", [Eregload(0)]);
Ecopy(Eregload(10), Eregstore(5));
Ecopy(Earith(Eadd, Eregload(10), Eintconst32(81)), Eregstore(10));
Ereloadretaddr(16, Eregstore(14));
Ecopy(Eshiftconst32(false, 3l, 0), Eregstore(6));
Ecopy(Earith(Eadd, Eregload(6), Eshiftconst32(false, 1l, 10)), Eregstore(6));
Ecopy(Eregload(6), Ewriteaddrw(Eoffset(Eregload(5), -4)));
Ecopy(Eregload(0), Ewriteaddrw(Eoffset(Eregload(5), 0)));
Ecopy(Eregload(5), Eregstore(0));
Ereturn(16);
Elabdef(Elabel("_camlShort__reduce_1138", 119));
Ecopy(Ereadaddrw(Eoffset(Eregload(0), 4)), Eregstore(7));
Ecopy(Ereadaddrw(Eoffset(Eregload(0), 8)), Eregstore(6));
Ecopy(Ereadaddrw(Eoffset(Eregload(0), 0)), Eregstore(0));
Ecopy(Eregload(6), Estackstoreref(4));
Ecopy(Ereadaddrb(Eoffset(Eregload(0), -4)), Eregstore(1));
Ecopy(Eregload(7), Estackstoreref(0));
Econd(Ecompare (Esigned (Eeq), Eregload(1), Eintconst32(31)), Egoto(Elabel("_camlShort__reduce_1138", 118)));
Ecall(Econstsym(true, "_camlShort__reduce_1138"), [Eregload(0)]);
Eresult("", [Eregload(0)]);
Ecopy(Eregload(10), Eregstore(3));
Ecopy(Earith(Eadd, Eregload(10), Eintconst32(161)), Eregstore(10));
Ecopy(Eshiftconst32(false, 3l, 1), Eregstore(4));
Ecopy(Earith(Eadd, Eregload(4), Eshiftconst32(false, 3l, 10)), Eregstore(4));
Ecopy(Eregload(4), Ewriteaddrw(Eoffset(Eregload(3), -4)));
Ecopy(Eregload(0), Ewriteaddrw(Eoffset(Eregload(3), 0)));
Ecopy(Estackloadref(0), Eregstore(0));
Ereloadretaddr(16, Eregstore(14));
Ecopy(Estackloadref(4), Eregstore(1));
Ecopy(Eregload(0), Ewriteaddrw(Eoffset(Eregload(3), 4)));
Ecopy(Eregload(1), Ewriteaddrw(Eoffset(Eregload(3), 8)));
Ecopy(Eregload(3), Eregstore(0));
Ereturn(16);
Elabdef(Elabel("_camlShort__reduce_1138", 118));
Ecopy(Ereadaddrw(Eoffset(Eregload(0), 0)), Eregstore(1));
Econd(Ecompare (Esigned (Eeq), Earith(Eand, Eregload(1), Eintconst32(1l)), Eintconst32(0l)), Egoto(Elabel("_camlShort__reduce_1138", 117)));
Ereloadretaddr(16, Eregstore(14));
Ecopy(Eregload(7), Eregstore(0));
Ereturn(16);
Elabdef(Elabel("_camlShort__reduce_1138", 117));
Ereloadretaddr(16, Eregstore(14));
Ecopy(Eregload(6), Eregstore(0));
Ereturn(16);
Elabdef(Elabel("_camlShort__reduce_1138", 116));
Ecopy(Ereadaddrw(Eoffset(Eregload(0), 0)), Eregstore(5));
Ecopy(Ereadaddrb(Eoffset(Eregload(5), -4)), Eregstore(6));
Econd(Ecompare (Esigned (Eeq), Eregload(6), Eintconst32(2l)), Egoto(Elabel("_camlShort__reduce_1138", 115)));
Ecopy(Eregload(5), Eregstore(0));
Ecall(Econstsym(true, "_camlShort__reduce_1138"), [Eregload(0)]);
Eresult("", [Eregload(0)]);
Ecopy(Eregload(10), Eregstore(4));
Ecopy(Earith(Eadd, Eregload(10), Eintconst32(81)), Eregstore(10));
Ereloadretaddr(16, Eregstore(14));
Ecopy(Eshiftconst32(false, 7l, 0), Eregstore(5));
Ecopy(Earith(Eadd, Eregload(5), Eshiftconst32(false, 1l, 10)), Eregstore(5));
Ecopy(Eregload(5), Ewriteaddrw(Eoffset(Eregload(4), -4)));
Ecopy(Eregload(0), Ewriteaddrw(Eoffset(Eregload(4), 0)));
Ecopy(Eregload(4), Eregstore(0));
Ereturn(16);
Elabdef(Elabel("_camlShort__reduce_1138", 115));
Ecopy(Ereadaddrw(Eoffset(Eregload(5), 0)), Eregstore(0));
Ecopy(Ereadaddrw(Eoffset(Eregload(5), 8)), Eregstore(2));
Ecopy(Eregload(10), Eregstore(7));
Ecopy(Earith(Eadd, Eregload(10), Eintconst32(241)), Eregstore(10));
Ecopy(Eshiftconst32(false, 1l, 1), Eregstore(8));
Ecopy(Earith(Eadd, Eregload(8), Eshiftconst32(false, 3l, 10)), Eregstore(8));
Ecopy(Eregload(8), Ewriteaddrw(Eoffset(Eregload(7), -4)));
Ecopy(Eregload(0), Ewriteaddrw(Eoffset(Eregload(7), 0)));
Ecopy(Ereadaddrw(Eoffset(Eregload(5), 4)), Eregstore(1));
Ecopy(Eshiftconst32(false, 7l, 0), Eregstore(3));
Ecopy(Earith(Eadd, Eregload(3), Eshiftconst32(false, 1l, 10)), Eregstore(3));
Ecopy(Eregload(1), Ewriteaddrw(Eoffset(Eregload(7), 4)));
```

```
Ecopy(Earith(Eadd, Eregload(7), Eintconst32(16l)), Eregstore(1));
Ecopy(Eregload(2), Ewriteaddrw(Eoffset(Eregload(7), 8)));
Ecopy(Eregload(3), Ewriteaddrw(Eoffset(Eregload(1), -4)));
Ecopy(Eregload(7), Ewriteaddrw(Eoffset(Eregload(1), 0)));
Eadj(16);
Ecopy(Estackloadref(-4), Eregstore(14));
Econd(Enop, Egoto(Econstsym(false, "_camlShort__subst_1115")));
Eadj(0);
Efundecl(0, _camlShort__fact_calc_1163, false);
Elabdef(Elabel("_camlShort__fact_calc_1163", 2000001));
Ecopy(Eregload(10), Eregstore(1));
Ecopy(Earith(Eadd, Eregload(10), Eintconst32(201)), Eregstore(10));
Ecopy(Econstsym(false, "_camlShort"), Eregstore(5));
Ecopy(Eshiftconst32(false, 3l, 0), Eregstore(2));
Ecopy(Earith(Eadd, Eregload(2), Eshiftconst32(false, 1l, 10)), Eregstore(2));
Ecopy(Eregload(2), Ewriteaddrw(Eoffset(Eregload(1), -4)));
Ecopy(Eregload(0), Ewriteaddrw(Eoffset(Eregload(1), 0)));
Ecopy(Earith(Eadd, Eregload(1), Eintconst32(81)), Eregstore(0));
Ecopy(Eshiftconst32(false, 1l, 0), Eregstore(4));
Ecopy(Earith(Eadd, Eregload(4), Eshiftconst32(false, 1l, 11)), Eregstore(4));
Ecopy(Eregload(4), Ewriteaddrw(Eoffset(Eregload(0), -4)));
Ecopy(Ereadaddrw(Eoffset(Eregload(5), 16)), Eregstore(6));
Ecopy(Eregload(6), Ewriteaddrw(Eoffset(Eregload(0), 0)));
Ecopy(Eregload(1), Ewriteaddrw(Eoffset(Eregload(0), 4)));
Ereturn(0);
Eadj(0);
Efundecl(8, _camlShort__reduce_n_1165, true);
Elabdef(Elabel("_camlShort__reduce_n_1165", 2000001));
Econd(Ecompare (Esigned (Eeq), Eregload(0), Eintconst32(1l)), Egoto(Elabel("_camlShort__reduce_n_1165", 130)));
Ecopy(Eregload(0), Estackstoreref(0));
Ecopy(Eregload(1), Eregstore(0));
Ecall(Econstsym(true, "_camlShort__reduce_1138"), [Eregload(0)]);
Eresult("", [Eregload(0)]);
Ecopy(Estackloadref(0), Eregstore(4));
Ecopy(Eregload(0), Eregstore(1));
Ecopy(Ereadaddrw(Eoffset(Eregload(4), 0)), Eregstore(0));
Econd(Enop, Egoto(Elabel("_camlShort__reduce_n_1165", 2000001)));
Elabdef(Elabel("_camlShort__reduce_n_1165", 130));
Ereloadretaddr(8, Eregstore(14));
Ecopy(Eregload(1), Eregstore(0));
Ereturn(8);
Eadj(0);
Efundecl(0, _camlShort__entry, false);
Einit( { Econstsym(false, "_camlShort__7");
        Econstsym(false, "_camlShort__6");
        Econstsym(false, "_camlShort__5");
        Econstsym(false, "_camlShort__4");
        Econstsym(false, "_camlShort__1");
        Econstsym(false, "_camlShort__3");
        Econstsym(false, "_camlShort__2") } , "_camlShort");
Ecopy(Eshiftconst32(false, 1l, 0), Eregstore(0));
Ereturn(0);
Eadj(0);
Elabdef(Elab("_camlShort__code_end"));
Eabort("camlShort__code_end");
Elabdef(Elab("_camlShort__data_begin"));
Elabdef(Elab("_camlShort__data_end"));
Elabdef(Elab("_camlShort__frametable"));
]));;
```

Appendix-H

This appendix contains the compiled output for the factorial example presented in section 6.5 (Simulation Example). The language is in the form of a behavioural Verilog case statement, corresponding to a state machine. This language may be simulated in parallel and is equivalent to the processor example of Appendix-C, supplemented by the logic of Figure 9.3.

