

**IMPROVING MEMORY AND VALIDATION SUPPORT IN FPGA  
ARCHITECTURE EXPLORATION**

by

Andrew Somerville

Bachelor of Computer Science, University of New Brunswick, 2010

A Thesis Submitted in Partial Fulfillment  
of the Requirements for the Degree of

**Masters of Computer Science**

in the Graduate Academic Unit of Computer Science

Supervisor: Kenneth B. Kent, PhD, Computer Science

Examining Board: David Bremner, PhD, Computer Science  
Gerhard W. Dueck, PhD, Computer Science  
Mary E. Kaye, MEng, Electrical and Computer Engineering

This thesis is accepted by the  
Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

April, 2013

© Andrew Somerville, 2013

## **Abstract**

The Verilog-to-Routing (VTR) computer aided design (CAD) flow provides researchers with a unique ability: to explore the properties of hypothetical field-programmable gate array (FPGA) architectures and to explore various improvements to different stages of the CAD flow. This work enhances the VTR CAD flow first by providing a high performance verification framework, as well as a verification of the elaboration stage of the CAD flow. The CAD flow is also extended to support the elaboration and exploration of soft logic memories as well as more flexible hard block memory splitting and padding. Experimental results show that these new capabilities are useful in architecture exploration. Results also show that small soft logic memories can provide superior area efficiency and critical path delay on homogenous memory architectures containing only large block memories.

## **Acknowledgements**

I would like to thank my wife, Louise, for her ongoing support and encouragement. I would also like to thank CMC Microsystems, the Natural Sciences and Engineering Research Council and the New Brunswick Innovation Foundation for their support of this research.

## List of Tables

Table 1: The 19 Large Benchmarks Presented in [10] Plus LU64PEEng, Ordered by Size .....	28
Table 2: A Summary of Issues Discovered in Odin II via Simulation .....	32
Table 3: Odin II Benchmark Compilation Times, Ordered by the Number of Nodes.....	47
Table 4: A Listing of Thousand-Vector Simulation Times for each Benchmark.....	48
Table 5: Parallel Simulation Times for One through Six Processors .....	51
Table 6: Parallel Speedups for Each Benchmark Using One Through Six Processors ....	52
Table 7: The Verilog Circuit Used in Memory Exploration.....	56

## List of Figures

Figure 1: A 2-LUT; The Four ( $2^k$ ) Configuration Inputs are Driven by the FPGA's Configuration RAM .....	3
Figure 2: A BLE; LUT Configuration and Flip-Flop Bypass are Driven by Configuration RAM .....	4
Figure 3: A CLB with a Cluster Size of 2 ( $N=2$ ), with Local Routing to the Left of the BLEs .....	5
Figure 4: A High Level View of Island-Style FPGA Architecture .....	6
Figure 5: A small area of an FPGA illustrating the layout of RAM and DSP hard blocks.	8
Figure 6: A Generic CAD Flow, from the Initial Design to the Final Bit Stream.....	9
Figure 7: The Verilog-To-Routing (VTR) CAD Flow .....	14
Figure 8: Odin II .....	16
Figure 9: An example netlist.....	17
Figure 10: (a) An Example Netlist; (b) The Order in Which the Nodes will be Computed by the Sequential Simulator; (c) Ordered Nodes Divided into Stages According to Dependencies .....	22
Figure 11: A Single Port Memory with Two Address Bits that has been Split into Two Single Port Memories with One Address Bit Each.....	43
Figure 12: A Soft Logic Single Port Memory with One Address Bit and One Data Bit..	45
Figure 13: A Soft Logic Dual Port Memory with One Address Bit and One Data Bit ....	45
Figure 14: Sequential Simulation Times Plotted Against the Number of Nodes in the Netlist.....	49
Figure 15: Average Parallel Speedup across all Benchmarks vs. Number of Processors	52

Figure 16: Silicon Area Occupied by Soft Logic Memories vs. Size..... 57

Figure 17: Areas of Various Hard Block Memories vs. Size with the Soft Logic Average  
Shown ..... 57

Figure 18: Critical Path Delay in Seconds vs. Memory Size for Soft Logic Memories... 59

## Table of Contents

Abstract .....	ii
Acknowledgements .....	iii
List of Tables .....	iv
List of Figures .....	v
List of Symbols, Nomenclature or Abbreviations .....	ix
1 Introduction .....	1
2 Background .....	3
2.1 Island-Style FPGAs .....	4
2.2 Hard Blocks .....	7
2.3 FPGA Configuration and CAD Flow .....	8
2.4 FPGA Memories .....	11
2.4.1 Memory Packing .....	12
2.4.2 Soft Logic Memories .....	13
2.5 The VTR CAD Flow .....	14
2.6 Odin II .....	15
2.7 The Odin II Simulator .....	16
3 The Performance of the Odin II Simulator .....	19
3.1 Previous Odin II Simulation Algorithm .....	19
3.2 Optimized Sequential Simulation .....	21
3.3 Letting $c$ be the number of cycles simulated, the total runtime of the original algorithm is .....	22
3.4 while the new algorithm has a total runtime of .....	22
3.5 Parallel Simulation .....	23
3.5.1 Selection of <b>n0</b> .....	25
3.5.2 Selection of OpenMP .....	25
3.6 Further Optimizations .....	26
4 Verification of Odin II .....	27
4.1 Methodology .....	27
4.2 Detection of Malformed Netlist Structures .....	30
4.3 Three Valued Logic .....	31
4.4 BLIF Reading and Verification .....	31

4.5	Results of Verification .....	32
4.5.1	Simulation Bugs .....	34
4.5.2	Elaboration Bugs .....	35
4.5.3	BLIF Bugs .....	37
4.6	Regression Testing .....	38
4.7	Compiler Performance.....	39
4.8	Current Status .....	39
5	Memory Support .....	41
5.1	Memory Instantiation .....	41
5.2	Memory Splitting and Padding.....	43
5.3	Soft Logic Memories .....	44
6	Performance Analysis .....	47
6.1	Benchmark Compilation.....	47
6.2	Sequential Simulation Results .....	48
6.3	Parallel Simulation Results.....	50
7	Memory Results and Analysis .....	54
7.1	Methodology.....	54
7.2	Soft Logic Memories vs. Hard Block Memories.....	56
8	Conclusions and Future Work.....	61
	Bibliography .....	64
	Curriculum Vitae .....	67

## **List of Symbols, Nomenclature or Abbreviations**

ABC – A logic synthesis tool, developed at Berkeley and used as part of the VTR flow

API – Application programming interface

ASIC – Application-specific integrated circuit

BLE – Basic logic element, the basic building block of a CLB within an FPGA

BLIF – Berkeley logic interchange format

CAD – Computer aided design

CLB – Cluster-based logic block, the basic programmable unit of an island-style FPGA

DSP – Digital signal processor

FCM – Field-configurable memory

FPGA – Field-programmable gate array

HDL – Hardware description language

IDE – Integrated development environment

LUT – Lookup table

ModelSim – An industry standard HDL simulation tool by Mentor Graphics

Netlist – A directed graph representing a digital circuit containing nodes, pins and nets

Odin II – An open source Verilog HDL elaboration tool which is part of the VTR flow.

OpenMP – An open API specification for adding implicit parallelism to programs written in C or FORTRAN

Pragma (OpenMP) – A type of preprocessor directive used by OpenMP to instruct the compiler to parallelize the following code block

Quartus II – A commercial FPGA IDE by Altera

RAM – Random access memory

Verilog – A popular C-like hardware description language used for describing the structure of digital circuits.

VPR – An FPGA packing, placement and routing tool which is part of the VTR CAD flow

VTR – Verilog-to-Routing, an open source FPGA CAD flow capable of targeting hypothetical FPGA architectures.

# 1 Introduction

The ability to explore hypothetical field-programmable gate array (FPGA) architectures at various stages of the computer aided design (CAD) flow is an important capability for FPGA researchers. Verilog-to-Routing (VTR) is a collection of open source CAD tools that provides a complete FPGA-targeted CAD flow from the Verilog [28] hardware description language (HDL) to routing [10], capable of exploring the potential of various improvements to different stages of the CAD flow [5, 21].

VTR consists of Odin II which performs Verilog HDL elaboration, ABC [9] which is responsible for logic synthesis and technology mapping and VPR which packs, places and routes the circuit to the target FPGA architecture [10]. This thesis details several important improvements to Odin II, the elaboration stage of VTR. These improvements span performance, reliability, and functionality.

A functional verification of Odin II will be conducted using simulation which will lay the groundwork for further work. New capabilities will be opened up in terms of memory architecture exploration with the addition of Odin II memory splitting and padding, as well as the addition of a soft logic memory elaboration. These new capabilities will be used to perform memory architecture exploration in order to determine if soft logic memory can provide comparable performance in terms of area and delay when compared with hard block memory.

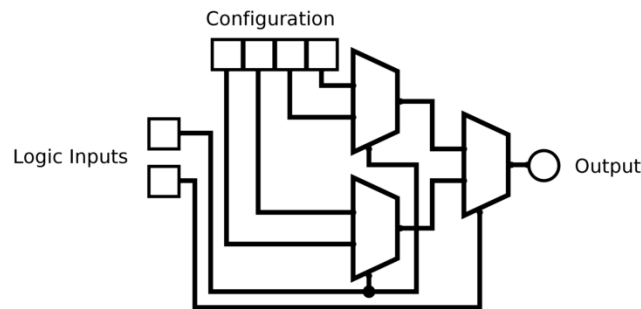
Chapter 2 details relevant background on FPGAs, VTR and Odin II. Chapter 3 contains performance improvements to the Odin II simulator, including a new parallel simulation algorithm using OpenMP. In Chapter 4 we conduct a functional verification of the Odin II compiler and simulator in order to show that the netlists (circuits) produced by Odin II are functionally correct. Chapter 5 describes improvements to memory elaboration support within Odin II, at the syntactic level as well as at the netlist level.

Chapter 6 presents the results of the performance improvements made in Chapter 3. Chapter 7 details the results of architecture explorations using Odin II's new memory splitting and padding as well as soft logic memories. Finally, Chapter 8 concludes the thesis and describes possible future work.

## 2 Background

A field-programmable gate array (FPGA) is a reconfigurable semiconductor device which is capable of being configured to behave like an arbitrary digital circuit. Modern FPGAs derive much of their programmability from the use of lookup tables (LUTs) [4, 19]. LUTs are capable of behaving like arbitrary Boolean functions, giving them great flexibility.

Figure 1 shows an implementation of a LUT with two logic inputs and one output. The configuration inputs are driven by the programmable configuration memory present within the FPGA.



**Figure 1: A 2-LUT; The Four ( $2^k$ ) Configuration Inputs are Driven by the FPGA's Configuration RAM**

LUTs may have various numbers of inputs depending on the FPGA architecture. While architectures with multi-output LUTs do exist [16], we will be concerned only with architectures consisting of single output LUTs.

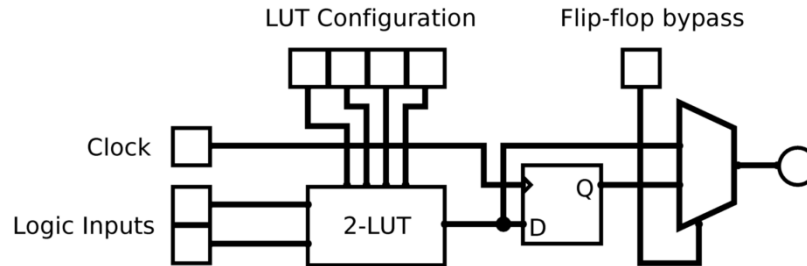
In general, a LUT with  $k$  inputs is referred to as a  $k$ -LUT. A  $k$ -LUT may implement any Boolean function of  $k$  inputs and one output [4, 19]. In [19] the effect of LUT size on

FPGA performance and density is explored, and [20] explores simultaneous (multi-objective) depth and area minimization of LUT-based FPGAs.

## 2.1 Island-Style FPGAs

This thesis will be concerned exclusively with island-style FPGAs. In island-style FPGAs, each lookup table is coupled with a flip-flop to produce a basic logic element (BLE) [4, 14]. An example BLE is depicted in Figure 2, consisting of a 2-LUT and a single flip-flop.

