Hardware Synthesis From a Traditional Programming Language

by

Dylan Radcliffe

A thesis submitted to the

School of Information Technology and Electrical Engineering
The University of Queensland

for the degree of

BACHELOR OF ENGINEERING IN THE DIVISION OF ELECTRICAL ENGINEERING

October 2002

14 Jenkinson Street
Indooroopilly, QLD 4068

The Head
School of Information Technology and Electrical Engineering
The University of Queensland
St Lucia, QLD 4068

Dear Professor Kaplan,

In accordance with the requirements of the degree of Bachelor of Engineering in the division of Electrical Engineering, I present the following thesis entitled “Hardware Synthesis From a Traditional Programming Language”. This work was performed under the supervision of Dr Peter Sutton.

I declare that the work presented in the thesis is, to the best of my knowledge and belief, original and my own work, except as acknowledged in the text and bibliography, and that the material has not been submitted, either in whole or in part, for a degree at this or any other university.

Yours Sincerely,

Dylan Radcliffe

Abstract

A key factor in the development of modern, complex digital technology has been the use of automated design tools. A significant area of research in electronic design automation is in the development of tools that can synthesize hardware from a design written in the form of a high-level programming language such as C. It is believed that these "hardware compilers" will help to decrease development time, thus shortening the crucial time-to-market for designs. This makes them particularly applicable to programmable hardware devices such as Field Programmable Gate Arrays (FPGAs) that are growing in popularity as fast, reconfigurable development platforms.

The aim of this thesis was to produce an open implementation of a hardware compiler that was flexible enough to be extended by future developers. This would offer an advantage over current commercial products that are closed, proprietary systems, not freely available and essentially unmodifiable. The development of a first version of this tool in C++, using non-proprietary libraries, is given in this document. Descriptions of the design issues surrounding the implementation of each of these modules are presented.

The outcome of this study was the hardware compiler, *Circe*, that accepts input in a variant of C and produces a structural hardware description in register transfer level VHDL. This output can be mapped to a specific technology using existing logic-level synthesis tools. Circe was tested and the designs produced were mapped to a Xilinx FPGA to demonstrate a complete design-flow. They were shown to accurately reflect the original C behaviour and the efficiency of the produced hardware was comparable with that of a similar commercial tool. A number of future directions for Circe are outlined to demonstrate the potential of this open system.

Acknowledgments

I wish to acknowledge all of the people who have helped me with this thesis, in particular:

My supervisor, Dr Peter Sutton, for his guidance throughout this project.

My family for providing support, in particular my mother for proofreading this report and my father for his valuable advice.

The librarians of UQ for their excellent provision of an invaluable service.

The writers of open software, particularly those that have contributed to the GNU project.

Last but not least, I would like to acknowledge my friends for the help and support they have given me over the years.

This thesis was typeset using LyX.

“It would appear we have reached the limits of what it is possible to achieve with computer technology, although one should be careful with such statements; they tend to sound pretty silly in five years."

John von Neumann ca. 1949.

Contents

Abstract **ii**

Acknowledgments **iii**

1 Introduction **1**

1.1 Introduction to Hardware Synthesis 1

1.2 Motivation 1

1.3 Outcomes 2

1.4 Outline of this document 3

2 Background and Review of Previous Work **5**

2.1 Design Domains and Synthesis 5

2.2 Hardware Description Languages (HDLs) 7

2.3 Programming Language Based Synthesis 7

2.4 Design Representations in Synthesis Systems 8

2.5 Efficiency and Optimization 8

2.6 Summary of Previous Work 9

2.6.1 Synopsys' Cocentric SystemC Compiler (Commercial) 9

2.6.2 Handel-C (Commercial) 9

2.6.3 Bach (Research) 10

2.6.4 SPARK (Research) 10

2.6.5 SpC (Research) 10

2.6.6 Summary of Previous Work 11

2.7 Summary 11

3 Problem Definition and Scope **12**

3.1 Design Input 12

3.2 Design output 13

3.2.1 Datapath 14

3.2.2 Controller 14

3.2.3 Output Format 14

3.3 Summary 15

4 Design and Implementation **17**

4.1 Selection of Environment 17

4.2 Overall System Design 18

4.3 The Parser 19

4.3.1 The Abstract Syntax Tree (AST) 20

4.3.2 The Symbol Table 21

4.3.3 Scanning and Parsing 21

4.3.4 Semantic Analysis 22

4.4 The Synthesis Engine 22

4.4.1 Intermediate Representation (IR) Overview 23

4.4.2 Control-flow Graph (CFG) Implementation 24

4.4.3 Data-flow Graph (DFG) Implementation 25

4.4.4 Translation to the IR 26

4.4.5 High-level Synthesis 28

4.5 The Backend 29

4.5.1 Functional Units and Registers 30

4.5.2 Structuring the Datapath 31

4.5.3 Structuring the Controller 32

4.6 Summary 33

5 Design Evaluation **35**

5.1 Testing 35

5.1.1 Methodology 35

5.1.2 Example Programs 36

5.1.3 Results 37

5.2 Discussion of Observations 41
5.2.1 Design Quality 41
5.2.2 Semantic Issues 44
5.3 Comparison with Specification 45
5.3.1 Design Input 45
5.3.2 Design Output 45
5.3.3 Extensibility 46
5.4 Comparison with related products 48
5.4.1 Overall Comparisons 48
5.4.2 Direct Comparison with Handel 49
5.5 Evaluation of Engineering performance 50
5.5.1 Process 50
5.5.2 Contribution 51
5.6 Summary 51

6 Future Developments **52**
6.1 The Input Language 52
6.2 The Synthesis Engine 54
6.3 Optimizations 54
6.4 Targets 55
6.5 Summary 56

7 Conclusions **57**

Bibliography **59**

Appendices **62**

A A Worked Example : Noughts and Crosses **62**

B A Brief Guide to the Source Code for Circe **79**

List of Figures

2.1 Gajski's "Y-Chart" [1] 6

4.1 Major components of the system 18
4.2 An example of a AST for an *if* statement 20
4.3 A example CFG fragment resulting from an if statement 25
4.4 A example DFG fragment resulting from an assignment statement 26
4.5 Translation to the IR (Partially completed) 27
4.6 An example of maintaining algorithmic integrity 32

5.1 The up/down counter simulation 39
5.2 The matrix multiply simulation 39
5.3 The noughts and crosses example 40
5.4 Circe generated datapath for a counter 42
5.5 The adder functional-unit before and after logic-level optimization 43

6.1 An example of moving behaviour from the control to the data-flow 55

List of Tables

2.1 Comparison of Previous Work 11

5.1 Area cost of produced designs targeted to a SpartanXL FPGA 41

5.2 Achievement of Objectives 47

5.3 Comparison of Previous Work with Circe 48

5.4 Direct Comparison With Handel-C 50

Chapter 1

Introduction

1.1 Introduction to Hardware Synthesis

The design of modern digital electronic systems involves specifying complicated VLSI (very large scale integration) circuits that may contain many millions of transistors. The ability to produce such designs relies on the use of various computer aided design (CAD) tools including tools for synthesis, simulation and verification [2]. Designing and synthesizing hardware from specialized hardware description languages (HDLs) is already commonplace

The Synthesis of hardware from a *traditional* or high-level programming language (HLL) such as *C* allows for an extra level of abstraction in hardware design enabling designers to concern themselves with algorithms rather than specific hardware techniques. These "hardware compilers" also provide an approach to hardware design that is similar to software design which is useful when designing components that may eventually be implemented in either hardware or software. Furthermore, a HLL based design process allows for quick prototyping of systems on programmable logic devices such as FPGAs.

1.2 Motivation

Hardware compilers have moved beyond research projects with commercial products such as Synopsis' Cocentric C and Celoxica's Handel-C being released. Despite this, skepticism of the usefulness of particularly *C* or *C++* based hardware design remains as expressed in a recent Design Automation Conference Panel [3]. One of the problems ex-

pressed is that there is a general misunderstanding of the advantages and disadvantages of this design methodology and its place in relation to other methods. Schaumont in [3] expresses a concern that the blanket advocation of HLL-based design as a replacement for HDLs leads to "false expectations". Areas such as System-level design, sequential and control dominated designs are, however, outlined as the strengths of hardware compilation.

One difficultly is the range of hardware compilation tools easily accessible to the general design community and to educational institutions is small and restrictive. Products are expensive and proprietary making modification or experimentation difficult or impossible. Electronic CAD vendors wishing to sell their particular products are ready to promote their approaches to hardware compilation as the "best" way or over-emphasize benefits and downplay disadvantages. What is needed is a way to make hardware compilation more accessible to a larger range of designers and educators. An open source[1] hardware compiler would provide a freely available and modifiable system giving designers, educators and students a tool to explore the possibilities of this design methodology. This was the motivation behind this thesis.

The aim of this thesis was to develop the framework for such an open system. In particular, to create a working design-flow from a HLL specification to a structural hardware description capable of being mapped to a particular technology by logic synthesis tools. The aim was to concentrate on producing a complete design-flow rather than perfecting a particular element of that design flow. Such a system would provide a basis for future work perhaps employing more sophisticated synthesis or optimization techniques or concentrating on more specific design areas.

1.3 Outcomes

A working, open, hardware compiler, Circe[2], was implemented in C++. This CAD tool takes an input design written in a variant of the C programming language and produces output in the form of a subset of VHDL chosen for the representation of a structural register-transfer level datapath and finite-state machine controller description. The tool includes: a parsing system that checks the syntax and semantics of the design and translates

[1] In this thesis the terms "open source" software and "open" or "free" software will be used interchangably. The differences between these terms as outlined by the Free Software Foundation[4] are unimportant for the purposes of the study.

[2] *Circe*, pronounced Sir-See, is a sorceress in Greek Mythology capable of transforming living creatures. The pronunciation is also a play on "Circuits from C".

the source code into a computer-manipulatable form; a synthesis engine that determines the specific controller states, functional units and registers that are needed and maps the behavioural input description to these resources; and a backend that handles the details of producing the output design.

This program was tested and shown to produce correct and synthesizable hardware designs. When the test designs were simulated (using a VHDL simulation tool) the simulation behaviour matched the behaviour of the original source given the semantics of the input language. The output designs were also mapped to both FPGA and VLSI technologies using commercial logic synthesis tools. The FPGA designs were implemented on a FPGA development board and were shown to function correctly.

Elements of the developed CAD tool need to be improved before it can reach the standard of commercial systems. Some restrictions on the input language and on the types of functional units were introduced to simplify the development of this initial system. In particular, function calls were removed from C as well as complex data types and functional units were restricted to single-cycle operation. Also, the high-level synthesis algorithms implemented were simple - enough to demonstrate the potential of the system but not to produce high quality designs. These aspects as well as other possibilities have been outlined as future extensions to the project.

1.4 Outline of this document

This document serves to present the background, aims, process and findings of this study. A brief introduction to hardware synthesis, hardware description languages, and synthesis from high-level languages is given in Chapter 2. This chapter also contains a guide to some previously implemented hardware compilation systems. Chapter 3 refines the overall aim of the thesis into a description of specific objectives for the proposed CAD tool. Decisions made as part of the scoping of the project rather than the design process proper are presented there.

Chapter 4 details the design and implementation of the CAD tool, Circe, focusing on the major decisions including environment selection, overall flow design and design representations at the input, intermediate and output stages of the flow. This implementation is then evaluated in Chapter 5 which also presents a brief comparison with other related products. Possible future improvements to this system are presented in Chapter 6. This is

done in terms of both improvements and extensions to each of the major components in the design flow.

Chapter 7 states the overall conclusions of the study.

Chapter 2

Background and Review of Previous Work

This chapter introduces the background theory needed for this thesis as well as presenting a summary of previous work in this area. An explanation of synthesis and how it relates to other computer-aided design (CAD) tools is given as well as an introduction to synthesis from hardware description languages and programming languages. A discussion of design representations in CAD packages as well as approaches to optimization in high-level synthesis is also included. The chapter concludes with a summary and comparison of some of the previous work in this field.

2.1 Design Domains and Synthesis

Gajski and Kuhn [1] divide VLSI design into three domains: Functional (also known as Behavioural), Structural and Geometrical (also known as Physical) with each of these domains being specified as a hierarchy of abstraction from high to low as shown in Figure 2.1. The functional domain is a "black box" view of a system or component that details behaviour without details of implementation. Design in the structural domain is concerned with the composition of the system or components by specifying how they break down into smaller and smaller blocks until the transistor level is reached. The geometrical domain is concerned with the details of how a design is physically mapped onto silicon.

Design synthesis can be defined as the transformation of a design to a level of lower abstraction. This definition is sometimes refined (for example in [5]) to the transformation

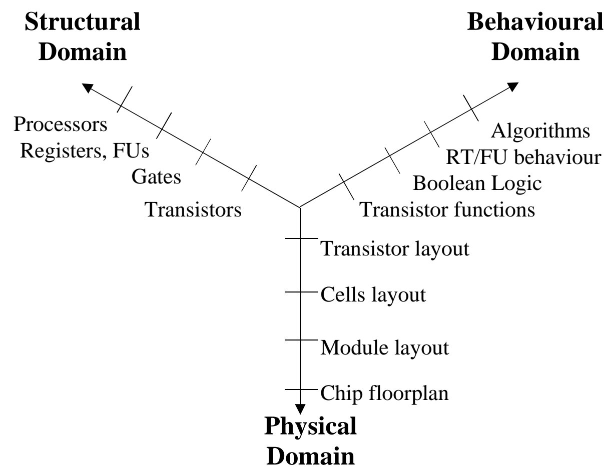


Figure 2.1: Gajski's "Y-Chart" [1]

of a design from a point in the functional domain to one in the structural domain. CAD tools can be used to perform synthesis tasks at different design points and with various levels of interactivity with the designer. Gerez [6] identifies logic synthesis and high-level synthesis as two of the major synthesis areas used in digital design. High-level synthesis (HLS) is the translation of an algorithmic design specification into an interconnection of combinational logic (functional units) and registers - the Register Transfer Level (RTL). The actual logic descriptions at the RTL are specified in a generic manner - perhaps as boolean equations or as generic gates. It is the task of logic synthesis to map these descriptions onto a specific structure suitable for the target architecture. This process will be referred to as *low-level* synthesis throughout this thesis. This is to distinguish it from *high-level* synthesis which is the form of synthesis that will be focused on.

High-level Synthesis (HLS), also known as architectural or behavioural synthesis), allows a design to be specified functionally at the level of algorithms and then be converted using a set of constraints into the RTL's structural connection of functional units and registers. During this process both the temporal and spatial locations of each function must be determined, i.e. during which clock cycles will a function occur and what physical functional unit will it be bound to. Gajski *et al.*[5] describe these two tasks as scheduling and allocation.

2.2 Hardware Description Languages (HDLs)

Specific languages for describing hardware have become a widely used tool for digital design, both as modeling languages and as inputs to synthesis tools. VHDL is an example of a HDL that was initially used to specify and simulate hardware but is now commonly used as an input to synthesis tools [7]. HDLs can suffer from their expressiveness with the same behavioural design written in different ways or even the same way across different HLS tools producing different results. Because of this, advice such as "think hardware" [8] is given to designers who wish to produce easily synthesizable VHDL or Verilog. This problem, as well as the difficulties in synthesis from languages designed initially for simulation, has led to attempts to define standard synthesizable subsets for HDLs. Both VHDL [9] and Verilog [10] have IEEE working groups working to produce such standards.

2.3 Programming Language Based Synthesis

Programming languages for software development are used to allow designers to operate at a level of a greater abstraction than the specific machine instructions of the CPU being targeted. Scott [11] identifies two major types of programming language: declarative and imperative. Declarative languages allow the designer to focus on what the program should do rather than how it should be done and examples include functional programming languages such as *Lisp* and *Haskell*. Alternatively, imperative languages are those for which the method, i.e. the algorithm, is expressed and these include what this thesis calls "traditional" programming languages such as *Pascal*, *Fortran* and *C*. Declarative languages will not be considered in this thesis and so the term "programming language" will refer exclusively to imperative languages.

The automatic generation of hardware from a traditional programming language or a high-level language (HLL), sometimes called "Hardware Compilation", is not a new idea but has recently attracted greater interest. Wirth [12] reasons that this is due to the improvement in semiconductor technology, particularly the availability of rapidly reconfigurable hardware devices such as field programmable gate arrays (FPGAs). Various fundamental difficulties with Hardware Compilation have been identified [13]. These include the lack of support in programming languages for describing the inherent concurrency and reactivity of hardware. Several research projects and some commercial products have

been developed in the area of hardware compilation and have attempted to address these problems. Some key ones are described in section 2.6.

2.4 Design Representations in Synthesis Systems

A key issue in all Computer Aided Design (CAD) is how the design is represented internally by the system. The decision on how this is to done can affect the performance of the tool and the complexity of algorithms that manipulate the design. In the field of HLS data structures based on variations on "graphs" dominate because of their inherent flexibility, for example they can be used to describe both behavioural and structural descriptions. Preiss [14] defines a graph as a set of points (vertices) with a set of lines (edges) connecting various points. Graphs can be directed - edges have a specific direction, from one vertex to another - or undirected - edges have no inherent direction.

Various examples of data structures for HLS have been proposed. Camposano and Tabet [15] present an example of how four separate directed graphs: control-flow graphs (CFGs), data-flow graphs (DFGs), control-automation graphs (CAG) and datapath graphs (DPGs), can be used in a behavioural VHDL synthesis engine. The behaviour of the design is extracted from the source to form a CFG and a DFG. HLS is performed on this representation refining the CFG to a CAG (a representation of a finite-state machine) and the DFG to a DPG (a structural datapath). This is an example of a "disjoint" control and data-flow representation. Gajski *et al.* [5] contrast this to a "hybrid" control and data-flow representation in which all the control and data information is embedded into a single graph. A more recent example of disjoint CFGs and DFGs is given by Bergamaschi [16]. This example is presented as an approach to the problem of estimating final hardware costs from the design representation used in HLS. It does this by annotating the CFG and DFG during HLS in a way that implies the final hardware precisely enough to provide good cost estimations.

2.5 Efficiency and Optimization

The same language "expressiveness" problem encountered when synthesizing VHDL can apply to synthesis from a programming language. For example, the reference manual for Celoxica's Handel-C [17] (a commercial tool) describes ways to adjust programming style

to increase the efficiency of the produced output. This is a similar problem to that of optimization in software compilation and to a certain extent can be improved by employing the same source level optimization techniques used in these compilers.

Furthermore, techniques from computer architecture such as pipelining [18] or other forms of parallel or overlapping execution are relevant. When compiling software for advanced architectures instructions can be reordered code to exploit instruction-level parallelism but data and control dependencies must be preserved using well known techniques [19, 20]. In the case of hardware compilation the same principles apply with operations being rescheduled to provide greater parallelism while maintaining the integrity of the algorithm. Weinhardt and Luk [21] concluded that the automatic synthesis of pipelines in a HLS system provided significant speedup over source-level optimizations alone.

2.6 Summary of Previous Work

Several previous approaches to synthesis from a traditional programming language are outlined below. A summary comparing features of these approaches appears in Table 2.1

2.6.1 Synopsys' Cocentric SystemC Compiler (Commercial)

SystemC [22] has been designed as a C++ based modeling language for hardware allowing for different levels of abstraction: from the structural, register-transfer level (RTL) up to the system design level. Synopsys has released a behavioural synthesis tool, Cocentric SystemC Compiler [23], to translate SystemC models into gate-level netlists or structural VHDL or Verilog descriptions. It is a high-level synthesis system, performing scheduling and allocation as well as providing support for automatic use of pipelined operations.

2.6.2 Handel-C (Commercial)

While SystemC is essentially a hardware description language, Celoxica's Handel-C [24] (the core of the DK1 Design Suite) was designed as a programming language that can be used to generate hardware. Handel-C is a variant of the C language with added syntax to support explicit parallelism as well as other features like bit manipulation. Full control over scheduling is given to the user by the implementation of timing semantics - each

assignment statement takes one clock-cycle [17]. As such Handel-C's compiler is not a full HLS tool. Handel-C is also designed specifically to target FPGAs.

2.6.3 Bach (Research)

Bach [25] is a research project by Sharp that is similar to Handel-C in that it aims to "feel" like a programming language rather than a hardware description language. Bach's difference is that it does not require the user to specify scheduling, but rather performs HLS on the input description. It does, however, allow for the specification of parallel code sections as well as communications between them.

2.6.4 SPARK (Research)

SPARK [26] is a project at the University of California, Irvine to develop an improved HLS tool using optimization techniques developed for parallelizing compilers. It takes behavioural ANSI-C as input and produces a structural (RTL) VHDL description as output.

2.6.5 SpC (Research)

SpC [27] (Synthesis of pointers in C) is a tool produced for a research project dealing with the issue of synthesizing the behaviour of pointers and dynamic allocation/de-allocation of memory in hardware. As such it supports arbitrary C data structures. The tool does not perform HLS itself but rather produces a behavioural Verilog description that can be synthesized using commercial tools.

2.6.6 Summary of Previous Work

Table 2.1: Comparison of Previous Work

Name	Input	Synthesis	Target
SystemC (Synopsys' Cocentric Compiler)	Modeling focus using C++ syntax/semantics. Behavioural model is essentially "programming", however.	HLS performed.	RTL (but tools included for FPGA, VLSI)
Handel-C (Celoxica)	Programming focus - C variant with explicit constructs for parallel execution. Supports arrays, pointers (but not dynamic allocation). RAM must be specifically targeted.	No Scheduling. Timing built into language - user determined scheduling.	FPGA (ready for place and route)
Bach	Programming focus - C variant. Explicit constructs for parallel execution.	HLS performed. Timing restrictions can be specified by user.	RTL VHDL
SPARK	Programming focus - ANSI C.	HLS performed.	RTL VHDL
SpC	Programming - C based. Includes dynamic memory allocation, pointers, and arbitrary C data structures.	No HLS performed (left in a behavioural form).	Verilog (Behavioural)

2.7 Summary

This chapter has introduced the theory relevant to this study as well as a summary of some of the previous work in this area. The background to CAD tools in digital electronics was presented, particularly the concept of high-level synthesis. The theory of the use of formal languages, both hardware description languages and more traditional programming languages, to design hardware was given as well as some of the issues surrounding the design of CAD tools that synthesize designs written in this form. Some aspects of design efficiency and optimization were also touched upon. Various hardware compilers, both commercial and research projects, were cited and the differences between them explained.

The next chapter defines the scope of this thesis, refining the broad statement of goals given in the introduction into a specific set of objectives.

Chapter 3

Problem Definition and Scope

The overall aim of this study was to develop a CAD system capable of producing a hardware structure from a design input given in a high-level programming language. This design flow was to be written with extensibility in mind and as such it was also a requirement that the software be developed in a way that would allow it to be improved through future work. The major practical restriction that this put on the software was that it had to be written using non-proprietary libraries. It was necessary to further refine the specified requirements of the CAD software before commencing the design process. The following sections present an overview of the chosen scope in terms of the inputs and outputs of the system.

3.1 Design Input

As there are a vast array of traditional programming languages there were many possible choices for an input language to the CAD system. The C programming language has become extremely popular for software development and could be considered the de-facto standard for embedded software. As such it is a good choice for a programming language to design hardware. Furthermore, much of the related work described in the previous chapter was based on C or C-like languages. Other possible languages include C++ and Java. Both of these are popular among software developers and allow for higher levels of abstraction in design. The downside of these languages is the greater complexity required in both compilation and (particularly for Java) runtime environments. This complexity could distract from the ultimate purpose of the study. From the considerations above it was decided that a C-like language would be the input choice for the CAD system.

A decision was also needed on the modifications to standard (ANSI) C that would be made for the input language. This includes both restrictions placed on C constructs as well as the addition or modification of constructs to allow for hardware specific features. Some existing tools, for example Handel-C [17] provide specific constructs for specifying portions of code to be run in parallel. Some aspects of C that can create difficulty for hardware synthesis are pointers and dynamic memory allocation. These problems are explained and addressed in [27].