```
subst_1115:  begin  write4(r[14], r[13]-4); r[13] = (r[13] - 32'h00000018);  end
subst_1115L10008209:  begin   end
subst_1115L2000001:  begin  read1((r[2] - 32'h00000004));  end
subst_1115L10008210:  begin  delayslot(3, 1'b1);  end
subst_1115L10008211:  begin  write4(r[2], (r[13] + 32'h0000000c));  end
subst_1115L10008212:  begin  write4(r[1], (r[13] + 32'h00000010));  end
subst_1115L10008213:  begin  write4(r[0], (r[13] + 32'h00000008));  end
subst_1115L10008214:  begin  if (r[3] == 32'h00000000)  begin  nxtpc = (subst_1115L113);  end  else  begin  end   end
subst_1115L10008215:  begin  if (r[3] == 32'h00000001)  begin  nxtpc = (subst_1115L111);  end  else  begin  end   end
subst_1115L10008216:  begin  if (r[3] == 32'h00000002)  begin  nxtpc = (subst_1115L110);  end  else  begin  end   end
subst_1115L10008217:  begin  if (r[3] == 32'h00000003)  begin  nxtpc = (subst_1115L107);  end  else  begin  end   end
subst_1115L10008218:  begin  if (r[3] == 32'h00000004)  begin  nxtpc = (subst_1115L106);  end  else  begin  end   end
subst_1115L10008219:  begin  if (r[3] == 32'h00000005)  begin  nxtpc = (subst_1115L105);  end  else  begin  end   end
subst_1115L10008220:  begin  if (r[3] == 32'h00000006)  begin  nxtpc = (subst_1115L104);  end  else  begin  end   end
subst_1115L10008221:  begin  if (r[3] == 32'h00000007)  begin  nxtpc = (subst_1115L103);  end  else  begin  end   end
subst_1115L10008222:  begin   end
subst_1115L113:  begin  read4((r[2] + 32'h00000000));  end
subst_1115L10008223:  begin  delayslot(1, 1'b0);  end
subst_1115L10008224:  begin  r[14] = (r[15] + 32'h00000004); if (1'b1)  begin  nxtpc = ((beq_id_1097));  end  else  begin  end   end
subst_1115L10008225:  begin   end
subst_1115L10008226:  begin  if (r[0] == 32'h00000001)  begin  nxtpc = (subst_1115L112);  end  else  begin  end   end
subst_1115L10008227:  begin  read4((r[13] + 32'h00000014)); end
subst_1115L10008228:  begin  delayslot(14, 1'b0);  end
subst_1115L10008229:  begin  read4((r[13] + 32'h00000010)); end
subst_1115L10008230:  begin  delayslot(0, 1'b0);  end
subst_1115L10008231:  begin  r[13] = (r[13] + 32'h00000018); if (1'b1)  begin  nxtpc = (r[14]);  end  else  begin  end   end
subst_1115L10008232:  begin   end
subst_1115L112:  begin  read4((r[13] + 32'h00000014)); end
subst_1115L10008233:  begin  delayslot(14, 1'b0);  end
subst_1115L10008234:  begin  read4((r[13] + 32'h0000000c)); end
subst_1115L10008235:  begin  delayslot(0, 1'b0);  end
subst_1115L10008236:  begin  r[13] = (r[13] + 32'h00000018); if (1'b1)  begin  nxtpc = (r[14]);  end  else  begin  end   end
subst_1115L10008237:  begin   end
subst_1115L111:  begin  read4((r[2] + 32'h00000004));  end
subst_1115L10008238:  begin  delayslot(2, 1'b0);  end
subst_1115L10008239:  begin  r[14] = (r[15] + 32'h00000004); if (1'b1)  begin  nxtpc = ((subst_1115));  end  else  begin  end   end
subst_1115L10008240:  begin   end
subst_1115L10008241:  begin  read4((r[13] + 32'h0000000c)); end
subst_1115L10008242:  begin  delayslot(1, 1'b0);  end
subst_1115L10008243:  begin  write4(r[0], r[13]);  end
subst_1115L10008244:  begin  read4((r[13] + 32'h00000008)); end
subst_1115L10008245:  begin  delayslot(0, 1'b0);  end
subst_1115L10008246:  begin  read4((r[1] + 32'h00000000));  end
subst_1115L10008247:  begin  delayslot(2, 1'b0);  end
subst_1115L10008248:  begin  read4((r[13] + 32'h00000010)); end
subst_1115L10008249:  begin  delayslot(1, 1'b0);  end
subst_1115L10008250:  begin  r[14] = (r[15] + 32'h00000004); if (1'b1)  begin  nxtpc = ((subst_1115));  end  else  begin  end   end
subst_1115L10008251:  begin   end
subst_1115L10008252:  begin  r[1] = r[10];  end
subst_1115L10008253:  begin  r[10] = (r[10] + 32'h0000000c);  end
subst_1115L10008254:  begin  read4((r[13] + 32'h00000014)); end
subst_1115L10008255:  begin  delayslot(14, 1'b0);  end
subst_1115L10008256:  begin  read4(r[13]); end
subst_1115L10008257:  begin  delayslot(4, 1'b0);  end
subst_1115L10008258:  begin  r[2] = 32'h00000001;  end
subst_1115L10008259:  begin  r[2] = (r[2] + 32'h00000800);  end
subst_1115L10008260:  begin  write4(r[2], (r[1] - 32'h00000004));  end
subst_1115L10008261:  begin  write4(r[0], (r[1] + 32'h00000000));  end
subst_1115L10008262:  begin  write4(r[4], (r[1] + 32'h00000004));  end
subst_1115L10008263:  begin  r[0] = r[1];  end
subst_1115L10008264:  begin  r[13] = (r[13] + 32'h00000018); if (1'b1)  begin  nxtpc = (r[14]);  end  else  begin  end   end
subst_1115L10008265:  begin   end
subst_1115L110:  begin  read4((r[2] + 32'h00000000));  end
subst_1115L10008266:  begin  delayslot(1, 1'b0);  end
subst_1115L10008267:  begin  read4((r[2] + 32'h00000008));  end
subst_1115L10008268:  begin  delayslot(3, 1'b0);  end
subst_1115L10008269:  begin  write4(r[1], r[13]);  end
subst_1115L10008270:  begin  write4(r[3], r[13]+4);  end
subst_1115L10008271:  begin  r[14] = (r[15] + 32'h00000004); if (1'b1)  begin  nxtpc = ((beq_id_1097));  end  else  begin  end   end
subst_1115L10008272:  begin   end
subst_1115L10008273:  begin  if (r[0] == 32'h00000001)  begin  nxtpc = (subst_1115L109);  end  else  begin  end   end
subst_1115L10008274:  begin  read4(r[13]+4); end
subst_1115L10008275:  begin  delayslot(6, 1'b0);  end
subst_1115L10008276:  begin  if (1'b1)  begin  nxtpc = (subst_1115L108);  end  else  begin  end   end
subst_1115L10008277:  begin   end
subst_1115L109:  begin  read4(r[13]+4); end
subst_1115L10008278:  begin  delayslot(2, 1'b0);  end
```