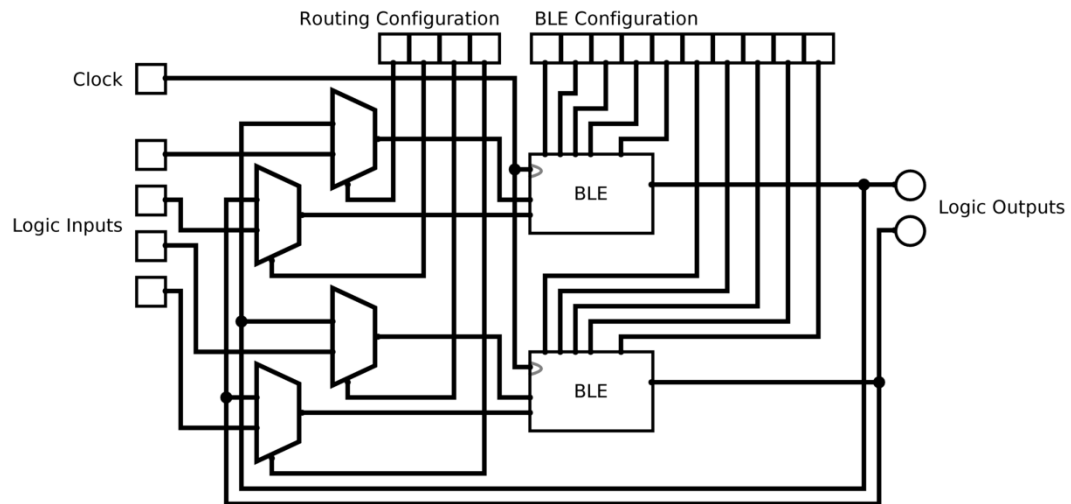
Within this BLE, the flip-flop's input is driven by the lookup table. The flip-flop may also be bypassed allowing only the LUT to contribute to the outputs. The LUT may also be configured as a pass-thru enabling the BLE to function as a simple flip-flop [19, 4].



**Figure 2: A BLE; LUT Configuration and Flip-Flop Bypass are Driven by Configuration RAM**  
 In an island-style FPGA, BLEs are clustered to form cluster-based logic blocks (CLBs). Each CLB consists of a number of BLEs connected with some local routing. The number of BLEs in each CLB,  $N$ , is referred to as the cluster size [4, 18].

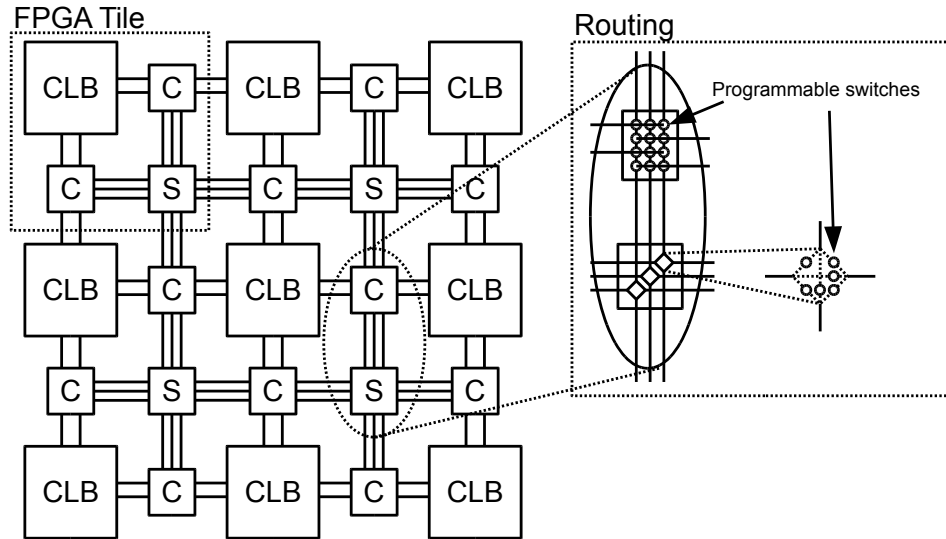
The local interconnect within each CLB allows the BLEs within to be interconnected with one another. Figure 3 shows a simple CLB with a cluster size of two. The local

routing within this CLB consists of the four multiplexers on the left. This routing configuration allows one input of each BLE to be connected to each of the outputs [14, 18].



**Figure 3: A CLB with a Cluster Size of 2 ( $N=2$ ), with Local Routing to the Left of the BLEs**

Figure 4 shows a high level view of an island-style FPGA. CLBs are arrayed within the FPGA in the grid pattern. Around each CLB there are several routing wires that are connected to the CLBs inputs and outputs via programmable switches, indicated by the letter *C* [14].



**Figure 4: A High Level View of Island-Style FPGA Architecture**

At each corner of the CLB, the routing wires terminate within grids of programmable switches, indicated by the letter *S*. These switches enable the routing from each CLB to be interconnected with the routing from other CLBs [14, 4].

Each CLB and its associated routing are collectively referred to as an FPGA tile [4].

Within an island-style FPGA the tile is replicated many times, producing a large array wherein any CLB may be connected to any other CLB [14, 4]. The routing on an FPGA accounts for most of the chip area and contributes significantly to its delay [18, 17].

An FPGA's programmable switches may be either directional, bidirectional or a mixture of the two depending on the architecture [18, 14]. In [14], it is concluded that directional single-driver wiring provides superior area and delay performance over bidirectional or mixed wiring. Switches may be implemented using tri-state buffers or a mixture of buffers and pass transistors to achieve greater area efficiency [18].

Configuration memory is also present for each FPGA tile. This memory stores the states of all configurable switches within the routing, inside the CLBs, within each BLE, as well as the configuration of each LUT [14].

At the edges of the FPGA, programmable I/O blocks allow the neighboring CLBs or routing channels to connect directly with the external pins of the FPGA [17, 14, 19].

## **2.2 Hard Blocks**

The key advantage of CLBs is their flexibility. They offer a great deal of configurability both in terms of the functions they implement and in terms of their interconnection. This flexibility comes at the cost of greater delay, area and power consumption when compared to traditional application specific integrated circuits (ASICs) [3, 13, 17].

As a result, modern FPGAs often implement commonly used design elements directly within the FPGA in order to achieve ASIC-like density, performance, and power consumption [7, 13, 17]. These fixed function areas are called hard blocks.

Common hard blocks present on modern FPGAs include memories, multipliers and digital signal processors (DSPs), all of which are common in modern designs and can otherwise occupy significant area on the FPGA [13, 17].

In [17] we see that the Altera Stratix and Stratix II both contain DSP blocks as well as various memory blocks interspersed amongst the CLBs. Smaller 512 byte and 4K distributed memories are striped vertically, while larger block RAMs are also present.

Figure 5 illustrates a small area at the corner of a hypothetical FPGA architecture. Small distributed RAMs and DSP blocks are striped vertically between the CLB tiles. Also present is a large block RAM in the lower right corner. This arrangement is similar to that of the Altera Stratix described in [17].

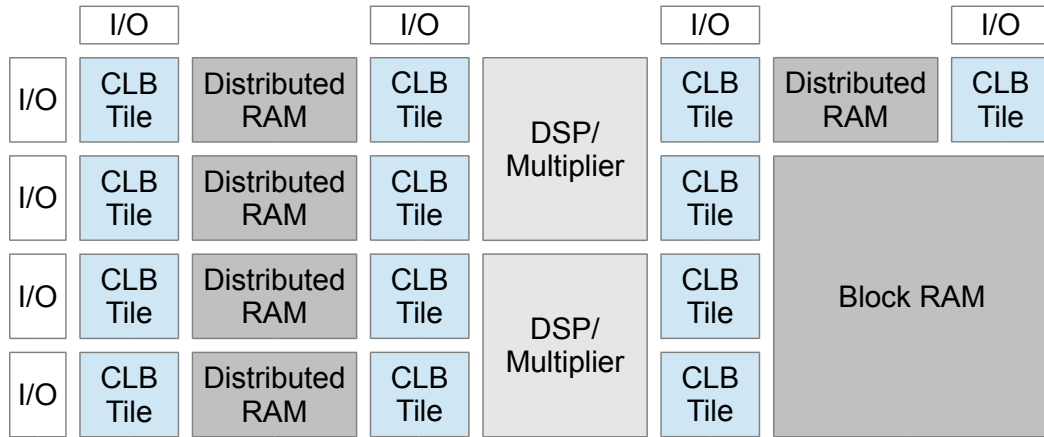


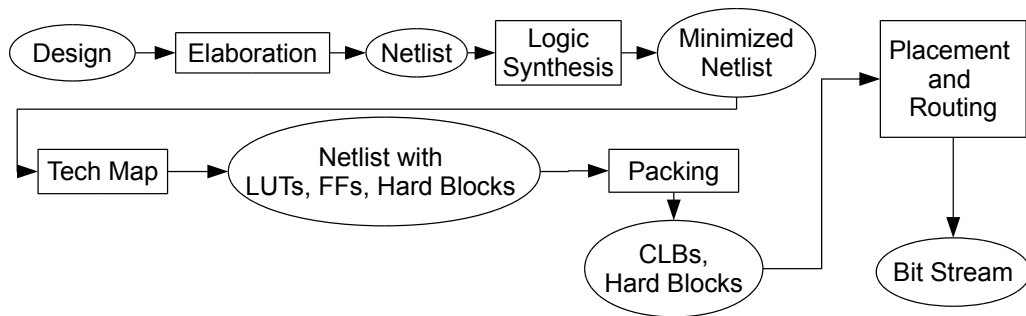
Figure 5: A small area of an FPGA illustrating the layout of RAM and DSP hard blocks.

### 2.3 FPGA Configuration and CAD Flow

The programming of FPGAs is ultimately accomplished by configuring the FPGA's programmable switches and LUTs. This process involves writing serialized programming data to the FPGA's configuration SRAM. This programming data is referred to as a bit stream and is stored in an FPGA configuration file. This configuration file is used to program the FPGA [17].

Since circuit designers work with higher level languages, several transformations must take place before the bit stream is generated. The translation of high level circuit descriptions such as those written in hardware description languages (HDLs) into an FPGA bit stream is referred to as the computer aided design (CAD) flow [3, 17, 19].

Figure 6 depicts a generic FPGA CAD flow from the high level user-generated design to the resulting bit stream. FPGA CAD flows generally include six key stages: elaboration (or design entry), synthesis (also logic synthesis or optimization), technology mapping, packing, placement, and finally routing [10, 17, 19].



**Figure 6: A Generic CAD Flow, from the Initial Design to the Final Bit Stream**

During elaboration, HDL compilation transforms the high level circuit description into a data structure called a netlist [1, 17]. This netlist provides a functional description of the circuit which directly reflects the contents of the high level design. The synthesis phase is then responsible for minimizing the netlist, producing a smaller but logically equivalent netlist [1, 19, 17].

The netlist is represented in terms of logical primitives present in a generic gate library, but may also contain architecture-specific functional units identified as being available during elaboration. These may include hard blocks such as memories or DSPs [1, 17].

Technology mapping then acts on this minimized netlist, converting all logical primitives not native to the target FPGA architecture (or the target manufacturing process in the case of ASICs) into ones which are available on the target device [17, 19]. To illustrate, the architecture in Figure 4 contains only 2-LUTs and flip-flops. Therefore the technology map needs to produce a netlist containing only those primitives.

The next phase of the CAD flow takes the technology mapped netlist and groups primitives together into their final logical structure. Again, using the architecture in Figure 4, the packed netlist would consist only of CLBs, each consisting of a cluster of two BLEs and associated internal routing. The LUTs and flip-flops are packed into the CLBs, and the routing within each CLB is configured [19].

The packed components must then be positioned on the FPGA's grid. The placement stage assigns each logic block created in the packing phase a physical location within the target FPGA. Hard blocks are also assigned to their physical counterparts during the placement phase [1, 17].

The blocks must then be interconnected via the FPGA's routing. The routing phase configures the routing switches, connecting the FPGA's logic blocks to form the final circuit [19, 17, 10].

It is now possible to program the physical FPGA. A configuration file containing the bit stream is produced which contains the information required to program the target FPGA. Some CAD flows such as VTR do not target real FPGAs but rather are used for architecture evaluation and CAD research [10, 19]. VTR therefore does not produce a bit stream but instead produces data and statistics describing the placement, routing and performance of the circuit on the hypothetical FPGA.

## **2.4 FPGA Memories**

Memory hard blocks have become ubiquitous in modern FPGAs [10, 17]. This raises questions regarding how memories are elaborated, packed, placed and routed on the FPGA.

Memories have two basic dimensions: width and depth. The width of a memory describes the number of bits contained within each word while the depth describes the number of words the memory can store.

The ratio of width to depth is referred to as the aspect ratio. Memories may also have multiple modes, meaning that a single memory block may be configurable to various

aspect ratios as in [7]. These field configurable memory (FCM) blocks are employed within FPGAs [10].

Memories are accessible via ports. The number of ports determines the number of memory words which may be addressed simultaneously. Each port may support reading, writing or both. We will deal exclusively with single and dual port memories where each port supports both reading and writing.

### **2.4.1 Memory Packing**

In order to fit into the fixed-size hard blocks which are present on modern FPGAs, logical memory blocks resulting from the elaboration of Verilog HDL circuits must be resized to the same port dimensions as one of the available modes of the physical memory hard blocks present on the FPGA.

If a logical memory block is larger than the physical memory block, it must be split into two or more logical memory blocks possessing the correct dimensions. If the logical memory is smaller than the physical hard block, it must be increased in size or padded to the size of the physical memory block.

FPGA memory blocks are also capable of being configured to act as two or more smaller memories [7]. This means that smaller memories can sometimes be packed into larger memories [2].

