2008

# A CAD Tool for Synthesizing Optimized Variants of Altera's Nios II Soft-Core Processor

Omar Al Rayahi
*University of Windsor*

# A CAD Tool for Synthesizing Optimized Variants of Altera's Nios II Soft-Core Processor

By

**Omar Al Rayahi**

A Thesis
Submitted to the Faculty of Graduate Studies
through Electrical and Computer Engineering
in Partial Fulfillment of the Requirements for the
Degree of Master of Applied Science at the
University of Windsor

Windsor, Ontario, Canada
2008

*Your file Votre référence*
*ISBN: 978-0-494-47050-3*
*Our file Notre référence*
*ISBN: 978-0-494-47050-3*

## Author's Declaration of Originality

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

# *Abstract*

Soft-core processors offer embedded system designers the benefits of customization, flexibility and reusability. Altera's NIOS II soft-core processor is a popular, commercially available soft-core processor that can be implemented on a variety of Altera FPGAs. In this thesis, the Nios II soft-core processor from Altera Corporation was studied and a VHDL implementation, called UW_Nios II, was developed. UW_Nios II was developed to enable us to perform design space exploration (DSE) for the Nios II processor. It was evaluated and compared with Altera Nios II and shown to be competitive. SCBuild is an existing CAD tool that was developed to enable DSE of soft-core processors. We modified SCBuild to automatically explore the design space of the UW_Nios II using a genetic algorithm. This tool can accurately estimate the area and critical path delay of different variants of the UW_Nios II on a field programmable gate array. Through experiments conducted using SCBuild, it was shown that employing a genetic algorithm to explore the design space of parameterized Nios II core, with a large design space, helps designers find optimized variants of UW_Nios II.

# *Acknowledgements*

I would like to express my sincere appreciation to my supervisor, Dr. Mohammed A. S. Khalid, who generously contributed his time and effort toward this project. His enthusiasm for, and belief in my research has always motivated me. He introduced me to this interesting research area and guided me throughout my thesis with great patience, which has made possible the completion of this thesis.

I am also grateful to my thesis committee members, Dr. N. Kar and Dr. Z. Kobti for their advice regarding the research process and their assistance in the preparation of this thesis.

I would also like to extend my deep appreciation to my parents for their constant support, patience and encouragement to complete this work. Special thanks to my sisters and their families (Salma and her son Omar, and Azza and her children Abdel Monem and Bana) and brothers (Ahmad and Usama) who helped in ways unknown to them. I extend special thanks to my uncle and aunt and their family as well.

Finally, I would like to acknowledge my fiends and fellow graduate students at the University of Windsor. Ian, I'm grateful for your constant advice and help in and out of the lab. Lastly, thanks to Jay, Marwan, Junsong, Hongmei, Lin Lin, Thuan, Aws, Ray, Harb, Amir, Yasser and everyone else who gave me good company and made this time of my life more enjoyable.

# *Table of Contents*

# *List of Figures*

*Page*

# *List of Tables*

*Page*

# *Abbreviations*

| | |
|---|---|
| ALU: | Arithmetic Logic Unit |
| ASIC: | Application-Specific Integrated Circuit |
| ASIP: | Application-Specific Instruction-set Processor |
| CAD: | Computer Aided Design |
| CMP: | Chip Multiprocessor |
| CPU: | Central Processing Unit |
| DOF: | Decode and Operand Fetch |
| DS: | Design Space |
| DSE: | Design Space Exploration |
| DSP: | Digital Signal Processing |
| EA: | Evolutionary Algorithm |
| EX: | Execute |
| FF: | Flip Flop |
| FPGA: | Field-Programmable Gate Array |
| GA: | Genetic Algorithm |
| HDL: | Hardware Description Language |
| I/O: | Input/Output |
| IC: | Integrated Circuit |
| IDE: | Integrated Development Environment |
| IF: | Instruction Fetch |
| IP: | Intellectual Property |
| IR: | Instruction Register |
| ISA: | Instruction Set Architecture |
| LE: | Logic Element |
| LPM: | Library of Parameterizable Megafunction |
| LUT: | Look Up Table |
| MAC: | Multiply-Accumulate |
| MUX: | Multiplexer |
| OP: | Opcode |
| OPX: | Opcode-Extension |
| OS: | Operating System |
| PC: | Program Counter |
| PLL: | Phase-locked Loop |

| | |
|---|---|
| RAM: | Random Access Memory |
| RISC: | Reduced Instruction Set Computer |
| ROM: | Read Only Memory |
| RTL: | Register Transfer Level |
| SCBuild: | Soft-Core Build |
| SEAMO: | Simple Evolutionary Algorithm for Multi-objective Optimization |
| SoC: | System on a Chip |
| SOPC: | System on a Programmable Chip |
| SPREE: | Soft Processor Rapid Exploration Environment |
| Tcl: | Tool Command Language |
| UART: | Universal Asynchronous Receiver/Transmitter |
| VHDL: | Very High Speed Integrated Circuit Hardware Description |
| VLSI: | Very Large Scale Integration |
| WB: | Write Back |
| XML: | Extensible Markup Language |

# *List of Symbols*

| Symbol | Definition |
|---|---|
| $P$ | Total number of parameters. |
| $p_i$ | The $i^{th}$ parameter. |
| $i$ | Parameter index. |
| $N$ | Size of genetic population. |
| $r_c$ | Crossover rate |
| $r_m$ | Mutation rate. |
| $K$ | Total number of objectives. |
| $k$ | Objective index. |
| $F_k\ (p_1, p_2, \ldots, p_P)$ | The $k^{th}$ objective function. |
| $f_{i,k}(p_i)$ | The functional form of $i^{th}$ term of the $k^{th}$ objective function. |
| $a_{i,k}$ | The $i^{th}$ regression coefficient for the $k^{th}$ objective function. |