```
subst_1115L10008279:  begin  read4((r[13] + 32'h00000010)); end
subst_1115L10008280:  begin  delayslot(1, 1'b0);  end
subst_1115L10008281:  begin  read4((r[13] + 32'h00000008)); end
subst_1115L10008282:  begin  delayslot(0, 1'b0);  end
subst_1115L10008283:  begin  r[14] = (r[15] + 32'h00000004); if (1'b1)  begin  nxtpc = ((subst_1115));  end  else  begin  end   end
subst_1115L10008284:  begin   end
subst_1115L10008285:  begin  r[6] = r[0];  end
subst_1115L10008286:  begin   end
subst_1115L108:  begin  r[0] = r[10];  end
subst_1115L10008287:  begin  r[10] = (r[10] + 32'h00000010);  end
subst_1115L10008288:  begin  read4(r[13]); end
subst_1115L10008289:  begin  delayslot(1, 1'b0);  end
subst_1115L10008290:  begin  r[8] = 32'h00000002;  end
subst_1115L10008291:  begin  r[8] = (r[8] + 32'h00000c00);  end
subst_1115L10008292:  begin  write4(r[8], (r[0] - 32'h00000004));  end
subst_1115L10008293:  begin  write4(r[1], (r[0] + 32'h00000000));  end
subst_1115L10008294:  begin  read4((r[13] + 32'h0000000c)); end
subst_1115L10008295:  begin  delayslot(1, 1'b0);  end
subst_1115L10008296:  begin  read4((r[13] + 32'h00000014)); end
subst_1115L10008297:  begin  delayslot(14, 1'b0);  end
subst_1115L10008298:  begin  read4((r[1] + 32'h00000004));  end
subst_1115L10008299:  begin  delayslot(1, 1'b0);  end
subst_1115L10008300:  begin  write4(r[1], (r[0] + 32'h00000004));  end
subst_1115L10008301:  begin  write4(r[6], (r[0] + 32'h00000008));  end
subst_1115L10008302:  begin  r[13] = (r[13] + 32'h00000018); if (1'b1)  begin  nxtpc = (r[14]);  end  else  begin  end   end
subst_1115L10008303:  begin   end
subst_1115L107:  begin  r[0] = r[10];  end
subst_1115L10008304:  begin  r[10] = (r[10] + 32'h00000008);  end
subst_1115L10008305:  begin  r[3] = 32'h00000003;  end
subst_1115L10008306:  begin  r[3] = (r[3] + 32'h00000400);  end
subst_1115L10008307:  begin  write4(r[3], (r[0] - 32'h00000004));  end
subst_1115L10008308:  begin  read4((r[2] + 32'h00000000));  end
subst_1115L10008309:  begin  delayslot(3, 1'b0);  end
subst_1115L10008310:  begin  read4((r[13] + 32'h00000014)); end
subst_1115L10008311:  begin  delayslot(14, 1'b0);  end
subst_1115L10008312:  begin  write4(r[3], (r[0] + 32'h00000000));  end
subst_1115L10008313:  begin  r[13] = (r[13] + 32'h00000018); if (1'b1)  begin  nxtpc = (r[14]);  end  else  begin  end   end
subst_1115L10008314:  begin   end
subst_1115L106:  begin  read4((r[2] + 32'h00000000));  end
subst_1115L10008315:  begin  delayslot(2, 1'b0);  end
subst_1115L10008316:  begin  r[14] = (r[15] + 32'h00000004); if (1'b1)  begin  nxtpc = ((subst_1115));  end  else  begin  end   end
subst_1115L10008317:  begin   end
subst_1115L10008318:  begin  r[6] = r[10];  end
subst_1115L10008319:  begin  r[10] = (r[10] + 32'h00000008);  end
subst_1115L10008320:  begin  read4((r[13] + 32'h00000014)); end
subst_1115L10008321:  begin  delayslot(14, 1'b0);  end
subst_1115L10008322:  begin  r[7] = 32'h00000004;  end
subst_1115L10008323:  begin  r[7] = (r[7] + 32'h00000400);  end
subst_1115L10008324:  begin  write4(r[7], (r[6] - 32'h00000004));  end
subst_1115L10008325:  begin  write4(r[0], (r[6] + 32'h00000000));  end
subst_1115L10008326:  begin  r[0] = r[6];  end
subst_1115L10008327:  begin  r[13] = (r[13] + 32'h00000018); if (1'b1)  begin  nxtpc = (r[14]);  end  else  begin  end   end
subst_1115L10008328:  begin   end
subst_1115L105:  begin  read4((r[2] + 32'h00000004));  end
subst_1115L10008329:  begin  delayslot(2, 1'b0);  end