### 2.4.2 Soft Logic Memories

If hard block memories are unavailable on the target device or if a logic memory block is too small to merit the use of a hard block, it may be desirable to construct a memory from configurable (soft) logic instead.

An ongoing question in FPGA architecture exploration is that of the trade-off between hard and soft resources in an FPGA [8]. In [13], the trade-off between soft logic and custom hard blocks is explored in the implementation of several constructs, including memories. The paper reports that the custom vs. FPGA bit density ratio is 7 times and the delay ratio is 9 times for a two read – one write memory.

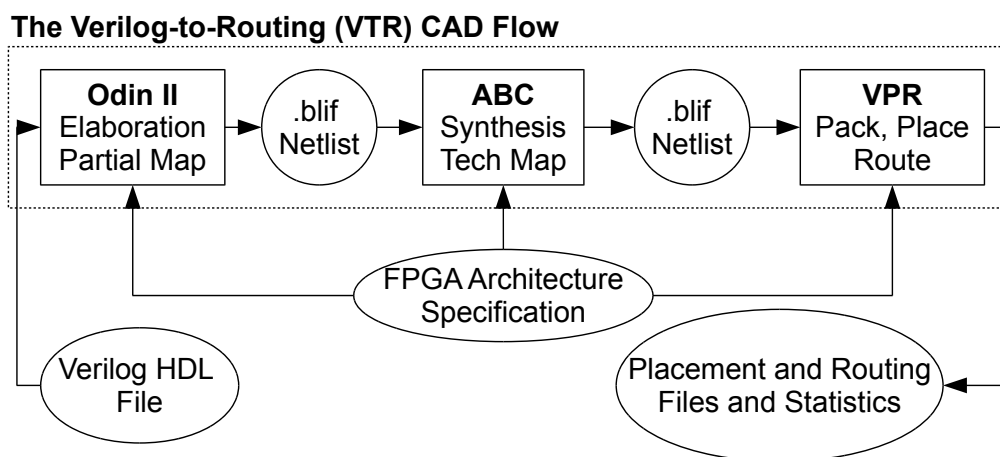
Kuon et al. detailed a comparison of ASIC performance with FPGA performance [7]. The authors report a gap between the area and critical path delay of benchmarks implemented in an FPGA versus an ASIC. However, the authors do not examine the impact of a soft versus hard memory implementation in the FPGA.

Given these results it is clear that when available, a hard block memory provides better performance over a soft logic memory. However, additional studies found that if hard circuits are not used, they are wasted (including the very expensive programmable routing that surrounds the logic) and have a negative impact on logic density [2]. Wilton showed that heterogeneous architectures containing different sizes of hard memories result in significantly denser implementations of logic circuits than architectures with only one size of memory [11].

This suggests a possible balance between usage of hard and soft memories may provide an optimal solution. It also suggests the use of soft logic may provide better area efficiency than under-utilized hard blocks, especially on homogenous architectures.

## 2.5 The VTR CAD Flow

The Verilog-to-Routing (VTR) [9] CAD flow is a publicly available FPGA CAD flow which is uniquely capable of exploring the properties of hypothetical FPGA architectures [10]. It is also used by researchers to investigate improvements to the various stages of the CAD flow [1].



**Figure 7: The Verilog-To-Routing (VTR) CAD Flow**

The VTR flow is depicted in Figure 7, and consists of Odin II, ABC, and VPR. All three tools are released open source. Odin II is a Verilog HDL elaboration (front-end synthesis) tool developed by researchers at the University of New Brunswick and the University of Miami in Ohio [1, 21]. ABC, developed at Berkeley, is a system for sequential synthesis

[6]. VPR, developed at the University of Toronto, performs packing, placement and routing to the target FPGA architecture [9, 10].

VTR uses the Berkley Logic Interchange Format (BLIF) [29] to share netlists between the three applications which make up the CAD flow. An Extensible Markup Language (XML) formatted architecture specification file is used to describe the target architecture. This file is read directly by Odin II and VPR; in the case of ABC, the architecture file is read by the VTR scripts and ABC is supplied with architecture-specific details such as the LUT size.

## **2.6 Odin II**

Figure 8 gives an overview of the internal functions of Odin II. Odin II parses Verilog HDL using a bison-based [30] parser, producing an Abstract Syntax Tree (AST). The AST is then traversed, producing a netlist using a set of supported gates and logic blocks. Supported logic blocks include memories and multipliers, as well as adders. All primitives present in this initial netlist are generic [1].

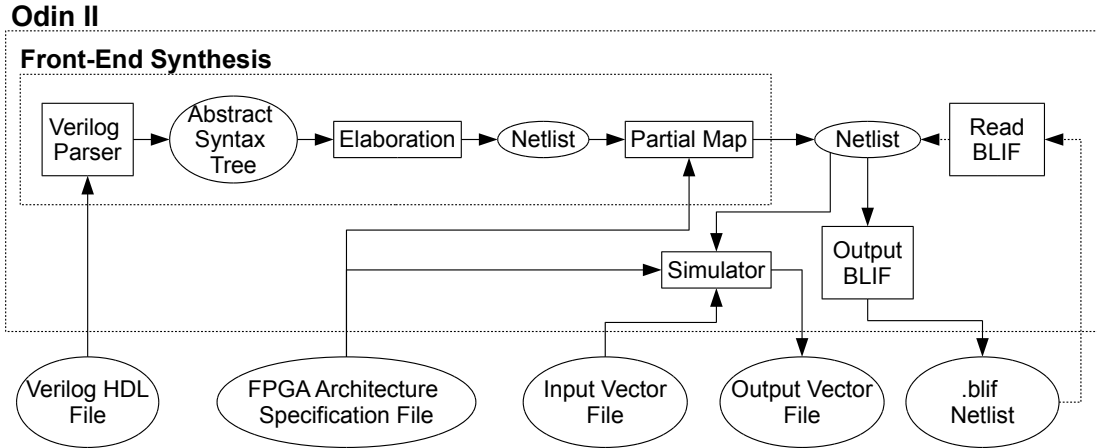


Figure 8: Odin II

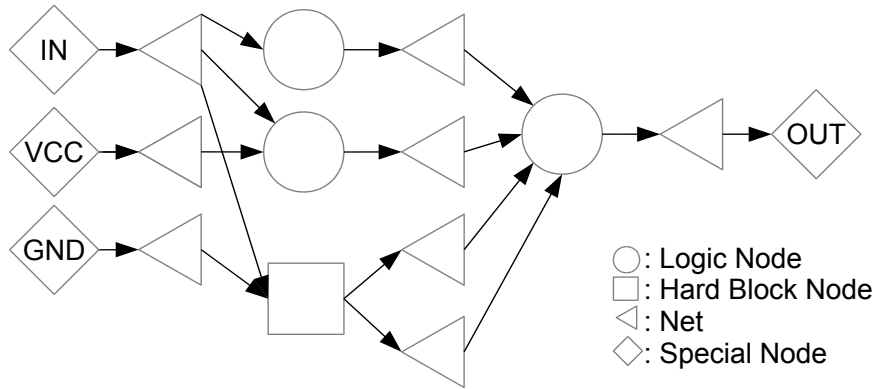
Odin II's partial map stage converts some of the generic primitives in the netlist to architecture specific ones present within the specific target architecture. Logic blocks not supported by the architecture (such as multipliers) are replaced by soft logic. Other blocks are resized (split or padded) to match those of the target architecture [1].

Odin II is capable of reading and writing netlists in BLIF format [5]. This allows Odin II to read and simulate BLIF files from other stages of the CAD flow. It also enables Odin II to be used to verify its own BLIF output which is critical to the rest of the CAD flow.

## 2.7 The Odin II Simulator

Odin II supports functional verification of the netlist through discrete event simulation [5]. The simulator reads input vectors from an input vector file (or generates them internally), propagates the values through the netlist, and writes the resulting values to the output vector file.

The simulator operates by traversing the netlist data structure. The netlist is a directed graph which contains nodes, nets, and pins. Figure 9 shows an example of how a netlist might be represented by Odin II.



**Figure 9: An example netlist.**

At the left, an input and two constant nodes represent the circuit's primary inputs. Every node except the primary output node at the right drives a single net for each output pin. Each net may drive one or more input pins to one or more nodes.

All nodes except hard block, multiplier and memory nodes have a single output pin driving a single net. Hard blocks, multipliers, and memories may have multiple output pins organized into ports.

Cycles may exist in Odin II netlists; however they always contain at least one clocked node such as a memory or a flip flop. Combinational loops consisting of cycles containing no clocked nodes are not permitted within Odin II's netlists.

Simulation occurs in discrete cycles. Each cycle involves the propagation of an input vector (a particular assignment of values to the primary inputs) through the netlist to the primary outputs. The outputs are then written to an output vector file. Each line in a vector file normally represents a single cycle, and each column represents a particular input or output.

Prior to this work, the simulation functionality of Odin II has not been significantly used or rigorously tested. The question of Odin II's functional correctness has received limited attention, with no rigorous verification of Odin II having been performed at the functional level. A previous verification effort which was presented as a proof of concept focused only on small micro benchmark circuits [5], and no effort was made to verify Odin II using larger real-world Verilog circuits. Simulation performance of Odin II is addressed in [5], but was not given significant attention.

### **3 The Performance of the Odin II Simulator**

Due to the ever-increasing complexity of circuits being used with the VTR flow [10], it is important that Odin II's verification framework be able to cope with the simulation of these circuits in a timely manner.

The performance of Odin II's simulator therefore needed to be addressed. This also provided an opportunity to address any design flaws and to perform a general audit on the algorithms and data structures used in its construction. Due to the prevalence of multi-core computer systems, parallel approaches were also prioritized [26].

#### **3.1 Previous Odin II Simulation Algorithm**

During simulation, a set of input vectors is propagated from the primary inputs through the netlist cycle by cycle to attain a set of output vectors. Each pin has an associated cycle attribute, indicating the last cycle it completed. When each node is simulated, its input pins are used to compute the values of its output pins. As each pin is calculated, its cycle attribute is advanced to indicate that it has completed the current cycle of simulation.

At the beginning of a cycle, an input vector is either read in from a file or generated randomly. If the vector is generated, it is appended to a file for reference. This vector is applied to the primary input pins of the netlist. Flip-flops are then computed using values from the previous cycle. All ready top level nodes are then added to a queue. Lastly, all ready constant nodes are added.

The nodes in the queue are then removed one by one and simulated by computing the values of their outputs and assigning those values to the output pins of the node. The cycle attribute of each pin is advanced as its value is assigned. After each node has been simulated, any nodes connected to its output pins are placed in the queue for simulation if they are not already in the queue, are ready for computation and have not already had their values computed for this cycle. After each node is added to the queue, its *in\_queue* attribute is set to prevent it from being added multiple times.

This process is repeated until there are no nodes left in the queue. After each cycle the values left on the output pins are appended to a file where they can be examined later. If there are still more input vectors to be simulated, the cycle is advanced and the next input vector is read. When there are no more input vectors, the simulator terminates.

This algorithm constitutes a breadth-first traversal of the netlist. With a netlist consisting of  $n$  nodes visited in this manner, the number of revisited nodes will depend on the degree of fan-in. Since each node not on the top-level will be visited once for every input pin, let  $d$  be the average degree of fan-in, or the average number of input pins per node.

Defined in terms of the number of connections,  $m$ ,  $d = m/n$ .

The total run time of the simulation cycle will be the run time of the sequential portion plus the run time of the parallel portion. The run time of the sequential portion depends primarily on the number of top input pins and constant nodes, which will be small compared to  $n$ , the total number of nodes in the netlist.

Assuming the initial setup has a run time of  $O(1)$ , the number of nodes visited by the algorithm will be

$$T_{Orig}^*(n, d) = O(nd)$$

which is also equal to the number of edges,  $m$ . But since each node is only calculated once, the number of nodes calculated will be  $n$ . But because the computation of each node is a relatively trivial task, the time to visit all nodes in the netlist is expected to dominate as the problem size increases.

### 3.2 Optimized Sequential Simulation

Currently, during each simulation cycle, each node is visited on average  $d$  times but is only computed once. Ideally we should be able to visit and compute each node only once. This would give us a run time complexity on the order of  $n$ .

This is made possible by the fact that for each cycle the original simulation algorithm will compute the nodes in exactly the same order. It is a simple matter to record the order in which they are computed during the first cycle, and simply compute the nodes in this order during all subsequent cycles. This amounts to a topological sort where the resulting nodes are arranged in a particular order according to their dependencies [31]. Since there are  $n$  nodes, storing the computation order has a complexity of  $O(n)$ , rendering the run time of this operation negligible when compared to the  $O(nd)$  of the traversal for  $d > 1$ .