To allow this study to be feasible it was decided that only a restricted subset of C would be allowed. In particular, pointers, dynamic memory allocation, structures, floating point data types and recursive functions were disallowed. Additional constructs were sparingly added to this specification as the aim was to allow the user to "program" hardware as if it were software. It was decided that any width of integer would be allowed to be declared (rather than fixed values such as 8,16 and 32) as this flexibility is desirable in hardware designs and easier to achieve than in software.

To make a design useful it must be able to communicate with the external world. When programming a microcontroller this is possible through a fixed set of I/O ports but when designing hardware it is desirable to have greater control over inputs and outputs. In order to allow this it was decided that the input language should have at least have a mechanism to declare variables as inputs and outputs. The keyword *in* should specify an unlatched input from the external world which behaves as a constant variable (the user is unable to write to it). The keyword *out* should specify an internally stored output - i.e. the same as a regular variable except that the output is provided externally.

3.2 Design output

As stated, the overall goal of the CAD tool was to produce a structural design derived from the behavioural input. As it was not a goal to optimize synthesis for a particular target technology it was decided to output at the *register transfer level* (RTL). At this level a hardware design is represented as a structural *datapath* with an associated *controller*.

Synchronous hardware output was chosen over asynchronous as it allows the design to be simpler and easier to simulate and debug. It also allows targeting of FPGAs where the use of asynchronous designs is problematic.

3.2.1 Datapath

The *datapath* is the interconnection of functional units (FUs), storage units (SUs) and multiplexers (MUXs) that describe the flow of data through the system. Examples of functional units include adders, multipliers and comparators - that is, they are the parts of the hardware that perform the calculations and comparisons. A single FU may perform multiple types of operations. For example, an adder may double as a subtracter. To enable this it was decided to give some FUs *control vectors* used to specify what specific operation is to be performed. In order to simplify the system only single-cycle FUs were considered.

Storage units allow data to be stored across multiple clock cycles. In the scope of this study the storage unit used is the *register*. A *register* is designed to store n-bit data on its input on the positive edge of the clock whenever its latch-enable input is asserted.

The third type of structure on the Datapath are Multiplexers. Multiplexers allow FUs and SUs to be shared by channeling the correct input based on a *select* signal. This allows an adder, for example, to be used for different calculations in different clock cycles.

3.2.2 Controller

The MUXs' *select*, the registers' *latch-enable* and the FUs' *control vectors* signals must be provided by the *controller*. In algorithmic terms the controller is the device that keeps track of which section of the program is currently being executed. At the RTL level a *controller* is generally represented as a finite-state machine (FSM) with state transitions dependent on values from the data path. While this FSM will eventually have to be translated into flip-flops and combinational logic this is generally not the RTL representation and therefore is generally not a concern for high-level synthesis but rather for a specialized process known as controller synthesis. Thus it was decided that the controller output of the design would be a FSM description of a controller rather than a connection of flip-flops and logic.

3.2.3 Output Format

One of the major goals of the design was to allow for a complete design-flow from programming language to an FPGA or VLSI layout. As such, the output of the CAD tool

needed to be in a form that could be used by lower-level CAD tools that would perform the logic synthesis as well as placement and routing.

One possible output format was the Electronic Design Interchange Format (EDIF). This format is used to exchange design data between CAD systems [28]. While it is possible to import EDIF netlists into a lower-level synthesis tool (such as Xilinx) there are serious disadvantages to this approach. Since an EDIF netlist describes purely structural connections it would not be possible to simply output the controller in the form of a FSM as described above. Furthermore, EDIF is not designed to be human readable so it would be difficult to debug such a system.

For these reasons it was decided that a subset of a hardware description language (HDL) would be used. It was necessary to restrict use of the HDL to a subset that was synthesizable to hardware. VHDL was chosen as a variety of simultaion and lower-level synthesis tools supporting it (Xilinx, Leonardo Spectrum, Active HDL) were available. This allowed both simulation of programs and synthesis to FPGA boards to be tested. The other restrictions placed on the output were that it must be an RTL description - an unambiguous datapath and controller - and it must have explicit registers and functional units, i.e. by using components, rather than implied ones.

3.3 Summary

This chapter clarified the scope of the problem addressed in this study. In particular it detailed the specifications for the design of the CAD tool to be described in the following chapters. It was determined that the design would need to:

- accept a design input in a variant of the C programming language;
- output a design consisting of a structural interconnection of function units, storage units and multiplexers (datapath) and a finite-state machine description of a controller;
- produce output in a subset of the VHDL language that is synthesizable by lower-level tools.;
- be able to produce a variety of output structures for a single behavioural input;
- be extensible to allow the addition of new optimization or synthesis techniques; and

- be developed using open or non-propriety libraries to allow the end product to be open source.

The Design and Implementation of the CAD tool is presented in the following chapter. Chapter 6 then discusses the end product and evaluates it against these specifications.

Chapter 4

Design and Implementation

The previous chapter outlined the required specification for the developed CAD tool in terms of the objectives of the study. It also detailed the specific inputs and outputs for the software. This chapter presents the design and implementation of *Circe*, the CAD tool developed according to these specifications. First an outline of the development environment and the overall approach to the problem is given. Following this is a more detailed explanation of the implementation of each of the major components.

4.1 Selection of Environment

Before development of the software could commence a programming language and operating system (or systems) needed to be selected. The languages considered (because of the author's familiarity with them) were C, C++ and Java. Both C++ and Java provide support for extensive abstraction of data structures together with libraries of commonly used data structures such as vectors, linked-lists, and maps whereas C does not. It was anticipated that these features would be necessary in order to meet the project objectives within the set time-frame and therefore C was rejected as a development language. Java had the advantage of providing extensive support for system portability but the disadvantage of non-native executables that are generally slower than C++ equivalents. Another advantage of C++ was its popularity among open-source developers which has led to extensive tool and library support including the GNU C++ Compiler (g++), the standard C++ library (formally the Standard Template Library) and open-source versions of compiler writing tools Yacc (Bison) and Lex (Flex)[1]. This support as well as the author's

[1] Although these programs technically produce C output it is possible to integrate this into a C++ program

previous experience with C++ and with the Yacc program led to the final selection of C++ as the development language.

The GNU C/C++ Compiler is available across a variety of operating systems including most UNIX environments and Microsoft Windows®. Therefore using this compiler allows for portability across these operating systems. This is beneficial as some lower-level CAD tools are available in UNIX and others in Windows, although the FPGA synthesis tools, such as Xilinx, are generally Windows based. It was decided that development would be undertaken in a UNIX environment due to the better default support for C++ tools. In particular the undergraduate UNIX (Solaris) machines available had support for g++, the standard C++ library, Bison and Flex already available. The author also had access to a Linux box with the same development environment and so development was shared across Linux and Solaris (System V) machines.

4.2 Overall System Design

The overall software system had to accept input written in the specified C-like language, perform synthesis on the algorithmic description and produce a structural VHDL output in the form specified. Given this, it was decided to divide the design into three major components: the Parser[2] (or front-end), the Synthesis Engine (or middle-system) and the Backend. These elements are illustrated in Figure 4.1. Their roles are described briefly below with details of their design and implementation in the following sections.

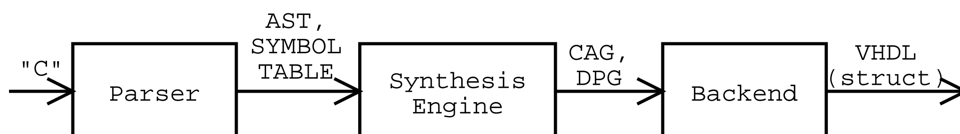


Figure 4.1: Major components of the system

The Parser's role is essentially to perform the role of the scanner, parser and semantic analyzer in a software compiler as described by many compiler references including [29]. The tasks of this component are:

1. Perform lexical analysis on the source file to translate raw characters into input tokens

[2]The word "Parser" (uppercase P) is used to describe a collection of components the major of which is the actual parser (lowercase p).

2. Perform syntax analysis on the input tokens to produce an Abstract Syntax Tree (AST)
3. Perform semantic analysis on the input, e.g. check that all variables that are used have been previously defined, and produce a Symbol Table

The role of the Synthesis engine is to map the algorithmic description described by the AST and Symbol Table to a form that can be converted to a hardware description by the Backend. The essential tasks of this component are:

1. Translation of the AST into an Intermediate Representation (IR)
2. Scheduling of operations to clock cycles to produce a Schedule Table
3. Binding and Allocation of operations and variables to functional units and storage elements to produce Allocation Tables

The role of the Backend is to take the IR representation along with associated Schedule and Allocation Tables from the Synthesis engine and produce structural hardware output. In this case the output is in the form of VHDL as described in the previous chapter. More specifically the tasks of the Backend are:

1. Determine MUX requirements for the datapath
2. Determine control signal values for different control states
3. Output VHDL descriptions of the datapath and controller

4.3 The Parser

The Parser's main role is to convert the human readable source code of a program into a representation that can be processed more readily by the synthesis engine. In order to achieve this two major data structures were selected: the Abstract Syntax Tree and the Symbol Table. These data structures were implemented using C++ classes. The Flex and Bison UNIX tools were used to generate code for scanning and parsing the textual input.

4.3.1 The Abstract Syntax Tree (AST)

Abstract Syntax Trees, or ASTs, are used in software compilers to store the syntactical details of a program in an analyzable and manipulatable form. For example an *if* statement may be described by a node with three branches: one for the condition expression, one for the statement to execute if true and one for the statement to execute if false. This is illustrated in Figure 4.2.

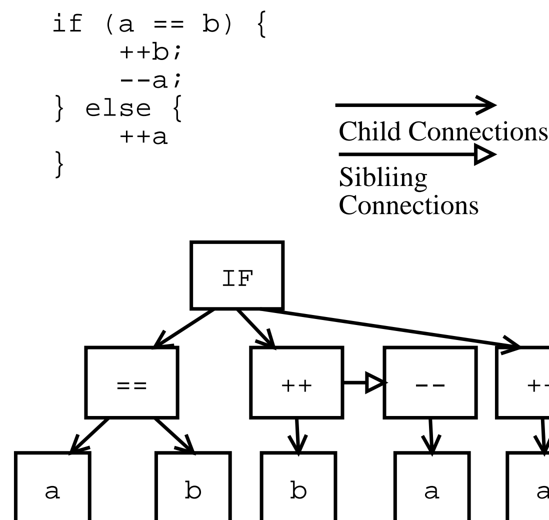


Figure 4.2: An example of a AST for an *if* statement

For this system a simple AST design was used consisting of nodes that can have children and sibling nodes attached to them. It is important to note that two "child" nodes of the same node are not in fact "sibling" nodes of each other in this model despite the use of those terms. Child nodes were used for representation of algorithmic hierarchy such as the *if* statement described above, whereas a sibling node is the next node in a list such as a statement list or a list of arguments to a function call. Both child and sibling nodes are illustrated in figure 4.2.

The AST nodes were implemented as a C++ class rather than a structure in order to allow the internal construction to be hidden behind an abstract interface. This was to allow changes to the internals without having to rewrite the rest of the Parser. A single node class with an enumerated type to identify node types was used rather than subclasses for each node specialization. This was done to avoid over-complicating the design with a large number of class definitions.

4.3.2 The Symbol Table

The Symbol Table is used to hold information on the variables and functions defined by the input program. This information is potentially required at all stages of the system but is gathered by the Parser. The table itself was implemented as a C++ *map* of pointers to Identifier Information objects with a *string* as the *map*'s key. Each Identifier object holds the relevant information on a symbol: its type (function or variable), its precision value (how many bits precision and whether signed or unsigned) and a textual representation of the symbol (for error and warning messages and for debugging purposes). Identifier objects were sub-classed into “function” and “variable identifier” objects in order to allow for specific storage and interface requirements of each type. For example, “function” objects store a list of calling parameters and “variable identifier” objects store other information such as whether a variable is reference or not.

4.3.3 Scanning and Parsing

Scanning and parsing are both well defined and explored areas in Computer Science. Scanners perform lexical analysis on a string of characters, breaking them into *tokens* with each token being a logic element of the source such as a keyword or variable name. This is generally done by deriving *regular expressions* for the possible input tokens and developing a lexical engine based around a finite-state machine[3] that is either derived by hand or more often, automatically. Flex (an open source version of Lex) can automatically create a C source implementation of a scanner based on a supplied set of regular expressions. As such, it was decided that Flex would be used to create the scanner for this system. An added advantage of this approach was that there was a publicly available Lex specification file [30] developed for the ANSI C language. This file was edited to produce the Lex specification for the C-variant language detailed in the previous chapter. Some of the major changes to C specification included removal of keywords (mainly data types like *char*) and the abolishment of floating point and string constants.

As described above the parser accepts a program written in a language described by a formal *grammar* and builds an AST representation of the program. There are various approaches to writing parsers, however a common one used for programming languages is the *LALR* parser which can be used to parse most grammatical constructs found in standard programming languages [29, 31]. The UNIX tool Bison (an open source version of

[3]In this context a fi nite-state machine is often referred to as a deterministic fi nite automation as in [29]

Yacc) can be used to automatically generate *LALR* parsers based on a formal grammatical description of the language. Semantic actions can be embedded inside this grammatical description to allow an AST to be built while parsing. This was the method used to generate the parser for this system. A pre-existing Yacc description for the ANSI C language was obtained [30] and used as the basis for the parser. This description had to be substantially rewritten, however, due to the differences between ANSI C and the chosen input language. Furthermore, all of the semantic actions for building the AST and maintaining the Symbol Table had to be written from scratch.

4.3.4 Semantic Analysis

For C-like programming languages, Semantic Analysis consists primarily of building the Symbol Table and type-checking the expressions in the program [29]. It is sometimes possible to implement all of this task to occur during the first source-file pass, particularly if no complicated analysis is needed. The other option is to perform multiple passes over the source. For this system, the first approach was possible as there is little semantic analysis necessary and as such it was considered simpler just to implement semantic checks where needed in the Yacc file. If this system were to be improved to include more sophisticated processing of the parse tree, for example performing source-level optimizations, then a second pass may be required. The design of the AST data structure would allow for additional parses simply by "walking" the tree.

4.4 The Synthesis Engine

The Synthesis engine is at the heart of the CAD system providing the bridge between the behavioural design representation and the structural one. It is responsible for determining how the operations, variables and control structures of the programming language are mapped to the functional units, registers and control logic of the hardware description. To achieve these goals the Synthesis Engine was designed around a flexible yet relatively simple set of data structures - the Intermediate Representation. Scheduling - the assignment of operations to control steps - and Allocation - the mapping of operations and variables to functional and storage units - were achieved in this design through manipulation of the Intermediate Representation.

4.4.1 Intermediate Representation (IR) Overview

The intermediate design representation had to be both flexible and simple while retaining the control and data dependencies of the original behavioural description. As high-level synthesis has been relatively well studied there were plenty of ideas to drawn on for this part of the design. Variations on data-flow graph (DFG) and control-flow graph (CFG) structures that can be used for this purpose are presented in the literature, including a good explanation of some basic structures in [15].

Before determining the Intermediate Representation that would be best for this system a series of design decisions had to be made:

- Whether to use a combined control-flow/data-flow structure or separate structures;
- Whether the final hardware design would be represented by an annotated version of the intermediate representation or by completely separate data structures; and
- Whether the intermediate representation should be hierarchical (matching the hierarchical structure of the Abstract Syntax Tree) or flat.

These decisions were made keeping in mind the main objectives of the study: to produce a working prototype within a set time-frame and to allow for a range of design outputs from a single behavioural input. The first of these objectives tends to favour a simpler, perhaps more rigid, synthesis system while the second creates a need for some flexibility in the Synthesis Engine. The final decisions made came as a result of finding a balance between these two almost competing objectives.

It was decided that in order to produce a relatively simple translation to hardware, the intermediate representation would be based around two distinct (but linked) structures: a Control-flow Graph (CFG) and a Data-flow Graph (DFG) that would be able to map onto the Controller and Datapath of the target architecture. There would be a separate CFG/DFG pair for each function declared in the design. These structures were designed so that a hardware structure could be extracted from them with minimal change to their initial form but that additional information (annotations) could be added and manipulation performed if a more sophisticated HLS routine was desired. This concept of a representation that can be manipulated and augmented but that, from the start is directly translatable to hardware was inspired by the structures described in [16], in particular the CFG to FSM technique. It was realized, however, in a different form (and for a different purpose) in

this project. It was this idea that allowed simplicity and flexibility to co-exist in the design so that both of the major objectives could be met.

With these fundamental decisions in place the details of the CFGs and DFGs, the translations to them form the AST and the performance of HLS on them could be developed. The following discussions make use of graph terminology from Computer Science that are defined in most introductory data-structure texts including [14]. For the implementation of all graph structures a C++ template library (The Graph Template Library - GTL [32]) was used. This library allows the creation of graph structures with an arbitrary number of named attributes for both the vertices and edges.

4.4.2 Control-flow Graph (CFG) Implementation

The CFGs developed for the Synthesis Engine represent all the conditional behaviour of the original program in terms of an interconnection of three basic vertex (node) types: *operations*, *forks* and *joins*. These three types were chosen as they are flexible enough to model any control structure but simple enough to allow for analysis and manipulation of the CFG.

Operation vertices represent the execution of one operation (such as an addition, comparison or a variable assignment) and their position in the CFG represent the logical order in which the various operations of the algorithm should be executed. In reality it would be expected that more than one operation would be executed in a single clock cycle and so an attribute called “fall-through” was added to each vertex of the CFG. This annotation is used to indicate whether or not two consecutive operations in the CFG have been scheduled to the same FSM control state. Each operation vertex in the CFG is also linked to a corresponding vertex representing the operation in the DFG. Operation vertices are restricted to having only a single incoming edge and a single outgoing edge. This was done to avoid complications if, for example, the HLS routines want to adjust the order of the operation nodes.

Fork vertices were created to allow for conditional branching in the control as is required, for example, by *if* and *while* statements. No operation is linked to a Fork vertex but instead a single output from an operation node in the DFG is associated with it. This output determines whether control will move to the left (output is zero) or right (output is non-zero) of the fork. As no operation is associated with a fork vertex it can always exist in the same control state as the preceding operation (or Join) vertex. Therefore there is an implicit “fall-through” on all vertices in the CFG that proceed a fork vertex. It is also

possible to link fork nodes together to allow for more complicated control branching to occur.

Join vertices were required to allow two separate control paths to recombine while maintaining the single incoming edge rule for operation and fork vertices. As no operation is associated with a Join vertex there is an implicit “fall-through” from a Join to the following vertex. Figure illustrates a CFG resulting from an if statement and shows the places were implciit “fall-throughs” exist.

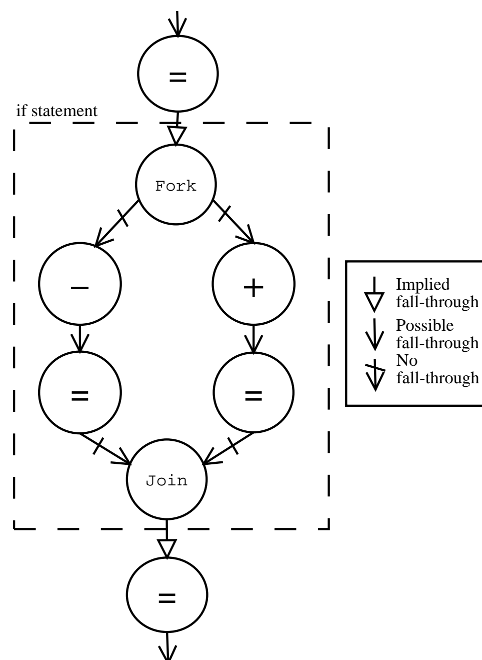


Figure 4.3: A example CFG fragment resulting from an if statement

4.4.3 Data-flow Graph (DFG) Implementation

The DFGs developed for the Synthesis Engine represent the flow of data in the program in terms of an interconnection of vertices representing operations and variables. The types of vertices that were used were *operations*, *constants, variables* and *assignments*.

Operation vertices were used to describe the flow of data to and from operations such as additions, multiplications and comparisons. Each operation vertex takes one or more inputs (incoming edges) and produce an output (outgoing edge). Each operation is linked to a corresponding vertex in the CFG, used to determine the control state the operation executes in. There is also a binding attribute associated with operations that provides a

reference to the actual functional unit (FU) that will be used to implement the operation. The allocation part of HLS determines the value of this attribute. Each operation may also require binding to a storage unit (a register) depending on which control state its outputs are to be used. Another attribute is used for this and it is also set during HLS.

Assignment vertices represent an assignment operation. Like operation vertices they are associated with a particular CFG vertex that determines what control state the assignment occurs in. Each assignment vertex has a single incoming edge representing the data to assign and a single outgoing edge that is connected to the variable vertex that the data is being assigned to.

Variable vertices are used to represent variables in the program. Incoming edges to variable vertices must come from assignment vertices. Outgoing edges represent reads on the variable.

Constant vertices represent constant data values. The output of these vertices are a fixed constant, that is no operation executes, and therefore they do not need a link to the CFG. There are also no incoming edges on a constant vertex. Figure 4.4 illustrates an example DFG.

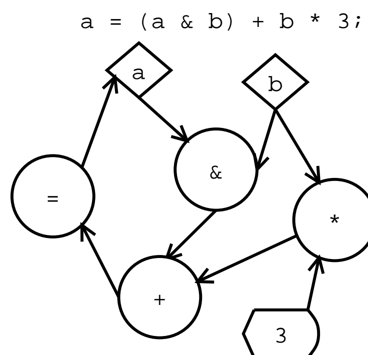


Figure 4.4: A example DFG fragment resulting from an assignment statement

4.4.4 Translation to the IR

A systematic process was developed to translate the hierarchical program flow found in the Abstract Syntax Tree (AST) into the flattened control-flow (CFG) and data-flow (DFG) required for the IR. The overall method was to make each AST node translate itself into CFG/DFG components, call its children to do the same and then link these

components up before passing them all back up the tree. In this way the entire DFG and CFG structures may be built up. The following paragraphs give a brief overview on the translation of different constructs.

Each expression node in the AST (corresponding to a unary, binary or ternary expression) is translated directly into an operation vertex on the DFG and a corresponding vertex on the CFG. The AST node's children are then processed with the outputs from their DFG vertices becoming the inputs to parent node's DFG vertex. In this way complicated expressions can be translated. Assignment AST nodes each become an assignment vertex on the DFG, linked to an operation vertex on the CFG. Reads from variables are translated into outgoing edges on the DFG's variable vertices while constants in the original program are translated directly into constant vertices in the DFG. Neither of these two constructs affects the CFG.

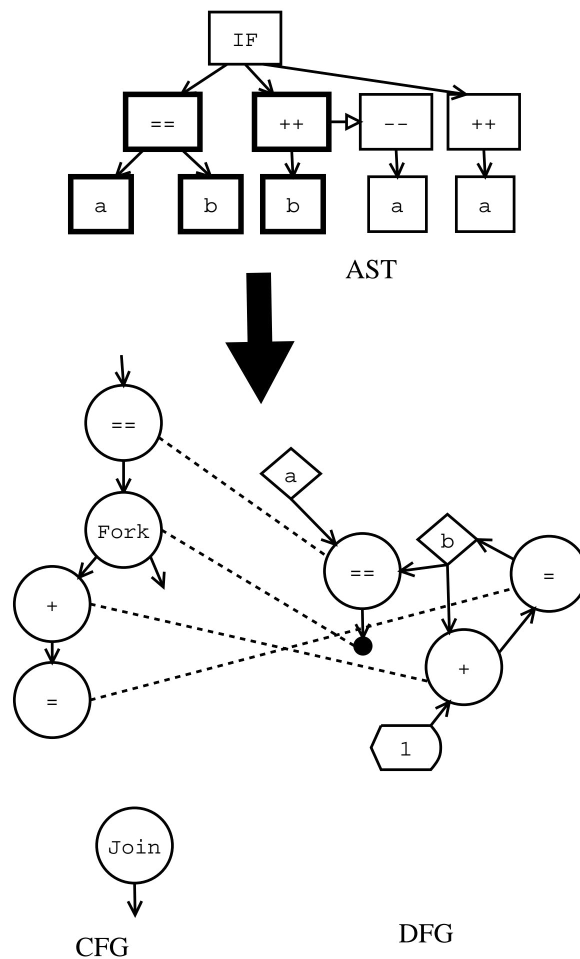


Figure 4.5: Translation to the IR (Partially completed)

The remaining constructs of the programming language involve control of the program flow and so translate to more complicated CFG structures while not affecting the DFG directly. For example, an *if* statement becomes a fork/join pair in the CFG with the two alternate control paths between them. A *while* statement becomes a join vertex followed by any operations needed for the condition statement and then a fork determining whether control should enter (or re-enter) the body of the while loop or exit it.