subst_1115L10008330:  begin  r[14] = (r[15] + 32'h00000004); if (1'b1)  begin  nxtpc = ((subst_1115));  end  else  begin  end   end
subst_1115L10008331:  begin   end
subst_1115L10008332:  begin  read4((r[13] + 32'h0000000c)); end
subst_1115L10008333:  begin  delayslot(2, 1'b0);  end
subst_1115L10008334:  begin  read4((r[13] + 32'h00000010)); end
subst_1115L10008335:  begin  delayslot(1, 1'b0);  end
subst_1115L10008336:  begin  read4((r[2] + 32'h00000000));  end
subst_1115L10008337:  begin  delayslot(2, 1'b0);  end
subst_1115L10008338:  begin  write4(r[0], r[13]);  end
subst_1115L10008339:  begin  read4((r[13] + 32'h00000008)); end
subst_1115L10008340:  begin  delayslot(0, 1'b0);  end
subst_1115L10008341:  begin  r[14] = (r[15] + 32'h00000004); if (1'b1)  begin  nxtpc = ((subst_1115));  end  else  begin  end   end
subst_1115L10008342:  begin   end
subst_1115L10008343:  begin  r[3] = r[10];  end
subst_1115L10008344:  begin  r[10] = (r[10] + 32'h0000000c);  end
subst_1115L10008345:  begin  read4((r[13] + 32'h00000014)); end
subst_1115L10008346:  begin  delayslot(14, 1'b0);  end
subst_1115L10008347:  begin  read4(r[13]); end
subst_1115L10008348:  begin  delayslot(5, 1'b0);  end
subst_1115L10008349:  begin  r[4] = 32'h00000005;  end
subst_1115L10008350:  begin  r[4] = (r[4] + 32'h00000800);  end
subst_1115L10008351:  begin  write4(r[4], (r[3] - 32'h00000004));  end
subst_1115L10008352:  begin  write4(r[0], (r[3] + 32'h00000000));  end
subst_1115L10008353:  begin  write4(r[5], (r[3] + 32'h00000004));  end
subst_1115L10008354:  begin  r[0] = r[3];  end
subst_1115L10008355:  begin  r[13] = (r[13] + 32'h00000018); if (1'b1)  begin  nxtpc = (r[14]);  end  else  begin  end   end
subst_1115L10008356:  begin   end
subst_1115L104:  begin  read4((r[2] + 32'h00000008));  end
subst_1115L10008357:  begin  delayslot(2, 1'b0);  end
subst_1115L10008358:  begin  r[14] = (r[15] + 32'h00000004); if (1'b1)  begin  nxtpc = ((subst_1115));  end  else  begin  end   end
subst_1115L10008359:  begin   end
subst_1115L10008360:  begin  read4((r[13] + 32'h0000000c)); end
subst_1115L10008361:  begin  delayslot(6, 1'b0);  end
subst_1115L10008362:  begin  read4((r[13] + 32'h00000010)); end
subst_1115L10008363:  begin  delayslot(1, 1'b0);  end
subst_1115L10008364:  begin  read4((r[6] + 32'h00000004));  end
subst_1115L10008365:  begin  delayslot(2, 1'b0);  end
subst_1115L10008366:  begin  write4(r[0], r[13]+4);  end
subst_1115L10008367:  begin  read4((r[13] + 32'h00000008)); end
subst_1115L10008368:  begin  delayslot(0, 1'b0);  end
subst_1115L10008369:  begin  r[14] = (r[15] + 32'h00000004); if (1'b1)  begin  nxtpc = ((subst_1115));  end  else  begin  end   end
subst_1115L10008370:  begin   end
subst_1115L10008371:  begin  write4(r[0], r[13]);  end
subst_1115L10008372:  begin  read4((r[13] + 32'h0000000c)); end
subst_1115L10008373:  begin  delayslot(0, 1'b0);  end
subst_1115L10008374:  begin  read4((r[13] + 32'h00000010)); end
subst_1115L10008375:  begin  delayslot(1, 1'b0);  end
subst_1115L10008376:  begin  read4((r[0] + 32'h00000000));  end
subst_1115L10008377:  begin  delayslot(2, 1'b0);  end
subst_1115L10008378:  begin  read4((r[13] + 32'h00000008)); end
subst_1115L10008379:  begin  delayslot(0, 1'b0);  end
subst_1115L10008380:  begin  r[14] = (r[15] + 32'h00000004); if (1'b1)  begin  nxtpc = ((subst_1115));  end  else  begin  end   end
subst_1115L10008381:  begin   end
subst_1115L10008382:  begin  r[2] = r[10];  end
```

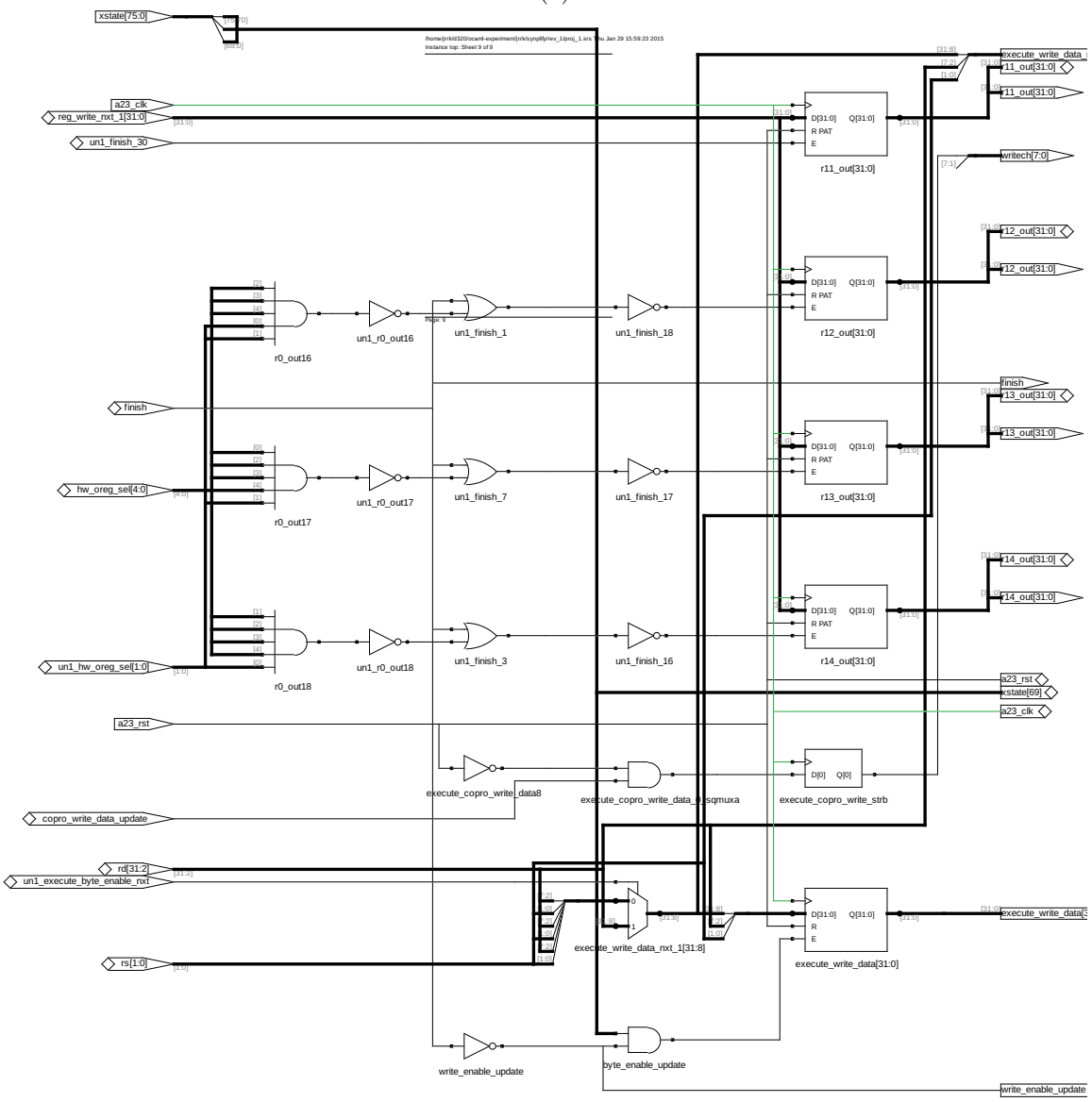