Figure 10 (a) illustrates an example netlist. Figure 10 (b) shows the order in which the netlist is traversed. On all subsequent cycles, simulation consists of nothing more than computing the nodes in that order. Since there are  $n$  nodes, this is a linear operation with a run time of

$$T_{Opt}^*(n) = O(n).$$

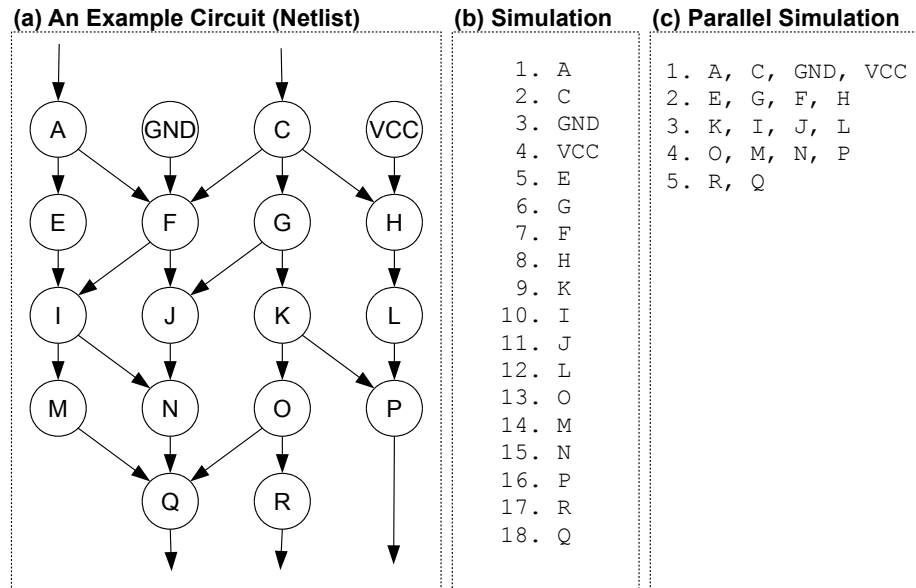


Figure 10: (a) An Example Netlist; (b) The Order in Which the Nodes will be Computed by the Sequential Simulator; (c) Ordered Nodes Divided into Stages According to Dependencies

3.3 Letting  $c$  be the number of cycles simulated, the total runtime of the original algorithm is

$$T_{Orig}^*(n, d, c) = O(c(nd))$$

3.4 while the new algorithm has a total runtime of

$$T_{Opt}^*(n, d, c) = O(nd + cn).$$

which approaches  $O(cn)$  as the number of cycles increases.

### 3.5 Parallel Simulation

Since we now have the nodes arranged in a fixed computable order as illustrated in Figure 10 (b), we can be certain that at any given point in computation that all dependencies for the current node have been met. But it may be the case that several nodes have their dependencies met simultaneously and may be computed independently of one-another.

The ordered nodes can therefore be grouped into stages: sets of nodes which are not directly related to one another. This will be referred to as staging and is accomplished by scanning our ordered list of nodes from start to finish. Each node is examined to see if it is directly connected to any other node already in the current stage. If the node has no direct relationship to any node in the current stage, it may be added. If there is a direct relationship, a new stage is created, and the node is placed into it. The result of carrying out this process on Figure 10 (b) is illustrated in Figure 10 (c).

Letting  $s$  be the average proportion of nodes added to each stage, the run time of staging the nodes will be

$$O(n \cdot snd + nd \cdot sn) \leq O(n^2 d)$$

where  $s \leq 1$ , due to the need to verify that each node is neither a child nor a parent of any node in the current stage. For large values of  $n$ , this would only pay off over a large number of simulation cycles, and only then if it provided a significant speedup for all subsequent cycles.

Two temporary hash tables are therefore employed during the construction of each stage. One contains all nodes currently in that stage, while the other contains all the children of those nodes. Assuming hash table look-ups occur in  $O(1)$  time, this leaves us with

$$O(n + n d) \in O(nd)$$

as our staging overhead, equating to one hash table look-up for each node plus one for each child of each node, where the latter dominates. As the algorithm must traverse the netlist on the first cycle anyway at a cost of  $O(nd)$ , this overhead is acceptable and will not affect the overall performance of the algorithm.

Now that our nodes have been staged, during simulation we must only ensure that each stage follows the previous one, as the nodes within each stage are not dependent on one another. We'll let  $n_0$  be the number of nodes required to merit parallel execution.

An OpenMP parallel for loop may now be employed to process any stage containing at least  $n_0$  nodes [22, 23]. The theoretical maximum speedup according to Amdahl's law will be

$$S_p(n) = \left( \frac{1}{1 - P} \right)$$

where  $P$  is the proportion of nodes in stages not smaller than  $n_0$ . Where  $P = 1$ , we can expect a parallel execution time of  $O(n/p)$  with  $p$  processors. With  $P < 1$  we get

$$T_p(n) = O\left(\frac{Pn}{p} + (1 - P)n\right)$$

corresponding to the case where  $Pn$  nodes are computed in parallel while  $(1 - P)n$  nodes must be computed sequentially. This ignores any practical considerations such as

memory bandwidth, threading overhead and load balancing, all of which will play a role in the real-world performance of the algorithm. It is also assumed that the core count will be relatively low since this software is primarily used on small shared memory computers such as desktops, workstations and laptops.

### **3.5.1 Selection of $n_0$**

Initially  $n_0$  was determined empirically to be large enough that no benchmark experienced parallel slowdown. A value of 146 was determined to meet this condition. However further experimentation revealed that a fixed value of  $n_0$  neglects the heterogeneity of the nodes, as in practice some extremely small stages were able to achieve parallel speedup.

As it was unclear what properties of a particular stage make it more amenable to parallelism, the current algorithm tests each stage's sequential and parallel execution time during the first few cycles of simulation and flags any stages which achieved parallel speedup. Stages which have been flagged are executed in parallel during all subsequent cycles.

### **3.5.2 Selection of OpenMP**

Odin II is a widely used tool which must continue to be usable on a wide range of platforms. These include GNU/Linux distributions, Microsoft Windows and Mac OS X. While low level threading libraries such as `pthread`s may sometimes provide better performance, they do not provide the portability offered by OpenMP [22, 23].

The issues of backward compatibility, readability and writability also played into the decision to use OpenMP. For the algorithm described above, a single OpenMP pragma is required to allow parallel execution. This pragma can easily be added or removed at compile time using the `_OPENMP` preprocessor constant, which is only set in the presence of OpenMP support during compilation.

### **3.6 Further Optimizations**

Since each pin connected to a net must necessarily carry the same value, it is only necessary to store the value for each net. Also, due to previously poor spatial locality, the value and cycle attributes have now been moved directly into the net structure in order to improve simulation performance. This eliminates the separate structure present in the original version of the simulator.

Similarly, to reduce the storage space required in each net, the storage type for each pin value has been changed to a `signed char` where previously an `int` was used. This potentially reduces the storage space required within each net by a factor of four.

A reduction in the number of reads and writes to disk during simulation was also achieved. During each cycle, the input and output vectors must be written and the input vectors for the next cycle must be read or written. This leads to frequent disk I/O which negatively impacts the performance of simulation. Simulation values are therefore buffered for several cycles before they are written in order to reduce the frequency of writes to disk.

## 4 Verification of Odin II

A verification of Odin II was carried out in order to demonstrate that the netlists produced by Odin II were reliable, and to show that the simulator produced correct results. It is essential that its results be rigorously verified over a broad range of circuits in order to provide maximum coverage and to maximize the chances of detecting a particular fault within the netlist or the simulator.

In [5] the authors conduct a basic proof of concept verification of Odin II using the Odin II simulator. This work provides a more extensive verification, discovering many simulation differences between Odin II and ModelSim [24]. The subsequent corrections to Odin II form a major contribution to [10].

### 4.1 Methodology

Verification of Odin II was carried out by simulating each circuit and comparing the simulation results with ModelSim [24], an industry standard HDL simulation tool. Odin II is capable of generating a ModelSim script for each circuit which tells ModelSim how to drive the inputs and clocks.

When differences were found, visualization and BLIF file examination were sometimes employed in order to isolate the differences to a particular operation or Verilog construct. Small test circuits were also used to test various hypotheses regarding the failures, and to verify that the relevant bugs had been fixed. Due to the risk of reproducing ModelSim

bugs, each case was individually considered and cross checked with other tools such as Quartus II [27].

In general, every available benchmark was employed, starting with the 19 which would eventually accompany the VTR release [10], plus one additional large benchmark not included with the release. Table 1 lists these benchmarks, including the number of nodes, connections and stages (or levels) contained within their post-elaboration netlists.

**Table 1: The 19 Large Benchmarks Presented in [10] Plus LU64PEEng, Ordered by Size**

Benchmark	Nodes	Connections	Stages
mcml	477776	1508498	663
LU64PEEng	433156	2066993	1401
LU32PEEng	217845	1019336	1337
bgm	165658	811401	353
LU8PEEng	66558	285831	1289
stereovision2	61573	188323	99
stereovision0	49395	155167	74
blob_merge	40523	170451	1436
stereovision1	39080	130605	60
mkDelayWorker32B	27629	97063	260
boundtop	14455	82935	58
orl200	13832	49144	122
mkSMAdapter4B	10279	32915	52
raygentop	9850	40357	44
sha	5647	152495	54
mkPktMerge	2503	7369	33
diffeq1	1664	4631	73
stereovision3	1501	5722	16
ch_intrinsics	1381	10698	14
diffeq2	1087	2901	73

One thousand random vectors were used to drive each benchmark, including appropriate reset signals. These benchmarks are large real-world circuits, some of which employ complex state machines in their design. Therefore it was sometimes necessary to use

custom made vectors in order to drive some of these circuits correctly. This was the case with `mcml` as well as with `sha`.

To facilitate the construction of custom made vectors, the simulator's vector parsing facilities were enhanced to allow advanced formatting of the vector files, including whitespace and comments for readability. Also added was an option to directly read in the custom input vectors by name. To speed verification of the output, an additional option was added to allow the output vectors to be compared to a user specified output vector file.

Many of the circuits also included reset signals which had to be driven correctly. To assist with testing, a feature was added to the simulator which allows the user to specify a list of active high or active low reset signals to drive during random vector generation.

In some cases the primary outputs were not very useful in diagnosing particular circuits. Access to internal nets such as those driven by state machine registers were often needed to determine if a circuit was behaving correctly and to isolate faults. An option was added to the simulator to allow an arbitrary list of items to be added to the output vector file, allowing additional nets to be observed during simulation. These items are textually matched at simulation time to named items within the netlist including nets, nodes, and pins.

In addition to the large benchmark circuits and the micro benchmarks, a set of small custom benchmark circuits was also employed in order to exhaustively test each circuit element individually. For example a binary operations benchmark was used to simultaneously check all binary operations. Simple memory benchmarks and multiplier benchmarks were also developed to test those subsystems. These were later integrated into the micro benchmark suite.

## **4.2 Detection of Malformed Netlist Structures**

Malformed netlists can sometimes be the cause of unusual behavior during simulation as well as incorrect BLIF output. During the traversal of the netlist, the simulator now detects many malformed netlist structures such as multiple drivers to the same net, areas of the netlist which fail to change value, undriven input pins and other inconsistencies.

If an area of the netlist fails to update, the simulator is capable of printing a trace, showing connected nodes and aiding the developer in isolating the cause. Failure to update some areas of the netlist points to some inputs being undriven. This could mean that certain gates are being computed incorrectly or that other problems exist within the netlist structure.

Inconsistencies in the netlist may point to incorrect use of the Odin II API in constructing the netlist. Detection of these inconsistencies allows the developer to rule them out when debugging incorrect BLIF or simulation output.

### **4.3 Three Valued Logic**

Odin II's simulation is carried out using three valued logic. The three values include the standard binary one and zero, along with the unknown value. Other tools also support the unknown value, such as ModelSim which represents it using an "x". In order for Odin II's simulation to conform to ModelSim's, all gate computations must be correct in three valued logic.

In order to facilitate testing of gates in three valued logic, an option was added to the simulator to automatically generate random logic containing the unknown value. This allowed small-scale testing of individual gates against ModelSim. In order to achieve conformity, all gates within the simulator were tested using small benchmarks to ensure that they conformed to ModelSim.

### **4.4 BLIF Reading and Verification**

In order to verify BLIF files from various stages of the VTR flow, Odin II is equipped with a BLIF reader. This BLIF reader had significant performance issues dealing with the 19 large benchmarks used in [10], and failed to read many of them at all.

Since the BLIF reader was written as a separate executable with widespread code duplication, it was first integrated into the main Odin II executable. A command line option was added to allow Odin II to load a BLIF file rather than compiling a Verilog file. This integration also allowed features such as Odin II's `graphviz` netlist visualization to work with netlists read from BLIF files.

The BLIF reader was then profiled using `gprof` to determine why it was performing poorly. Portions of the BLIF reader were rewritten in order to improve its performance and reduce its memory usage such that it could handle the large benchmarks.

In order to test the correctness of the netlists read from the BLIF files, BLIF files compiled from already verified Verilog code were used. The simulation results for each BLIF file were compared to the results from the original Verilog code for the same input vectors.