Figure 4.5 illustrates a the generation of parts of a CFG and DFG from the AST. In the figure the translation is partially completed (nodes of the AST tree that are highlighted have been translated) and the links between the CFG and DFG are shown with dotted lines.

4.4.5 High-level Synthesis

The process of High-level Synthesis (HLS) determines the precise timing for each operation as well as the actual functional and structural units that operations and variables will be bound to. The design chosen for the intermediate representation allowed for quite a range of alternative HLS approaches. Only a few simple approaches to HLS were implemented in this study as a proof of the flexibility of the design. The aim was to demonstrate the possible range of structural options possible with the system rather than to provide the best HLS system possible. More sophisticated HLS options that could be implemented are considered in Chapter 6.

The simplest scheduling method possible in this system is to not set any of the CFG's "fall-through" attributes, therefore giving a separate control state to each operation. While this *trivial* scheduling will create more states than necessary for most designs, it is still useful for the first tests of the system as a whole. A slight improvement was made to this system to schedule assignments to occur in the same control state as the operation that precedes them. This prevents unnecessary registers on the outputs of operations that are to be assigned directly to variables.

A second relatively simple scheduling method was added to demonstrate that different schedules of the same design were possible. As such this second method was designed to produce a significantly different type of schedule to the first. The method used was to schedule all operations that form a linear chain in the CFG to occur in the same control state and thus can be described as *minimal cycle*[4] scheduling. That is, chains of operations

[4]This is somewhat of a misnomer as a design with less states could be produced with *manipulation* rather than just annotation of the graphs.

not interrupted by forks or joins all execute in a single clock cycle. This was achieved by simply setting the "fall-through" attribute on all operation vertices. Simple command-line options were implemented to allow the desired scheduling routine to be selected.

Binding and allocation are the processes in HLS that determine what functional and storage units will be required and how specific operations will be mapped onto these units. In the implemented HLS routines in this system these processes were run on a design that had already been scheduled. The one binding and allocation routine implemented was designed to be compatible with both the scheduling algorithms as well as generally any other independent[5] scheduling algorithm.

Binding was accomplished through a lookup table associating operations with functional units (FUs). For example, multiplication may map onto a specific multiplication unit while addition and subtractions may map on to a generic "adder" unit. As well as by their generic types, FUs were also classified on the basis of their *order*. An FU's order was defined as the predominant bus width of the FU. Four-bit numbers would not require a 16-bit adder, for instance.

It was determined that for the purposes of simplicity the allocation algorithm would act on a previously scheduled design rather than work in conjunction with the scheduler. While this did not produce a particularly sophisticated allocation strategy it was deemed sufficient for the purposes of this study. The first constraint that had to be obeyed by the allocation algorithm was that each FU and register in the design could be used at most once per control state. The second constraint was that for each particular type of FU there is a limit to the number of multiplexed inputs that are feasible before the MUX cost outweighs the cost of simply adding another FU. An extreme example is that of an AND gate for which it makes no sense to multiplex. A simple "greedy" algorithm was used for allocation. This algorithm allocated the first unit found that was both available in that control state and had less than the maximum number of allowed multiplexed inputs. It created a new instance of that unit if necessary.

4.5 The Backend

The final stage of the system had to convert the IR with HLS annotations into VHDL output of the form described in the previous chapter. The IR, once scheduled and allocated, can be considered a structural design as the datapath and controller have been determined.

[5]More advanced HLS would have a greater interdependence between scheduling and allocation/binding.

The terms Control Automation Graph (CAG) and Data Path Graph (DPG) will be used to describe the CFG and DFG respectively once HLS information has been added. It was decided that separate C++ classes would be used to represent the CAG and DPG in the system. This allowed the Backend to be kept separate from the other parts of the system so that modifications could be made to one part with minimal effect on the other. One particular consideration was that it should be possible to rewrite the Backend to target another design representation format without having to change the Synthesis Engine. The following sections describe various aspects of the Backend's implementation.

4.5.1 Functional Units and Registers

For a design described at the register-transfer level, the building blocks are the functional units and registers. The Synthesis Engine described above determines the number and type of each of these units and how they will connect together in the datapath. As such, these elements are the primitives of any design produced by the system in the same way that the machine code instructions of a CPU are the primitives for a software compiler. Therefore, a standard "library" containing these primates was necessary for this synthesis system. The only requirement for the development of these components was that they must be able to possess a VHDL *entity* interface to allow them to be instigated in the datapath description.

It was decided that synthesizable VHDL descriptions would be used to implement this library. The only restriction placed on these descriptions were that they must be synthesizable by the lower-level synthesis packages used to map the design onto specific targets. Writing libraries specific to particular synthesis packages or technology targets was also a possibility but it was decided that this would be left as a possible future improvement to the system.

Another problem that was addressed was that of how to represent the arbitrary bit-widths (called *order*) of FUs and Registers that were possible. One alternative was to create a separate description for each possible *order* of each component type, for example separately specify "reg1", "reg2", "reg64" as registers with bit-widths of 1, 2 and 64 respectively. The major disadvantage of this approach was that it would restrict the possible bit-widths to those specified in the library (thus implying a maximum bit-width). An improved version of this would be to generate specific component descriptions at runtime from a library of generic components. For example, a generic description of a register, "reg", could be produced with the specific details of signal widths substituted at runtime

to create "reg1", "reg2" or "reg64" as needed. This was essentially the approach that was taken but instead of performing the substitutions explicitly the VHDL *generic* keyword was used in the library descriptions. In this way the final substitution of signal widths is specified within the datapath description and occurs when the design is mapped by the lower-level synthesis tools.

4.5.2 Structuring the Datapath

The restrictions on the VHDL output, as outlined in the previous chapter, were that it must be an RTL description and that it must consist of explicit, rather than implied, components. It was decided that the best way to achieve this in the datapath was to use a structural VHDL description consisting of *component* instances for each FU and register and to link these with signals that are assigned concurrently. *Process* statements were not used as sequential behaviour was not needed to describe the interconnections making up the RTL datapath. Furthermore, registers were not implied with process statements in the datapath but rather by explicit register components. Signals were defined for each input and output of each component.

In order to retain as much information as possible it was decided to keep the structure of the original DFG present in the VHDL output. This was done by declaring a signal for each input and output for all nodes in the DFG and connecting these as per the DFG. Concurrent signal assignments are just "wires"[6] so this structure does not itself imply any hardware. The hardware is implied by the *component* statements, the signals of which are connected to the DFG-based structure again using signal assignments. Several subtleties had to be handled, however. The major ones being component sharing and maintaining algorithmic integrity.

Allocation ensures that each FU or register is only used one way in a *single* control state but in general, across all the control states, these components will need to be shared by different parts of the datapath. It was necessary therefore to multiplex the inputs of each component with a signal driven by the controller. The actual MUXs were implied by VHDL using the concurrent *with* construct. This construct is actually not-structural in the sense that it implies a MUX rather than explicitly connecting a MUX component. The reason for this decision was two-fold: it was found to be quite complex to create a generic MUX description allowing for a variable number of inputs; and it was decided

[6] A signal concurrently (and unconditionally) assigned to another is just an alias of the original.

that a MUX in an RTL description was better described as switching behaviour than as specific logic connections.

The DFG describes the abstract flow of data in the program and while the annotations added by the Synthesis Engine are sufficient to fully define the specific functional unit and register requirements and the mapping to these structural elements, the specific connections needed are only implied. Thus it is the responsibility of the Backend to ensure that connections are made at the correct places to ensure algorithmic integrity. An example of this problem is shown in Figure 4.6. As both assignments are scheduled to occur during the same clock cycle the value of “a” in the second assignment is *not* available from the “a” register at the time it is needed. It is necessary, therefore, to “short-circuit” the value from the expression that *will* be assigned to “a” to the multiplication. An optimization was also introduced that removed the register of any non-output variable that was never read from. This included variables that were “read” only in the same control state as they were assigned, i.e. where the register outputs were never actually used.

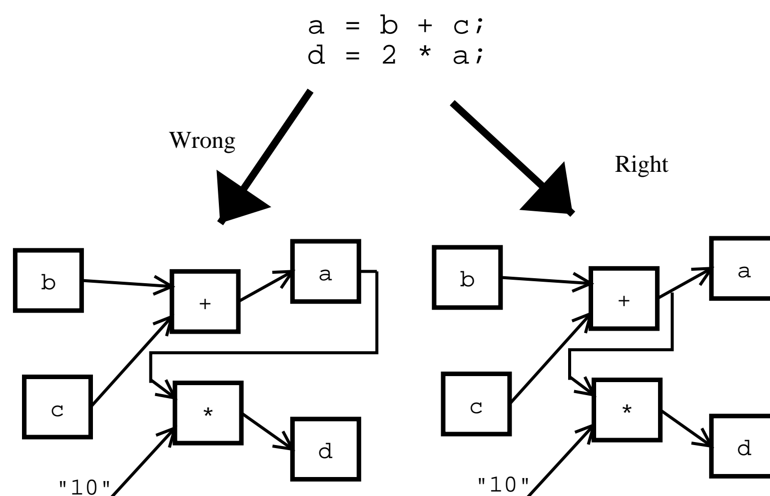


Figure 4.6: An example of maintaining algorithmic integrity

4.5.3 Structuring the Controller

The CAG produced by the Synthesis Engine describes the controller as a series of states with either a single transition out to a following state or a choice of transition based on a chain of one or more fork nodes. It the role of the Backend to output this structure as a VHDL controller description. As each state will uniquely define the MUX, latch-enable

and FU configuration signals required, the controller can be expressed as a Moore FSM. As explained in the previous chapter the output controller description was to be left as a FSM description rather than a description of the actual logic needed to implement it. To do this in VHDL it was decided that two *process* statements - one for combinational logic and one for implying transition on a positive clock edge - would be used. This is a common method of describing a FSM in VHDL and employs a “currentstate” implied register to hold the state which is used to determine the “nextstate” signal as well as the outputs of the FSM using purely combinational logic. The Combinational logic is described using *if* and *when* statements. This method of FSM description is detailed in several places including [33]. An example of a state machine described in this way is given in Listing 1.

To avoid implying unwanted latches in a VHDL controller, it is necessary to specify the values of every control signal for every possible state. In the process description used this can be done by either specifying a signal assignment for every control signal in every *when* clause (i.e. one for each state) or by specifying “default” assignments before the *case* statement and specifying overriding assignments in the relevant states. This second approach was chosen as it leads to more readable VHDL. The chosen default values were “0” for latch-enables and “X” (extended out to the number of X’s required) for MUX selects and FU configuration vectors.

One problem when implementing a program in hardware is that a program can have an ending whereas hardware essentially “runs” forever. To counter this problem there had to be a way for the hardware to become idle at the end of the program, that is, to display no visible changes. This was done by adding a separate *end* state when producing the controller. The final state in the program was linked to this end state which would loop back on itself continuously. By ensuring that all latch-enable signals were set to “0” (the default) during this final state there will be no changes to outputs after the end of the program.

4.6 Summary

This chapter has explained the design and implementation of Circe. The overall design as well as the details of the Parser, Synthesis Engine and the Backend were presented.

Chapter 5 presents an evaluation of this system comparing it against both the specfications and related work. Chapter 6 details possible future improvements to Circe.

Listing 1 A Moore FSM in VHDL

```
type state_type is (S1, S2, S3);
signal curState : state_type;
signal nextState : state_type;
signal controlSignal1, controlSignal2 : std_logic;
begin
   process (clk, res)
   begin
       if res = '0' then
           curState <= S1;
       elsif clk = '1'  and clk'event then
           curState <= nextState;
       end if ;
   end process ;
   process (curState, cond1, cond2)
   begin
       -- default values
       controlSignal1 <= 'X'; -- don't care
       controlSignal2 <= '0';
       case curState is
           when S1 =>
               if cond1 = '1' then
                   nextState <= S4;
               else
                   nextState <= S3;
               end if ;
               -- assign control signals here
               controlSignal1 <= '1';
           when S3 =>
               if cond2 = '1' then
                   nextState <= S1;
               else
                   nextState <= S4;
               end if ;
                -- assign control signals here
                controlSignal1 <= '0';
                controlSignal2 <= '1';
           when others =>
               nextState <= S1
           end case;
   end process;
```

Chapter 5

Design Evaluation

The previous chapter outlined the design and implementation of the CAD tool, Circe. This chapter presents an evaluation of Circe consisting of testing the system with a few example designs and discussing the findings. Circe is also compared to other hardware compilers and against the specifications outlined in Chapter 2. An evaluation of engineering performance is also given.

5.1 Testing

5.1.1 Methodology

Functionality testing was performed using the VHDL simulation features of Active HDL. This allowed the expected behaviour (as described in the input program) to be compared against the actual behaviour of output VHDL. It was necessary first to establish a clear *expected* functionality for each program. As there was no simulation facility implemented for the particular C-variant chosen this process was done by hand. The specific timings of the design are determined by the Synthesis Engine rather than being fully specified in the source. Thus the simulation waveforms were used to compare the specific operation timings produced by different HLS techniques. Both of the two implemented HLS algorithms (*trivial* and *minimal cycle*) were tested for each of the three designs.

As the output of the system is target-independent RTL VHDL code, testing the quality of the output consisted of using it as input to lower-level synthesis packages. *Xilinx Foundation Series* and *Leonardo Spectrum* were the two tools chosen for this task as they were

readily available tools, capable of mapping a RTL VHDL design onto a specific target. For the tests using Xilinx tools a 30k gate SpartanXL FPGA (S30XLTQ144-4) was chosen as the target device. This was done as a development board incorporating this FPGA was available and this allowed the ultimate test to be performed - implementation on a physical device. Leonardo Spectrum was used to view the RTL schematics for the designs and confirm that it would be possibile to target VLSI. The actual implementation to VLSI was not performed. Default synthesis options were used at all times (unless otherwise stated in the results). The reports from the synthesis tools were used to determine the area and timing characteristics of the designs.

5.1.2 Example Programs

The results of three example programs are presented below. The three examples were chosen to demonstrate: a simple program (a up/down counter); a data-intensive program (a transformation matrix multiplier); and a more complicated program with significant control structures (a noughts and crosses game).

The simple up/down counter program was designed to take two single bit inputs from the user: a user "clock" (not to be confused with the system clock) and a direction indicator - "0" for up and "1" for down. The output is a four-bit unsigned number indicating the current count. The counter stops counting up when the maximum count is reached and stops counting down when zero is reached. The code is given in Listing 2.

The transformation matrix program applies a 3x3 8-bit transformation matrix to a 3-dimensional vector. All the elements of the matrix are inputs as is the original vector. The transformed vector forms the three outputs. It was necessary to modify this program when testing with the Xilinx FPGA synthesis tools (see Listing 3) as the number of inputs and outputs was too high.

A more complicated program (listed in Appendix A) was chosen for the final example. The program implements a noughts and crosses game taking a 9-bit input corresponding to buttons for each of the nine squares and two 9-bit outputs corresponding to the 'X' and 'O' masks for each of the squares (only either the 'X' or the 'O' or neither output is asserted for each square). The user places a piece by pressing a button corresponding to the square they choose. The "machine"[1] then makes its move using a basic AI algorithm.

[1] As the program maps to custom hardware the word "computer" is avoided.

Listing 2 Up/down counter example

```
void main() {
        in int:1 direction, user_clock;
        out int:4 count;
        while (1) {
                // wait for rising edge
                // (button down)
                while(user_clock);
                while(!user_clock);
                if (direction) {
                        if (count != 0) {
                                --count;
                        }
                } else {
                        if (count != 15) {
                                ++count;
                        }
                }
        }
}
```

5.1.3 Results

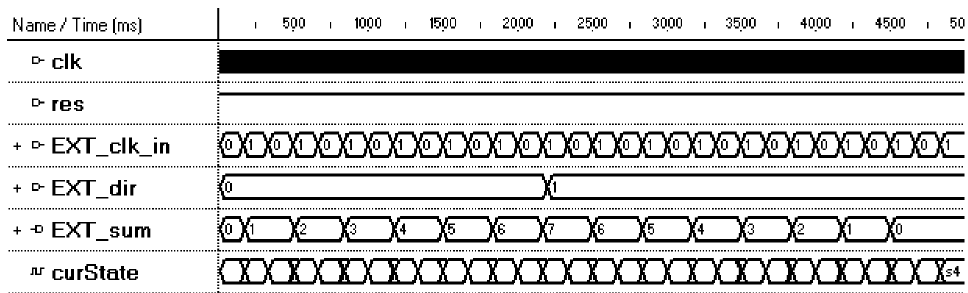
The VHDL output for both HLS algorithms and for all three programs matched the behaviour of the C input. This was confirmed via simulation using Active HDL. Example simulations waveforms are given in Figures 5.1, 5.2 and 5.3. These waveforms were generated by Active HDL. The *external* variables (inputs and outputs) of each design have signals prefixed by “EXT_”. The clock (“clk”) and asynchronous reset (“res”) signals are also shown. These simulation results match what was expected.

The designs were also mapped to a SpartanXL FPGA and tested using this device. Table 5.1 lists the chip area (from the place and route report) and minimum period (from the post-layout timing report) reported by the Xilinx Foundation tools for each of the implementations. As the two different scheduling methods produce designs with different specific operation timings the delay is itself not a particularly good measure. As such an experimental measure of the typical cycles[2] in the main loop for each program was multiplied to the delay to give some measure of the latency of a single iteration of the main

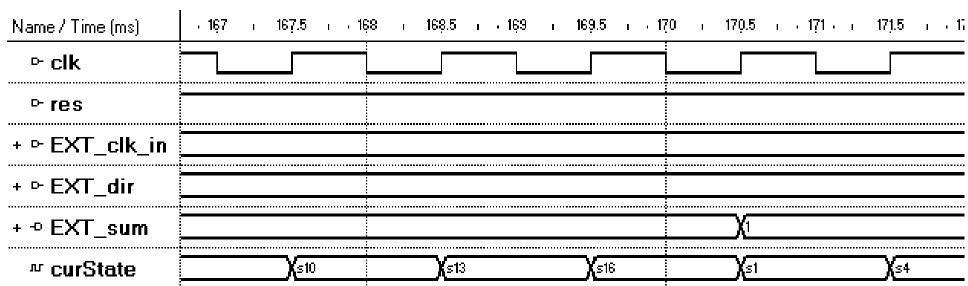
[2]The typical number of cycles between a user action and when the system is at equilibrium again was taken from simulation results. As the same number was used for each implementation the accuracy of this measure is not so important.

Listing 3 Matrix multiply example

```
void main() {
        // a limited number of inputs are
        // possible on a FPGA
        // the attempt is to try and make
        // it hard to optimize
        // out any of the 12 variables
        int:8 a1, a2, a3, b1, b2, b3, c1, c2, c3, x, y, z;
        in int:8 a;
        out int:8 xnew, ynew, znew;
        a1 = a;
        a2 = a + 1;
        a3 = a + 2;
        b1 = a + 3;
        b2 = a + 4;
        b3 = a + 5;
        c1 = a + 6;
        c2 = a + 7;
        c3 = a + 8;
        x = a + 9;
        y = a + 10;
        z = a + 11;
        while (1) {
                xnew = a1 * x + a2 * y + a3 * z;
                ynew = b1 * x + b2 * y + b3 * z;
                znew = c1 * x + c2 * y + c3 * z;
        }
}
```



(a) Whole simulation



(b) Magnified Portion

Figure 5.1: The up/down counter simulation

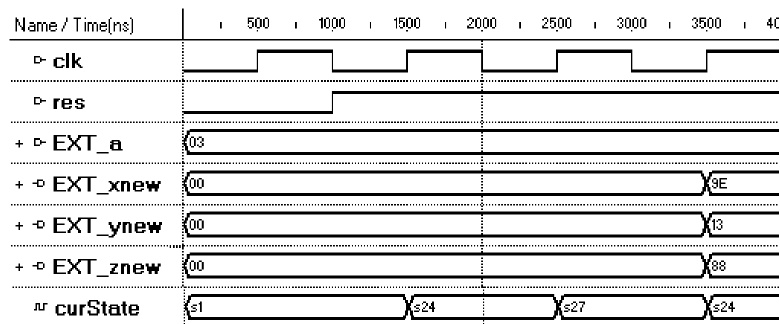


Figure 5.2: The matrix multiply simulation

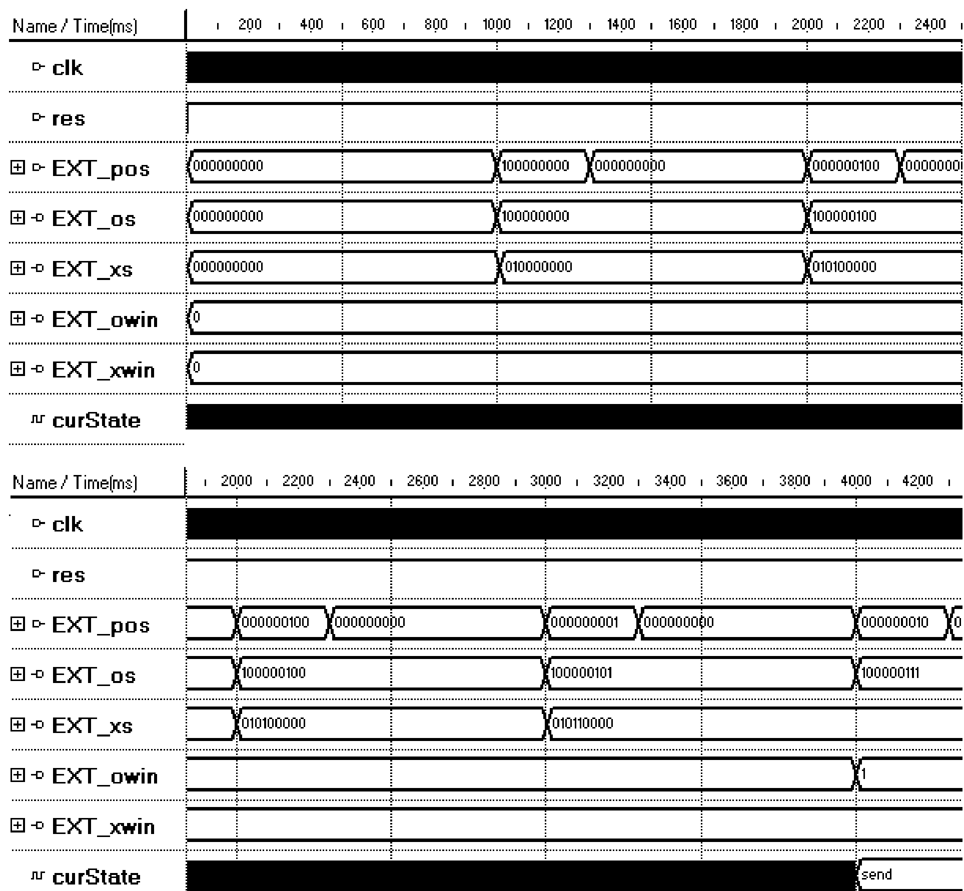


Figure 5.3: The noughts and crosses example

loop.

Table 5.1: Area cost of produced designs targeted to a SpartanXL FPGA

Program	Area (CLBs/% CLBs)	Minimum Period (ns)	Typical cycles in main loop	Typical main loop period (ns)
Counter (trivial)	10 (1%)	7.9	4	31.6
Counter (minimal cycles)	10 (1%)	7.9	4	31.6
Matrix (trivial)	163 (10%)	47.8	15	717
Matrix (minimal cycle)	390 (67%)	52.4	2	105
Noughts and Crosses (trivial)	177 (30%)	39.5	525	20740
Noughts and Crosses (minimal cycle)	82 (14%)	19.8	80	1584

5.2 Discussion of Observations

A number of observations were made during the testing of the system relating to the effectiveness of the synthesis process. Presented below are several issues (both positive and negative) that were identified. These findings indicate which parts of the system should be improved in future versions. Specific possible improvements are discussed in the next chapter.