```
subst_1115L10008383:  begin  r[10] = (r[10] + 32'h00000010);  end
subst_1115L10008384:  begin  r[3] = 32'h00000006;  end
subst_1115L10008385:  begin  r[3] = (r[3] + 32'h00000c00);  end
subst_1115L10008386:  begin  write4(r[3], (r[2] - 32'h00000004));  end
subst_1115L10008387:  begin  read4(r[13]); end
subst_1115L10008388:  begin  delayslot(3, 1'b0);  end
subst_1115L10008389:  begin  read4((r[13] + 32'h00000014)); end
subst_1115L10008390:  begin  delayslot(14, 1'b0);  end
subst_1115L10008391:  begin  read4(r[13]+4); end
subst_1115L10008392:  begin  delayslot(4, 1'b0);  end
subst_1115L10008393:  begin  write4(r[0], (r[2] + 32'h00000000));  end
subst_1115L10008394:  begin  write4(r[3], (r[2] + 32'h00000004));  end
subst_1115L10008395:  begin  write4(r[4], (r[2] + 32'h00000008));  end
subst_1115L10008396:  begin  r[0] = r[2];  end
subst_1115L10008397:  begin  r[13] = (r[13] + 32'h00000018); if (1'b1)  begin  nxtpc = (r[14]);  end  else  begin  end   end
subst_1115L10008398:  begin   end
subst_1115L103:  begin  read4((r[2] + 32'h00000000));  end
subst_1115L10008399:  begin  delayslot(2, 1'b0);  end
subst_1115L10008400:  begin  r[14] = (r[15] + 32'h00000004); if (1'b1)  begin  nxtpc = ((subst_1115));  end  else  begin  end   end
subst_1115L10008401:  begin   end
subst_1115L10008402:  begin  r[6] = r[10];  end
subst_1115L10008403:  begin  r[10] = (r[10] + 32'h00000008);  end
subst_1115L10008404:  begin  read4((r[13] + 32'h00000014)); end
subst_1115L10008405:  begin  delayslot(14, 1'b0);  end
subst_1115L10008406:  begin  r[7] = 32'h00000007;  end
subst_1115L10008407:  begin  r[7] = (r[7] + 32'h00000400);  end
subst_1115L10008408:  begin  write4(r[7], (r[6] - 32'h00000004));  end
subst_1115L10008409:  begin  write4(r[0], (r[6] + 32'h00000000));  end
subst_1115L10008410:  begin  r[0] = r[6];  end
subst_1115L10008411:  begin  r[13] = (r[13] + 32'h00000018); if (1'b1)  begin  nxtpc = (r[14]);  end  else  begin  end   end
subst_1115L10008412:  begin  r[13] = (r[13] + 32'h00000000);  end
subst_1115L10008413:  begin   end
```

Appendix-I

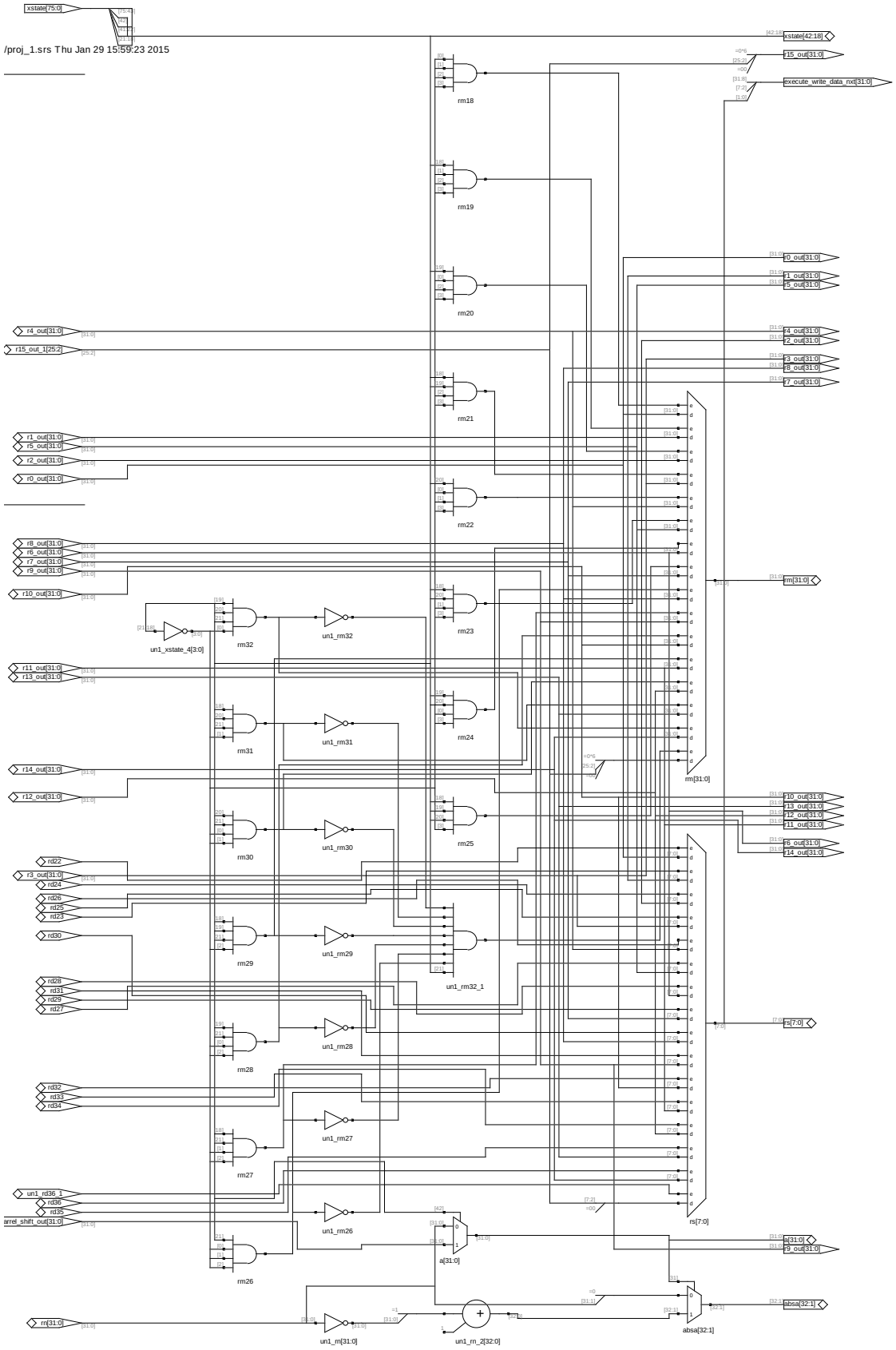
This appendix contains the schematics of the standalone processor template of Appendix-C. This is the plain version (without fault-tolerance).

Figure I.1: Internals of Amber-derived processor

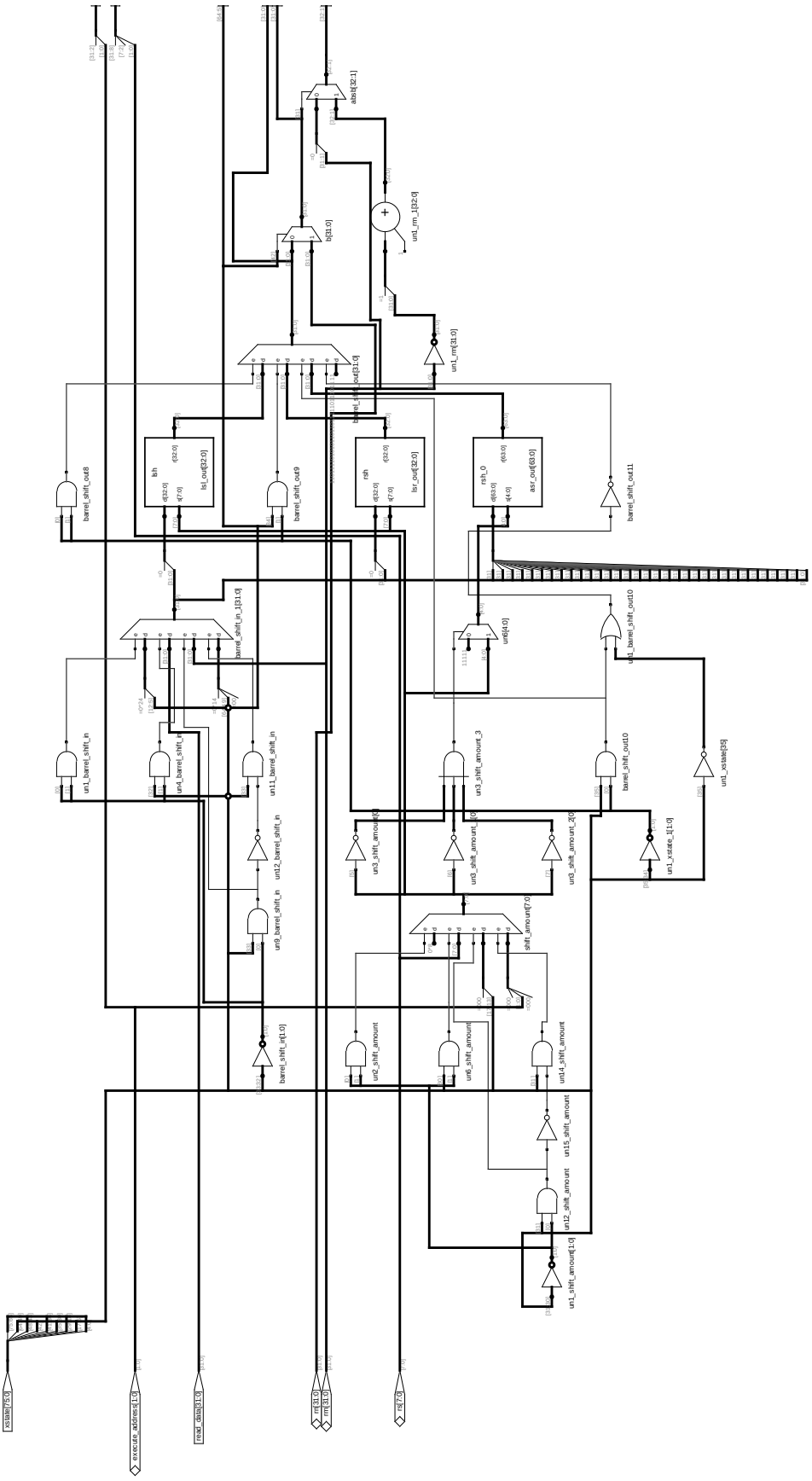
(a)



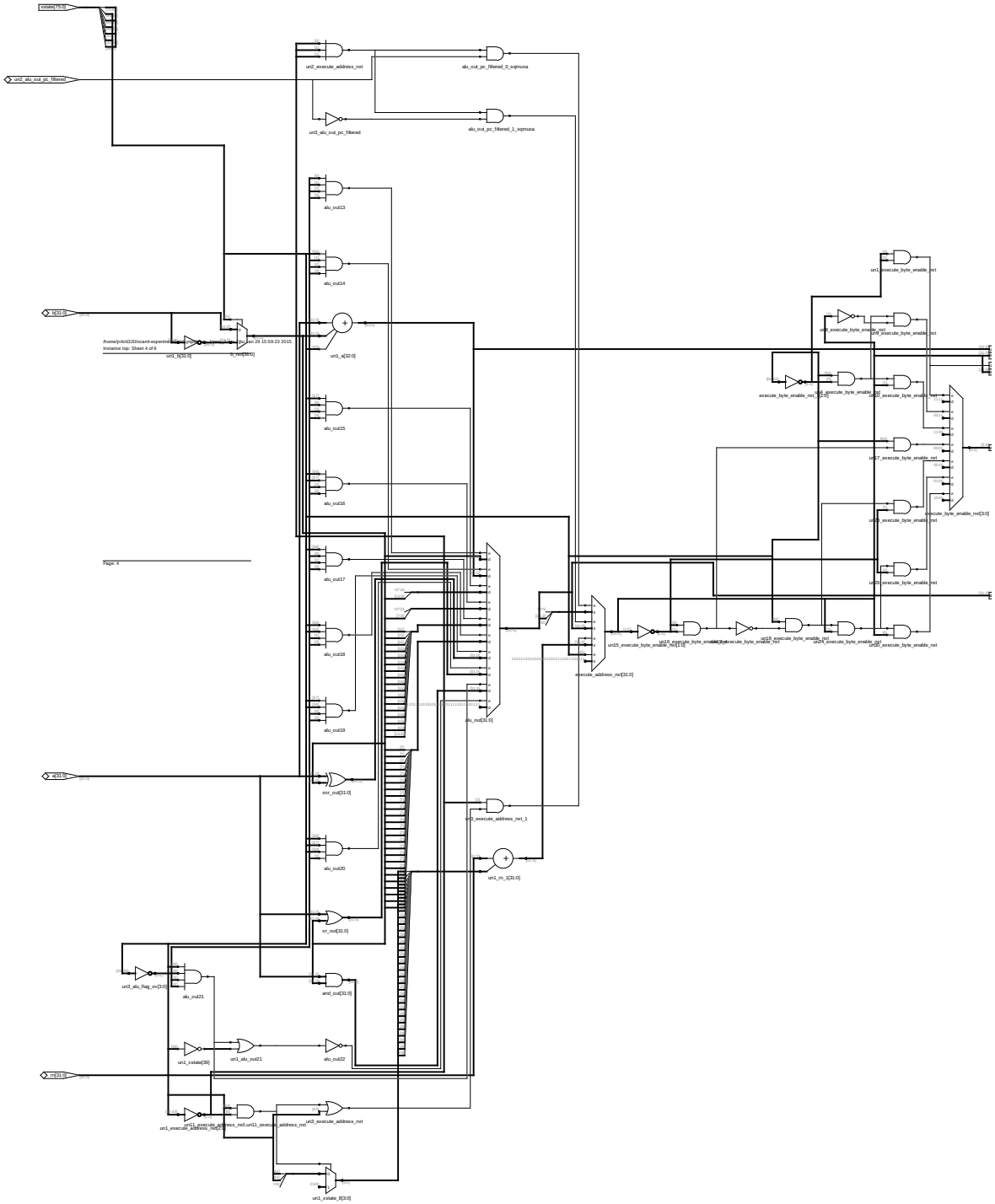
(b)


/proj_1.srs Thu Jan 29 15:59:23 2015
xstate[75:0]
xstate[42:18]
r15_out[31:0]
execute_write_data_nxt[31:0]
rm18
rm19
rm20
rm21
rm22
rm23
rm24
rm25
rm26
rm27
rm28
rm29
rm30
rm31
rm32
r0_out[31:0]
r1_out[31:0]
r5_out[31:0]
r4_out[31:0]
r15_out_1[25:2]
r2_out[31:0]
r3_out[31:0]
r8_out[31:0]
r7_out[31:0]
r6_out[31:0]
r9_out[31:0]
r10_out[31:0]
r11_out[31:0]
r13_out[31:0]
r14_out[31:0]
r12_out[31:0]
rd22
rd24
rd26
rd25
rd23
rd30
rd28
rd31
rd29
rd27
rd32
rd33
rd34
un1_rd36_1
rd36
rd35
arrel_shift_out[31:0]
rn[31:0]
un1_xstate_4[3:0]
un1_rm32
un1_rm31
un1_rm30
un1_rm29
un1_rm32_1
un1_rm28
un1_rm27
un1_rm26
un1_rm[31:0]
un1_rn_2[32:0]
a[31:0]
rm[31:0]
rs[7:0]
absa[32:1]
r0_out[31:0]
r1_out[31:0]
r5_out[31:0]
r4_out[31:0]
r2_out[31:0]
r3_out[31:0]
r8_out[31:0]
r7_out[31:0]
r10_out[31:0]
r13_out[31:0]
r12_out[31:0]
r11_out[31:0]
r6_out[31:0]
r14_out[31:0]
a[31:0]
r9_out[31:0]

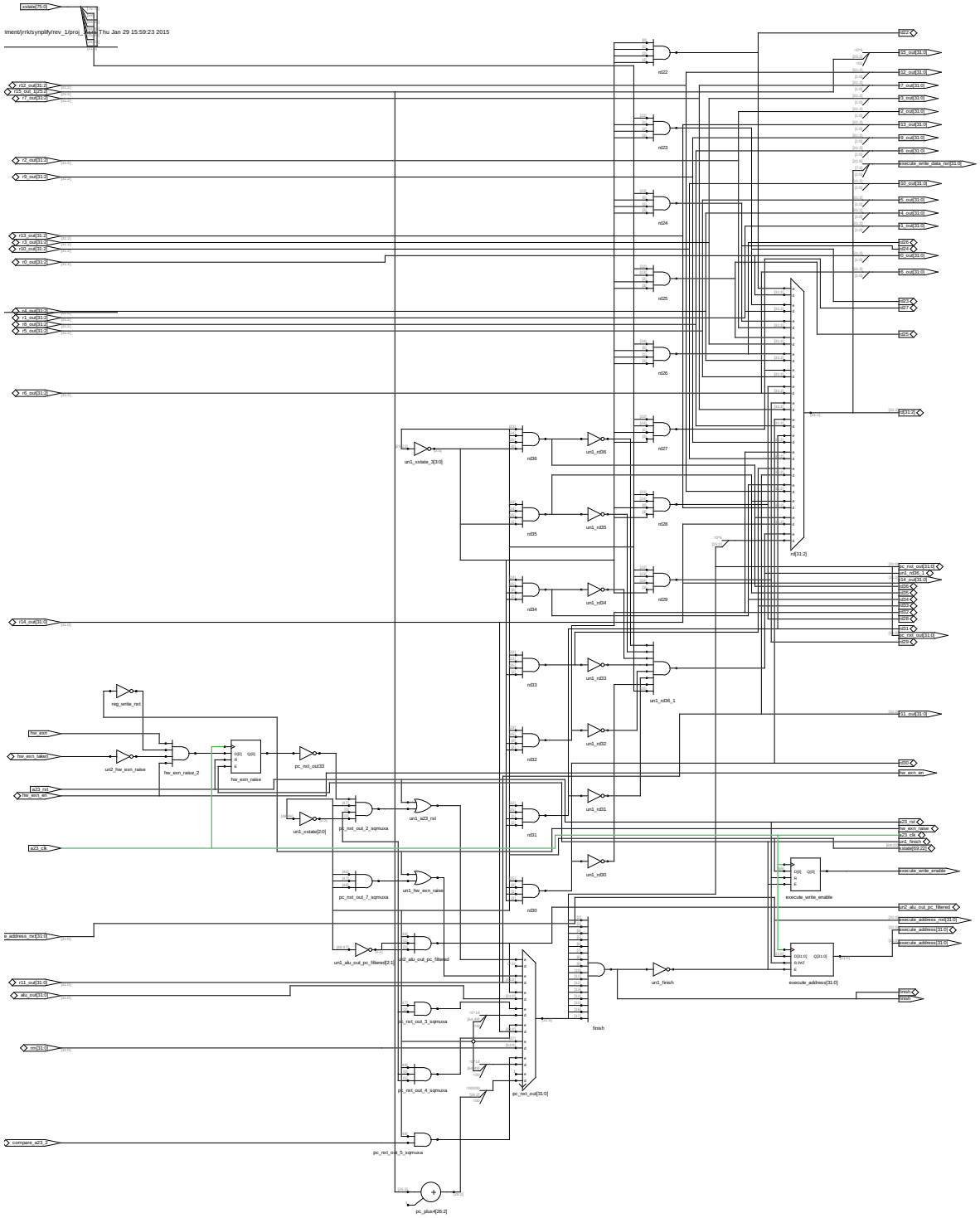
(c)


abs[32:1]
b[31:0]
un1_rm_1[32:0]
un1_rm[31:0]
barrel_shift_out[31:0]
barrel_shift_out8
barrel_shift_out9
barrel_shift_out11
lsh
lsl_out[32:0]