This allowed the simultaneous testing of Odin II’s BLIF output stage as well as the BLIF reader. Any differences found were then isolated by visualizing the netlists produced by reading the BLIF and by examining the BLIF. Small tests circuits were again employed to isolate the causes of particular failures.

#### 4.5 Results of Verification

Table 2: A Summary of Issues Discovered in Odin II via Simulation summarizes issues discovered within Odin II. Text in italics represents outstanding or uncorrectable issues. Separate status statements are given for the simulator. The “Odin II Status” column includes issues in parsing and elaboration as well as the BLIF file input and output stages.

**Table 2: A Summary of Issues Discovered in Odin II via Simulation**

	<b>Odin II Status</b>	<b>Simulator Status</b>
<b>Concatenation</b>	Bits reversed for all concatenated constants. All expressions produced zero when concatenated.	N/A

<b>Constants</b>	- Long constants (> 31 bits) silently turned to -1. - Constants were truncated and zero-extended incorrectly.	N/A
<b>Arithmetic and Boolean Operators</b>	- Incorrect netlists for some arithmetic cases involving constants. Constant folding optimizations were found to be at fault. - Order of operations parsed incorrectly for unary operands. - Adder carry-out discarded; changed to conform to ModelSim. - Multiple drivers failed silently. - Incorrect multiplier padding lead to multiple drivers for the same net. - BLIF LUT output was incorrect for carry functions.	- Multiply and carry functions were incorrect. <i>- Adders are structured differently and therefore cannot fully conform in three valued logic.</i>
<b>Comparisons</b>	Incorrect != and <=	
<b>Basic logic gates (AND, OR, etc...)</b>		Most dealt with the unknown value incorrectly. These were all rewritten.
<b>Memory</b>		Memories were simulated incorrectly. This had to be rewritten.
<b>Flip-Flops, Sequential Logic</b>		Driven incorrectly, producing timing differences in some cases.
<b>ModelSim Script Output</b>		Clock was driven incorrectly.
<b>BLIF Output</b>	- Incorrect or inconsistent naming leading to incorrect connections. (Eg: Split memories.) - Memory models were incorrect or missing from BLIF files.	N/A
<b>BLIF Input</b>	Incorrect for hard blocks.	<i>Differences in three valued logic for generic gates (LUTs) are unavoidable. This also affects the simulation of soft logic adders which are defined as LUTs in the BLIF file.</i>
<b>Netlist Structure</b>	Multiple drivers found in multiplier outputs due to incorrect use of the Odin II APT in padding the multipliers.	Addition of multiple consistency checks.

### 4.5.1 Simulation Bugs

The most serious bug discovered in the simulator was the incorrect handling of the unknown value by most of the basic logic functions. These functions were therefore audited, and all of them are now in conformity to ModelSim.

Memory simulation was also rewritten to conform to ModelSim. Memories will be covered in more detail in later chapters. The multiplier and adder/carry functions needed work as well to bring them into conformity.

If-else logic was also found to be handled differently by ModelSim. When simulating the resulting multiplexers in the case where some select inputs carried the unknown value, ModelSim would favor the selection which corresponded to the else clause. This was therefore implemented within Odin II, with the else clause being flagged during elaboration so that it could be correctly identified during simulation.

During the testing of some sequential circuits, it was discovered that differences arose from the failure of Odin II to compute all gates on both edges of the clock. Therefore Odin II's simulation algorithm was modified to simulate the netlist twice for each clock cycle: once for the falling edge of the clock and once more for the rising edge. Each input vector is applied to the inputs on both edges of the clock. This more realistic clocking allows values to be propagated through the netlist between flip-flop and memory updates.

This also means that the flip-flops are no longer computed as a special case at the beginning of the cycle, but are processed in-cycle. This allows Odin II to simulate flip-flops driven by both edges of the clock, although currently only rising edge flip-flops are supported.

To take advantage of this new capability, two additional options were added to Odin II. By default, Odin II now writes to the output vector file only on the falling edge. These new options allow the user to write to the output vector file on either the rising edge, or on both edges of the clock. The additional information about what the circuit is doing on each edge is useful in debugging and allows a more detailed comparison with ModelSim's output.

#### **4.5.2 Elaboration Bugs**

Constant folding optimizations were added to Odin II prior to this verification which allow constant arithmetic operations to be computed into fixed constants prior to elaboration. Through simulation it was found that several constant operations involving small integers were producing incorrect results due to a truncation bug within the constant folding code. This was subsequently corrected.

Several issues were found with netlists involving incorrect cardinality of the bits in some operations including concatenation and comparisons. Concatenation was eventually rewritten when it was discovered that it failed silently when anything but a constant was present within a concatenation. Concatenations now support arbitrary expressions.

It was also discovered that Odin II discarded the last carry from an adder while other tools such as ModelSim and Quartus II [27] append this bit to the output from the adder to be optionally captured by the designer. Odin II was enhanced to include this feature in order to conform to ModelSim. Adder output ports are now one bit wider than the widest input except in the case where the adder's output is being concatenated.

Some unary operations (such as unary OR) within expressions were found to have incorrect order of operations during parsing. The addition of parser precedence directives solved this issue.

Odin II was also found to be reading all constants into variables of type `long` prior to elaborating them. However the constant parsing functions were returning -1 in cases where the parsed constant was longer than 31 bits. These -1's were then elaborated into 2's complement constants within the resulting circuits. Support for constants was rewritten such that arbitrarily long binary, hexadecimal and octal constants are now supported. Decimal constants continue to be limited to the length of a `long long` (or a `long` on some systems), but an error is correctly issued should a decimal constant exceed the allowed length. This is considered acceptable since ModelSim limits decimal constants to only 31 bits.

Truncation and zero-extension of constants was also found to be faulty. Constants are now correctly truncated should they exceed the user-specified length. They are correspondingly zero-extended if they fall below the specified length.

Problems were also found with the netlist structure itself due to incorrect use of the Odin II API in constructing the netlist. One example of this is the padding of multiplier. The additional outputs which should have been unconnected were found to be erroneously driving one of the connected outputs. Various improvements were made to the API to facilitate the construction of valid netlist structures.

### **4.5.3 BLIF Bugs**

During the evaluation of the BLIF reading feature of Odin II, bugs were discovered both in the BLIF reader and in the BLIF output from Odin II. In most cases the netlists resulting from reading BLIF files were found to be incorrect due to bugs affecting the correct identification some of the elementary gates. It was found that the reading of hard blocks including memories and multipliers was not functioning correctly. In some cases duplicate nets were falsely detected by the BLIF reader.

An analysis revealed several design flaws in the BLIF reader. An audit involving widespread code cleanup was conducted in order to improve the maintainability and functionality of this code. The decision was also made to rewrite hard block support for the BLIF reader as well as relevant elementary logic identification code, necessary for constructing the correct netlist structures based on function definitions present within the

BLIF file. Part of this process was the addition of robust error detection which was previously absent from the BLIF reader.

Within the BLIF file, each type of hard block is accompanied by a model that defines which input and output ports the hard block is expected to have. As a result of recent enhancements and further testing, it was discovered that Odin II often produced models which did not match the hard blocks present within the circuit. This led to a restructuring of some of the memory and multiplier functionality within Odin II in order to correctly evaluate the hard blocks present within the circuit during the output stage.

It was also found that under certain circumstances the BLIF output contains incorrect names for some nets, leading to a netlist which is either unconnected or incorrectly connected. An example of this occurred when a memory node was split; the outputs were not renamed correctly, leading to an invalid BLIF file.

#### **4.6 Regression Testing**

To ensure that further developments to Odin II do not break currently working functionality, all circuits which are known to agree with ModelSim along with their known-good input and output vectors have been added to two regression tests. One consists of only small micro-benchmark circuits while the other consists of larger real-world circuits.

Each of these regression tests is accompanied by a shell script which simulates all relevant benchmarks to verify that their output vectors match those which are known to agree with ModelSim. The script accompanying the micro benchmarks also verifies the BLIF output by reading it back into Odin II and simulating it.

Additional benchmarks were added to cover new functionality as it was supported.

#### **4.7 Compiler Performance**

Some circuits were found to be taking a very long time to compile. As a result of profiling Verilog compilation in Odin II using gprof it was discovered that Odin II was spending nearly all of its time executing a bubble sort on lists of pins. When this bubble sort was replaced with a call to qsort, it was found that compilation times improved substantially.

#### **4.8 Current Status**

All supported micro-benchmarks currently agree exactly with ModelSim when simulated with 10,000 randomly generated vectors. Additionally, the BLIF files produced during their compilation also produce identical simulation results when they are read back into Odin II and simulated using the same input vectors. 14 of the 19 large benchmark circuits presented in [10] agree completely with ModelSim. The other five have minor variations [10].

Some of the full benchmarks in [10] do not produce identical results when their BLIF files are simulated. While differences are expected due to the lack of three valued logic

support in some cases while reading BLIFs as well as a lack of semantic information about if-else constructs, it is not known if these issues alone are the cause of these minor differences.

## 5 Memory Support

Improvements to the memory infrastructure of Odin II fall under three categories: improvements to memory instantiation support at the Verilog level, in splitting and padding of memories and finally in available elaboration options with the addition of soft logic memory support.

### 5.1 Memory Instantiation

Odin II previously supported the following explicit syntax for the instantiation of a single port memory.

```
single_port_ram ram(  
    .addr(address),  
    .data(data),  
    .we(write_enable),  
    .out(output),  
    .clk(clock)  
);
```

In this example, the ports `addr`, `data`, `we`, and `clk` are inputs representing the address, data to be written, write enable and clock, respectively. The port named `out` is the memory output.

Similarly, a dual port memory is instantiated as follows.

```
dual_port_ram ram(  
    .addr1(port1_address),  
    .addr2(port2_address),  
    .data1(port1_data),  
    .data2(port2_data),  
    .we1(port1_write_enable),  
    .we2(port2_write_enable),  
    .out1(port1_output),  
    .out2(port2_output),  
    .clk(clock)  
);
```

Here, a second address, data, write enable and output have been added creating a dual port memory.

Recently an implicit syntax was added to Odin II which enables a designer to specify memories using the following technique.

```
reg [`WIDTH-1:0] memory [`DEPTH-1:0];
```

A memory specified in this manner may be read from or written to from a Verilog sequential block using non-blocking assignments in the same manner as a register.

```
reg [`WIDTH-1:0] temp;
input [`WIDTH-1:0] data;
always @ (posedge clock)
begin
    memory[address] <= data;
    temp <= memory[address];
end
```

It may also be read via a continuous assignment as follows.

```
output [`WIDTH-1:0] out;
assign out = memory[address];
```

Implicit memories are synthesized as dual port RAMs, enabling the memory to be read and written simultaneously. The first port is always used for reading the memory, while the second is dedicated to writing.

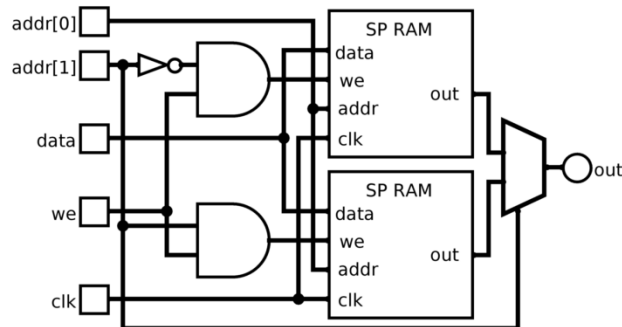
If one of the ports is unused the memory will be collapsed to a single port RAM. While the detection of this condition is currently limited to one of the ports being entirely unassigned, the possibility exists for more intelligent optimizations to detect the condition where the same address is assigned to both ports. This would require a semantic analysis

of the logic driving the address ports to determine if they are functionally equivalent and may only be realistic in simple cases.

## 5.2 Memory Splitting and Padding

The width of a memory will easily divide into a number of smaller memories, each receiving an arbitrary share of the original width. However if the logical width is not evenly divisible by the physical width, one of the resulting memories will require padding to conform to the size of the physical memory.

Alternately, this remaining memory can be converted to soft logic in order to avoid the use of an additional hard block memory. For this purpose, Odin II now has configurable width and depth cutoffs which govern the conversion of small memories into soft logic.



**Figure 11: A Single Port Memory with Two Address Bits that has been Split into Two Single Port Memories with One Address Bit Each**

When splitting memory depth, a memory with  $n$  address bits will be split into two memories with  $n-1$  address bits each. Figure 11 gives an example of a single port memory with two address bits and one data bit which has been split into two single port memories with one address bit and one data bit each. For this reason, padding of a memory's

address is only necessary if the logical memory is smaller than the available physical memory block.

Odin II received an overhaul of its memory splitting and padding capabilities which were previously limited to depth splitting and padding as well as width splits down on a single data bit. This single bit split avoided the need for any padding of the output or data ports. Odin II now has the ability to split and pad large memories and to pad smaller memories to arbitrary widths and depths in both dimensions allowing for correct memory sizing early in the CAD flow.