5.2.1 Design Quality

The quality of designs produced by a synthesis tool is crucial in determining the usefulness of that system. As previously stated the aim of this project was to produce the first version of an open programming-based design-flow and therefore the quality of the system is not up to commercial standards. It is possible, however, to make observations on what areas of the system as developed influence quality thus providing assistance to any future developers.

The role of the low-level (logic) synthesis tools must be recognized when analyzing design quality. Circe produces an RTL level design, leaving it up to other tools to synthesize

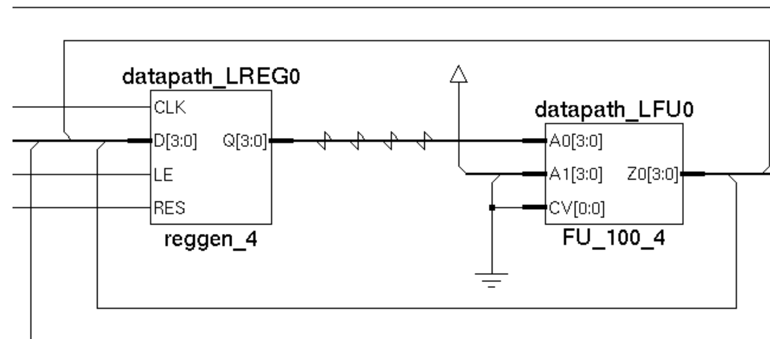
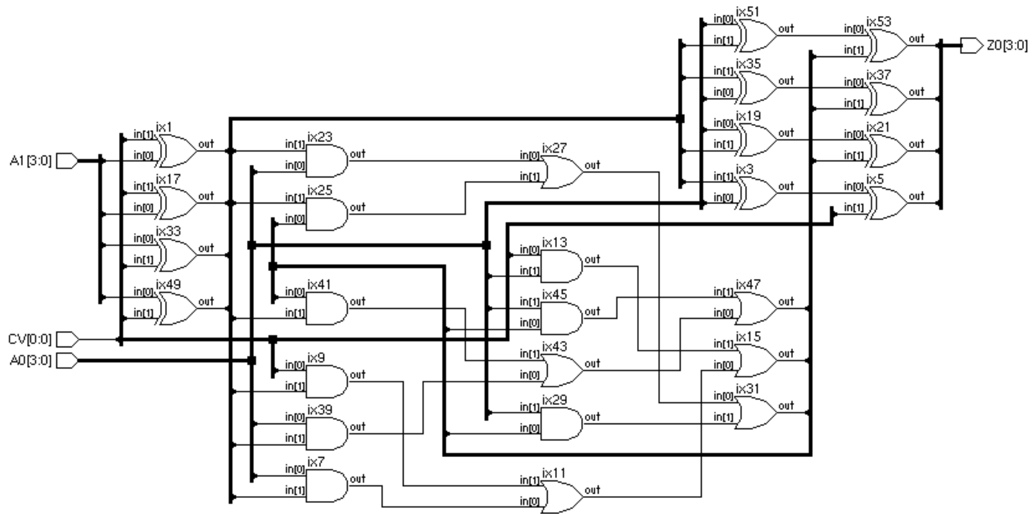


Figure 5.4: Circe generated datapath for a counter

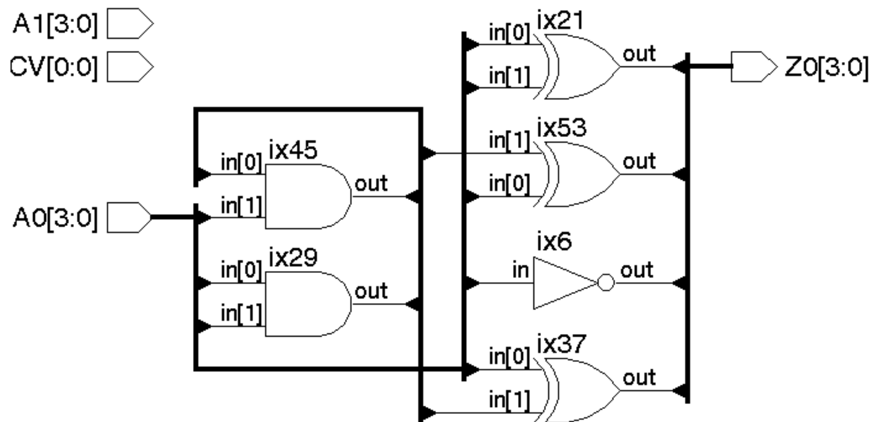
the specific logic circuitry that will be used. This choice will depend on the technology that is being targeted. It was observed that a side-effect of this mapping process was that various logic optimizations that Circe was not designed to make were being made during logic-synthesis. For example, consider a simple counter like that presented in the first example above. Assuming the adder functional unit is not shared, the hardware produced by Circe for the line “++counter” (for a 4-bit counter) is given in Figure 5.4[3]. Circe has synthesized an addition and register assignment, not making any attempt to extract a simplified behaviour, for example by inserting an explicit counter into the datapath. Despite this the logic-synthesis tool will simplify the adder because one input is permanently connected to “0001”. Leonardo Spectrum was used to synthesize this simple four-bit counter example and the results before and after “pre-optimization” (the name given by Leonardo for a step that includes various logic optimizations) are given in Figures 5.5 (a) and 5.5 (b) respectively. These illustrate an example of the possible improvements to the design that can be made at a lower-level than Circe.

While these improvements made by the logic-synthesis tools seem beneficial there are some downsides to leaving these sorts of optimization out of Circe. One problem is that without at least some basic estimates of what optimizations can be performed at the logic-level it is difficult for HLS to make good scheduling and allocation decisions. For example, it may be determined by the Synthesis Engine that the cost of two adders would be more than the cost of a single adder that is multiplexed. However, this could be the wrong decision if both addition operations involve constants that could cause a lot of the adder logic to be optimized away during logic-synthesis. This sort of problem is

[3]This figure was generated by using Leonardo Spectrum and selecting “view RTL schematic” after inputting the VHDL design



(a) The 4-bit adder/subtracter functional-unit



(b) The 4-bit adder/subtracter optimized to add by “0001”

Figure 5.5: The adder functional-unit before and after logic-level optimization

not particularly evident in the current version of Circe as there is no use of formal cost functions during HLS[4]. It would be a problem if more sophisticated HLS techniques were added, however.

5.2.2 Semantic Issues

Another factor in the evaluation of behavioural design tools (for either software or hardware) is how well they can consistently extract meaning from a design. That is, what level of influence does the style of the design, ie the code style, have on the synthesized design. Ideally if two different code fragments have the same meaning then they should lead to the same hardware implementation being produced as the role of the synthesis tool is to *extract* behaviour rather than simply translate a design from one form to another. Two examples of this issue that were observed during testing are outlined briefly below.

In the input language to Circe, there are two different ways to assign values based on a conditional expression: the *if* statement and the *conditional* (ternary) operator. Listing 4 gives an example of these two methods. Circe produces predominantly control structure for the *if* statement (at least two new states, one for each branch of the *if*) whereas a purely datapath structure is produced for the *conditional* operator method (there may be one or several states produced depending on the scheduling method). Ideally Circe should be able to treat both methods in the same way. A possible addition to Circe to enable this is discussed in Chapter 6.

Listing 4 Two different ways to assign a variable based on a condition

```
// If statement
if (cond) {
    foo = a * 3;
} else {
    foo = b + c;
}
...
// Conditional operator
foo = cond ? (a * 3) : (b + c);
```

An example of where Circe performs better at extracting behaviour comes when making use of “temporary” variables. When using the *minimal cycle* scheduling algorithm Circe

[4]Allocation does use a very basic notion of “cost”, however, that limits the number of inputs that should be multiplexed to a particular functional-unit type.

produces the same hardware for each of the two code fragments shown in Listing 5 assuming that “temp” is not an output. This is because Circe optimizes out registers that are not needed. As both the assignment to “temp” and to “foo” occur in the same clock cycle the “temp” register is never read from and therefore not needed.

Listing 5 Two different ways to write an assignment expression

```
// A complicated expression
foo = a * b + ((c << 2) * d);
...
// Use of temporary variable
// NB 'temp' is not used elsewhere
temp = (c << 2) * d;
foo = a * b + temp;
```

5.3 Comparison with Specification

In Chapter 3 a number of specifications for the proposed system were outlined. The extent to which each these objectives were achieved is described below. A summary of how well the objectives were achieved is given in Table 5.2.

5.3.1 Design Input

The implemented system achieved its objective of accepting input in a variant of the C programming language. Specifically, the input language allowed for most basic C constructs and allowed reasonably complicated example programs to be implemented. The specific C variant used in the implementation was more restrictive, however, than the original differences outlined in the specifications. Most significantly function calls and switch statements were not implemented in the system. These omissions stemmed from time limitations rather than serious theoretical hurdles of the project. As such an outline of how these restrictions may be removed is given in the next chapter.

5.3.2 Design Output

The primary objective relating to design output was that it must be in a RTL form - i.e. a structural interconnection of functional units (FUs), registers and multiplexers (MUXs)

with a finite-state machine (FSM) for a controller - and that it must be synthesizable by lower-level synthesis tools. These objectives were satisfied by making the instances of FUs and registers explicit in the VHDL output as well as by using a standard form of FSM descriptions. The design outputs were shown experimentally to be synthesizable by both of the synthesis tools (Xilinx Foundation Series and Leonardo Spectrum) used.

Another objective was that a variety of output structures should be possible for a single behavioural input. The purpose of this was to demonstrate that high-level synthesis (HLS) was possible with the system. As well as the default (trivial) mode a second (minimize cycles) mode was added. While neither of these modes represents a particularly sophisticated HLS algorithm they are enough to demonstrate that different outputs may be generated by varying the HLS algorithm used. The next chapter outlines possible improved HLS algorithms that could be implemented to extend the system.

5.3.3 Extensibility

One of the purposes for having an open source system of this kind was to allow new synthesis or optimization techniques to be experimented with easily. As such an important objective was to design the system to be extensible in these areas. The system as implemented achieves this objective through two design choices: the simple yet flexible representation of designs; and the modularization of the design flow into different stages. The design representation allows for a large range of HLS possibilities through simple annotations (as used by the basic HLS algorithms implemented) to the data and control graphs. Even more HLS options as well as optimization techniques are possible by manipulating the graphs, for example by swapping nodes or transferring behaviour from the controller to the datapath. More detail on the possible extensions is given in the next chapter.

The use of open libraries and development tools will allow this system to remain open source. The open source GNU C++ compiler (g++) was used for the compilation. Most required functionality came from the standard C++ library which comes standard with the GNU compiler. Extra functionality was needed for graph manipulation and this was found in the Graph Template Library (GTL) [32] which is also open source.

Table 5.2: Achievement of Objectives

Objective	Level of Achievement
To accept a design input in a variant of the C programming language	Most basic C constructs were supported. However function calls and switch statements were not implemented. It is possible for these to be added in future improvements, however. Also only simple data types are currently supported.
To output a structural design (datapath and controller)	This objective was achieved. The datapath outputted was a structural interconnection of FUs, registers and MUXs. The controller was outputted as a FSM description.
To produce Output in a subset of synthesizable VHDL	This objective was shown to be met through use of both Xilinx Foundation and Leonardo spectrum VHDL synthesis tools. The RTL VHDL descriptions output by the system was successfully mapped to FPGAs and VLSI using these tools.
To produce a variety of output structures for a single behavioural input	Two HLS algorithms were implemented to demonstrate that it was possible to produce more than a single structure for one behaviour. These algorithms were not particularly sophisticated or extensive however.
To allow extensions to be added	The choice of the intermediate representation for the design as well as the modularization of the system will allow extensions to be added or improvements made to the system relatively easily.
To be developed using non-proprietary libraries/ development tools	Open source libraries (standard C++ and GTL) were used as well as the open compiler g++.

5.4 Comparison with related products

Some comparisons can be made between Circe and some of the other Hardware Compilers outlined in Chapter 2, however the ability to do this is limited by access to these tools. For the most part the comparisons made are general. The Celoxica's Handel-C compiler version 3.0.2505 (DK1 design suite version 1.0 SP1) was accessible for this study and so it was possible to perform some more direct comparisons with that product.

5.4.1 Overall Comparisons

Table 5.3: Comparison of Previous Work with Circe

Name	Input	Synthesis	Target
Circe	Programming focus - Restricted C Variant, no constructs for timing. Arbitrary width integer data types.	Simple HLS performed	RTL (plus FSM controller description) VHDL
SystemC (Synopsys' Cocentric Compiler)	Modeling focus using C++ syntax/semantics. behavioural model is essentially "programming", however.	HLS performed.	RTL (but tools included for FPGA, VLSI)
Handel-C (Celoxica)	Programming focus - C variant with explicit constructs for parallel execution. Supports arrays, pointers (but not dynamic allocation). RAM must be specifically targeted.	No Scheduling. Timing built into language - user determined scheduling.	FPGA (ready for place and route)
Bach	Programming focus - C variant. Explicit constructs for parallel execution.	HLS performed. Timing restrictions can be specified by user.	RTL VHDL
SPARK	Programming focus - ANSI C.	HLS performed.	RTL VHDL
SpC	Programming - C based. Includes dynamic memory allocation, pointers, and arbitrary C data structures.	No HLS performed (left in a behavioural form).	Verilog (Behavioural)

Table 5.3 attempts to summarize Circe within the same table as that used when comparing previous work (Table 2.1). Overall Circe is currently a less advanced synthesis tool than the others mentioned. This is in terms of the input constructs supported and the synthesis

options available. Circe is focused on programming and performs HLS (albeit a simple algorithm in this version) and this makes it perhaps more similar to *Bach* than to some of the other hardware compilers mentioned. Despite its curent shortcomings, Circe should be extensible enough to allow it to closer approach the standard of existing tools in future versions.

5.4.2 Direct Comparison with Handel

It is difficult to directly compare Handel-C with Circe for a number of reasons:

- Circe has greater restrictions on the input language than Handel;
- Handel-C scheduling is fixed, based on the source rather than being determined by its synthesis engine;
- Handel-C performs its own low-level synthesis rather than requiring a tool such as *Xilinx Foundation* to do this (*Xilinx* is still needed for implementation); and
- It is difficult to compare synthesis tools with only a small range of examples and without detailed analysis of output designs.

Keeping these problems in mind the steps taken to compare Circe examples to Handel-C were as follows:

1. Take the program written for Circe and translate the syntactical differences for Handel (eg integer width is specified by ":" in Circe but with a space in Handel)
2. Synthesize and implement the program using Handel. This is the "natural" scheduling of the program in Handel.
3. Modify the program to give it the same specific timing as the *minimal cycle* schedule of Circe.
4. Synthesize and implement the modified program. This is the "matched" scheduling of the program in Handel.

The default options were used for both Handel and Xilinx and the results alongside those of Circe are presented in Table 5.4. Due to the difficulties in comparison outlined above, not much can be concluded from these results. What can be said is that at least in the examples the designs synthesized by Circe to RTL and mapped to the SpartanXL with Xilinx are of a similar quality to those synthesized by Handel.

Table 5.4: Direct Comparison With Handel-C

Program	Circe (min-cycle)		Handel (matched)		Handel ("natural")	
	Area (CLBs)	Period (ns)	Area (CLBs)	Period (ns)	Area (CLBs)	Period (ns)
Counter	10	7.9	10	3.7	9	5.1
Matrix	390	52.4	236	55.9	253	48.1
Noughts and Crosses	82	19.8	77	32.5	75	26.8

5.5 Evaluation of Engineering performance

5.5.1 Process

The overall process followed in this thesis was methodical and as a result the major objectives of the project were achieved. Specifically, the broad specification as outlined in Chapter 1 was refined into a set of measurable and feasible goals (as detailed in Chapter 2). The design work was broken into three major parts and a basic plan was drawn up along these lines. As the primary goal was to achieve a *working* design-flow, emphasis was placed on completing this first and then coming back to improve individual components if time allowed. This plan was followed quite closely but the time required to complete a first version of the design-flow was longer than expected. As a result there were more limitations to the input language than originally desired.

One aspect of the process that could be improved upon was in the testing procedure. While a formal testing plan was included on the schedule - each component was to have tests designed for and performed on it as well as formal testing of the system as a whole - this process was slackened in reality. As a result there were bugs in individual components that were not found until system testing but that should have been detected earlier. The general bug fixing process was systematic and effective, however. Various techniques including isolation, code inspection and the use of a debugger (gdb) were used to resolve problems.

The outcomes of the project demonstrate the success of the process followed. The stated objectives were achieved.

5.5.2 Contribution

The main contribution of this thesis was the development of a complete, first version of an open-source Hardware Compiler, Circe. This program was written using all open-source, non-proprietary tools: GNU C++, Flex, Bison and Boost's Graph Template Library, allowing it to be distributed and extended freely. The more specific contributions are:

- Ability to form a complete design flow from source to VLSI or an FPGA by combining Circe with logic synthesis, place and route tools;
- A modularized design that allows independent update of parsing, synthesis and design outputting functionality; and
- Data structures that allow the implementation of a large range of scheduling and allocation algorithms.

5.6 Summary

This chapter presented an evaluation of the work done in this thesis. The testing procedure for Circe was presented as was both the simulation and synthesis results of a few designs produced using this system. Some of the general observations made when testing the system were also discussed.

An evaluation of Circe against the specifications derived in Chapter 3 was performed. This indicated that the major goals of the project were achieved. The more open-ended objectives were also achieved to a reasonable level. Circe was compared with some existing products, in particular Handel-C, and despite currently being less advanced shows promise in producing designs of a similar quality. An evaluation of engineering performance during this thesis was also given.

The next chapter indicates some of the improvements that could be made to Circe in future versions.

Chapter 6

Future Developments

Chapters 3, 4 and 5 discussed the specification, implementation and evaluation of the Circe hardware compiler produced in this study. In these chapters comments were made regarding decisions that were necessary to ensure completion of the tool but that limited its flexibility or performance. In this chapter these issues are addressed with a look at what parts of the system could be improved for future versions. Possible improvements to the existing input language, high-level synthesis system and the design output are all discussed as are possible new features, such as optimizations.

6.1 The Input Language

The input language used to specify designs for this system could possibly be improved in the following ways:

- Addition of further C constructs (specifically *switch* and function calls);
- Addition of constructs to allow more explicit control over timing; and
- Addition of constructs to allow more explicit control over parallelism.

Each of these possibilities will be briefly discussed below.

The addition of function calls would be a major improvement to the design language as it would allow for greater design abstraction. This addition would, however, require some major decisions on the meaning of functions to be addressed. One of the issues is

how parameters would be passed to and from the hardware representing functions. This is particularly a problem as hardware blocks in general would have arbitrary numbers of outputs whereas a C function has just one return value. The problem is even more subtle than this as it would be useful to allow functions to have direct access to inputs and outputs of the system or variables in the callee function. One method envisioned to tackle these problems is to allow a form of pass-by-reference and pass-by-constant reference for function calls. Passing an argument by constant reference would be the equivalent of an unlatched input to the function. Passing by (non-constant) reference could be used to imply direct outputs from the function.

Switch statements could be added to the language relatively easily. In the trivial case a *switch* statement could be implemented with a comparison operation for every *case* clause and a chain of fork nodes. The "fall-through" behaviour of the *switch* statement could be made possible by the addition of join nodes (where needed). This trivial implementation would suffer however as it would most likely lead to a proliferation of comparators in the design where perhaps a single decoder might be more applicable to situations where *switch* is used.

Some similar systems such as Handel-C and Bach allow the user to have more specific control over the specific timing behaviour of designs. Handel-C in particular is built around the principle of strict timing semantics. The emphasis of this project was towards synthesis from a *programming* language rather than from a hardware description language designed to look like C and so enforcing such strict timing rules at the expense of high-level synthesis possibilities was not considered. It would be useful, however, to allow some specific control over scheduling of operations to clock cycles to allow for more timing critical designs. The consideration of this would have to weigh up the benefits of these constructs versus the benefits of keeping the design specification at a more abstract level.

Additional constructs to allow parallel or threaded semantics could also be an improvement. It would be preferable to do this in a way resembling the use of threads in software development. This would allow for the user to specify where course grain parallelism of algorithms can be exploited. The fine grain parallelism could be found automatically by use of more sophisticated synthesis and optimization techniques (see below).

6.2 The Synthesis Engine

The HLS techniques already implemented are quite basic and primarily serve to demonstrate that the underlying data structures (and the design-flow as a whole) works. More sophisticated techniques that minimize a cost function while obeying a set of user constraints (e.g., delay, area or timing) could be implemented. In order to do this the models of functional units would have to be extended to include delay and area estimates and there would have to be a method of estimating the cost of the controller as well as multiplexers. Functional units are also presently restricted to being single cycle operations. Lifting this restriction would provide greater synthesis options that should help improve the quality of produced designs.

6.3 Optimizations

The quality of produced designs could also be improved by various optimization techniques. There are a number of optimization techniques that have not yet been explored but that could be added to Circe in subsequent versions. The various types of possible optimizations include source-level, data and control-flow analysis and techniques to improve fine-grain parallelism.

Source-level optimizations are used in software compilers and include constant-folding and common sub-expression elimination. These techniques would be performed on the representation of the source code, that is, on the abstract syntax tree. Constant-folding is performed by locating expressions containing multiple constant terms that can be simplified at compile-time. For example, the expression "3 * 2" in the source could be replaced by "6". Common sub-expression elimination is the identification and combining of repeated sub-expressions in the source. This stops the same expression being evaluated more than once unnecessarily.

More subtle optimizations could be implemented through analysis of the control-flow and data-flow graph (CFG and DFG) structures. For example, the range for which variables are "active" could be analyzed to allow registers to be shared across multiple variables. It is also possible to eliminate portions of the code that are unreachable or have no lasting effect on the program (dead-code elimination). In some cases it may also be beneficial to transfer behaviour from the CFG to the DFG. For example, an *if* statement that is used simply to choose what value to assign to a variable will generate at least 2 states whereas

it might be better to perform the choice within the datapath by using a multiplexer (the equivalent of the *C select* operator). Figure 6.1 illustrates an example of this situation. The resulting graphs are as if the user had used the conditional expression operator (?, :) instead of an *if* statement. There are more complicated situations than this but the same basic principle is present.

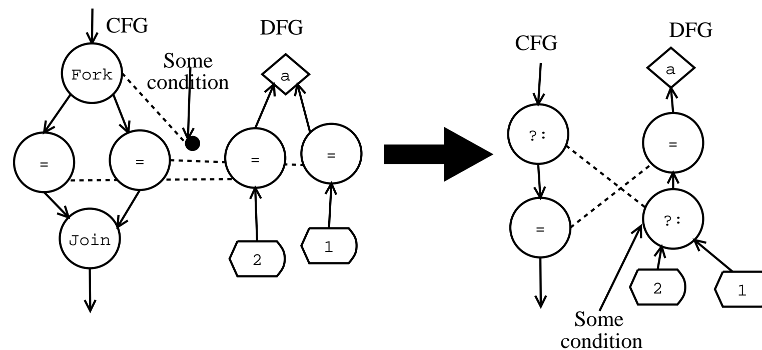


Figure 6.1: An example of moving behaviour from the control to the data-flow

Another source of improvement would be to better exploit the parallel nature of hardware where possible. Some constructs imply sequential execution when parallel execution may produce a better[1] design. For example a *for* loop may be used to calculate a parity bit by xor-ing data bits together one bit at a time (See algorithm 6). In a hardware implementation it may be better to treat this loop as if all of the loop iterations were written out explicitly, allowing them to be scheduled in the same control step and also allowing the "dummy" variable to be removed. This technique of "loop-unrolling" is used in compilers for CPUs that support instruction-level parallelism [18].

6.4 Targets

The system could be further developed by extending it so that it can produce different output formats or target different technologies. For example, a controller synthesis module could be added so that a pure RTL target (i.e. one without a finite-state machine description) could be produced. The system could also be extended to perform logic synthesis or technology mapping. The advantages of this would be less reliance on commercial tools as well as a potential for greater feedback during the synthesis process leading to better design quality.