rsh
lsr_out[32:0]
rsh_0
asr_out[63:0]
barrel_shift_in_1[31:0]
un4[4:0]
un1_barrel_shift_out10
un1_barrel_shift_in
un4_barrel_shift_in
un11_barrel_shift_in
un2_barrel_shift_in
un3_barrel_shift_in
un3_shift_amount_3
un3_shift_amount[3:0]
un3_shift_amount_1[0]
un3_shift_amount_2[0]
barrel_shift_out10
un1_xstate[95]
un1_xstate_1[1:0]
shift_amount[7:0]
barrel_shift_in[1:0]
un2_shift_amount
un6_shift_amount
un14_shift_amount
un15_shift_amount
un12_shift_amount
un1_shift_amount[1:0]
xstate[75:0]
execute_address[31:0]
read_data[31:0]
rm[31:0]
rm[31:0]
rs[7:0]

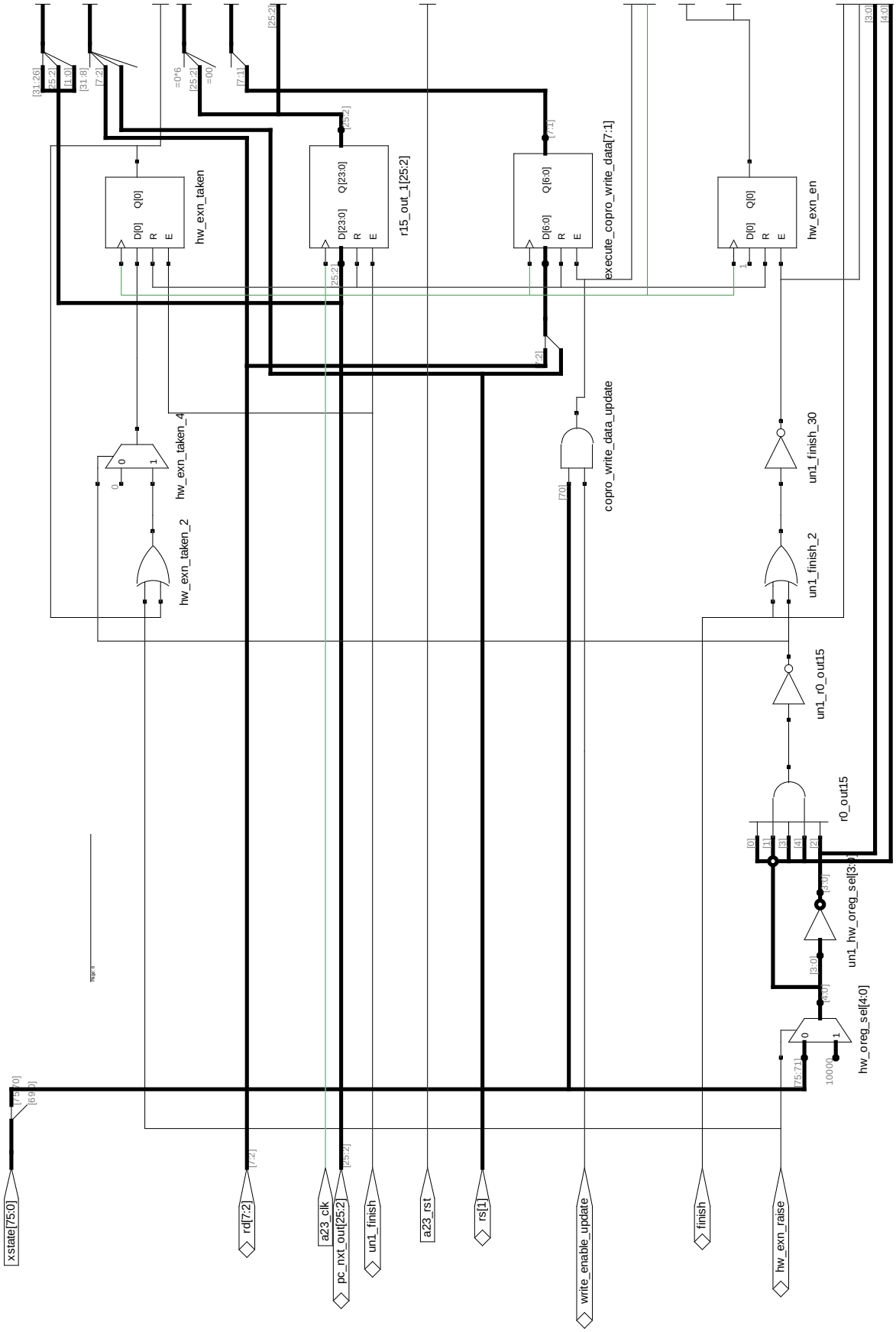
(d)



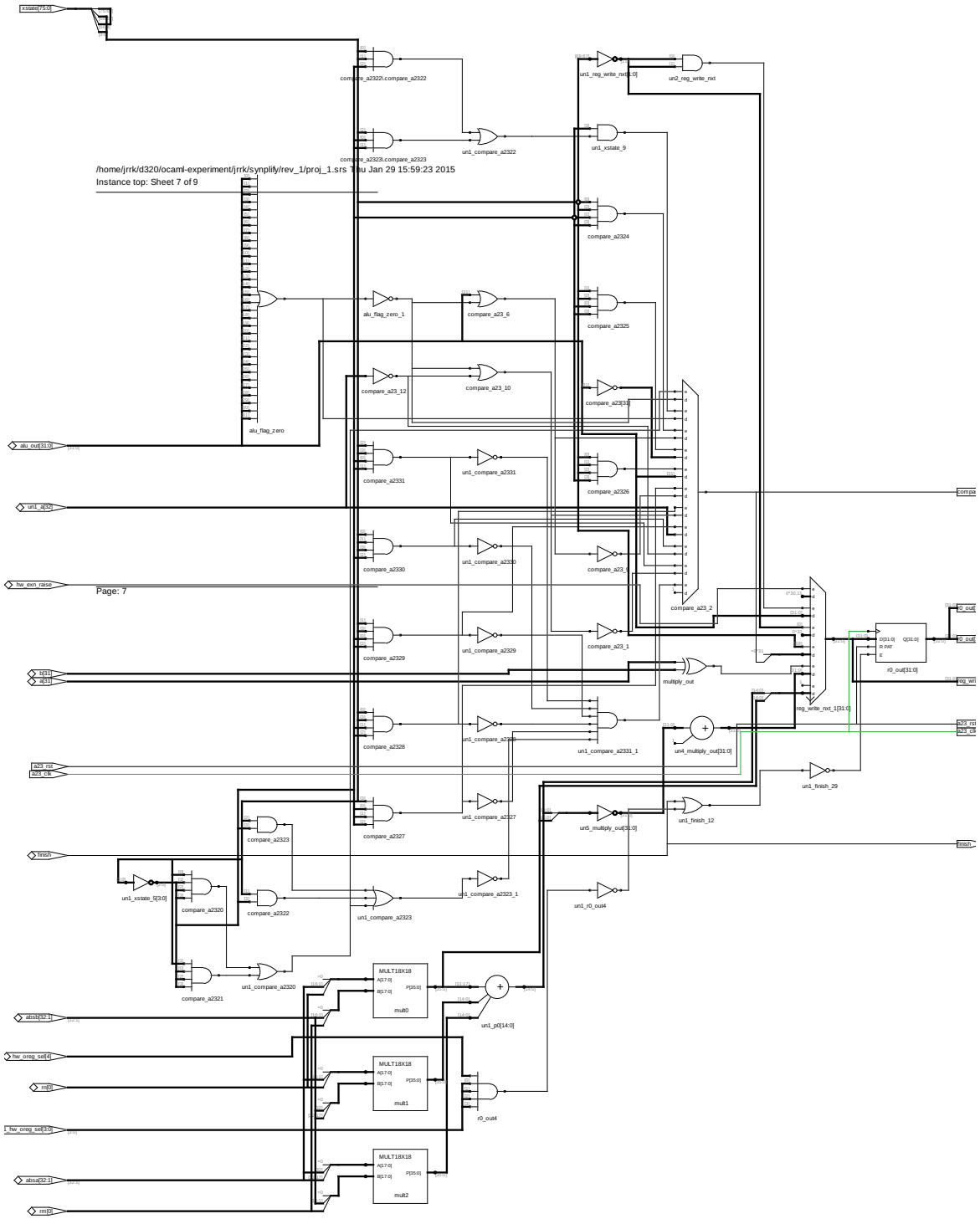
(e)



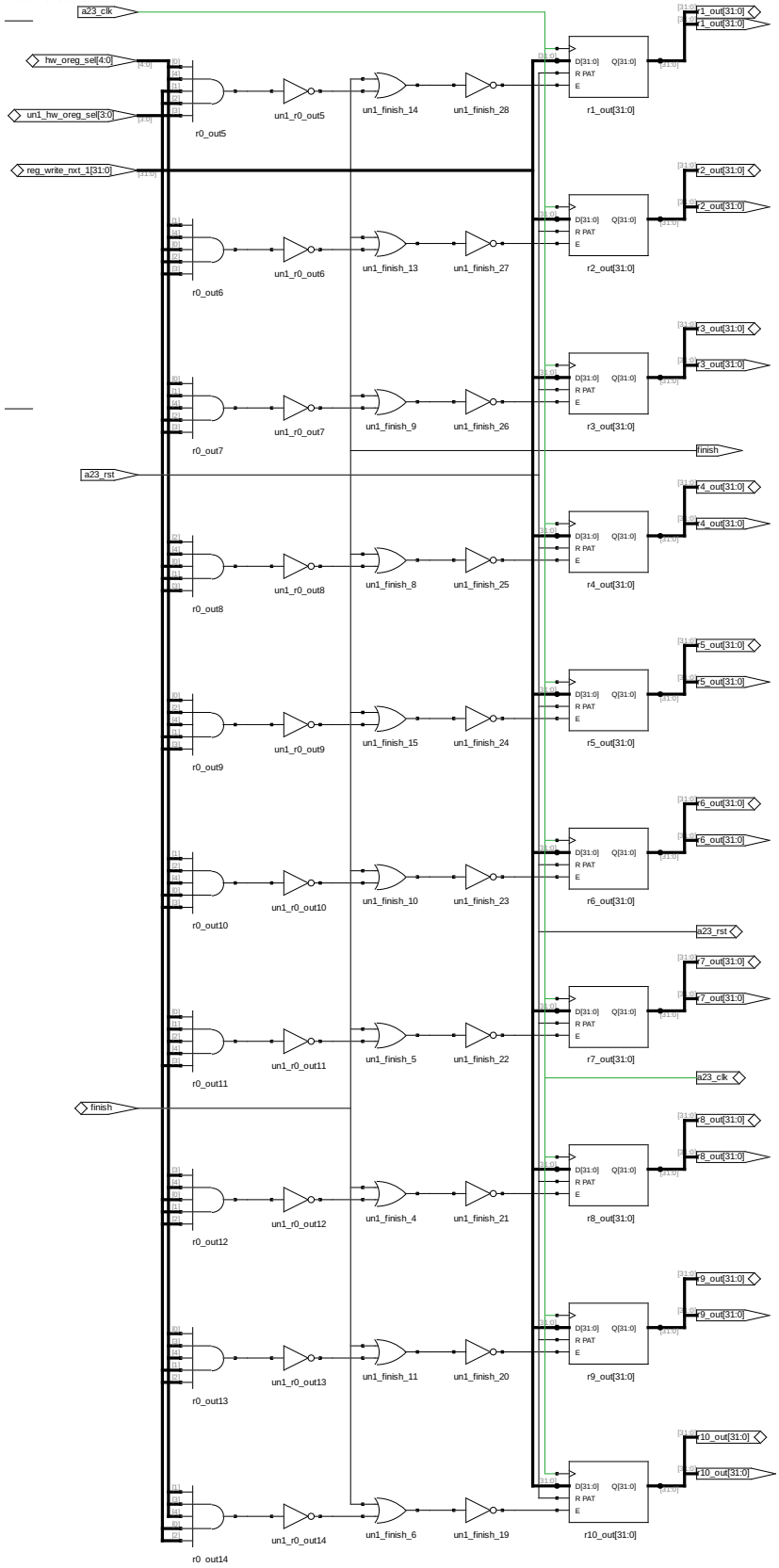
(f)


hw_exn_taken
r15_out_1[25:2]
execute_copro_write_data[7:1]
hw_exn_en
hw_exn_taken_4
hw_exn_taken_2
copro_write_data_update
un1_finish_30
un1_finish_2
un1_r0_out15
r0_out15
un1_hw_oreg_sel[3:0]
hw_oreg_sel[4:0]
xstate[75:0]
rd[7:2]
a23_clk
pc_nxt_out[25:2]
un1_finish
a23_rst
rs[1]
write_enable_update
finish
hw_exn_raise

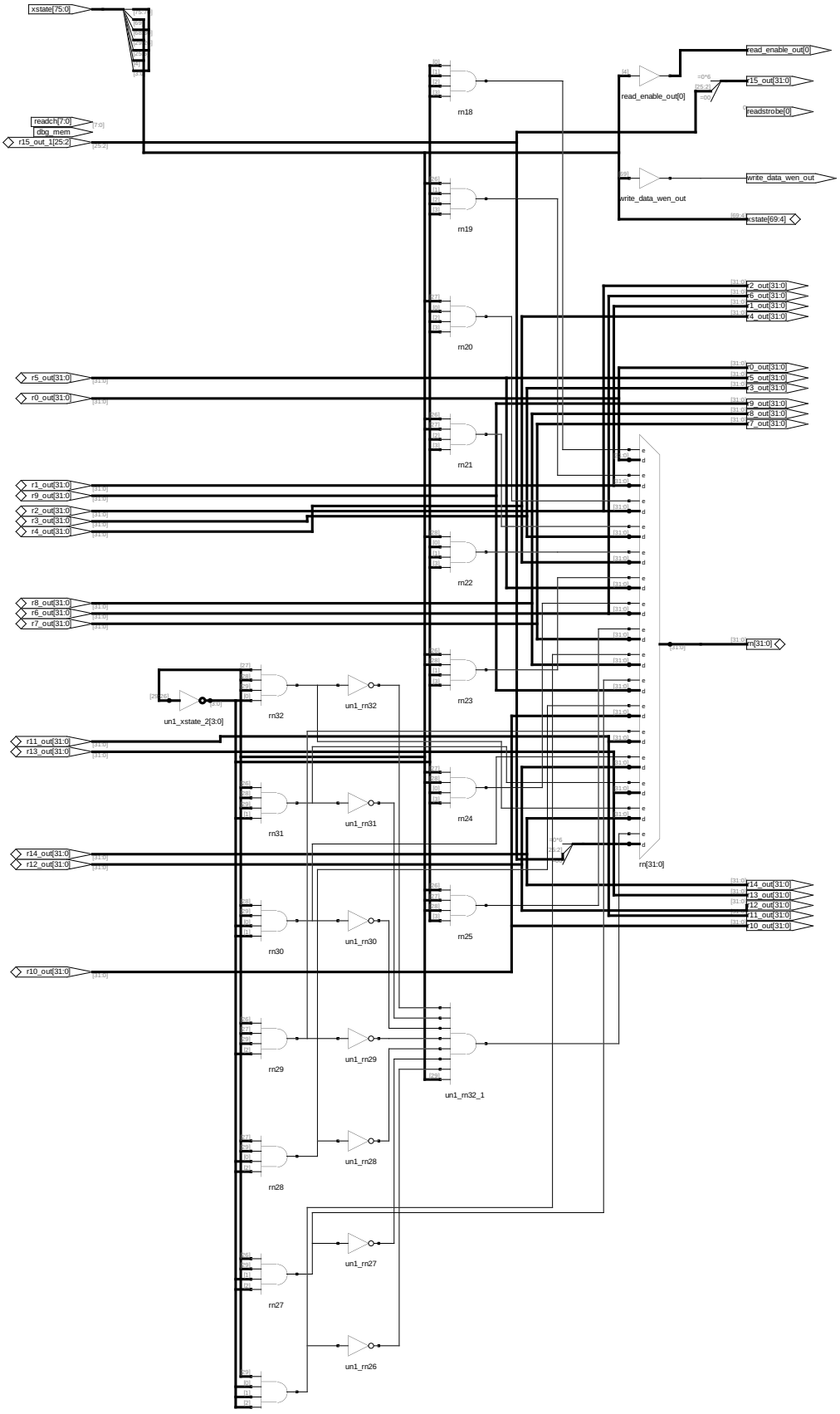
(g)


/home/jrrk/d320/ocaml-experiment/jrrk/synplify/rev_1/proj_1.srs Thu Jan 29 15:59:23 2015
Instance top: Sheet 7 of 9
Page: 7
xstate[75:0]
alu_out[31:0]
un1_a[32]
hw_exn_raise
b[31]
a[31]
a23_rst
a23_clk
finish
absb[32:1]
hw_oreg_sel[4]
rm[0]
hw_oreg_sel[3:0]
absa[32:1]
rm[0]
alu_flag_zero
alu_flag_zero_1
compare_a2322\.compare_a2322
compare_a2323\.compare_a2323
un1_compare_a2322
un1_reg_write_nxt[31:0]
un2_reg_write_nxt
un1_xstate_9
compare_a2324
compare_a2325
compare_a23_6
compare_a23_12
compare_a23_10
compare_a2326
compare_a2331
un1_compare_a2331
compare_a2330
un1_compare_a2330
compare_a23_3
compare_a23_2
compare_a2329
un1_compare_a2329
compare_a23_1
multiply_out
compare_a2328
un1_compare_a2328
un1_compare_a2331_1
un4_multiply_out[31:0]
reg_write_nxt_1[31:0]
r0_out[31:0]
un1_finish_29
compare_a2327
un1_compare_a2327
un5_multiply_out[31:0]
un1_finish_12
compare_a2323
un1_xstate_5[3:0]
compare_a2320
compare_a2322
un1_compare_a2323
un1_compare_a2323_1
un1_r0_out4
compare_a2321
un1_compare_a2320
MULT18X18
mult0
mult1
mult2
un1_p0[14:0]
r0_out4

(h)


a23_clk
hw_oreg_sel[4:0]
un1_hw_oreg_sel[3:0]
reg_write_nxt_1[31:0]
a23_rst
finish
r0_out5
r0_out6
r0_out7
r0_out8
r0_out9
r0_out10
r0_out11
r0_out12
r0_out13
r0_out14
un1_r0_out5
un1_r0_out6
un1_r0_out7
un1_r0_out8
un1_r0_out9
un1_r0_out10
un1_r0_out11
un1_r0_out12
un1_r0_out13
un1_r0_out14
un1_finish_14
un1_finish_13
un1_finish_9
un1_finish_8
un1_finish_15
un1_finish_10
un1_finish_5
un1_finish_4
un1_finish_11
un1_finish_6
un1_finish_28
un1_finish_27
un1_finish_26
un1_finish_25
un1_finish_24
un1_finish_23
un1_finish_22
un1_finish_21
un1_finish_20
un1_finish_19
D[31:0]
Q[31:0]
R PAT
E
r1_out[31:0]
r2_out[31:0]
r3_out[31:0]
r4_out[31:0]
r5_out[31:0]
r6_out[31:0]
r7_out[31:0]
r8_out[31:0]
r9_out[31:0]
r10_out[31:0]
finish
a23_rst
a23_clk

(i)


xstate[75:0]
readch[7:0]
dbg_mem
r15_out_1[25:2]
rn18
rn19
rn20
rn21
rn22
rn23
rn24
rn25
rn32
rn31
rn30
rn29
rn28
rn27
un1_rn32
un1_rn31
un1_rn30
un1_rn29
un1_rn28
un1_rn27
un1_rn26
un1_rn32_1
un1_xstate_2[3:0]
read_enable_out[0]
write_data_wen_out
xstate[69:4]
readstrobe[0]
r15_out[31:0]
r5_out[31:0]
r0_out[31:0]
r1_out[31:0]
r9_out[31:0]
r2_out[31:0]
r3_out[31:0]
r4_out[31:0]
r8_out[31:0]
r6_out[31:0]
r7_out[31:0]
r11_out[31:0]
r13_out[31:0]
r14_out[31:0]
r12_out[31:0]
r10_out[31:0]
rn[31:0]

Appendix-J

This appendix contains the extended LRTT parser example of section 9.1.