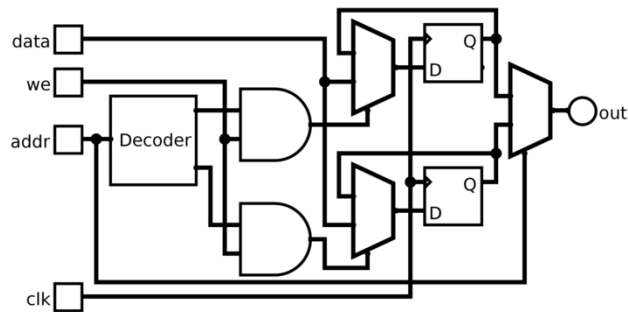
In order to facilitate architecture mapping in later stages of the CAD flow, Odin II now has configurable memory depth and width thresholds which may be read from the FPGA architecture or specified via the configuration file at runtime. During elaboration, all memories will be split or padded to conform to these depth and width thresholds.

Odin II also has the ability to split each memory down to a single data bit wide; the VTR CAD flow has the ability to repackage these small memories into one or more large memory blocks. This provides the packer with some additional flexibility [10].

### **5.3 Soft Logic Memories**

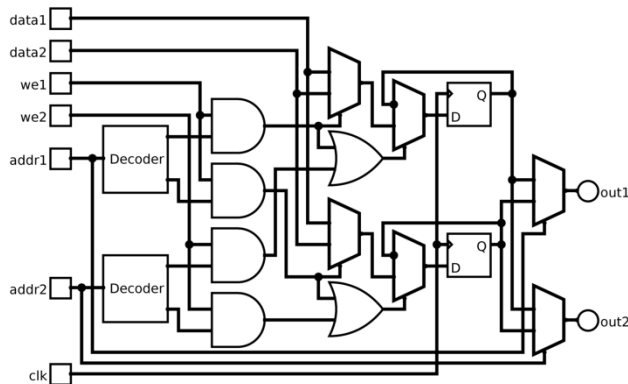
Odin II previously supported only hard block memories. However for very small memories we will show that it is not always worthwhile to use hard blocks. Additionally, not all FPGA architectures contain such hard blocks, and those that do contain a limited number. In each case, it may be useful to synthesize memories transparently as soft logic.

Therefore soft logic memories were recently added to Odin II in order to explore the potential of using soft logic in these and other situations.



**Figure 12: A Soft Logic Single Port Memory with One Address Bit and One Data Bit**

Figure 12 depicts the soft logic which can now be generated by Odin II in place of a 1x1 single port memory hard block. Notice that each single port memory requires an  $n$ -to- $2^n$  address decoder and  $m$   $2^n$ -port output multiplexers, where  $n$  is the number of address bits and  $m$  is the number of bits in each word.



**Figure 13: A Soft Logic Dual Port Memory with One Address Bit and One Data Bit**

Likewise, Figure 13 depicts the soft logic which can now be generated by Odin II in place of a 1x1 dual port memory. Notice that in addition to a second address decoder and

an additional output multiplexer for each data bit, we have also added a second multiplexer in order to switch between the two data inputs.

## 6 Performance Analysis

This work has produced substantial performance improvements within Odin II which allow it to be used for the elaboration and simulation of larger circuits than was previously possible.

### 6.1 Benchmark Compilation

Table 3 lists benchmark compilation times with Odin II, ordered by the number of nodes in the resulting netlist. These times were gathered on a GNU/Linux desktop computer system with an Intel Core 2 Duo E8400 CPU running at 3.0GHz and 8GB of DDR2 667MHz RAM. Times are averaged over ten trials. Since our interest lies primarily with the longer compile times, ten trials are considered sufficient.

**Table 3: Odin II Benchmark Compilation Times, Ordered by the Number of Nodes**

Benchmark	Nodes	Connections	Time (s)	Previous Time	Speedup
diffeq2	1087	2901	0.005	0.005	1.00
ch_intrinsics	1381	10698	0.019	0.021	1.11
stereovision3	1501	5722	0.027	0.072	2.67
diffeq1	1664	4631	0.008	0.009	1.13
mkPktMerge	2503	7369	0.074	0.072	0.97
sha	5647	152495	0.357	31.133	87.21
raygentop	9850	40357	0.130	0.156	1.20
mkSMAadapter4B	10279	32915	0.154	0.184	1.19
or1200	13832	49144	0.210	0.225	1.07
boundtop	14455	82935	0.188	0.239	1.27
mkDelayWorker32B	27629	97063	0.523	0.854	1.63
stereovision1	39080	130605	0.857	0.892	1.04
blob_merge	40523	170451	0.429	0.494	1.15
stereovision0	49395	155167	0.803	0.855	1.06
stereovision2	61573	188323	0.820	0.821	1.00
LU8PEEng	66558	285831	1.605	2.547	1.59
bgm	165658	811401	3.177	7.477	2.35
LU32PEEng	217845	1019336	6.559	11.216	1.71
LU64PEEng	433156	2066993	15.767	28.625	1.82
mcml	477776	1508498	14.051	302.809	21.55

The longest compilation time for the current version of Odin II is 15.8 seconds for the LU64PEEng benchmark. Previous compilation times are also given.

The most substantial difference is in the compile time of `mcm1` which dropped from 303 seconds down to only 14 seconds, more than twenty times faster. The differences are largely due to the fact that `mcm1` contains wide ports which were previously sorted using the slow bubble sort algorithm present in the earlier version of Odin II.

The `sha` benchmark also shows a similarly substantial gap between the current time and the previous time for the same reason stated above, this time amounting to an improvement of more than eighty times.

## **6.2 Sequential Simulation Results**

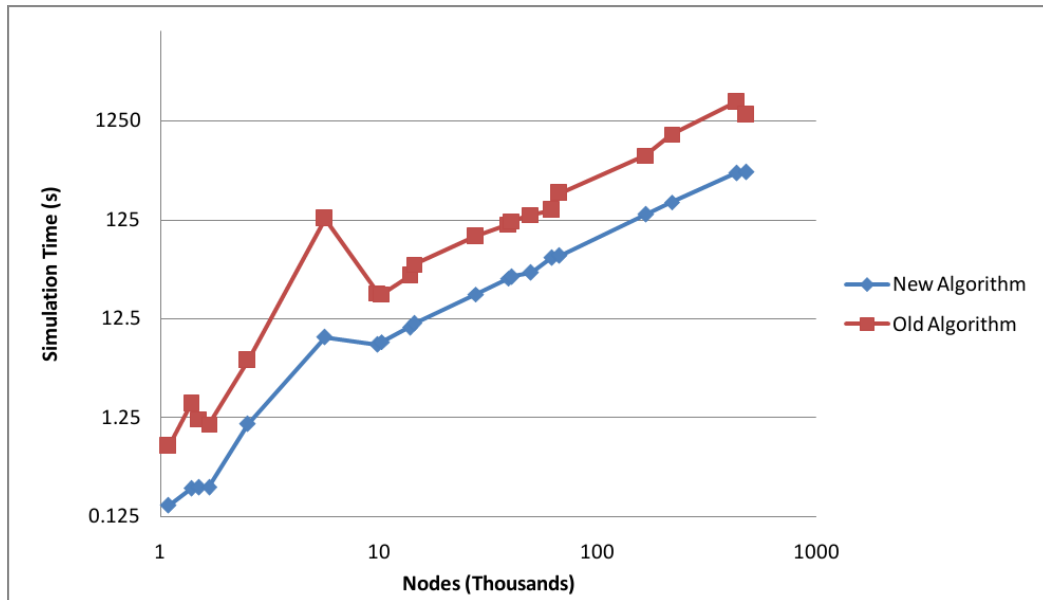
Sequential simulation runtimes were gathered on a desktop system with a Core 2 Duo E8400 CPU running at 3.0GHz and 8GB of DDR2 667MHz RAM. The longest runtime with the old algorithm was 1927 seconds on LU64PEEng. The new algorithm improves substantially on this, bringing the time down to 370 seconds, a difference of more than five times.

Table 4 gives simulation times for each benchmark using one thousand randomly generated vectors sorted by the number of nodes in each benchmark. Times are averaged over ten trials. The longest runtime with the old algorithm was 1927 seconds on LU64PEEng. The new algorithm improves substantially on this, bringing the time down to 370 seconds, a difference of more than five times.

**Table 4: A Listing of Thousand-Vector Simulation Times for each Benchmark**

Benchmark	Nodes	Connections	New (s)	Old (s)	Speedup
diffeq2	1087	2901	0.161	0.640	3.98
ch_intrinsics	1381	10698	0.241	1.723	7.15
stereovision3	1501	5722	0.246	1.209	4.91
diffeq1	1664	4631	0.248	1.061	4.28
mkPktMerge	2503	7369	1.079	4.732	4.39
sha	5647	152495	8.148	127.591	15.66
raygentop	9850	40357	6.806	22.640	3.33
mkSMAdapter4B	10279	32915	7.233	22.022	3.04
or1200	13832	49144	10.177	33.978	3.34
boundtop	14455	82935	11.196	43.863	3.92
mkDelayWorker32B	27629	97063	21.961	84.613	3.85
stereovision1	39080	130605	31.677	111.647	3.52
blob_merge	40523	170451	33.221	120.026	3.61
stereovision0	49395	155167	36.332	139.189	3.83
stereovision2	61573	188323	51.700	155.136	3.00
LU8PEEng	66558	285831	54.112	229.360	4.24
bgm	165658	811401	139.974	551.916	3.94
LU32PEEng	217845	1019336	186.843	906.378	4.85
LU64PEEng	433156	2066993	369.864	1926.721	5.21
mcml	477776	1508498	376.751	1417.124	3.76

Figure 14 plots the logarithm of these times against the logarithm of the number of nodes in each netlist, shown in thousands. Times tend to increase as the number of nodes increases with some exceptions.



**Figure 14: Sequential Simulation Times Plotted Against the Number of Nodes in the Netlist**

The spike present at 5647 nodes represents the `sha` benchmark which has an unusually large number of connections relative to the number of nodes (or a high degree) when compared with the other benchmarks. The spike is also present with the new algorithm, although it is substantially diminished. In the context of the old algorithm, a large degree is expected to substantially affect runtime, as the older algorithm had an overall runtime complexity  $O(nd)$  node computations.

With the new algorithm the factor of  $d$  is removed, however each node computation is also affected by the number of output pins which must be updated. While the total number of output pins may be less than the number of connections, this means that extremely high values of  $d$  clearly can have an impact on the new algorithm's runtime. The same effect is present to a lesser extent with `LU64PEEng` which also has a higher degree than the larger `mcm1` circuit.

When we look at the correlation coefficients, we see that a correlation of 0.996 exists between the old algorithm's runtime and the number of connections, whereas a correlation of only 0.975 exists between its runtime and the number of nodes. Comparing this to the new algorithm, we see that a stronger correlation of 0.999 exists between the new algorithm's runtime, and the number of nodes in the netlist, compared with 0.982 when correlated with the number of connections. This further supported the above analysis.

### 6.3 Parallel Simulation Results

Table 5 lists parallel simulation times for one through six processors. Times are given in seconds and were gathered on a GNU/Linux desktop computer system equipped with an AMD Phenom X6 1100T processor running at 3.3GHz and 8GB of DDR2 800MHz RAM. All times were averaged over ten trials. Sequential times ( $p=1$ ) on this system are comparable to the sequential times in Table 4.