[1]better according to the particular design constraints and cost functions

Listing 6 Parity using a for loop (above) and the unrolled loop (below)

```
// Looped parity for 4 bits - i is a dummy variable
for (i=0; i<4; ++i) {
    parity = parity ^ (data & 0x01);
    data = data >> 1;
}
...
// The unrolled loop
parity = parity ^ (data & 0x01);
data = data >> 1;
parity = parity ^ (data & 0x01);
data = data >> 1;
parity = parity ^ (data & 0x01);
data = data >> 1;
parity = parity ^ (data & 0x01);
data = data >> 1;
```

Another more significant extension would be to allow the user to target either hardware or software from the same input design. For example, instead of custom hardware they may wish to target a "softcore" processor for their FPGA design. If Circe was extended to be able to target both of these options then the designer would be able to try both implementations and compare the benefits of each alternative.

6.5 Summary

This chapter outlined some possible future enhancements to the CAD tool presented in this thesis. Improvements to the input language, such as the addition of further C constructs and the inclusion of a method to specify precise timings were suggested. The addition of more sophisticated synthesis methods as well as optimization techniques were canvassed as modifications that would improve the quality of the produced designs.

Chapter 7

Conclusions

This thesis has presented an overview of the development and evaluation of an open-source synthesis tool, Circe, that produces a structural, RTL output from a behavioural description written in a variant of the C programming language. The input language implemented allowed most basic C constructs but restricted data to simple integer types and did not allow function calls. The output produced by this tool is in the form of a VHDL datapath and controller. The datapath consisted of a connection of functional-units and registers specified from a hand-written generic library of components. The controller consisted of a standard, synthesizable finite-state machine description. The produced designs were shown to be synthesizable by logic-synthesis tools and able to be mapped to both FPGA and VLSI targets.

The process followed in the study was presented in this document. First, some introductory material including the motivation for the study was given. This set out the reasons for the project, thus influencing the goals that were decided upon. Following this introduction, background from the areas of hardware synthesis, particularly programming language-based, along with a summary of previous work in this area was given. This provided some basic theory that was drawn upon during the design process. The project was then formally scoped and specified and a list of measurable goals was presented. The process of design and implementation for Circe was then given including the design breakdown that was used as well as reasons behind the major design decisions that were made. The results of testing the system were then presented and the design evaluated against the given specifications as well as compared to other related projects. Future directions for Circe were then outlined.

The project presented in this thesis provides a working, open hardware compilation tool

and a solid foundation for future work. Although there are elements of the design that need to be improved before Circe reaches the standard of commercial tools the implementation presented does have a complete design-flow from "C" code to RTL VHDL and has been designed to be open to allow improvement and extension. Testing has shown that Circe produces designs that are correct both when simulated and when implemented on an FPGA. As such the major objectives of this study were achieved.

Bibliography

[1] D. Gajski and R. Kuhn, "Guest Editors' Introduction: New VLSI Tools," *IEEE Computer*, vol. 16, pp. 11–14, December 1983.

[2] J. Rabaey, *Digital Integrated Circuits*. Prentice Hall, Upper Saddle River, NJ, 1996.

[3] R. Gupta, G. Berry, R. Chandra, D. Gajski, K. Konigsfeld, P. Schaumont, and I. Verbauhede, "The next HDL (panel session): if C++ is ;the answer, what was the question?," in *Proceedings of the 38th conference on Design automation*, pp. 71–72, ACM Press, 2001.

[4] "The Free Software Definition." http://www.fsf.org/philosophy/free-sw.html, (online, April 2002).

[5] D. Gajski, N. Dutt, A. Wu, and S. Lim, *High Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, Boston, 1992.

[6] S. Gerez, *Algorithms for VLSI Design Automation*. John Wiley and Sons, Chichester, 1999.

[7] J. Wakerly, *Digital Design: Principles and Practices, 3rd Ed.* Prentice Hall, Upper Saddle River, NJ, 2001.

[8] H. Bhathagar, *Advanced ASIC chip synthesis: using Synopsys Design Compiler and PrimeTime*. Kluwer Academic Publishers, Boston, 1999.

[9] "VHDLSynth (1076.3) Home Page." http://www.vhdl.org/vhdlsynth/, (online, April 2002).

[10] "Verilog Synthesis Interoperability 1364.1." http://www.eda.org/vlog-synth/, (online, April 2002).

[11] M. Scott, *Programming Language Pragmatics*. Morgan Kaufmann, San Francisco, 2000.

[12] N. Wirth, “Hardware Compilation: Translating Programs into Circuits,” *IEEE Computer*, vol. 31, pp. 25–31, June 1998.

[13] J. K. A. Ghosh, “Hardware Synthesis from C/C++,” in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition 1999, IEEE*, pp. 387–9, 1999.

[14] B. Preiss, *Data Structures and Algorithms*. John Wiley & Sons, Inc., New York, 1999.

[15] R. Camposano and R. M. Tabet, “Design Representation for the Synthesis of Behavioral VHDL Models,” in *Proceedings of the 9th International Conference on Computer Hardware Description Languages and their Applications*.

[16] R. A. Bergamaschi, “Bridging the Domain of High-level and Logic Synthesis,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, pp. 582–596, May 2002.

[17] *Handel-C Language Reference Manual, Version 2.1*. Celoxica, 2001.

[18] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach, 2nd Ed.* Morgan Kaufmann, 1996.

[19] A. Appel, *Modern Compiler Implementation in C*. Cambridge University Press, Cambridge, 1998.

[20] M. Wolfe, *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, 1995.

[21] M. Weinhardt and W. Luk, “Evaluating Hardware Compilation Techniques,” *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines, IEEE Comput. Soc.*, pp. 333–4, 2000.

[22] “The SystemC Community.” http://www.systemc.org/, (online, April 2002).

[23] “CoCentric SystemC Compiler.” http://www.synopsis.com/products/cocentric_systemC/cocentric_systemC_ds.html, (online, April 2002).

[24] "Celoxica DK1 Design Suite." http://www.celoxica.com/products/design_suite/index.htm, (Online, April 2002).

[25] T. Kambe, A. Yamada, K. Nishida, K. Okada, M. Ohnishi, A. Kay, P. Boca, V. Zammit, and T. Nomura, "A C-based synthesis system, Bach, and its application (invited talk)," in *Proceedings of the conference on Asia South Pacific Design Automation Conference*, pp. 151–155, ACM Press, 2001.

[26] "SPARK: Synthesis using Parallelizing Compiler Techniques." http://www.ics.uci.edu/ spark/, (online, April 2002).

[27] L. Séméria, K. Sato, and G. D. Micheli, "Synthesis of hardware models in c with pointers and complex data structures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 6, pp. 743–756, 2001.

[28] "Electronic Design Interchange Format." http://www.edif.org/, (online, April 2002).

[29] K. Louden, *Compiler Construction: Principles and Practice*. PWS Publishing, Boston, 1997.

[30] J. D. J. Lee, edited by T. Stockfisch, "ANSI C Grammar." http://www.lysator.liu.se/c/ANSI-C-grammar-y.html, 1995 (online, April 2002).

[31] T. Pittman and J. Peters, *The Art of Compiler Design: Theory and Practice*. Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[32] "The Graph Template Library." http://www.infosun.fmi.uni-passau.de/GTL/, (online, July 2002).

[33] R. Airiau, J. Berge, and V. Olive, *Circuit Synthesis with VHDL*. Kluwer Academic Publishers, Boston, 1994.

Appendix A

A Worked Example : Noughts and Crosses

Specification

The Noughts and Crosses program implements a noughts and crosses game taking a 9-bin input corresponding to buttons for each of the nine squares and two 9-bit outputs corresponding to the 'X' and 'O' masks for each of the squares (only either the 'X' or the 'O' or neither output is asserted for each square). The user places a piece by pressing a button corresponding to the square they choose. The machine then makes its move using a simple AI algorithm.

Source Code

The "C" source for this program is given below:

```
void main() {
  // All these start by default at 0
  in unsigned int:9 pos;
        unsigned int:9 possaved;
  out unsigned int:9 os, xs;
  out unsigned int:1 owin, xwin;
  out unsigned int:9 temp, aitemp, aitemp2, move;
  int:1 turn; // 0 player (o's), 1 me
  unsigned int:2 i;
  turn = 0; // player starts
  while (!(owin || xwin)) {
    if (turn) {
      // AI algorithm here
      //
      // for now check for a possible win by me
      // or by my opponent and move there
```

```
        // else go anywhere
        move = 0;
        for (i = 0; i < 2 && !move; ++i) {
          aitemp = 0b100000000;
          while (aitemp) {
            aitemp2 = aitemp | (i ? xs : os );
            if ((!(((~aitemp2) & 0b111000000) && // rows
                  ((~aitemp2) & 0b000111000) &&
                  ((~aitemp2) & 0b000000111) &&
                  ((~aitemp2) & 0b100100100) && // cols
                  ((~aitemp2) & 0b010010010) &&
                  ((~aitemp2) & 0b001001001) &&
                  ((~aitemp2) & 0b100010001) && // diags
                  ((~aitemp2) & 0b001010100)))
                && (!(xs & aitemp)) && (!(os & aitemp))
                ) {
              move = aitemp;
              break;
            }
            aitemp = aitemp >> 1;
          }
        }
        if (!move) {
          aitemp = 0b100000000;
          while ((aitemp & os) || (aitemp & xs))
            aitemp = aitemp >> 1;
          move = aitemp;
        }
        xs = xs | move;
        temp = xs;
      } else {
        // Save the button states
        possaved = pos;
        // wait for user
        while ((!possaved) ||
               // pos is already filled
               (possaved & os) || (possaved & xs))
                  possaved = pos;
        os = os | possaved;
        temp = os;
      }
      // check for a win
      if (!(((~temp) & 0b111000000) && // rows
            ((~temp) & 0b000111000) &&
            ((~temp) & 0b000000111) &&
            ((~temp) & 0b100100100) && // cols
            ((~temp) & 0b010010010) &&
            ((~temp) & 0b001001001) &&
            ((~temp) & 0b100010001) && // diags
            ((~temp) & 0b001010100))) {
        if (turn) {
          xwin = 1;
        } else {
          owin = 1;
        }
      }
      // check for draw
      else if (!(~(os | xs))) {
        owin = 1;
        xwin = 1;
      }
      turn = ~turn;
    }
  }
```