```
open Mylib
open Ml_decls
open LRTTtypes

    (* instantiating the parser functor *)

module BackusParse = struct
  type t = syntax_t
  type parsevalue = t
  and grammar = production list
  and production = string * choice list       (* rule name -> e1 | e2 | ... *)
  and choice = string * string * parsing list (* choice name, comment, sequence *)
  and semantic = int -> int -> parsevalue -> parsevalue
  and messaging = int -> int -> string -> parsevalue -> unit
  and acceptor = int -> string -> int
  and parsing =
    | A of acceptor * string              (* parsing test callback *)
    | S of semantic                       (* semantic action *)
    | D of semantic                       (* error recovery by deletion *)
    | M of messaging * string             (* debugging action executed "en passant" *)
    | Z of parsing                        (* e* *)
    | P of parsing                        (* e+ *)
    | Opt of parsing                      (* e? *)
    | L of parsing                        (* lookahead, parsing resumes at the START *)
    | NT of string                        (* nonterminal 'name' *)
    | Empty                               (* epsilon *)
  and parsepoint = {
      mutable from : parsepoint;              (* source node back pointer *)
      mutable source : string;                (* source node id *)
      mutable pos : int;                      (* text position at start of production *)
      mutable stamp : int;                    (* node stamp for debugging *)
      mutable prod : parsing list;            (* start of production *)
      mutable curs : parsing list;            (* end of production segment for this node  *)
      mutable recursive : parsing list;       (* location of recursive call in this production *)
      mutable hook : parsepoint list;         (* current recursive call of this node *)
      mutable successors : parsepoint list }  (* recursion stack + production list + node successor *)
  let full_debug = 4096
  let run_debug = 8192
  let flush_memo = 16384
  let single_parse = 32768
  let node_stamping = 65536
end

open BackusParse

exception Failed

let demo impl streamparse f =

   (* incremental parser input *)

let parseloc = ref 0 in
let lines = ref [0] in
let chanstack = ref [] in
let filestack = ref [] in
let startstack = ref [[]] in

let input = ref "" in

let len = ref 0 in

let startrule = ref "" in

let rec stream_line (ix,strm,fd) =
  if !ix >= String.length strm then raise (Failure strm);
  let ch = strm.[!ix] in incr ix;
  if ch <> '\n' then String.make 1 ch^stream_line (ix,strm,fd) else "" in
let rec stream_close (ix,strm,fd) = () in
let charfeed ask lng =
  let pos = ask in
  if ask < 0 || lng < 0 then
    ""
  else begin
    while pos + lng > !len do
      try
        let line = stream_line (List.hd !chanstack) in
        input := !input ^ "\n" ^ line;
        len := String.length !input;
```

```
      with
        _ ->
          if (List.tl !chanstack) = [] then raise Not_found;
          stream_close (List.hd !chanstack);
          chanstack := List.tl !chanstack;
          filestack := List.tl !filestack;
          lines := List.hd !startstack;
          startstack := List.tl !startstack;
    done;

    for i = (if !lines = [] then 0 else (List.hd !lines) + 1) to pos+lng-1 do
      if !input.[i] = '\n' then lines := i :: !lines;
    done;

    if pos > !parseloc then parseloc := pos;   (* for the 'read' stream interleave *)

    (String.sub !input pos lng)
  end in
    (* "on-the-fly" lexing functions *)
let testeof pos _ =
  (try
    let _ = charfeed pos 0 in ();
    (try
      let _ = charfeed pos 1 in ();
      0                     (* before eof *)
    with Not_found -> -1)   (* exactly at eof *)
  with Not_found -> 0) in   (* BEYOND 'eof' isn't eof *)
let tsymbol pos name =
  let k = String.length name in
  if String.eqb (try charfeed pos k with Not_found -> "") name then k else 0 in
let nquotchar pos =
  if (charfeed pos 1).[0] = '\\' then
      (* peek one extra char beyond to account for the closing quote  *)
      let _ = charfeed (pos+2) 1 in
      2
  else
    1 in
let lineskip ptr = while (not (String.eqb (charfeed !ptr 1) "\n")) do  ptr := !ptr +1; done in
let tspace pos _ =
  let ptr = ref pos  in
  (try
    while (let ch = String.get (charfeed !ptr 1) 0 in ch = ' ' || ch = '\n' || ch = '\t') do
      ptr := !ptr +1;
    done;
    if String.eqb (charfeed !ptr 2) "--" then begin
      ptr := !ptr +2;
      lineskip ptr;
      ptr := !ptr +1;
  end;
  with Not_found -> ());
  (!ptr - pos) in
let stk = ref [] in
let pushvalue pval = stk := pval :: !stk in
let popvalue () =
    let pop = List.hd !stk in
    stk := List.tl !stk;
    pop in
let slet recurse beg pos pval =
  (try let tok = charfeed pos (pos - beg) in
  if impl.debug.debugall then print_endline ("let_rec "^tok);
  with Not_found -> ());
  pushvalue pval; Bool recurse in

let keywords = [ "let"; "rec"; "in"; "if"; "then"; "else"; "mod" ] in

let grammar : grammar = [
  "start", [
    "", "", [NT "osp"; NT "expr"];
    "", "", [NT "osp"; NT "top"];
  ];

  "top", [
    "", "", [NT "osp"; NT "topexp"; NT "expr"; S (fun beg pos pval ->
      print_endline ("top: "^impl.dump pval);
      match popvalue() with List lst ->
        LET (List.rev lst, pval) | oth -> failwith (String.concat ", " (List.map impl.dump [oth;pval])))];
  ];

  "lets", [
    "", "", [A(tsymbol, "let"); NT "sp"; A(tsymbol, "rec"); NT "sp"; NT "composition"; S (slet true)];
    "", "", [A(tsymbol, "let"); NT "sp"; NT "composition"; S (slet false)]
  ];

  "let", [
  "let", "declaration", [NT "lets"; NT "assign"; NT "start"; NT "sp"; A(tsymbol, "in"); NT "sp";
                         S (fun beg pos pval ->
      let flg = popvalue() in
      let lft,mid = match popvalue() with List lst ->
        let rlst = List.rev lst in (List.hd rlst,List(List.tl rlst)) | oth -> failwith (impl.dump oth) in
      if impl.debug.debugall then print_endline (String.concat ", " (List.map impl.dump [flg;lft;mid;pval]));
      Declare ((match flg with Bool t -> t | oth -> failwith (impl.dump oth)), lft, mid, pval))]
  ];