**Table 5: Parallel Simulation Times for One through Six Processors**

Benchmark	Nodes	p=1	P=2	p=3	p=4	p=5	p=6
diffeq2	1087	0.283	0.218	0.181	0.164	0.161	0.165
ch_intrinsics	1381	0.537	0.465	0.221	0.152	0.137	0.133
stereovision3	1501	0.456	0.415	0.188	0.132	0.124	0.125
diffeq1	1664	0.450	0.345	0.275	0.246	0.225	0.222
mkPktMerge	2503	2.108	1.280	0.930	0.738	0.615	0.450
sha	5647	8.553	6.002	5.526	4.865	4.472	4.164
raygentop	9850	6.121	3.755	2.955	2.303	1.855	1.773
mkSMAdapter4B	10279	6.478	3.952	3.040	2.349	2.050	1.852
or1200	13832	9.238	5.643	4.483	3.816	3.187	3.047
boundtop	14455	10.306	6.364	4.828	4.025	3.516	3.314
mkDelayWorker32B	27629	21.203	14.094	11.765	10.698	9.495	9.105
stereovision1	39080	31.285	18.658	13.990	11.689	9.820	10.043
blob_merge	40523	36.646	23.377	18.520	16.075	15.403	14.478
stereovision0	49395	33.494	20.129	15.440	13.142	11.567	11.826
stereovision2	61573	59.382	33.755	24.880	20.380	18.594	16.915
LU8PEEng	66558	58.171	38.577	30.715	27.198	25.012	23.787
bgm	165658	167.442	98.198	74.426	61.420	54.311	50.365
LU32PEEng	217845	191.649	129.560	101.615	86.085	80.041	74.251
LU64PEEng	433156	384.055	260.684	205.545	172.445	160.742	148.879
mcml	477776	380.725	240.561	185.702	158.136	139.925	130.285

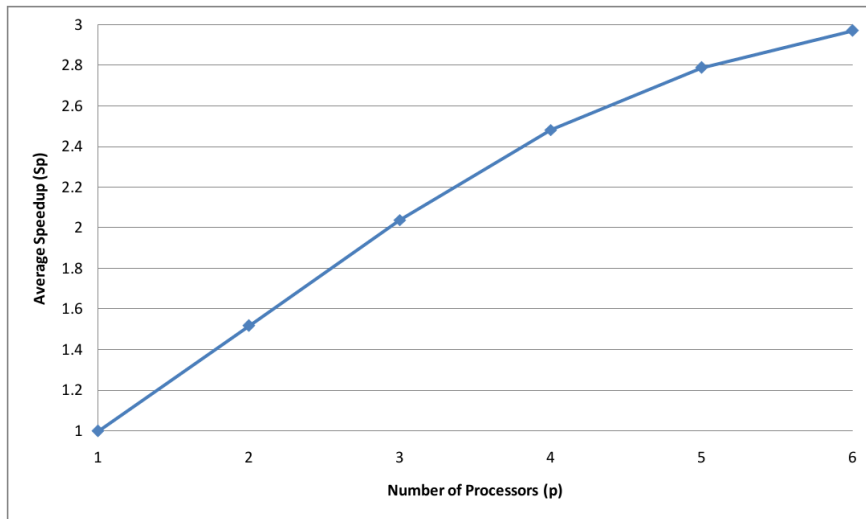
In Table 6, speedup values are computed based on the times from Table 5 [25]. Notice that the best speedup (4.69) is achieved on mkPktMerge using six processors. The average speedup is also presented in Table 6, showing an upward trend with the number of processors.

When correlated with the number of nodes, speedup with six processors shows a small negative correlation of -0.13, indicating that achievable speedup is not strongly influenced by the size of the benchmark.

**Table 6: Parallel Speedups for Each Benchmark Using One Through Six Processors**

<b>Benchmark</b>	<b>Nodes</b>	<b>p=1</b>	<b>p=2</b>	<b>p=3</b>	<b>p=4</b>	<b>p=5</b>	<b>p=6</b>
diffeq2	1087	1.00	1.30	1.56	1.72	1.76	1.71
ch_intrinsics	1381	1.00	1.16	2.44	3.53	3.93	4.03
stereovision3	1501	1.00	1.10	2.43	3.46	3.69	3.66
diffeq1	1664	1.00	1.30	1.64	1.83	2.00	2.03
mkPktMerge	2503	1.00	1.65	2.27	2.86	3.43	4.69
sha	5647	1.00	1.42	1.55	1.76	1.91	2.05
raygentop	9850	1.00	1.63	2.07	2.66	3.30	3.45
mkSMAadapter4B	10279	1.00	1.64	2.13	2.76	3.16	3.50
or1200	13832	1.00	1.64	2.06	2.42	2.90	3.03
boundtop	14455	1.00	1.62	2.13	2.56	2.93	3.11
mkDelayWorker32B	27629	1.00	1.50	1.80	1.98	2.23	2.33
stereovision1	39080	1.00	1.68	2.24	2.68	3.19	3.12
blob_merge	40523	1.00	1.57	1.98	2.28	2.38	2.53
stereovision0	49395	1.00	1.66	2.17	2.55	2.90	2.83
stereovision2	61573	1.00	1.76	2.39	2.91	3.19	3.51
LU8PEEng	66558	1.00	1.51	1.89	2.14	2.33	2.45
bgm	165658	1.00	1.71	2.25	2.73	3.08	3.32
LU32PEEng	217845	1.00	1.48	1.89	2.23	2.39	2.58
LU64PEEng	433156	1.00	1.47	1.87	2.23	2.39	2.58
mcml	477776	1.00	1.58	2.05	2.41	2.72	2.92
<i>Average</i>		<i>1.00</i>	<i>1.52</i>	<i>2.04</i>	<i>2.48</i>	<i>2.79</i>	<i>2.97</i>

Figure 15 plots the average parallel speedups from Table 6 against the number of processors. An upward trend is visible, with a maximum value of 2.97 with six processors and a speedup of 2.48 with four processors.



**Figure 15: Average Parallel Speedup across all Benchmarks vs. Number of Processors**

When compared to the original simulation algorithm in Table 4, runtimes are on average 9.6 times faster with four processors, and 11.4 times faster with six processors.

## 7 Memory Results and Analysis

In order to evaluate the new memory infrastructure, a series of experiments were devised to explore the effects these changes had on the architectural level. These experiments may also serve as a template for future architecture explorations using soft logic memories.

Experiments were run using the VTR 1.0 full release which is freely available online [9, 10]. The release was locally patched with Odin II r274 from the Odin II Google Code repository [21].

### 7.1 Methodology

All memory area and delay experiments were performed using the `k6_N10_memDepth16384_memData64_40nm_timing` architecture included with the VTR release. This is a classic island-style FPGA architecture based on the Altera Stratix IV, which includes timing and area measurements from that device [10]. It uses 6-LUTs with a cluster size of ten.

Where hard block memory is concerned, this is a homogenous architecture with large 144 kilobit field configurable memory blocks. During these experiments, these memory blocks are configured to a 36 bit width allowing for 4096 words per block.

While this architecture is supplied with accurate timing and area measurements for the CLBs and routing as well as accurate timing information for the memory hard blocks,

area measurements for the hard blocks are lacking. We will therefore assume that the memory blocks are comprised primarily of transistors using a standard 6T memory cell arrangement. Based on these assumptions, each memory bit will occupy an area equivalent to six minimum width transistors (*mwt*). This gives a reasonable lower bound estimate to the size of each memory block of

$$147456 \text{ bits} * 6 \text{ mwt/bit} = 884736 \text{ mwt}.$$

The `k6_N10_memDepth16384_memData64_40nm_timing` architecture was modified, incorporating this area measurement. This supports a rudimentary area measurement of the final circuit including hard blocks using the VTR flow.

For each of these experiments, the total silicon area is reported. This number is the sum of the total logic area and total routing area reported by VPR, measured in minimum width transistors. Critical path delay is reported in seconds as measured by VPR using the VTR 1.0 release [9].

The `k6_N10_memDepth16384_memData64_40nm_timing` architecture was also modified to allow Odin II to fully control the splitting and padding of memories. This modification was tested to ensure that Odin II's architecture aware splitting and padding performed as well as the previous VPR-level memory packing. CAD flow experiments done using the existing VTR `regression_memory_archs` task demonstrated that this modification made no difference to the resulting number of memory blocks and CLBs in the final circuit.

Table 7 lists the Verilog circuit used for memory exploration. This circuit uses a single implicit memory with configurable width and depth. Depths of 2 through 8 address bits were used. Width was varied by powers of two from 1 to 128 data bits per word for soft logic and as high as 1024 data bits per word for the hard blocks.

**Table 7: The Verilog Circuit Used in Memory Exploration**

```

`define DATA_BITS 1      // Bits per word
`define ADDRESS_BITS 2   // Address length
`define MEMORY_WORDS 4   // 2^ADDRESS_BITS

module bm_base_memory(clock, we, address_in,
address_out, out1, out2, value_out, value_in);
input clock, we;
input  [`DATA_BITS-1:0] value_in;
input  [`ADDRESS_BITS-1:0] address_in, address_out;
output [`DATA_BITS-1:0] value_out, out1, out2;
wire  [`DATA_BITS-1:0] value_out;
reg   [`DATA_BITS-1:0] out1, out2;
reg   [`ADDRESS_BITS-1:0] address;

reg  [`DATA_BITS-1:0] memory [ `MEMORY_WORDS-1:0];

always @(posedge clock)
begin
address <= address_in;
if (we == 1'b1)
memory[address] <= value_in;
end

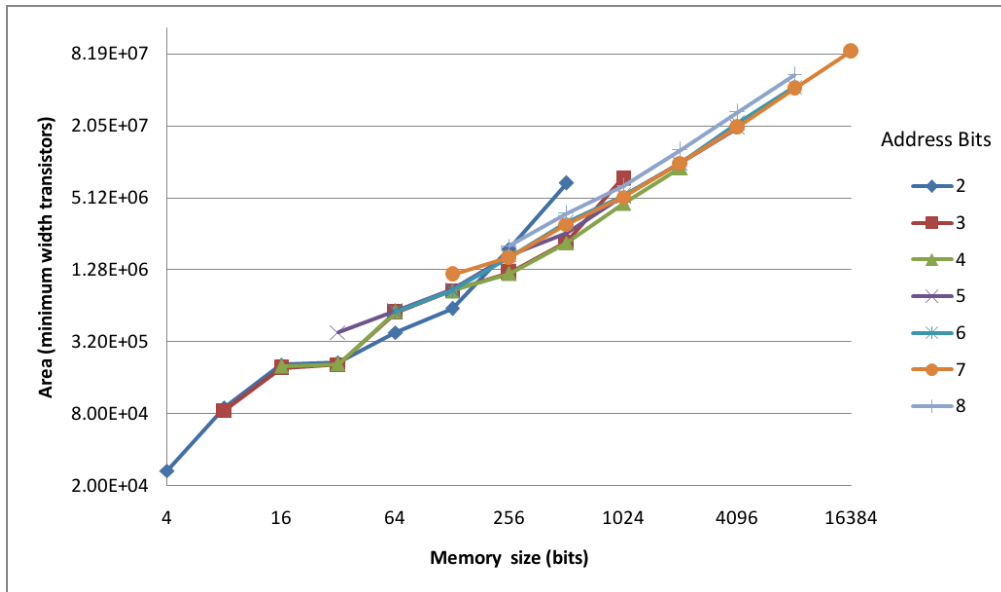
assign value_out = memory[address_out];

always @(posedge clock)
begin
out1 <= value_out & address_in;
out2 <= out1 & 1'b0;
end
endmodule

```

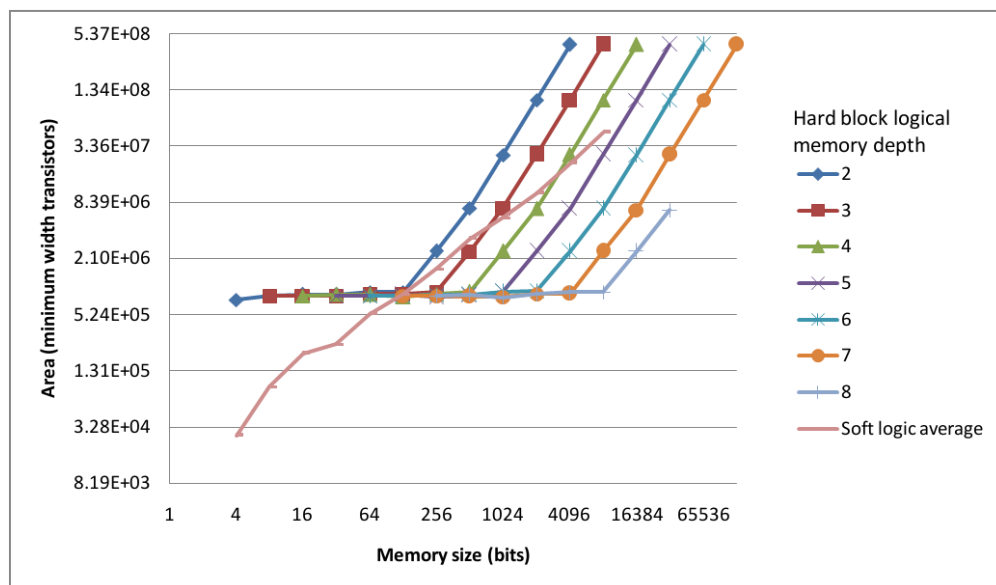
## 7.2 Soft Logic Memories vs. Hard Block Memories

Figure 16 shows the silicon area occupied by soft logic memories with widths ranging from 1 to 128 and depths from 2 through 8, segregated by depth. An upward trend is visible in terms of area as the number of memory bits increases. Aspect ratio appears to have little effect on the silicon area of the soft logic memory indicating that the configuration of the bits is less critical than the number of bits when predicting the area of a soft logic memory.



**Figure 16: Silicon Area Occupied by Soft Logic Memories vs. Size**

Figure 17 shows area measurements for various hard block memories. The average of all lines from Figure 16 is shown for reference. Hard block areas remain constant showing that only a single hard block is used until the width of a memory exceeds 36 bits. After this, one additional hard block is used for every additional 36 bits of width.