Output

The output from Circe for the noughts and crosses program is given below. To save space only the output from the minimal-cycle (option “mc” on Circe) method is presented here.

```
Library ieee;
Use ieee.std_logic_1164.all;
Use ieee.std_logic_misc.all; -- for reduce
Use ieee.std_logic_arith.all; -- for Ext
entity osandxs_main is
    port(
        clk : in std_logic;
        res : in std_logic;
        EXT_owin : out std_logic_vector(0 downto 0);
        EXT_xwin : out std_logic_vector(0 downto 0);
        EXT_pos : in std_logic_vector(8 downto 0);
        EXT_os : out std_logic_vector(8 downto 0);
        EXT_xs : out std_logic_vector(8 downto 0);
        EXT_temp : out std_logic_vector(8 downto 0);
        EXT_move : out std_logic_vector(8 downto 0);
        EXT_aitemp : out std_logic_vector(8 downto 0);
        EXT_aitemp2 : out std_logic_vector(8 downto 0)
    );
end osandxs_main;
architecture osandxs_main_arch of osandxs_main is
    -- Node: 1, type: 2
    signal node1_o : std_logic_vector(0 downto 0);
    -- Node: 2, type: 0, op: 100
    signal node2_i0 : std_logic_vector(0 downto 0);
    signal node2_o : std_logic_vector(0 downto 0);
    -- Node: 3, type: 3
    signal node3_o : std_logic_vector(0 downto 0);
    -- Node: 4, type: 3
    signal node4_o : std_logic_vector(0 downto 0);
    -- Node: 5, type: 3
    signal node5_o : std_logic_vector(0 downto 0);
    -- Node: 6, type: 0, op: 21
    signal node6_i0 : std_logic_vector(0 downto 0);
    signal node6_i1 : std_logic_vector(0 downto 0);
    signal node6_o : std_logic_vector(0 downto 0);
    -- Node: 7, type: 0, op: 23
    signal node7_i0 : std_logic_vector(0 downto 0);
    signal node7_o : std_logic_vector(0 downto 0);
    -- Node: 8, type: 3
    signal node8_o : std_logic_vector(8 downto 0);
    -- Node: 9, type: 0, op: 100
    signal node9_i0 : std_logic_vector(8 downto 0);
    signal node9_o : std_logic_vector(8 downto 0);
    -- Node: 10, type: 3
    signal node10_o : std_logic_vector(8 downto 0);
    -- Node: 11, type: 0, op: 23
    signal node11_i0 : std_logic_vector(8 downto 0);
    signal node11_o : std_logic_vector(0 downto 0);
    -- Node: 12, type: 3
    signal node12_o : std_logic_vector(8 downto 0);
    -- Node: 13, type: 0, op: 10
    signal node13_i0 : std_logic_vector(8 downto 0);
    signal node13_i1 : std_logic_vector(8 downto 0);
    signal node13_o : std_logic_vector(8 downto 0);
    -- Node: 14, type: 0, op: 21
    signal node14_i0 : std_logic_vector(0 downto 0);
    signal node14_i1 : std_logic_vector(8 downto 0);
    signal node14_o : std_logic_vector(0 downto 0);
    -- Node: 15, type: 3
    signal node15_o : std_logic_vector(8 downto 0);
    -- Node: 16, type: 0, op: 10
    signal node16_i0 : std_logic_vector(8 downto 0);
    signal node16_i1 : std_logic_vector(8 downto 0);
    signal node16_o : std_logic_vector(8 downto 0);
    -- Node: 17, type: 0, op: 21
    signal node17_i0 : std_logic_vector(0 downto 0);
    signal node17_i1 : std_logic_vector(8 downto 0);
    signal node17_o : std_logic_vector(0 downto 0);
    -- Node: 18, type: 0, op: 100
    signal node18_i0 : std_logic_vector(8 downto 0);
    signal node18_o : std_logic_vector(8 downto 0);
    -- Node: 19, type: 0, op: 11
    signal node19_i0 : std_logic_vector(8 downto 0);
    signal node19_i1 : std_logic_vector(8 downto 0);
    signal node19_o : std_logic_vector(8 downto 0);
    -- Node: 20, type: 0, op: 100
    signal node20_i0 : std_logic_vector(8 downto 0);
    signal node20_o : std_logic_vector(8 downto 0);
    -- Node: 21, type: 0, op: 100
    signal node21_i0 : std_logic_vector(8 downto 0);
    signal node21_o : std_logic_vector(8 downto 0);
    -- Node: 22, type: 3
    signal node22_o : std_logic_vector(8 downto 0);
    -- Node: 23, type: 2
    signal node23_o : std_logic_vector(0 downto 0);
    -- Node: 24, type: 0, op: 100
    signal node24_i0 : std_logic_vector(0 downto 0);
    signal node24_o : std_logic_vector(0 downto 0);
    -- Node: 25, type: 3
    signal node25_o : std_logic_vector(8 downto 0);
    -- Node: 26, type: 2
    signal node26_o : std_logic_vector(0 downto 0);
    -- Node: 27, type: 0, op: 100
    signal node27_i0 : std_logic_vector(0 downto 0);
    signal node27_o : std_logic_vector(0 downto 0);
    -- Node: 28, type: 3
    signal node28_o : std_logic_vector(1 downto 0);
    -- Node: 29, type: 2
    signal node29_o : std_logic_vector(1 downto 0);
    -- Node: 30, type: 0, op: 33
    signal node30_i0 : std_logic_vector(1 downto 0);
    signal node30_i1 : std_logic_vector(1 downto 0);
    signal node30_o : std_logic_vector(0 downto 0);
    -- Node: 31, type: 0, op: 23
    signal node31_i0 : std_logic_vector(8 downto 0);
    signal node31_o : std_logic_vector(0 downto 0);
    -- Node: 32, type: 0, op: 20
    signal node32_i0 : std_logic_vector(0 downto 0);
    signal node32_i1 : std_logic_vector(0 downto 0);
    signal node32_o : std_logic_vector(0 downto 0);
    -- Node: 33, type: 2
    signal node33_o : std_logic_vector(8 downto 0);
    -- Node: 34, type: 0, op: 100
    signal node34_i0 : std_logic_vector(8 downto 0);
    signal node34_o : std_logic_vector(8 downto 0);
    -- Node: 35, type: 3
    signal node35_o : std_logic_vector(8 downto 0);
    -- Node: 36, type: 0, op: 80
    signal node36_i0 : std_logic_vector(1 downto 0);
    signal node36_i1 : std_logic_vector(8 downto 0);
    signal node36_i2 : std_logic_vector(8 downto 0);
    signal node36_o : std_logic_vector(8 downto 0);
    -- Node: 37, type: 0, op: 11
    signal node37_i0 : std_logic_vector(8 downto 0);
    signal node37_i1 : std_logic_vector(8 downto 0);
    signal node37_o : std_logic_vector(8 downto 0);
    -- Node: 38, type: 0, op: 100
    signal node38_i0 : std_logic_vector(8 downto 0);
    signal node38_o : std_logic_vector(8 downto 0);
    -- Node: 39, type: 3
    signal node39_o : std_logic_vector(8 downto 0);
    -- Node: 40, type: 0, op: 13
    signal node40_i0 : std_logic_vector(8 downto 0);
    signal node40_o : std_logic_vector(8 downto 0);
    -- Node: 41, type: 4
    signal node41_i0 : std_logic_vector(8 downto 0);
    signal node41_o : std_logic_vector(8 downto 0);
    -- Node: 42, type: 0, op: 13
    signal node42_i0 : std_logic_vector(8 downto 0);
    signal node42_o : std_logic_vector(8 downto 0);
    -- Node: 43, type: 4
    signal node43_i0 : std_logic_vector(8 downto 0);
    signal node43_o : std_logic_vector(8 downto 0);
    -- Node: 44, type: 0, op: 20
    signal node44_i0 : std_logic_vector(8 downto 0);
    signal node44_i1 : std_logic_vector(8 downto 0);
    signal node44_o : std_logic_vector(0 downto 0);
    -- Node: 45, type: 0, op: 13
    signal node45_i0 : std_logic_vector(8 downto 0);
    signal node45_o : std_logic_vector(8 downto 0);
    -- Node: 46, type: 4
    signal node46_i0 : std_logic_vector(8 downto 0);
    signal node46_o : std_logic_vector(8 downto 0);
    -- Node: 47, type: 0, op: 20
    signal node47_i0 : std_logic_vector(0 downto 0);
    signal node47_i1 : std_logic_vector(8 downto 0);
    signal node47_o : std_logic_vector(0 downto 0);
    -- Node: 48, type: 0, op: 13
    signal node48_i0 : std_logic_vector(8 downto 0);
    signal node48_o : std_logic_vector(8 downto 0);
    -- Node: 49, type: 4
    signal node49_i0 : std_logic_vector(8 downto 0);
    signal node49_o : std_logic_vector(8 downto 0);
    -- Node: 50, type: 0, op: 20
    signal node50_i0 : std_logic_vector(0 downto 0);
    signal node50_i1 : std_logic_vector(8 downto 0);
    signal node50_o : std_logic_vector(0 downto 0);
    -- Node: 51, type: 0, op: 13
    signal node51_i0 : std_logic_vector(8 downto 0);
    signal node51_o : std_logic_vector(8 downto 0);
    -- Node: 52, type: 4
    signal node52_i0 : std_logic_vector(8 downto 0);
    signal node52_o : std_logic_vector(8 downto 0);
    -- Node: 53, type: 0, op: 20
    signal node53_i0 : std_logic_vector(0 downto 0);
    signal node53_i1 : std_logic_vector(8 downto 0);
    signal node53_o : std_logic_vector(0 downto 0);
    -- Node: 54, type: 0, op: 13
    signal node54_i0 : std_logic_vector(8 downto 0);
    signal node54_o : std_logic_vector(8 downto 0);
```

```
-- Node: 55, type: 4
signal node55_i0 : std_logic_vector(8 downto 0);
signal node55_o : std_logic_vector(8 downto 0);
-- Node: 56, type: 0, op: 20
signal node56_i0 : std_logic_vector(0 downto 0);
signal node56_i1 : std_logic_vector(8 downto 0);
signal node56_o : std_logic_vector(0 downto 0);
-- Node: 57, type: 0, op: 13
signal node57_i0 : std_logic_vector(8 downto 0);
signal node57_o : std_logic_vector(8 downto 0);
-- Node: 58, type: 4
signal node58_i0 : std_logic_vector(8 downto 0);
signal node58_o : std_logic_vector(8 downto 0);
-- Node: 59, type: 0, op: 20
signal node59_i0 : std_logic_vector(0 downto 0);
signal node59_i1 : std_logic_vector(8 downto 0);
signal node59_o : std_logic_vector(0 downto 0);
-- Node: 60, type: 0, op: 13
signal node60_i0 : std_logic_vector(8 downto 0);
signal node60_o : std_logic_vector(8 downto 0);
-- Node: 61, type: 4
signal node61_i0 : std_logic_vector(8 downto 0);
signal node61_o : std_logic_vector(8 downto 0);
-- Node: 62, type: 0, op: 20
signal node62_i0 : std_logic_vector(0 downto 0);
signal node62_i1 : std_logic_vector(8 downto 0);
signal node62_o : std_logic_vector(0 downto 0);
-- Node: 63, type: 0, op: 23
signal node63_i0 : std_logic_vector(0 downto 0);
signal node63_o : std_logic_vector(0 downto 0);
-- Node: 64, type: 0, op: 10
signal node64_i0 : std_logic_vector(8 downto 0);
signal node64_i1 : std_logic_vector(8 downto 0);
signal node64_o : std_logic_vector(8 downto 0);
-- Node: 65, type: 0, op: 23
signal node65_i0 : std_logic_vector(8 downto 0);
signal node65_o : std_logic_vector(0 downto 0);
-- Node: 66, type: 0, op: 20
signal node66_i0 : std_logic_vector(0 downto 0);
signal node66_i1 : std_logic_vector(0 downto 0);
signal node66_o : std_logic_vector(0 downto 0);
-- Node: 67, type: 0, op: 10
signal node67_i0 : std_logic_vector(8 downto 0);
signal node67_i1 : std_logic_vector(8 downto 0);
signal node67_o : std_logic_vector(8 downto 0);
-- Node: 68, type: 0, op: 23
signal node68_i0 : std_logic_vector(8 downto 0);
signal node68_o : std_logic_vector(0 downto 0);
-- Node: 69, type: 0, op: 20
signal node69_i0 : std_logic_vector(0 downto 0);
signal node69_i1 : std_logic_vector(0 downto 0);
signal node69_o : std_logic_vector(0 downto 0);
-- Node: 70, type: 0, op: 100
signal node70_i0 : std_logic_vector(8 downto 0);
signal node70_o : std_logic_vector(8 downto 0);
-- Node: 71, type: 4
signal node71_i0 : std_logic_vector(8 downto 0);
signal node71_o : std_logic_vector(7 downto 0);
-- Node: 72, type: 0, op: 100
signal node72_i0 : std_logic_vector(7 downto 0);
signal node72_o : std_logic_vector(7 downto 0);
-- Node: 73, type: 2
signal node73_o : std_logic_vector(0 downto 0);
-- Node: 74, type: 0, op: 1
signal node74_i0 : std_logic_vector(1 downto 0);
signal node74_i1 : std_logic_vector(0 downto 0);
signal node74_o : std_logic_vector(1 downto 0);
-- Node: 75, type: 0, op: 100
signal node75_i0 : std_logic_vector(1 downto 0);
signal node75_o : std_logic_vector(1 downto 0);
-- Node: 76, type: 0, op: 23
signal node76_i0 : std_logic_vector(8 downto 0);
signal node76_o : std_logic_vector(0 downto 0);
-- Node: 77, type: 2
signal node77_o : std_logic_vector(8 downto 0);
-- Node: 78, type: 0, op: 100
signal node78_i0 : std_logic_vector(8 downto 0);
signal node78_o : std_logic_vector(8 downto 0);
-- Node: 79, type: 0, op: 10
signal node79_i0 : std_logic_vector(8 downto 0);
signal node79_i1 : std_logic_vector(8 downto 0);
signal node79_o : std_logic_vector(8 downto 0);
-- Node: 80, type: 0, op: 10
signal node80_i0 : std_logic_vector(8 downto 0);
signal node80_i1 : std_logic_vector(8 downto 0);
signal node80_o : std_logic_vector(8 downto 0);
-- Node: 81, type: 0, op: 21
signal node81_i0 : std_logic_vector(8 downto 0);
signal node81_i1 : std_logic_vector(8 downto 0);
signal node81_o : std_logic_vector(0 downto 0);
-- Node: 82, type: 4
signal node82_i0 : std_logic_vector(8 downto 0);
signal node82_o : std_logic_vector(7 downto 0);
-- Node: 83, type: 0, op: 100
signal node83_i0 : std_logic_vector(7 downto 0);
signal node83_o : std_logic_vector(7 downto 0);
-- Node: 84, type: 0, op: 100
signal node84_i0 : std_logic_vector(8 downto 0);
signal node84_o : std_logic_vector(8 downto 0);
-- Node: 85, type: 0, op: 11
signal node85_i0 : std_logic_vector(8 downto 0);
signal node85_i1 : std_logic_vector(8 downto 0);
signal node85_o : std_logic_vector(8 downto 0);
-- Node: 86, type: 0, op: 100
signal node86_i0 : std_logic_vector(8 downto 0);
signal node86_o : std_logic_vector(8 downto 0);
-- Node: 87, type: 0, op: 100
signal node87_i0 : std_logic_vector(8 downto 0);
signal node87_o : std_logic_vector(8 downto 0);
-- Node: 88, type: 0, op: 13
signal node88_i0 : std_logic_vector(8 downto 0);
signal node88_o : std_logic_vector(8 downto 0);
-- Node: 89, type: 4
signal node89_i0 : std_logic_vector(8 downto 0);
signal node89_o : std_logic_vector(8 downto 0);
-- Node: 90, type: 0, op: 13
signal node90_i0 : std_logic_vector(8 downto 0);
signal node90_o : std_logic_vector(8 downto 0);
-- Node: 91, type: 4
signal node91_i0 : std_logic_vector(8 downto 0);
signal node91_o : std_logic_vector(8 downto 0);
-- Node: 92, type: 0, op: 20
signal node92_i0 : std_logic_vector(8 downto 0);
signal node92_i1 : std_logic_vector(8 downto 0);
signal node92_o : std_logic_vector(0 downto 0);
-- Node: 93, type: 0, op: 13
signal node93_i0 : std_logic_vector(8 downto 0);
signal node93_o : std_logic_vector(8 downto 0);
-- Node: 94, type: 4
signal node94_i0 : std_logic_vector(8 downto 0);
signal node94_o : std_logic_vector(8 downto 0);
-- Node: 95, type: 0, op: 20
signal node95_i0 : std_logic_vector(0 downto 0);
signal node95_i1 : std_logic_vector(8 downto 0);
signal node95_o : std_logic_vector(0 downto 0);
-- Node: 96, type: 0, op: 13
signal node96_i0 : std_logic_vector(8 downto 0);
signal node96_o : std_logic_vector(8 downto 0);
-- Node: 97, type: 4
signal node97_i0 : std_logic_vector(8 downto 0);
signal node97_o : std_logic_vector(8 downto 0);
-- Node: 98, type: 0, op: 20
signal node98_i0 : std_logic_vector(0 downto 0);
signal node98_i1 : std_logic_vector(8 downto 0);
signal node98_o : std_logic_vector(0 downto 0);
-- Node: 99, type: 0, op: 13
signal node99_i0 : std_logic_vector(8 downto 0);
signal node99_o : std_logic_vector(8 downto 0);
-- Node: 100, type: 4
signal node100_i0 : std_logic_vector(8 downto 0);
signal node100_o : std_logic_vector(8 downto 0);
-- Node: 101, type: 0, op: 20
signal node101_i0 : std_logic_vector(0 downto 0);
signal node101_i1 : std_logic_vector(8 downto 0);
signal node101_o : std_logic_vector(0 downto 0);
-- Node: 102, type: 0, op: 13
signal node102_i0 : std_logic_vector(8 downto 0);
signal node102_o : std_logic_vector(8 downto 0);
-- Node: 103, type: 4
signal node103_i0 : std_logic_vector(8 downto 0);
signal node103_o : std_logic_vector(8 downto 0);
-- Node: 104, type: 0, op: 20
signal node104_i0 : std_logic_vector(0 downto 0);
signal node104_i1 : std_logic_vector(8 downto 0);
signal node104_o : std_logic_vector(0 downto 0);
-- Node: 105, type: 0, op: 13
signal node105_i0 : std_logic_vector(8 downto 0);
signal node105_o : std_logic_vector(8 downto 0);
-- Node: 106, type: 4
signal node106_i0 : std_logic_vector(8 downto 0);
signal node106_o : std_logic_vector(8 downto 0);
-- Node: 107, type: 0, op: 20
signal node107_i0 : std_logic_vector(0 downto 0);
signal node107_i1 : std_logic_vector(8 downto 0);
signal node107_o : std_logic_vector(0 downto 0);
-- Node: 108, type: 0, op: 13
signal node108_i0 : std_logic_vector(8 downto 0);
signal node108_o : std_logic_vector(8 downto 0);
-- Node: 109, type: 4
signal node109_i0 : std_logic_vector(8 downto 0);
```

```
signal node109_o : std_logic_vector(8 downto 0);
-- Node: 110, type: 0, op: 20
signal node110_i0 : std_logic_vector(0 downto 0);
signal node110_i1 : std_logic_vector(8 downto 0);
signal node110_o : std_logic_vector(0 downto 0);
-- Node: 111, type: 0, op: 23
signal node111_i0 : std_logic_vector(0 downto 0);
signal node111_o : std_logic_vector(0 downto 0);
-- Node: 112, type: 0, op: 11
signal node112_i0 : std_logic_vector(8 downto 0);
signal node112_i1 : std_logic_vector(8 downto 0);
signal node112_o : std_logic_vector(8 downto 0);
-- Node: 113, type: 0, op: 13
signal node113_i0 : std_logic_vector(8 downto 0);
signal node113_o : std_logic_vector(8 downto 0);
-- Node: 114, type: 0, op: 23
signal node114_i0 : std_logic_vector(8 downto 0);
signal node114_o : std_logic_vector(0 downto 0);
-- Node: 115, type: 2
signal node115_o : std_logic_vector(0 downto 0);
-- Node: 116, type: 0, op: 100
signal node116_i0 : std_logic_vector(0 downto 0);
signal node116_o : std_logic_vector(0 downto 0);
-- Node: 117, type: 2
signal node117_o : std_logic_vector(0 downto 0);
-- Node: 118, type: 0, op: 100
signal node118_i0 : std_logic_vector(0 downto 0);
signal node118_o : std_logic_vector(0 downto 0);
-- Node: 119, type: 2
signal node119_o : std_logic_vector(0 downto 0);
-- Node: 120, type: 0, op: 100
signal node120_i0 : std_logic_vector(0 downto 0);
signal node120_o : std_logic_vector(0 downto 0);
-- Node: 121, type: 2
signal node121_o : std_logic_vector(0 downto 0);
-- Node: 122, type: 0, op: 100
signal node122_i0 : std_logic_vector(0 downto 0);
signal node122_o : std_logic_vector(0 downto 0);
-- Node: 123, type: 0, op: 13
signal node123_i0 : std_logic_vector(0 downto 0);
signal node123_o : std_logic_vector(0 downto 0);
-- Node: 124, type: 0, op: 100
signal node124_i0 : std_logic_vector(0 downto 0);
signal node124_o : std_logic_vector(0 downto 0);

component FU_100 is  --Add/Sub
    generic (order : Integer);
    port(
    A0 : in std_logic_vector((order-1) downto 0);
    A1 : in std_logic_vector((order-1) downto 0);
    Z0 : out std_logic_vector((order-1) downto 0);
    CV : in std_logic_vector(0 downto 0)
    );
end component ;
signal FU35_A0 : std_logic_vector(1 downto 0);
signal FU35_A1 : std_logic_vector(1 downto 0);
signal FU35_Z0 : std_logic_vector(1 downto 0);
signal FU35_CV : std_logic_vector(0 downto 0);

component FU_201 is  -->/<=/>=/<
    generic (order : Integer);
    port(
    A0 : in std_logic_vector((order) downto 0);
    A1 : in std_logic_vector((order) downto 0);
    Z0 : out std_logic_vector(0 downto 0);
    CV : in std_logic_vector(1 downto 0)
    );
end component ;
signal FU8_A0 : std_logic_vector(2 downto 0);
signal FU8_A1 : std_logic_vector(2 downto 0);
signal FU8_Z0 : std_logic_vector(0 downto 0);
signal FU8_CV : std_logic_vector(1 downto 0);

component FU_300 is  --BAnd
    generic (order : Integer);
    port(
    A0 : in std_logic_vector((order-1) downto 0);
    A1 : in std_logic_vector((order-1) downto 0);
    Z0 : out std_logic_vector((order-1) downto 0)
    );
end component ;
signal FU3_A0 : std_logic_vector(8 downto 0);
signal FU3_A1 : std_logic_vector(8 downto 0);
signal FU3_Z0 : std_logic_vector(8 downto 0);
signal FU5_A0 : std_logic_vector(8 downto 0);
signal FU5_A1 : std_logic_vector(8 downto 0);
signal FU5_Z0 : std_logic_vector(8 downto 0);
signal FU29_A0 : std_logic_vector(8 downto 0);
signal FU29_A1 : std_logic_vector(8 downto 0);
signal FU29_Z0 : std_logic_vector(8 downto 0);
signal FU32_A0 : std_logic_vector(8 downto 0);
signal FU32_A1 : std_logic_vector(8 downto 0);
signal FU32_Z0 : std_logic_vector(8 downto 0);
signal FU37_A0 : std_logic_vector(8 downto 0);
signal FU37_A1 : std_logic_vector(8 downto 0);
signal FU37_Z0 : std_logic_vector(8 downto 0);
signal FU38_A0 : std_logic_vector(8 downto 0);
signal FU38_A1 : std_logic_vector(8 downto 0);
signal FU38_Z0 : std_logic_vector(8 downto 0);

component FU_301 is  --BOr
    generic (order : Integer);
    port(
    A0 : in std_logic_vector((order-1) downto 0);
    A1 : in std_logic_vector((order-1) downto 0);
    Z0 : out std_logic_vector((order-1) downto 0)
    );
end component ;
signal FU7_A0 : std_logic_vector(8 downto 0);
signal FU7_A1 : std_logic_vector(8 downto 0);
signal FU7_Z0 : std_logic_vector(8 downto 0);
signal FU12_A0 : std_logic_vector(8 downto 0);
signal FU12_A1 : std_logic_vector(8 downto 0);
signal FU12_Z0 : std_logic_vector(8 downto 0);
signal FU40_A0 : std_logic_vector(8 downto 0);
signal FU40_A1 : std_logic_vector(8 downto 0);
signal FU40_Z0 : std_logic_vector(8 downto 0);
signal FU57_A0 : std_logic_vector(8 downto 0);
signal FU57_A1 : std_logic_vector(8 downto 0);
signal FU57_Z0 : std_logic_vector(8 downto 0);

component FU_303 is  --BInv
    generic (order : Integer);
    port(
    A0 : in std_logic_vector((order-1) downto 0);
    Z0 : out std_logic_vector((order-1) downto 0)
    );
end component ;
signal FU60_A0 : std_logic_vector(0 downto 0);
signal FU60_Z0 : std_logic_vector(0 downto 0);
signal FU13_A0 : std_logic_vector(8 downto 0);
signal FU13_Z0 : std_logic_vector(8 downto 0);
signal FU14_A0 : std_logic_vector(8 downto 0);
signal FU14_Z0 : std_logic_vector(8 downto 0);
signal FU16_A0 : std_logic_vector(8 downto 0);
signal FU16_Z0 : std_logic_vector(8 downto 0);
signal FU18_A0 : std_logic_vector(8 downto 0);
signal FU18_Z0 : std_logic_vector(8 downto 0);
signal FU20_A0 : std_logic_vector(8 downto 0);
signal FU20_Z0 : std_logic_vector(8 downto 0);
signal FU22_A0 : std_logic_vector(8 downto 0);
signal FU22_Z0 : std_logic_vector(8 downto 0);
signal FU24_A0 : std_logic_vector(8 downto 0);
signal FU24_Z0 : std_logic_vector(8 downto 0);
signal FU26_A0 : std_logic_vector(8 downto 0);
signal FU26_Z0 : std_logic_vector(8 downto 0);
signal FU41_A0 : std_logic_vector(8 downto 0);
signal FU41_Z0 : std_logic_vector(8 downto 0);
signal FU42_A0 : std_logic_vector(8 downto 0);
signal FU42_Z0 : std_logic_vector(8 downto 0);
signal FU44_A0 : std_logic_vector(8 downto 0);
signal FU44_Z0 : std_logic_vector(8 downto 0);
signal FU46_A0 : std_logic_vector(8 downto 0);
signal FU46_Z0 : std_logic_vector(8 downto 0);
signal FU48_A0 : std_logic_vector(8 downto 0);
signal FU48_Z0 : std_logic_vector(8 downto 0);
signal FU50_A0 : std_logic_vector(8 downto 0);
signal FU50_Z0 : std_logic_vector(8 downto 0);
signal FU52_A0 : std_logic_vector(8 downto 0);
signal FU52_Z0 : std_logic_vector(8 downto 0);
signal FU54_A0 : std_logic_vector(8 downto 0);
signal FU54_Z0 : std_logic_vector(8 downto 0);
signal FU58_A0 : std_logic_vector(8 downto 0);
signal FU58_Z0 : std_logic_vector(8 downto 0);

component FU_400 is  --LAnd
    generic (order : Integer);
    port(
    A0 : in std_logic_vector((order-1) downto 0);
    A1 : in std_logic_vector((order-1) downto 0);
    Z0 : out std_logic_vector(0 downto 0)
    );
end component ;
signal FU10_A0 : std_logic_vector(0 downto 0);
signal FU10_A1 : std_logic_vector(0 downto 0);
signal FU10_Z0 : std_logic_vector(0 downto 0);
signal FU31_A0 : std_logic_vector(0 downto 0);
signal FU31_A1 : std_logic_vector(0 downto 0);
signal FU31_Z0 : std_logic_vector(0 downto 0);
signal FU34_A0 : std_logic_vector(0 downto 0);
```

```
signal FU34_A1 : std_logic_vector(0 downto 0);
signal FU34_Z0 : std_logic_vector(0 downto 0);
signal FU15_A0 : std_logic_vector(8 downto 0);
signal FU15_A1 : std_logic_vector(8 downto 0);
signal FU15_Z0 : std_logic_vector(0 downto 0);
signal FU17_A0 : std_logic_vector(8 downto 0);
signal FU17_A1 : std_logic_vector(8 downto 0);
signal FU17_Z0 : std_logic_vector(0 downto 0);
signal FU19_A0 : std_logic_vector(8 downto 0);
signal FU19_A1 : std_logic_vector(8 downto 0);
signal FU19_Z0 : std_logic_vector(0 downto 0);
signal FU21_A0 : std_logic_vector(8 downto 0);
signal FU21_A1 : std_logic_vector(8 downto 0);
signal FU21_Z0 : std_logic_vector(0 downto 0);
signal FU23_A0 : std_logic_vector(8 downto 0);
signal FU23_A1 : std_logic_vector(8 downto 0);
signal FU23_Z0 : std_logic_vector(0 downto 0);
signal FU25_A0 : std_logic_vector(8 downto 0);
signal FU25_A1 : std_logic_vector(8 downto 0);
signal FU25_Z0 : std_logic_vector(0 downto 0);
signal FU27_A0 : std_logic_vector(8 downto 0);
signal FU27_A1 : std_logic_vector(8 downto 0);
signal FU27_Z0 : std_logic_vector(0 downto 0);
signal FU43_A0 : std_logic_vector(8 downto 0);
signal FU43_A1 : std_logic_vector(8 downto 0);
signal FU43_Z0 : std_logic_vector(0 downto 0);
signal FU45_A0 : std_logic_vector(8 downto 0);
signal FU45_A1 : std_logic_vector(8 downto 0);
signal FU45_Z0 : std_logic_vector(0 downto 0);
signal FU47_A0 : std_logic_vector(8 downto 0);
signal FU47_A1 : std_logic_vector(8 downto 0);
signal FU47_Z0 : std_logic_vector(0 downto 0);
signal FU49_A0 : std_logic_vector(8 downto 0);
signal FU49_A1 : std_logic_vector(8 downto 0);
signal FU49_Z0 : std_logic_vector(0 downto 0);
signal FU51_A0 : std_logic_vector(8 downto 0);
signal FU51_A1 : std_logic_vector(8 downto 0);
signal FU51_Z0 : std_logic_vector(0 downto 0);
signal FU53_A0 : std_logic_vector(8 downto 0);
signal FU53_A1 : std_logic_vector(8 downto 0);
signal FU53_Z0 : std_logic_vector(0 downto 0);
signal FU55_A0 : std_logic_vector(8 downto 0);
signal FU55_A1 : std_logic_vector(8 downto 0);
signal FU55_Z0 : std_logic_vector(0 downto 0);

component FU_401 is  --LOr
    generic (order : Integer);
    port(
    A0 : in std_logic_vector((order-1) downto 0);
    A1 : in std_logic_vector((order-1) downto 0);
    Z0 : out std_logic_vector(0 downto 0)
    );
end component ;
signal FU0_A0 : std_logic_vector(0 downto 0);
signal FU0_A1 : std_logic_vector(0 downto 0);
signal FU0_Z0 : std_logic_vector(0 downto 0);
signal FU4_A0 : std_logic_vector(8 downto 0);
signal FU4_A1 : std_logic_vector(8 downto 0);
signal FU4_Z0 : std_logic_vector(0 downto 0);
signal FU6_A0 : std_logic_vector(8 downto 0);
signal FU6_A1 : std_logic_vector(8 downto 0);
signal FU6_Z0 : std_logic_vector(0 downto 0);
signal FU39_A0 : std_logic_vector(8 downto 0);
signal FU39_A1 : std_logic_vector(8 downto 0);
signal FU39_Z0 : std_logic_vector(0 downto 0);

component FU_403 is  --LInv
    generic (order : Integer);
    port(
    A0 : in std_logic_vector((order-1) downto 0);
    Z0 : out std_logic_vector(0 downto 0)
    );
end component ;
signal FU1_A0 : std_logic_vector(0 downto 0);