  "assign", [
    "", "", [NT "osp"; A(tsymbol, "="); NT "osp"]
  ];

  "topexp", [
  "inlst", "" , [NT "topexp"; NT "let"; S (fun beg pos pval ->
    print_endline ("inlst: "^impl.dump pval);

    match popvalue() with
```

```
    List lst -> List (pval :: lst)
  | oth -> failwith (String.concat ", " (List.map impl.dump [oth;pval])))];
  "topone", "", [NT "let"; S (fun beg pos pval ->
    print_endline ("topone: "^impl.dump pval);
    List [pval])];
];

"expr", [
"fundecl", "function declaration",
[NT "osp"; A(tsymbol, "fun"); NT "sp"; NT "patt"; NT "osp"; A(tsymbol, "->"); NT "osp"; NT "expr";
  S (fun beg pos pval -> Abstraction (popvalue (), pval))];
"fadd", "", [NT "subexp"; NT "osp"; A(tsymbol, "+."); NT "osp"; NT "subexp";
              S (fun beg pos pval -> FAdd (popvalue (), pval))];
"fsub", "", [NT "subexp"; NT "osp"; A(tsymbol, "-."); NT "osp"; NT "subexp";
              S (fun beg pos pval -> FSub (popvalue (), pval))];
"addition", "", [NT "subexp"; NT "osp"; A(tsymbol, "+"); NT "osp"; NT "subexp";
                  S (fun beg pos pval -> Add (popvalue (), pval))];
"subtraction", "", [NT "subexp"; NT "osp"; A(tsymbol, "-"); NT "osp"; NT "subexp";
                     S (fun beg pos pval -> Sub (popvalue (), pval))];
"if", "", [NT "osp"; A (tsymbol, "if") ; NT "osp"; NT "boolean"; NT "osp"; A (tsymbol, "then") ;
           NT "expr"; NT "osp"; A (tsymbol, "else"); NT "osp"; NT "expr";
             S (fun beg pos pval -> let mid = popvalue () in If (popvalue (), mid, pval))];
  "subexp", "", [NT "subexp"];
  "eof", "", [A (testeof, "EOF")];
];

"subexp", [

"funappl", "", [NT "primary"; NT "sp"; NT "funargs"; S (fun beg pos pval -> App (popvalue (),
    match pval with List lst -> List.rev lst | oth -> failwith (impl.dump oth)))];
"fmult", "", [NT "primary"; NT "osp"; A(tsymbol, "*."); NT "osp"; NT "primary";
               S (fun beg pos pval -> FMul (popvalue (), pval))];
"fdiv", "", [NT "primary"; NT "osp"; A(tsymbol, "/."); NT "osp"; NT "primary";
              S (fun beg pos pval -> FDiv (popvalue (), pval))];
"multiplication", "", [NT "subexp"; NT "osp"; A(tsymbol, "*"); NT "osp"; NT "primary";
                       S (fun beg pos pval -> Mul (popvalue (), pval))];
"division", "", [NT "primary"; NT "osp"; A(tsymbol, "/"); NT "osp"; NT "primary";
                  S (fun beg pos pval -> Div (popvalue (), pval))];
"modulo", "", [NT "primary"; NT "osp"; A(tsymbol, "mod"); NT "osp"; NT "primary";
                S (fun beg pos pval -> Mod (popvalue (), pval))];
"primary", "", [NT "primary"];
"eof", "", [A (testeof, "EOF")];
];

"funargs", [
  "arglist", "" , [NT "funargs"; NT "sp"; NT "primary"; S (fun beg pos pval ->
    match popvalue() with List lst -> List (pval :: lst) | oth ->
      failwith (String.concat ", " (List.map impl.dump [oth;pval])))];
  "onearg", "", [NT "primary"; S (fun beg pos pval -> List [pval])];
  "eof", "", [A (testeof, "EOF"); S (fun beg pos pval -> List [])];
];

"primary", [
  "unit", "", [NT "osp"; A (tsymbol, "("); A(tsymbol, ")"); S (fun _ _ pval -> pushvalue pval; Unit)];
  "paren", "", [NT "osp"; A (tsymbol, "("); NT "osp"; NT "start"; NT "osp"; A(tsymbol, ")")];
  "ident", "", [NT "osp"; NT "patt"];
  "float", "", [NT "osp"; NT "float"];
  "integer", "", [NT "osp"; NT "integer"];
  "eof", "", [A (testeof, "EOF")];
];

 "float", [
  "float", "", [A ((fun pos _ ->
let ptr = ref pos in
(try lineskip ptr; with Not_found -> ());
let line = charfeed pos (!ptr - pos) and need_dot = ref true and k = ref 0 in
while (!k < String.length line) && (((line.[!k] >= '0') && (line.[!k] <= '9')) || ((line.[!k] = '.') && !k > 0)) do
  if line.[!k] = '.' then need_dot := false;
  incr k
done;
if !need_dot then 0 else
  begin
    if impl.debug.debugall && !k > 0 then
      print_endline ("Float "^string_of_int !ptr^" "^string_of_int(!k)^" '"^(String.sub line 0 !k)^"'");
    !k
  end),"float"); S (fun beg pos pval -> pushvalue pval;
let res = ref Unit in
let str = charfeed (beg) (pos - beg) in
if impl.debug.debugall then print_endline ("Float "^str);
(try
  res := Float (impl.float_of_string str)
with _ -> ());
!res)];
];

 "integer", [
"", "", [A ((fun pos _ -> let ptr = ref pos in (try lineskip ptr; with Not_found -> ());
let line = charfeed pos (!ptr - pos) in
let k = ref 0 in
while (!k < String.length line) && (line.[!k] >= '0') && (line.[!k] <= '9') do incr k done;
if !k > 0 && impl.debug.debugall then print_endline ("Integer "^string_of_int !k^" '"^(String.sub line 0 !k)^"'\n");
!k), "integer"); S (fun beg pos pval -> pushvalue pval;
let res = ref Unit in
let str = charfeed (beg) (pos - beg) in
if impl.debug.debugall then print_endline ("Integer "^str);
(try
  res := Int (int_of_string str)
with _ -> ());
!res) ];
];

"boolean", [
```

```
  "paren", "", [NT "osp"; A(tsymbol, "("); NT "osp"; NT "boolean"; NT "osp"; A(tsymbol, ")")];
  "equality", "", [NT "expr"; NT "assign"; NT "expr";
                   S (fun beg pos pval -> Eq (popvalue (), pval))];
  "less", "", [NT "expr"; NT "osp"; A(tsymbol, "<"); NT "osp"; NT "expr";
               S (fun beg pos pval -> LT (popvalue (), pval))];
  "greater", "", [NT "expr"; NT "osp"; A(tsymbol, ">"); NT "osp"; NT "expr";
                  S (fun beg pos pval -> GT (popvalue (), pval))];
  "lessequal", "", [NT "expr"; NT "osp"; A(tsymbol, "<="); NT "osp"; NT "expr";
                    S (fun beg pos pval -> LE (popvalue (), pval))];
  "greaterequal", "", [NT "expr"; NT "osp"; A(tsymbol, ">="); NT "osp"; NT "expr";
                       S (fun beg pos pval -> GE (popvalue (), pval))];
    "", "", [NT "primary"];
    "false", "", [A (tsymbol, "false") ; S (fun beg pos pval -> pushvalue pval; Bool false)];
    "true", "", [A (tsymbol, "true"); S (fun beg pos pval -> pushvalue pval; Bool true)];
    "eof", "", [A (testeof, "EOF")];
  ];

  "composition", [
    "leftrecurs", "" , [NT "composition"; NT "sp"; NT "patt"; S  (fun beg pos pval ->
      match popvalue() with
        List lst -> List (pval :: lst)
      | oth -> failwith (String.concat ", " (List.map impl.dump [oth;pval])))];
    "initial", "", [NT "patt"; S (fun beg pos pval -> List [pval])];
  ];

  "patt", [ "", "", [A ((fun pos _ -> let ptr = ref pos  in
  (try
    while (let ch = charfeed !ptr 1 in ((ch.[0]) >= 'a' && (ch.[0]) <= 'z') || (ch.[0] = '_')) do
      ptr := !ptr +1;
    done;
  with Not_found -> ());
  let oth = charfeed pos (!ptr - pos) in
  if impl.debug.debugall then print_endline oth;
  if List.mem' String.eqb oth keywords then 0 else (!ptr - pos)), "-id-"); S (fun beg pos pval ->
  pushvalue pval;
  Var (charfeed beg (pos - beg)))];
  ];

  "tstring", [ "", "", [A ((fun pos _ ->
  (try
    if String.eqb (charfeed pos 1) "\"" then
      let ptr = ref (pos + 1) in
      while not (String.eqb (charfeed !ptr 1) "\"") do
        ptr := !ptr + (nquotchar !ptr);
      done;
      ((!ptr - pos) + 1)
    else
      0
  with Not_found -> 0)), "string"); S (fun beg pos pval ->
  pushvalue pval;
  String (charfeed (beg + 1) (pos - (beg + 2))))];
  ];

  "tquotchar", [ "", "", [A ((fun pos _ ->
  (try
    let k = nquotchar (pos+1) in
    if (charfeed pos 1).[0] = '\'' && (charfeed (pos+k+1) 1).[0] = '\'' then
      k + 2
    else
      0
  with Not_found -> 0)), "quotchar"); S (fun beg pos pval ->
  pushvalue pval;
  let ch = (charfeed (beg + 1) 1).[0] in
  if ch <> '\\' then Char ch
  else match (charfeed (beg + 2) 1).[0] with
    | 'n' -> Char '\n'
    | 't' -> Char '\t'
    | '\'' -> Char '\''
    | oth -> Char oth)];
  ];

  "sp", [
    "", "", [P (A(tspace, "_"))];
  ];
  "osp", [
    "", "", [Z (A(tspace, "_"))];
  ];
]
in
  begin
    chanstack := (ref 0,(f^"\n"),Std.in_) :: !chanstack;
    filestack := "-" :: !filestack;
    startstack := !lines :: !startstack;

  (try
    input := stream_line (List.hd !chanstack);
    len := String.length !input;
  with
    _ ->
      print_endline "NO input";
      );

  let (parsed,i) = streamparse !parseloc Unit grammar
      (if String.eqb !startrule "" then "start" else !startrule)
      ((if impl.debug.debugrun then run_debug else 0)
         + (if impl.debug.debugall then full_debug else 0)
         + (if impl.debug.stamping then node_stamping else 0)
         + (if impl.debug.memflush then flush_memo else 0)
         + (impl.debug.memosize land 4095))
  in

  if i <= 0 then
```

```
  failwith ("parse error: parsing failed "^string_of_int i)
else if i < !len then
  failwith ("parse error: "^string_of_int (!len - i)^
            " extra characters after end of acceptable input: "^String.sub !input 0 i)
else
  begin
  if impl.debug.debugall then print_endline ("parser returned: "^ !input);
  parsed
  end
end
```

END

OF

DISSERTATION