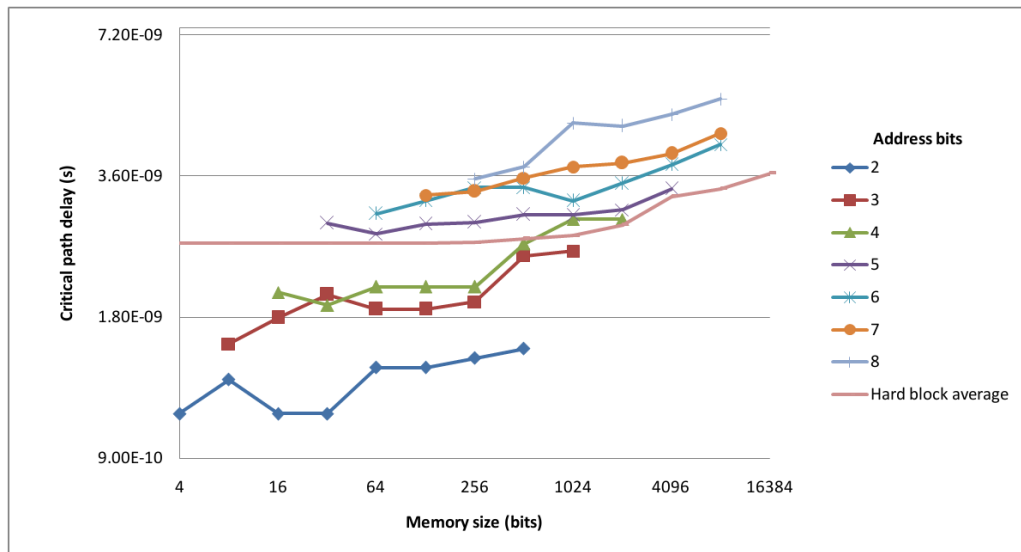


**Figure 17: Areas of Various Hard Block Memories vs. Size with the Soft Logic Average Shown**

It is clear that the area of the soft logic memories is below that of the hard block for a range of sizes. This is the case for all memories of depth two and three, among others. Due to the shallow nature of the memories in this circuit and the large physical hard blocks, the majority of each hard block remains unused, with a maximum of 256 out of a total of 4096 words being used with an address width of 8 bits.

This waste gives the soft logic memory the advantage as the soft logic is able to occupy less area than a single hard block when sufficiently small soft logic memories are used. Due to the homogenous nature of this architecture (as well as the choice of mode), more appropriately sized memory blocks are not available. If this architecture were heterogeneous, containing small distributed memories as well as large block memories, this waste could be largely eliminated simply by packing the smaller logical memories into more appropriately sized physical memory blocks [11].

Figure 18 gives critical path delays for various soft logic memory sizes, segregated by depth. Notice that each individual memory depth has a distinctly different delay curve, with the shallowest memories showing the shortest critical path delay.



**Figure 18: Critical Path Delay in Seconds vs. Memory Size for Soft Logic Memories**

The average delay curve for the hard blocks is also shown in Figure 18. The hard block delay does not respond to the varying memory depth within this range, as the number of hard blocks used remains relatively unaffected by this parameter. It is noteworthy that the shallower soft logic memories do compete effectively with the hard block in terms of delay, remaining well below the hard block average for depths of two and three address bits and not significantly exceeding the hard block delay for a depth of four.

Again the hard block is largely wasted when mapping the smaller memories.

Homogenous architectures with a greater variety of available memory blocks would be more appropriate. Smaller memory hard blocks would certainly have more competitive delays when compared to the large block RAM used here [11].

Never the less, it is clear from these findings that small soft logic memories can have advantages both in terms of delay and in terms of area when dealing with some homogenous memory architectures.

## 8 Conclusions and Future Work

Substantial improvements have been made to the VTR flow through this work. The performance of Odin II has been substantially improved, both in terms of the performance of its compiler and in terms of the performance of its simulator. A new parallel simulation algorithm has been developed which achieves an average speedup of 2.97 with six processors across twenty large benchmark circuits when compared with the improved sequential simulator, and a speedup of 11.4 when compared to the original simulator.

The reliability of the VTR flow has been substantially improved through verification of Odin II via simulation and the resulting corrections. Regression tests have been added which allow developers to quickly verify their changes to Odin II. Benchmark support has been greatly improved through the addition of support for implicit memory syntax.

Architecture exploration has been performed using Odin II's new soft logic memory support. These results show that small soft logic memories may outperform their hard block counterparts both in terms of area and delay on heterogeneous memory architectures with large block memories.

The improvements to the verification framework of Odin II support future development of the CAD flow by allowing developers to rigorously verify their improvements. This affords researchers a greater degree of confidence in the correctness of their findings.

Future work could extend the Odin II simulator to support timing simulation and power

modeling by annotating the netlist and enhancing the simulator to support these computations.

Thanks to Odin II's BLIF reading capability, netlists from various other stages in the CAD flow can now be verified as well including post-synthesis netlists from ABC, and post-routing netlists which may be produced by VPR. Extension of VPR to produce such netlists would enable Odin II to perform timing and power estimation based on the actual placement of logic elements within the final FPGA layout.

Extension of Odin II to support new Verilog HDL is ongoing. Support for new architectural features such as hard block adder/carry chains is also ongoing, which will improve CAD flow results when using large adders, and will open up new research questions surrounding these architectures.

This work also opens up possibilities surrounding soft logic memories. Future work could explore the possibility of tuning elaboration parameters in order to take best advantage of a mixture of hard and soft logic memories within a particular architecture based on certain elaboration objectives. These may include the minimization of critical path delay to support higher clock speeds or the minimization of silicon area on the FPGA in order to reduce power consumption.

In the future, possibilities exist for Odin II to perform such objective-driven elaboration across the entire netlist, exploring such tradeoffs in various aspects of netlist construction and optimization.

## Bibliography

- [1] P. Jamieson, K. B. Kent, F. Gharibian and L. Shannon. Odin II - An Open-Source Verilog HDL Synthesis Tool for CAD Research. Presented at Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on. 2010.
- [2] P. A. Jamieson and J. Rose. Enhancing the area efficiency of FPGAs with hard circuits using shadow clusters. *Very Large Scale Integration (VLSI) Systems*, IEEE Transactions on 18(12), pp. 1696-1709. 2010.
- [3] I. Kuon and J. Rose. Measuring the gap between FPGAs and ASICs. *Computer-Aided Design of Integrated Circuits and Systems*, IEEE Transactions on 26(2), pp. 203-215. 2007.
- [4] I. Kuon and J. Rose. Area and delay trade-offs in the circuit and architecture design of FPGAs. Presented at Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays. 2008, Available: <http://doi.acm.org/10.1145/1344671.1344695>.
- [5] J. C. Libby, A. Furrow, P. O'Brien and K. B. Kent. A framework for verifying functional correctness in Odin II. Presented at Field-Programmable Technology (FPT), 2011 International Conference on. 2011.
- [6] A. Mishchenko. ABC: A System for Sequential Synthesis and Verification. <http://www.eecs.berkeley.edu/~alanmi/abc/>, 2009, Accessed: April 2012.
- [7] T. Ngai, J. Rose and S. J. E. Wilton. An SRAM-programmable field-configurable memory. Presented at Custom Integrated Circuits Conference, 1995., Proceedings of the IEEE 1995. 1995.
- [8] J. Rose. Hard vs. soft: The central question of pre-fabricated silicon. Presented at Multiple-Valued Logic, 2004. Proceedings. 34th International Symposium on. 2004.
- [9] J. Rose et al., vtr-verilog-to-routing. <http://code.google.com/p/vtr-verilog-to-routing/>, 2012, Accessed: April 2012.
- [10] J. Rose, J. Luu, C. W. Yu, O. Densmore, J. Goeders, A. Somerville, K. B. Kent, P. Jamieson and J. Anderson. The VTR project: Architecture and CAD for FPGAs from verilog to routing. Presented at Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays. 2012, Available: <http://doi.acm.org/10.1145/2145694.2145708>.

- [11] S. J. E. Wilton. Implementing logic in FPGA memory arrays: Heterogeneous memory architectures. Presented at Field-Programmable Technology, 2002. (FPT). Proceedings. 2002 IEEE International Conference on. 2002.
- [12] S. J. E. Wilton, J. Rose and Z. G. Vranesic. Architecture of centralized field-configurable memory. Presented at Field-Programmable Gate Arrays, 1995. FPGA '95. Proceedings of the Third International ACM Symposium on. 1995.
- [13] H. Wong, V. Betz and J. Rose. Comparing FPGA vs. custom cmos and the impact on processor microarchitecture. Presented at Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays. 2011, Available: <http://doi.acm.org/10.1145/1950413.1950419>.
- [14] G. Lemieux, E. Lee, M. Tom and A. Yu. Directional and single-driver wires in FPGA interconnect. Presented at Field-Programmable Technology, 2004. Proceedings. 2004 IEEE International Conference on Field-Programmable Technology. December 2004.
- [15] P. Chow, Soon Ong Seo, J. Rose, K. Chung, G. Paez-Monzon and I. Rahardja. The design of an SRAM-based field-programmable gate array. I. architecture. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on 7(2), pp. 191-197. 1999.
- [16] T. Ahmed, P. D. Kundarewich, J. H. Anderson, B. L. Taylor and R. Aggarwal. Architecture-specific packing for virtex-5 FPGAs. Presented at Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays. 2008, Available: <http://doi.acm.org/10.1145/1344671.1344675>.
- [17] V. Manohararajah, G. R. Chiu, D. P. Singh and S. D. Brown. Predicting interconnect delay for physical synthesis in a FPGA CAD flow. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on 15(8), pp. 895-903. 2007.
- [18] M. Sheng and J. Rose. Mixing buffers and pass transistors in FPGA routing architectures. Presented at Proceedings of the 2001 ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays. 2001, Available: <http://doi.acm.org/10.1145/360276.360302>.
- [19] E. Ahmed and J. Rose. The effect of LUT and cluster size on deep-submicron FPGA performance and density. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on 12(3), pp. 288-298. 2004.
- [20] J. Cong and Yean-Yow Hwang. Simultaneous depth and area minimization in LUT-based FPGA mapping. Presented at Field-Programmable Gate Arrays,

1995. FPGA '95. Proceedings of the Third International ACM Symposium on. 1995.
- [21] K. B. Kent et al., Odin II – Verilog Synthesis tool generally targeting FPGAs. <http://code.google.com/p/odin-ii/>, 2012, Accessed: April 2012.
- [22] OpenMP.org. The OpenMP® API specification for parallel programming. <http://openmp.org/>, 2012, Accessed: April 2012.
- [23] OpenMP.org. The OpenMP® API specification for parallel programming: OpenMP Compilers. <http://openmp.org/wp/openmp-compilers/>, 2012, Accessed: April 2012.
- [24] Model.com. ModelSim - Advanced Simulation and Debugging. <http://model.com/>, 2012, Accessed: April 2012.
- [25] T. Rauber and G. Runger. Performance analysis of parallel programs. In *Parallel Programming: for Multicore and Cluster Systems*. Springer-Verlag Berlin Heidelberg, 2010.
- [26] T. Rauber and G. Runger. Introduction. In *Parallel Programming: for Multicore and Cluster Systems*. Springer-Verlag Berlin Heidelberg, 2010.
- [27] Altera.com. Quartus II Web Edition Software. <http://www.altera.com/products/software/quartus-ii/web-edition/qts-we-index.html>, 2012, Accessed: April 2012.
- [28] Verilog.com. Verilog Resources. <http://www.verilog.com>, 2011, Accessed: May 2012.
- [29] UC Berkeley. Berkeley logic interchange format (BLIF). Oct Tools Distribution, vol. 2.
- [30] Akim Demaille, Joel E. Denny, and Paul Eggert. Bison - GNU parser generator. GNU.org, <http://www.gnu.org/software/bison/>. 2011, Accessed: May 2012.
- [31] Thomas H. Cormen, et al. Topological Sort. *Introduction to Algorithms*, Third Edition, pp. 612-615. The MIT Press, 2009.

## Curriculum Vitae

Candidate's full name: Lewis Andrew Somerville

Universities attended:

1. University of New Brunswick, 2010, Bachelor of Computer Science

Publications:

1. A. Somerville and K.B. Kent. Acceleration of Simulation in Odin II with OpenMP. Technical Report TR11-210, University of New Brunswick, September 2011.
2. J. Rose, J. Luu, C. W. Yu, O. Densmore, J. Goeders, A. Somerville, K. B. Kent, P. Jamieson and J. Anderson. The VTR project: Architecture and CAD for FPGAs from verilog to routing. Presented at the ACM/SIGDA International Symposium on Field Programmable Gate Arrays. 2012, Available: <http://doi.acm.org/10.1145/2145694.2145708>.
3. A. Somerville, K. B. Kent. Improving Memory Support in the VTR Flow. Presented at the 22<sup>nd</sup> International Conference on Field Programmable Logic and Applications, 2012.
4. S. Wang, A. Somerville, Kenneth B. Kent. A Simple Connect6 Threat-Based Hardware Design. 4th International Workshop on Highly Efficient Accelerators and Reconfigurable Technologies. Okinawa, Japan, May 30, 2012.