signal FU1_Z0 : std_logic_vector(0 downto 0);
signal FU28_A0 : std_logic_vector(0 downto 0);
signal FU28_Z0 : std_logic_vector(0 downto 0);
signal FU56_A0 : std_logic_vector(0 downto 0);
signal FU56_Z0 : std_logic_vector(0 downto 0);
signal FU2_A0 : std_logic_vector(8 downto 0);
signal FU2_Z0 : std_logic_vector(0 downto 0);
signal FU9_A0 : std_logic_vector(8 downto 0);
signal FU9_Z0 : std_logic_vector(0 downto 0);
signal FU30_A0 : std_logic_vector(8 downto 0);
signal FU30_Z0 : std_logic_vector(0 downto 0);
signal FU33_A0 : std_logic_vector(8 downto 0);
signal FU33_Z0 : std_logic_vector(0 downto 0);
signal FU36_A0 : std_logic_vector(8 downto 0);
signal FU36_Z0 : std_logic_vector(0 downto 0);
signal FU59_A0 : std_logic_vector(8 downto 0);
    signal FU59_Z0 : std_logic_vector(0 downto 0);

    component FU_500 is  --Tern
        generic (order : Integer);
        port(
        A0 : in std_logic_vector((order-1) downto 0);
        A1 : in std_logic_vector((order-1) downto 0);
        A2 : in std_logic_vector((order-1) downto 0);
        Z0 : out std_logic_vector((order-1) downto 0)
        );
    end component ;
    signal FU11_A0 : std_logic_vector(8 downto 0);
    signal FU11_A1 : std_logic_vector(8 downto 0);
    signal FU11_A2 : std_logic_vector(8 downto 0);
    signal FU11_Z0 : std_logic_vector(8 downto 0);
    component reggen is
        generic( width : integer );
        port( CLK, RES, LE : in std_logic;
              D : in std_logic_vector(width-1 downto 0);
              Q : out std_logic_vector(width-1 downto 0));
    end component reggen;
    signal REG1_Q : std_logic_vector(0 downto 0);
    signal REG1_D : std_logic_vector(0 downto 0);
    signal REG1_LE : std_logic;
    signal MUXREG1 : std_logic_vector(0 downto 0);
    signal REG2_Q : std_logic_vector(0 downto 0);
    signal REG2_D : std_logic_vector(0 downto 0);
    signal REG2_LE : std_logic;
    signal MUXREG2 : std_logic_vector(0 downto 0);
    signal REG0_Q : std_logic_vector(0 downto 0);
    signal REG0_D : std_logic_vector(0 downto 0);
    signal REG0_LE : std_logic;
    signal MUXREG0 : std_logic_vector(0 downto 0);
    signal REG9_Q : std_logic_vector(1 downto 0);
    signal REG9_D : std_logic_vector(1 downto 0);
    signal REG9_LE : std_logic;
    signal MUXREG9 : std_logic_vector(0 downto 0);
    signal REG3_Q : std_logic_vector(8 downto 0);
    signal REG4_Q : std_logic_vector(8 downto 0);
    signal REG4_D : std_logic_vector(8 downto 0);
    signal REG4_LE : std_logic;
    signal MUXREG4 : std_logic_vector(0 downto 0);
    signal REG5_Q : std_logic_vector(8 downto 0);
    signal REG5_D : std_logic_vector(8 downto 0);
    signal REG5_LE : std_logic;
    signal REG6_Q : std_logic_vector(8 downto 0);
    signal REG6_D : std_logic_vector(8 downto 0);
    signal REG6_LE : std_logic;
    signal REG7_Q : std_logic_vector(8 downto 0);
    signal REG7_D : std_logic_vector(8 downto 0);
    signal REG7_LE : std_logic;
    signal MUXREG7 : std_logic_vector(0 downto 0);
    signal REG8_Q : std_logic_vector(8 downto 0);
    signal REG8_D : std_logic_vector(8 downto 0);
    signal REG8_LE : std_logic;
    signal MUXREG8 : std_logic_vector(1 downto 0);
    signal REG10_Q : std_logic_vector(8 downto 0);
    signal REG10_D : std_logic_vector(8 downto 0);
    signal REG10_LE : std_logic;
    signal MUXREG10 : std_logic_vector(1 downto 0);
    signal REG11_Q : std_logic_vector(8 downto 0);
    signal REG11_D : std_logic_vector(8 downto 0);
    signal REG11_LE : std_logic;

begin
    datapath: block

        begin

            node2_i0 <= node1_o;

            node6_i0 <= node4_o;
            node6_i1 <= node5_o;

            node7_i0 <= node6_o;

            node9_i0 <= node8_o;

            node11_i0 <= node10_o;

            node13_i0 <= node10_o;
            node13_i1 <= node12_o;

            node14_i0 <= node11_o;
            node14_i1 <= node13_o;

            node16_i0 <= node10_o;
            node16_i1 <= node15_o;

            node17_i0 <= node14_o;
```

```
node17_i1 <= node16_o;

node18_i0 <= node8_o;

node19_i0 <= node12_o;
node19_i1 <= node10_o;

node20_i0 <= node19_o;

node21_i0 <= node20_i0;


node24_i0 <= node23_o;


node27_i0 <= node26_o;


node30_i0 <= node28_o;
node30_i1 <= node29_o;

node31_i0 <= node25_o;

node32_i0 <= node30_o;
node32_i1 <= node31_o;


node34_i0 <= node33_o;

node36_i0 <= node28_o;
node36_i1 <= node15_o;
node36_i2 <= node12_o;

node37_i0 <= node35_o;
node37_i1 <= node36_o;

node38_i0 <= node37_o;

node40_i0 <= node38_i0;

node41_i0 <= node40_o;

node42_i0 <= node38_i0;

node43_i0 <= node42_o;

node44_i0 <= node41_o;
node44_i1 <= node43_o;

node45_i0 <= node38_i0;

node46_i0 <= node45_o;

node47_i0 <= node44_o;
node47_i1 <= node46_o;

node48_i0 <= node38_i0;

node49_i0 <= node48_o;

node50_i0 <= node47_o;
node50_i1 <= node49_o;

node51_i0 <= node38_i0;

node52_i0 <= node51_o;

node53_i0 <= node50_o;
node53_i1 <= node52_o;

node54_i0 <= node38_i0;

node55_i0 <= node54_o;

node56_i0 <= node53_o;
node56_i1 <= node55_o;

node57_i0 <= node38_i0;

node58_i0 <= node57_o;

node59_i0 <= node56_o;
node59_i1 <= node58_o;

node60_i0 <= node38_i0;

node61_i0 <= node60_o;

node62_i0 <= node59_o;
node62_i1 <= node61_o;

node63_i0 <= node62_o;

node64_i0 <= node15_o;
node64_i1 <= node35_o;

node65_i0 <= node64_o;

node66_i0 <= node63_o;
node66_i1 <= node65_o;

node67_i0 <= node12_o;
node67_i1 <= node35_o;

node68_i0 <= node67_o;

node69_i0 <= node66_o;
node69_i1 <= node68_o;

node70_i0 <= node35_o;

node71_i0 <= node35_o;

node72_i0 <= node71_o;


node74_i0 <= node28_o;
node74_i1 <= node73_o;

node75_i0 <= node74_o;

node76_i0 <= node25_o;


node78_i0 <= node77_o;

node79_i0 <= node35_o;
node79_i1 <= node12_o;

node80_i0 <= node35_o;
node80_i1 <= node15_o;

node81_i0 <= node79_o;
node81_i1 <= node80_o;

node82_i0 <= node35_o;

node83_i0 <= node82_o;

node84_i0 <= node35_o;

node85_i0 <= node15_o;
node85_i1 <= node25_o;

node86_i0 <= node85_o;

node87_i0 <= node86_i0;

node88_i0 <= node22_o;

node89_i0 <= node88_o;

node90_i0 <= node22_o;

node91_i0 <= node90_o;

node92_i0 <= node89_o;
node92_i1 <= node91_o;

node93_i0 <= node22_o;

node94_i0 <= node93_o;

node95_i0 <= node92_o;
node95_i1 <= node94_o;

node96_i0 <= node22_o;

node97_i0 <= node96_o;

node98_i0 <= node95_o;
node98_i1 <= node97_o;

node99_i0 <= node22_o;

node100_i0 <= node99_o;

node101_i0 <= node98_o;
node101_i1 <= node100_o;
```

```
node102_i0 <= node22_o;

node103_i0 <= node102_o;

node104_i0 <= node101_o;
node104_i1 <= node103_o;

node105_i0 <= node22_o;

node106_i0 <= node105_o;

node107_i0 <= node104_o;
node107_i1 <= node106_o;

node108_i0 <= node22_o;

node109_i0 <= node108_o;

node110_i0 <= node107_o;
node110_i1 <= node109_o;

node111_i0 <= node110_o;

node112_i0 <= node12_o;
node112_i1 <= node15_o;

node113_i0 <= node112_o;

node114_i0 <= node113_o;


node116_i0 <= node115_o;


node118_i0 <= node117_o;


node120_i0 <= node119_o;


node122_i0 <= node121_o;

node123_i0 <= node3_o;

node124_i0 <= node123_o;

node1_o <=  "0" ;
node2_o <= node2_i0;

node3_o <= REG0_Q;

node4_o <= REG1_Q;

node5_o <= REG2_Q;

node6_o <= FU0_Z0;

node7_o <= FU1_Z0;

node8_o <= REG3_Q;

node9_o <= node9_i0;

node10_o <= REG4_Q;

node11_o <= FU2_Z0;

node12_o <= REG5_Q;

node13_o <= FU3_Z0;

node14_o <= FU4_Z0;

node15_o <= REG6_Q;

node16_o <= FU5_Z0;

node17_o <= FU6_Z0;

node18_o <= node18_i0;

node19_o <= FU7_Z0;

node20_o <= node20_i0;

node21_o <= node21_i0;

node22_o <= REG7_Q;

node23_o <=  "0" ;

node24_o <= node24_i0;

node25_o <= REG8_Q;

node26_o <=  "0" ;
node27_o <= node27_i0;

node28_o <= REG9_Q;

node29_o <=  "10" ;
node30_o <= FU8_Z0;

node31_o <= FU9_Z0;

node32_o <= FU10_Z0;

node33_o <=  "100000000" ;
node34_o <= node34_i0;

node35_o <= REG10_Q;

node36_o <= FU11_Z0;

node37_o <= FU12_Z0;

node38_o <= node38_i0;

node39_o <= REG11_Q;

node40_o <= FU13_Z0;

node41_o <= node41_i0 and  "111000000" ;

node42_o <= FU14_Z0;

node43_o <= node43_i0 and  "000111000" ;

node44_o <= FU15_Z0;

node45_o <= FU16_Z0;

node46_o <= node46_i0 and  "000000111" ;

node47_o <= FU17_Z0;

node48_o <= FU18_Z0;

node49_o <= node49_i0 and  "100100100" ;

node50_o <= FU19_Z0;

node51_o <= FU20_Z0;

node52_o <= node52_i0 and  "010010010" ;

node53_o <= FU21_Z0;

node54_o <= FU22_Z0;

node55_o <= node55_i0 and  "001001001" ;

node56_o <= FU23_Z0;

node57_o <= FU24_Z0;

node58_o <= node58_i0 and  "100010001" ;

node59_o <= FU25_Z0;

node60_o <= FU26_Z0;

node61_o <= node61_i0 and  "001010100" ;

node62_o <= FU27_Z0;

node63_o <= FU28_Z0;

node64_o <= FU29_Z0;

node65_o <= FU30_Z0;

node66_o <= FU31_Z0;

node67_o <= FU32_Z0;

node68_o <= FU33_Z0;

node69_o <= FU34_Z0;

node70_o <= node70_i0;
```

```
node71_o <= node71_i0(8 downto 1);

node72_o <= node72_i0;

node73_o <=  "1" ;
node74_o <= FU35_Z0;

node75_o <= node75_i0;

node76_o <= FU36_Z0;

node77_o <=  "100000000" ;
node78_o <= node78_i0;

node79_o <= FU37_Z0;

node80_o <= FU38_Z0;

node81_o <= FU39_Z0;

node82_o <= node82_i0(8 downto 1);

node83_o <= node83_i0;

node84_o <= node84_i0;

node85_o <= FU40_Z0;

node86_o <= node86_i0;

node87_o <= node87_i0;

node88_o <= FU41_Z0;

node89_o <= node89_i0 and  "111000000" ;

node90_o <= FU42_Z0;

node91_o <= node91_i0 and  "000111000" ;

node92_o <= FU43_Z0;

node93_o <= FU44_Z0;

node94_o <= node94_i0 and  "000000111" ;

node95_o <= FU45_Z0;

node96_o <= FU46_Z0;

node97_o <= node97_i0 and  "100100100" ;

node98_o <= FU47_Z0;

node99_o <= FU48_Z0;

node100_o <= node100_i0 and  "010010010" ;

node101_o <= FU49_Z0;

node102_o <= FU50_Z0;

node103_o <= node103_i0 and  "001001001" ;

node104_o <= FU51_Z0;

node105_o <= FU52_Z0;

node106_o <= node106_i0 and  "100010001" ;

node107_o <= FU53_Z0;

node108_o <= FU54_Z0;

node109_o <= node109_i0 and  "001010100" ;

node110_o <= FU55_Z0;

node111_o <= FU56_Z0;

node112_o <= FU57_Z0;

node113_o <= FU58_Z0;

node114_o <= FU59_Z0;

node115_o <=  "1" ;
node116_o <= node116_i0;

node117_o <=  "1" ;
node118_o <= node118_i0;

node119_o <=  "1" ;
node120_o <= node120_i0;

node121_o <=  "1" ;
node122_o <= node122_i0;

node123_o <= FU60_Z0;

node124_o <= node124_i0;


LFU35: component FU_100
    generic map( 2 )
    port map( FU35_A0,FU35_A1,FU35_Z0,FU35_CV);

FU35_A0 <= node74_i0;

FU35_A1 <= Ext(node74_i1,2);

LFU8: component FU_201
    generic map( 2 )
    port map( FU8_A0,FU8_A1,FU8_Z0,FU8_CV);

FU8_A0 <= Ext(node30_i0,3);

FU8_A1 <= Ext(node30_i1,3);

LFU3: component FU_300
    generic map( 9 )
    port map( FU3_A0,FU3_A1,FU3_Z0);

FU3_A0 <= node13_i0;

FU3_A1 <= node13_i1;

LFU5: component FU_300
    generic map( 9 )
    port map( FU5_A0,FU5_A1,FU5_Z0);

FU5_A0 <= node16_i0;

FU5_A1 <= node16_i1;

LFU29: component FU_300
    generic map( 9 )
    port map( FU29_A0,FU29_A1,FU29_Z0);

FU29_A0 <= node64_i0;

FU29_A1 <= node64_i1;

LFU32: component FU_300
    generic map( 9 )
    port map( FU32_A0,FU32_A1,FU32_Z0);

FU32_A0 <= node67_i0;

FU32_A1 <= node67_i1;

LFU37: component FU_300
    generic map( 9 )
    port map( FU37_A0,FU37_A1,FU37_Z0);

FU37_A0 <= node79_i0;

FU37_A1 <= node79_i1;

LFU38: component FU_300
    generic map( 9 )
    port map( FU38_A0,FU38_A1,FU38_Z0);

FU38_A0 <= node80_i0;

FU38_A1 <= node80_i1;

LFU7: component FU_301
    generic map( 9 )
    port map( FU7_A0,FU7_A1,FU7_Z0);

FU7_A0 <= node19_i0;

FU7_A1 <= node19_i1;

LFU12: component FU_301
    generic map( 9 )
    port map( FU12_A0,FU12_A1,FU12_Z0);
```

```
FU12_A0 <= node37_i0;

FU12_A1 <= node37_i1;

LFU40: component FU_301
    generic map( 9 )
    port map( FU40_A0,FU40_A1,FU40_Z0);

FU40_A0 <= node85_i0;

FU40_A1 <= node85_i1;

LFU57: component FU_301
    generic map( 9 )
    port map( FU57_A0,FU57_A1,FU57_Z0);

FU57_A0 <= node112_i0;

FU57_A1 <= node112_i1;

LFU60: component FU_303
    generic map( 1 )
    port map( FU60_A0,FU60_Z0);

FU60_A0 <= Sxt(node123_i0,1);

LFU13: component FU_303
    generic map( 9 )
    port map( FU13_A0,FU13_Z0);

FU13_A0 <= node40_i0;

LFU14: component FU_303
    generic map( 9 )
    port map( FU14_A0,FU14_Z0);

FU14_A0 <= node42_i0;

LFU16: component FU_303
    generic map( 9 )
    port map( FU16_A0,FU16_Z0);

FU16_A0 <= node45_i0;

LFU18: component FU_303
    generic map( 9 )
    port map( FU18_A0,FU18_Z0);

FU18_A0 <= node48_i0;

LFU20: component FU_303
    generic map( 9 )
    port map( FU20_A0,FU20_Z0);

FU20_A0 <= node51_i0;

LFU22: component FU_303
    generic map( 9 )
    port map( FU22_A0,FU22_Z0);

FU22_A0 <= node54_i0;

LFU24: component FU_303
    generic map( 9 )
    port map( FU24_A0,FU24_Z0);

FU24_A0 <= node57_i0;

LFU26: component FU_303
    generic map( 9 )
    port map( FU26_A0,FU26_Z0);

FU26_A0 <= node60_i0;

LFU41: component FU_303
    generic map( 9 )
    port map( FU41_A0,FU41_Z0);

FU41_A0 <= node88_i0;

LFU42: component FU_303
    generic map( 9 )
    port map( FU42_A0,FU42_Z0);

FU42_A0 <= node90_i0;

LFU44: component FU_303
    generic map( 9 )
    port map( FU44_A0,FU44_Z0);

FU44_A0 <= node93_i0;

LFU46: component FU_303
    generic map( 9 )
    port map( FU46_A0,FU46_Z0);

FU46_A0 <= node96_i0;

LFU48: component FU_303
    generic map( 9 )
    port map( FU48_A0,FU48_Z0);

FU48_A0 <= node99_i0;

LFU50: component FU_303
    generic map( 9 )
    port map( FU50_A0,FU50_Z0);

FU50_A0 <= node102_i0;

LFU52: component FU_303
    generic map( 9 )
    port map( FU52_A0,FU52_Z0);

FU52_A0 <= node105_i0;

LFU54: component FU_303
    generic map( 9 )
    port map( FU54_A0,FU54_Z0);

FU54_A0 <= node108_i0;

LFU58: component FU_303
    generic map( 9 )
    port map( FU58_A0,FU58_Z0);

FU58_A0 <= node113_i0;

LFU10: component FU_400
    generic map( 1 )
    port map( FU10_A0,FU10_A1,FU10_Z0);

FU10_A0 <= node32_i0;

FU10_A1 <= node32_i1;

LFU31: component FU_400
    generic map( 1 )
    port map( FU31_A0,FU31_A1,FU31_Z0);

FU31_A0 <= node66_i0;

FU31_A1 <= node66_i1;

LFU34: component FU_400
    generic map( 1 )
    port map( FU34_A0,FU34_A1,FU34_Z0);

FU34_A0 <= node69_i0;

FU34_A1 <= node69_i1;

LFU15: component FU_400
    generic map( 9 )
    port map( FU15_A0,FU15_A1,FU15_Z0);

FU15_A0 <= node44_i0;

FU15_A1 <= node44_i1;

LFU17: component FU_400
    generic map( 9 )
    port map( FU17_A0,FU17_A1,FU17_Z0);

FU17_A0 <= Ext(node47_i0,9);

FU17_A1 <= node47_i1;

LFU19: component FU_400
    generic map( 9 )
    port map( FU19_A0,FU19_A1,FU19_Z0);

FU19_A0 <= Ext(node50_i0,9);

FU19_A1 <= node50_i1;

LFU21: component FU_400
    generic map( 9 )
    port map( FU21_A0,FU21_A1,FU21_Z0);
```

```
FU21_A0 <= Ext(node53_i0,9);

FU21_A1 <= node53_i1;

LFU23: component FU_400
    generic map( 9 )
    port map( FU23_A0,FU23_A1,FU23_Z0);

FU23_A0 <= Ext(node56_i0,9);

FU23_A1 <= node56_i1;

LFU25: component FU_400
    generic map( 9 )
    port map( FU25_A0,FU25_A1,FU25_Z0);

FU25_A0 <= Ext(node59_i0,9);

FU25_A1 <= node59_i1;

LFU27: component FU_400
    generic map( 9 )
    port map( FU27_A0,FU27_A1,FU27_Z0);

FU27_A0 <= Ext(node62_i0,9);

FU27_A1 <= node62_i1;

LFU43: component FU_400
    generic map( 9 )
    port map( FU43_A0,FU43_A1,FU43_Z0);

FU43_A0 <= node92_i0;

FU43_A1 <= node92_i1;

LFU45: component FU_400
    generic map( 9 )
    port map( FU45_A0,FU45_A1,FU45_Z0);

FU45_A0 <= Ext(node95_i0,9);

FU45_A1 <= node95_i1;

LFU47: component FU_400
    generic map( 9 )
    port map( FU47_A0,FU47_A1,FU47_Z0);

FU47_A0 <= Ext(node98_i0,9);

FU47_A1 <= node98_i1;

LFU49: component FU_400
    generic map( 9 )
    port map( FU49_A0,FU49_A1,FU49_Z0);

FU49_A0 <= Ext(node101_i0,9);

FU49_A1 <= node101_i1;

LFU51: component FU_400
    generic map( 9 )
    port map( FU51_A0,FU51_A1,FU51_Z0);

FU51_A0 <= Ext(node104_i0,9);

FU51_A1 <= node104_i1;

LFU53: component FU_400
    generic map( 9 )
    port map( FU53_A0,FU53_A1,FU53_Z0);

FU53_A0 <= Ext(node107_i0,9);

FU53_A1 <= node107_i1;

LFU55: component FU_400
    generic map( 9 )
    port map( FU55_A0,FU55_A1,FU55_Z0);

FU55_A0 <= Ext(node110_i0,9);

FU55_A1 <= node110_i1;

LFU0: component FU_401
    generic map( 1 )
    port map( FU0_A0,FU0_A1,FU0_Z0);

FU0_A0 <= node6_i0;

FU0_A1 <= node6_i1;

LFU4: component FU_401
    generic map( 9 )
    port map( FU4_A0,FU4_A1,FU4_Z0);

FU4_A0 <= Ext(node14_i0,9);

FU4_A1 <= node14_i1;

LFU6: component FU_401
    generic map( 9 )
    port map( FU6_A0,FU6_A1,FU6_Z0);

FU6_A0 <= Ext(node17_i0,9);

FU6_A1 <= node17_i1;

LFU39: component FU_401
    generic map( 9 )
    port map( FU39_A0,FU39_A1,FU39_Z0);

FU39_A0 <= node81_i0;

FU39_A1 <= node81_i1;

LFU1: component FU_403
    generic map( 1 )
    port map( FU1_A0,FU1_Z0);

FU1_A0 <= node7_i0;

LFU28: component FU_403
    generic map( 1 )
    port map( FU28_A0,FU28_Z0);

FU28_A0 <= node63_i0;

LFU56: component FU_403
    generic map( 1 )
    port map( FU56_A0,FU56_Z0);

FU56_A0 <= node111_i0;

LFU2: component FU_403
    generic map( 9 )
    port map( FU2_A0,FU2_Z0);

FU2_A0 <= node11_i0;

LFU9: component FU_403
    generic map( 9 )
    port map( FU9_A0,FU9_Z0);

FU9_A0 <= node31_i0;

LFU30: component FU_403
    generic map( 9 )
    port map( FU30_A0,FU30_Z0);

FU30_A0 <= node65_i0;

LFU33: component FU_403
    generic map( 9 )
    port map( FU33_A0,FU33_Z0);

FU33_A0 <= node68_i0;

LFU36: component FU_403
    generic map( 9 )
    port map( FU36_A0,FU36_Z0);

FU36_A0 <= node76_i0;

LFU59: component FU_403
    generic map( 9 )
    port map( FU59_A0,FU59_Z0);

FU59_A0 <= node114_i0;

LFU11: component FU_500
    generic map( 9 )
    port map( FU11_A0,FU11_A1,FU11_A2,FU11_Z0);

FU11_A0 <= Ext(node36_i0,9);

FU11_A1 <= node36_i1;

FU11_A2 <= node36_i2;
```

```
LREG1: component reggen
    generic map( 1 )
    port map( clk, res, REG1_LE, REG1_D, REG1_Q);
EXT_owin <= REG1_Q;
with MUXREG1 select
REG1_D <=
    node116_o when  "0" ,
    node120_o when  "1" ,
    node120_o when others
;

LREG2: component reggen
    generic map( 1 )
    port map( clk, res, REG2_LE, REG2_D, REG2_Q);
EXT_xwin <= REG2_Q;
with MUXREG2 select
REG2_D <=
    node118_o when  "0" ,
    node122_o when  "1" ,
    node122_o when others
;

LREG0: component reggen
    generic map( 1 )
    port map( clk, res, REG0_LE, REG0_D, REG0_Q);
with MUXREG0 select
REG0_D <=
    Ext(node2_o,1) when  "0" ,
    node124_o when  "1" ,
    node124_o when others
;

LREG9: component reggen
    generic map( 2 )
    port map( clk, res, REG9_LE, REG9_D, REG9_Q);
with MUXREG9 select
REG9_D <=
    Ext(node27_o,2) when  "0" ,
    node75_o when  "1" ,
    node75_o when others
;

REG3_Q <= EXT_pos;
LREG4: component reggen
    generic map( 9 )
    port map( clk, res, REG4_LE, REG4_D, REG4_Q);
with MUXREG4 select
REG4_D <=
    node9_o when  "0" ,
    node18_o when  "1" ,
    node18_o when others
;

LREG5: component reggen
    generic map( 9 )
    port map( clk, res, REG5_LE, REG5_D, REG5_Q);
EXT_os <= REG5_Q;
REG5_D <= node20_o;

LREG6: component reggen
    generic map( 9 )
    port map( clk, res, REG6_LE, REG6_D, REG6_Q);
EXT_xs <= REG6_Q;
REG6_D <= node86_o;

LREG7: component reggen
    generic map( 9 )
    port map( clk, res, REG7_LE, REG7_D, REG7_Q);
EXT_temp <= REG7_Q;
with MUXREG7 select
REG7_D <=
    node21_o when  "0" ,
    node87_o when  "1" ,
    node87_o when others
;

LREG8: component reggen
    generic map( 9 )
    port map( clk, res, REG8_LE, REG8_D, REG8_Q);
EXT_move <= REG8_Q;
with MUXREG8 select
REG8_D <=
    Ext(node24_o,9) when  "00" ,
    node70_o when  "01" ,
    node84_o when  "10" ,
    node84_o when others
;

LREG10: component reggen
    generic map( 9 )
            port map( clk, res, REG10_LE, REG10_D, REG10_Q);
        EXT_aitemp <= REG10_Q;
        with MUXREG10 select
        REG10_D <=
            node34_o when  "00" ,
            Ext(node72_o,9) when  "01" ,
            node78_o when  "10" ,
            Ext(node83_o,9) when  "11" ,
            Ext(node83_o,9) when others
        ;

        LREG11: component reggen
            generic map( 9 )
            port map( clk, res, REG11_LE, REG11_D, REG11_Q);
        EXT_aitemp2 <= REG11_Q;
        REG11_D <= node38_o;

end block datapath;
controller: block
    type state_type is (SEnd, S1, S2, S6, S8, S9,
          S10, S17, S18, S22, S24, S29, S30, S31,
          S33, S34, S60, S61, S67, S68, S69, S74,
          S75, S97, S98, S103, S107, S108);
    signal curState : state_type;
    signal nextState : state_type;

    begin
        process (clk, res)
        begin
            if res = '0' then
                curState <= S1;
            elsif clk = '1'  and clk'event then
                curState <= nextState;
            end if ;
        end process ;
        process (curState,
        node7_o,
        node3_o,
        node17_o,
        node32_o,
        node35_o,
        node69_o,
        node76_o,
        node81_o,
        node111_o,
        node114_o,
        node3_o)
        begin
            FU35_CV <=  "X" ;
            FU8_CV <=  "XX" ;
            REG1_LE <=  '0' ;
            MUXREG1 <=  "X" ;
            REG2_LE <=  '0' ;
            MUXREG2 <=  "X" ;
            REG0_LE <=  '0' ;
            MUXREG0 <=  "X" ;
            REG9_LE <=  '0' ;
            MUXREG9 <=  "X" ;
            REG4_LE <=  '0' ;
            MUXREG4 <=  "X" ;
            REG5_LE <=  '0' ;
            REG6_LE <=  '0' ;
            REG7_LE <=  '0' ;
            MUXREG7 <=  "X" ;
            REG8_LE <=  '0' ;
            MUXREG8 <=  "XX" ;
            REG10_LE <=  '0' ;
            MUXREG10 <=  "XX" ;
            REG11_LE <=  '0' ;

            case curState is
                when S1 =>
                    nextState <= S2;
                    REG0_LE <=  '1' ;
                    MUXREG0 <=  "0" ;
                when S2 =>
                    if or_reduce(node7_o) = '1' then
                        if or_reduce(node3_o) = '1' then
                            nextState <= S22;
                        else
                            nextState <= S9;
                        end if ;
                    else
                        nextState <= S6;
                    end if ;
                when S6 =>
                    nextState <= SEnd;
                when S8 =>
                    if or_reduce(node111_o) = '1' then
                        if or_reduce(node3_o) = '1' then
```

```
                nextState <= S108;
            else
                nextState <= S107;
            end if ;
        else
            nextState <= S98;
        end if ;
    when S9 =>
        nextState <= S10;
        REG4_LE <=  '1' ;
        MUXREG4 <=  "0" ;
    when S10 =>
        if or_reduce(node17_o) = '1' then
            nextState <= S18;
        else
            nextState <= S17;
        end if ;
    when S17 =>
        nextState <= S8;
        REG5_LE <=  '1' ;
        REG7_LE <=  '1' ;
        MUXREG7 <=  "0" ;
    when S18 =>
        nextState <= S10;
        REG4_LE <=  '1' ;
        MUXREG4 <=  "1" ;
    when S22 =>
        nextState <= S24;
        REG9_LE <=  '1' ;
        REG8_LE <=  '1' ;
        MUXREG9 <=  "0" ;
        MUXREG8 <=  "00" ;
    when S24 =>
        if or_reduce(node32_o) = '1' then
            nextState <= S30;
        else
            nextState <= S29;
        end if ;
        FU8_CV <=  "11" ;
    when S29 =>
        if or_reduce(node76_o) = '1' then
            nextState <= S68;
        else
            nextState <= S67;
        end if ;
    when S30 =>
        nextState <= S31;
        REG10_LE <=  '1' ;
        MUXREG10 <=  "00" ;
    when S31 =>
        if or_reduce(node35_o) = '1' then
            nextState <= S34;
        else
            nextState <= S33;
        end if ;
    when S33 =>
        nextState <= S24;
        FU35_CV <=  "0" ;
        REG9_LE <=  '1' ;
        MUXREG9 <=  "1" ;
    when S34 =>
        if or_reduce(node69_o) = '1' then
            nextState <= S61;
        else
            nextState <= S60;
                    end if ;
                    REG11_LE <=  '1' ;
                when S60 =>
                    nextState <= S31;
                    REG10_LE <=  '1' ;
                    MUXREG10 <=  "01" ;
                when S61 =>
                    nextState <= S33;
                    REG8_LE <=  '1' ;
                    MUXREG8 <=  "01" ;
                when S67 =>
                    nextState <= S8;
                    REG6_LE <=  '1' ;
                    REG7_LE <=  '1' ;
                    MUXREG7 <=  "1" ;
                when S68 =>
                    nextState <= S69;
                    REG10_LE <=  '1' ;
                    MUXREG10 <=  "10" ;
                when S69 =>
                    if or_reduce(node81_o) = '1' then
                        nextState <= S75;
                    else
                        nextState <= S74;
                    end if ;
                when S74 =>
                    nextState <= S67;
                    REG8_LE <=  '1' ;
                    MUXREG8 <=  "10" ;
                when S75 =>
                    nextState <= S69;
                    REG10_LE <=  '1' ;
                    MUXREG10 <=  "11" ;
                when S97 =>
                    nextState <= S2;
                    REG0_LE <=  '1' ;
                    MUXREG0 <=  "1" ;
                when S98 =>
                    if or_reduce(node114_o) = '1' then
                        nextState <= S103;
                    else
                        nextState <= S97;
                    end if ;
                when S103 =>
                    nextState <= S97;
                    REG1_LE <=  '1' ;
                    REG2_LE <=  '1' ;
                    MUXREG1 <=  "0" ;
                    MUXREG2 <=  "0" ;
                when S107 =>
                    nextState <= S97;
                    REG1_LE <=  '1' ;
                    MUXREG1 <=  "1" ;
                when S108 =>
                    nextState <= S97;
                    REG2_LE <=  '1' ;
                    MUXREG2 <=  "1" ;
                when others =>
                    nextState <= SEnd;
            end case ;
        end process ;

end block controller;
end osandxs_main_arch;
```

Xilinx Implementation Reports

Sections of the reports generated by Xilinx during the synthesis and implementation of the noughts and crosses design are given below.

Place and Route Report

```
Release 3.3.08i - Par D.27
Sun Oct 13 16:29:46 2002
```

```
par -w -ol 2 -d 0 map.ncd osandxs.ncd osandxs.pcf
Constraints file: osandxs.pcf
Loading design for application par from file map.ncd.
   "osandxs_main" is an NCD, version 2.35, device xcs30xl, package tq144, speed
-4
Loading device for application par from file 's30xl.nph' in environment
C:/Xilinx.
Device speed data version:  FINAL 1.19 2000-02-10.
Device utilization summary:
   Number of External IOBs            67 out of 192    59%
      Flops:                          18
      Latches:                         0
   Number of IOBs driving Global Buffers    1 out of 8      12%
   Number of CLBs                     82 out of 576    14%
      Total Latches:                   0 out of 1152    0%
      Total CLB Flops:                76 out of 1152    6%
      4 input LUTs:                  141 out of 1152   12%
      3 input LUTs:                   19 out of 576     3%
   Number of BUFGLSs                   1 out of 8      12%
   Number of STARTUPs                  1 out of 1     100%
Overall effort level (-ol):   2 (set by user)
Placer effort level (-pl):    2 (set by user)
Placer cost table entry (-t): 1
Router effort level (-rl):    2 (set by user)
Starting initial Placement phase.  REAL time: 2 secs
Finished initial Placement phase.  REAL time: 2 secs
Starting Constructive Placer.  REAL time: 2 secs
Placer score = 98610
Placer score = 71970
Placer score = 68700
Placer score = 63660
Placer score = 61320
Placer score = 56400
Placer score = 54720
Placer score = 51840
Placer score = 50220
Placer score = 47070
Placer score = 46500
Placer score = 45510
Placer score = 44760
Placer score = 44280
Placer score = 42660
Placer score = 41160
Placer score = 40440
Placer score = 39960
Placer score = 39690
Placer score = 39300
Placer score = 39030
Placer score = 38880
Placer score = 38820
Placer score = 38490
Placer score = 38160
Placer score = 38040
Placer score = 37920
Finished Constructive Placer.  REAL time: 7 secs
Dumping design to file osandxs.ncd.
Starting Optimizing Placer.  REAL time: 7 secs
Optimizing
Swapped 4 comps.
Xilinx Placer [1]   37830   REAL time: 8 secs
Finished Optimizing Placer.  REAL time: 8 secs
Dumping design to file osandxs.ncd.
Total REAL time to Placer completion: 8 secs
Total CPU time to Placer completion: 7 secs
0 connection(s) routed; 615 unrouted active, 3 unrouted PWR/GND.
Starting router resource preassignment
Completed router resource preassignment. REAL time: 8 secs
Starting iterative routing.
Routing active signals.
End of iteration 1
615 successful; 0 unrouted active,
   3 unrouted PWR/GND; (0) REAL time: 11 secs
End of iteration 2
615 successful; 0 unrouted active,
   3 unrouted PWR/GND; (0) REAL time: 11 secs
Constraints are met.
Routing PWR/GND nets.
Power and ground nets completely routed.
Dumping design to file osandxs.ncd.
Starting cleanup
Improving routing.
```

```
End of cleanup iteration 1
618 successful; 0 unrouted; (0) REAL time: 19 secs
Dumping design to file osandxs.ncd.
Total REAL time: 19 secs
Total CPU  time: 17 secs
End of route.  618 routed (100.00%); 0 unrouted.
No errors found.
Completely routed.
This design was run without timing constraints.  It is likely that much better
circuit performance can be obtained by trying either or both of the following:
  - Enabling the Delay Based Cleanup router pass, if not already enabled
  - Supplying timing constraints in the input design
Total REAL time to Router completion: 19 secs
Total CPU time to Router completion: 18 secs
Generating PAR statistics.
   The Delay Summary Report
   The Score for this design is: 356
The Number of signals not completely routed for this design is: 0
   The Average Connection Delay for this design is:        2.299 ns
   The Maximum Pin Delay is:                               9.546 ns
   The Average Connection Delay on the 10 Worst Nets is:   6.352 ns
   Listing Pin Delays by value: (ns)
    d < 2.00    < d < 4.00  < d < 6.00  < d < 8.00  < d < 10.00  d >= 10.00
   _________   _________   _________   _________   _________   _________
        286         246          74          10           2           0
Dumping design to file osandxs.ncd.
All signals are completely routed.
Total REAL time to PAR completion: 20 secs
Total CPU time to PAR completion: 18 secs
Placement: Completed - No errors found.
Routing: Completed - No errors found.
PAR done.
```

Post Layout Timing Report

```
--------------------------------------------------------------------------------
Xilinx TRACE, Version D.27
Copyright (c) 1995-2000 Xilinx, Inc.  All rights reserved.
trce osandxs.ncd osandxs.pcf -e 3 -o osandxs.twr -xml osandxs_trce.xml
Design file:              osandxs.ncd
Physical constraint file: osandxs.pcf
Device,speed:             xcs30xl,-4 (FINAL 1.19 2000-02-10)
Report level:             error report
--------------------------------------------------------------------------------
WARNING:Timing:2491 - No timing constraints found, doing default enumeration.
================================================================================
Timing constraint: Default period analysis
 1170 items analyzed, 0 timing errors detected.
 Minimum period is  19.812ns.
================================================================================
================================================================================
Timing constraint: Default net enumeration
 165 items analyzed, 0 timing errors detected.
 Maximum net delay is   9.546ns.
--------------------------------------------------------------------------------
All constraints were met.
Data Sheet report:
-----------------
All values displayed in nanoseconds (ns)
Setup/Hold to clock clk
---------------+------------+------------+
               |  Setup to  |  Hold to   |
Source Pad     | clk (edge) | clk (edge) |
---------------+------------+------------+
EXT_pos<0>     |    5.700(R)|    0.000(R)|
EXT_pos<1>     |    5.700(R)|    0.000(R)|
EXT_pos<2>     |    5.700(R)|    0.000(R)|
EXT_pos<3>     |    5.700(R)|    0.000(R)|
EXT_pos<4>     |    5.700(R)|    0.000(R)|
EXT_pos<5>     |    5.700(R)|    0.000(R)|
EXT_pos<6>     |    5.700(R)|    0.000(R)|
EXT_pos<7>     |    5.700(R)|    0.000(R)|
EXT_pos<8>     |    5.700(R)|    0.000(R)|
---------------+------------+------------+
Clock clk to Pad
```

```
---------------+------------+
               | clk (edge) |
Destination Pad|   to PAD   |
---------------+------------+
EXT_aitemp2<0> |   7.571(R)|
EXT_aitemp2<1> |   7.594(R)|
EXT_aitemp2<2> |   7.578(R)|
EXT_aitemp2<3> |   7.566(R)|
EXT_aitemp2<4> |   7.619(R)|
EXT_aitemp2<5> |   7.565(R)|
EXT_aitemp2<6> |   7.579(R)|
EXT_aitemp2<7> |   7.609(R)|
EXT_aitemp2<8> |   7.596(R)|
EXT_aitemp<0>  |  12.086(R)|
EXT_aitemp<1>  |  14.457(R)|
EXT_aitemp<2>  |  10.864(R)|
EXT_aitemp<3>  |  12.304(R)|
EXT_aitemp<4>  |  10.694(R)|
EXT_aitemp<5>  |  13.348(R)|
EXT_aitemp<6>  |  10.908(R)|
EXT_aitemp<7>  |  11.044(R)|
EXT_aitemp<8>  |  10.934(R)|
EXT_move<0>    |   9.877(R)|
EXT_move<1>    |  12.106(R)|
EXT_move<2>    |  10.730(R)|
EXT_move<3>    |  10.397(R)|
EXT_move<4>    |  10.671(R)|
EXT_move<5>    |  10.663(R)|
EXT_move<6>    |  10.668(R)|
EXT_move<7>    |  12.888(R)|
EXT_move<8>    |  10.162(R)|
EXT_os<0>      |  11.947(R)|
EXT_os<1>      |  11.670(R)|
EXT_os<2>      |  10.873(R)|
EXT_os<3>      |  11.257(R)|
EXT_os<4>      |  10.463(R)|
EXT_os<5>      |  10.484(R)|
EXT_os<6>      |  11.031(R)|
EXT_os<7>      |  10.468(R)|
EXT_os<8>      |  11.671(R)|
EXT_owin<0>    |  11.476(R)|
EXT_temp<0>    |  11.684(R)|
EXT_temp<1>    |  12.233(R)|
EXT_temp<2>    |  11.434(R)|
EXT_temp<3>    |  12.339(R)|
EXT_temp<4>    |  13.885(R)|
EXT_temp<5>    |  12.713(R)|
EXT_temp<6>    |  12.374(R)|
EXT_temp<7>    |  11.732(R)|
EXT_temp<8>    |  12.496(R)|
EXT_xs<0>      |  11.798(R)|
EXT_xs<1>      |  11.999(R)|
EXT_xs<2>      |  10.162(R)|
EXT_xs<3>      |  10.482(R)|
EXT_xs<4>      |  11.171(R)|
EXT_xs<5>      |  12.686(R)|
EXT_xs<6>      |  15.005(R)|
EXT_xs<7>      |  13.371(R)|
EXT_xs<8>      |  11.241(R)|
EXT_xwin<0>    |  11.299(R)|
---------------+------------+
Clock to Setup on destination clock clk
---------------+---------+---------+---------+---------+
               | Src/Dest| Src/Dest| Src/Dest| Src/Dest|
Source Clock   |Rise/Rise|Fall/Rise|Rise/Fall|Fall/Fall|
---------------+---------+---------+---------+---------+
clk            |   19.812|         |         |         |
```

```
---------------+---------+---------+---------+---------+
Timing summary:
---------------
Timing errors: 0  Score: 0
Constraints cover 1170 paths, 165 nets, and 615 connections (100.0% coverage)
Design statistics:
   Minimum period:  19.812ns (Maximum frequency:  50.474MHz)
   Maximum net delay:   9.546ns
Analysis completed Sun Oct 13 16:30:09 2002
--------------------------------------------------------------------------------
```

Appendix B

A Brief Guide to the Source Code for Circe

Source Files

A brief description of each of the source files for Circe is given below. They are divided into the three modules of the design: the Parser, the Synthesis Engine and the Backend. A fourth group of global utility files are listed separately.

Parser Module

- ASTNode.h - The declarations for the abstract syntax tree (AST) class
- ASTNode.cpp - The implementation of the main functions of AST class
- SymbolTable.h - The declarations for the Symbol Table class
- SymbolTable.cpp - The implementation of the Symbol Table class
- scanner.l - The Flex input file (scanner)
- cadgram.y - The Bison input file (parser)

Synthesis Engine Module

- ASTNode_Graph.cpp - The implementation of the functions to convert the AST into the intermediate representation

- Bindings.h - The interface for binding and allocation algorithms
- BindingsSimp.cpp - The implementation of a simple binding and allocation algorithm
- scheduling.h - The interface for scheduling algorithms
- schedulingsimp.cpp - The implementation of a simple scheduling algorithm (the *trivial* one).
- schedulingsimpfast.cpp - The implementation of the *minimum-cycle* scheduling algorithm
- graph.h - The wrapper (wrapper for the Boost library) class used as a basis (template) for all graph classes
- graph_tempfunc.h - The wrapper template functions used for all graph classes
- cfg.h, dfg.h - Header files for control and data-flow graphs
- cfg.cpp, dfg.cpp - Implementation of specific functions for the CFG and DFG classes
- hls.h - Header file for the main functions of the Synthesis Engine
- hls.cpp - Functions for coordinating the Synthesis Engine

Backend Module

- cag.h, dpg.h - Header files for the controller and datapath graphs
- cag.cpp, dpg.cpp - Implementation of specific functions for the CAG and DPG classes
- backend.h - Header file for the main Backend functions
- backend.cpp - Function for coordinating the Backend

Global Source Files

- vhdlconst.h - Constants declared for VHDL output
- vhdlutil.cpp - Utility functions to assist in outputting VHDL
- datareader.h - Header file for configuration reading utility functions
- datareader.cpp - Utility functions to assist in reading configuration files (needed for FU information)
- global.h - Global declarations
- global.cpp - General utility functions
- main.cpp - The main program

Code Fragments

The full source code of Circe is too long to list in this document. As such, a few brief code fragments have been chosen from key areas and are included below.

Building the Intermediate Representation (IR)

The following code fragment is taken from "ASTNode_Graph.cpp" and shows part of routine used to convert the abstract syntax tree into the control and data-flow graphs.

```
bool CASTNode::buildGraphs(tGraphBuildInfo& graphInfo) const
{
    // Graph depends on what the node type is
    switch (m_type) {
        case AST_BINEXP:
        case AST_UNIEXP:
        /* -- snip -- */
        case AST_TRIEXP: {
            // vector used to store the dfg outputs
            vector <tIndex> dfgOuts(getNumChildren());
            for (int i = 0; i < getNumChildren(); i++) {
                if (getChild(i) != NULL) {
                    getChild(i)->buildFullGraphs(graphInfo);
                    dfgOuts[i] = graphInfo.out;
                }
            }
            //////////
            // CFG
            /////////
            graphInfo.addTo =
                graphInfo.cfg.addNode(CFG_OP,
                                      graphInfo.addTo);
            /////////
```

```
        // DFG
        /////////
        tIndex opNode = graphInfo.dfg.addNode(DFG_OP);
        // Set the op attribute
        graphInfo.dfg.setOp(opNode, m_op);
        setOpPrec(graphInfo, dfgOuts, opNode);
        // Connect all the inputs up
        for (int i = 0; i< getNumChildren(); i++) {
            tIndex newEdge =
                graphInfo.dfg.connectNode(dfgOuts[i],
                                          opNode);
        }
        graphInfo.out = opNode;
        // Set up the links between the op nodes
        graphInfo.cfg.setLink(graphInfo.addTo, opNode);
        graphInfo.dfg.setLink(opNode, graphInfo.addTo);
        break;
    }
   /* -- snip -- */
  case AST_IF: {
        getChild(0)->buildFullGraphs(graphInfo);
        tIndex condNode = graphInfo.out;
        graphInfo.addTo =
            graphInfo.cfg.addNode(CFG_FORK,
                                  graphInfo.addTo);
        tIndex forkNode = graphInfo.addTo;
        tIndex joinNode = graphInfo.cfg.addNode(CFG_JOIN);
        // First connection to the fork must be the
        // false (else)
        if (getChild(2) != NULL) {
            // graphInfo has 'addTo' at the fork
            getChild(2)->buildFullGraphs(graphInfo);
            graphInfo.cfg.connectNode(graphInfo.addTo,
                                      joinNode);
            graphInfo.addTo = forkNode;
        } else {
          graphInfo.cfg.connectNode(forkNode,
                                    joinNode);
        }
        if (getChild(1) != NULL) {
            // graphInfo has 'addTo' at the fork
            getChild(1)->buildFullGraphs(graphInfo);
            graphInfo.cfg.connectNode(graphInfo.addTo,
                                      joinNode);
            graphInfo.addTo = forkNode;
        } else {
          graphInfo.cfg.connectNode(forkNode, joinNode);
        }
        graphInfo.addTo = joinNode;
        // Set up the links between the op nodes
        // The link from the DFG to the CFG is from
        // the operation that produces the test output
        // to the fork node
        graphInfo.cfg.setLink(forkNode, condNode);
        break;
    }
    case AST_DO: {
        // save stuff
        tIndex savedBreakPoint = graphInfo.breakPoint;
        tIndex savedContinuePoint = graphInfo.continuePoint;
        // Join node
        graphInfo.addTo =
            graphInfo.cfg.addNode(CFG_JOIN,
                                  graphInfo.addTo);
        tIndex joinNode = graphInfo.addTo;
        graphInfo.continuePoint = joinNode;
        // Fork node
        tIndex forkNode = graphInfo.cfg.addNode(CFG_FORK);
        // The first edge on the fork node - false edge
        // Break point join
        graphInfo.breakPoint =
            graphInfo.cfg.addNode(CFG_JOIN, forkNode);
        // link back
        graphInfo.cfg.connectNode(forkNode, joinNode);
        // c0 and c1
        getChild(0)->buildFullGraphs(graphInfo);
        getChild(1)->buildFullGraphs(graphInfo);
```

```
            tIndex condNode = graphInfo.out;
            // Link up
            graphInfo.cfg.connectNode(graphInfo.addTo, forkNode);
            graphInfo.addTo = graphInfo.breakPoint;
            //Reset break and continue
            graphInfo.breakPoint = savedBreakPoint;
            graphInfo.continuePoint = savedContinuePoint;
            // Set up the links between the op nodes
            // The link from the DFG to the CFG is from
            // the operation that produces the test output
            // to the fork node
            graphInfo.cfg.setLink(forkNode, condNode);
            break;
        }
       /* -- etc -- */
```

The *trivial* Scheduling Algorithm

The full listing of the “schedulingsimp.cpp” is given below.

```
// schedulingsimp.cpp
#include "scheduling.h"
// CSimpleScheduler
// Set the initial-fall throughs
void CSimpleScheduler::initialFallthrough:: operator()(tIndex n) {
    // For this very simple scheduler every operation can have
    // its own state except for assignments which it would be
    // silly to leave in a seperate state to the operation
    // that proceeds them
    if (cfg.getType(n) == CFG_OP) {
        if (dfg.getOp(cfg.getLink(n)) == OP_ASSIGN) {
            tIndex prev;
            if ((prev = cfg.getSinglePrev(n)) > 0) {
                // make sure we don't drop through two assignments
                // to the same variable
                tIndex prevDFG;
                tIndex nDFG;
                prevDFG = cfg.getLink(prev);
                nDFG = cfg.getLink(n);
                if (dfg.getOp(prevDFG) == OP_ASSIGN) {
                    if (dfg.getSingleNext(nDFG) == dfg.getSingleNext(prevDFG)) {
                        return;
                    }
                }
                cfg.setFallthroughBelow(prev, true);
            }
        }
    }
}
void CSimpleScheduler::performInitialSchedule(const CFuncInfo &funcInfo){
    CCFG &cfg = *funcInfo.getCFG();
    CDFG &dfg = *funcInfo.getDFG();
    cfg.numberedVisit(initialFallthrough(cfg, dfg));
}
```

The *Minimum-Cycle* Scheduling Algorithm

The full listing of the “schedulingsimpfast.cpp” is given below.

```
// schedulingsimp.cpp
#include "scheduling.h"
```

```
// This scheduler algorithm is very simple
// It forces a linear control section to be executed
// in a single clock cycle (unless Sync's are present)
// Set the initial fall-throughs
void CSimpleFastScheduler::initialFallthrough:: operator()(tIndex n) {
    // We can rely on the lower down functions to stop us
    // setting fallthroughs on forks/joins
    cfg.setFallthroughBelow(n, true);
}
void CSimpleFastScheduler::performInitialSchedule(const CFuncInfo &funcInfo){
    CCFG &cfg = *funcInfo.getCFG();
    CDFG &dfg = *funcInfo.getDFG();
    cfg.numberedVisit(initialFallthrough(cfg, dfg));
}
```

The Allocation Algorithm

The code below shows part of the allocation algorithm used. Specifically, it shows the function that finds and attaches a functional unit to a particular operation.

```
// Creates an initial allocation
CSimpleBinder::tFUBindIt CSimpleBinder::initialBind(tFUCode FUCode,
            tPrec prec, tIndex DPNode, tControlStep start, tControlStep end = -1) {
    FUBindingKey key(FUCode, prec);
    const tFULookupSpecsVal &val = FULookupSpecs[FUCode];
    // Search all current bindings of this FUCode/prec.
     // Don't share FUs if flag says not too
    if (!(gSynthFlags & SYNTH_FUNOSHARE)) {
        pair<tFUBindIt, tFUBindIt> pairi = FUMap.equal_range(key);
        while (pairi.first != pairi.second) {
            // attempt to bind to it
            // so long as the use isn't too high (if that flag is set)
            if ( !(gSynthFlags & SYNTH_FUSMARTSHARE) ||
                pairi.first->second.bindings.useCount() < val.maxShare) {
                if (pairi.first->second.bindings.fill(DPNode, start, end)) {
                    // success
                    return pairi.first;
                }
            }
            pairi.first++;
        }
    }
    // Have to add a new one
    tFUBindIt i = FUMap.insert(tFUBindingMap::value_type(key, FUBindingValue()));
    if (!i->second.bindings.fill(DPNode, start, end)) {
        internal_error(
            "CSimpleBinder::initialBind - probem binding new FUMap entry",
             0);
    }
    return i;
}
```