# Chapter 1

# *Introduction*

With the increased variety and complexity of digital electronic devices, demand for systems that perform a specific set of tasks for a particular application increases. Embedded systems are used for this purpose; they are designed to do a specific task, rather than be a general-purpose computer for multiple tasks. In general, an embedded system has a hardware component and a software component, sometimes referred to as firmware, that's designed to execute on the hardware. The software component is usually stored in read-only memory or Flash memory chips rather than a disk drive. It often runs with limited computer hardware resources: small or no keyboard, screen, and little memory. The hardware component usually consists of a microprocessor and associated peripherals.

Since the hardware component (i.e., the microprocessor) is only required to run a single software application, it can be optimized to run it as efficiently as possible. This has led to the development of Application Specific Instruction-Set Processors (ASIP's). ASIP's are processors designed and optimized to run only one application. The architecture is therefore optimized to run that specific application efficiently. With recent advancements in IC process technology, embedded systems have become more complex and are performing more tasks. More complex embedded systems introduced new design challenges.

In the past, embedded systems used to be developed by designing the hardware component first, and then developing the software component to run on the designed

hardware. Designers later realized that by following this approach, they missed out on potential optimizations that could be exploited if the hardware and software were designed concurrently. This has led to a second design approach for embedded systems known as the hardware/software co-design approach [1, 2, 3, 4]. As embedded systems got more and more complex, it has become impractical and time consuming to design every hardware component of embedded systems from scratch. Thus, a third approach known as the platform-based design approach [5, 6, 7] took shape. In this approach, designers depend on pre-designed and pre-tested hardware components, known as intellectual property (IP) cores, to build their hardware systems.

Soft-cores are one class of hardware IP cores. A soft core is a synthesizable hardware component that is described at the register transfer level using one or more hardware description languages (HDLs), such as Verilog or VHDL. Many soft-cores are parameterized, meaning that one or more of the core's features can be changed at design time prior to synthesis. A parameter is a specific aspect of the core's architecture that can be changed and assigned values from a finite set by the designer [8, 9]. Some examples of parameters include variable bus width, multiple implementations of functional units, and multiple memory sizes to name a few. Core parameterization makes soft IP cores flexible because they can be easily configured to suit different applications in a short time, which makes them attractive to designers.

FPGA's are a special class of programmable logic devices that can be programmed and re-programmed any number of times to act virtually like any digital circuit, subject to the logic capacity of the FPGA. FPGAs serve as a real-time prototyping and implementation medium on which complete embedded systems can be implemented to test and verify their functionality. This has encouraged embedded systems designers to increasingly use FPGA's as their implementation medium to in order to minimize design costs and time.

When designing embedded systems, it's necessary that the hardware component be well optimized and configured so that the software component can run efficiently. This is

important to avoid ending up with a sub-optimal system. The set of all possible hardware design configurations that can be used to perform the system's intended tasks is referred to as the system's design space (DS). As systems become more parameterized, their design spaces expand; design spaces can easily contain thousands of possible hardware configurations or more. Therefore, the task of selecting the most optimal hardware platform configuration for the hardware component of an embedded system becomes difficult.

Designers usually find it necessary to explore the design spaces of their systems in search of the optimal configuration for their target application. This process is known as design space exploration (DSE) [10]. As design spaces expand, it becomes impractical and time consuming to consider and evaluate each configuration individually. Therefore, the DSE process needs to be automated.

In this thesis, a methodology to automatically explore the design space of a parameterized soft-core microprocessor targeted for implementation on FPGA platforms and the necessary CAD tool are developed. In this work, a parameterized soft-core processor, called UW_Nios II, that supports the same instruction set as Altera's Nios II soft-core processor was initially developed using VHDL. Then, an existing CAD tool was modified to automatically explore the design space of the UW_Nios II soft-core processor.

## 1.1 Thesis Objectives

The microarchitecture of hard core processors targeting ASICs has been studied by researchers and manufacturers in detail for a long time. However, design features and trade-offs of FPGA-based soft-core processors are significantly different than those implemented in VLSI design flows [11, 12]. As a result, conclusions drawn from research conducted on hard core processors may not be transferable to soft-core processors targeting FPGA platforms. Therefore, the main goal of this research is to enhance the understanding of the design process of commercial soft-core microprocessors targeting FPGA platforms including their microarchitectures and associated CAD tools and design methodologies. An

exploration of the design space of UW_Nios II soft-core processor targeting Altera FPGAs was conducted to achieve this goal. This thesis has the following objectives:

1. Develop a parameterized VHDL implementation of Altera's Nios II soft-core processor, and investigate different architectural variations of it.
2. Modify an existing CAD tool, called SCBuild, and enable it to automatically explore the design space of the developed soft-core processor using a genetic algorithm. This tool should be able to accurately estimate the area and critical path delay of different variants of the processor on a field programmable gate array.
3. Compare the different variants of the processor with Altera's Nios II commercial soft-core processors in terms of performance and area utilization on an FPGA.

To satisfy the first objective, the Nios II soft-core processor from Altera Corporation was studied and a VHDL implementation of it, called UW_Nios II, was developed and its functionality was tested. Different architectural variations of it were developed and analyzed. For the second objective, an existing CAD tool, called SCBuild ("Soft-Core Build"), was modified using C++. This tool employs a genetic-based algorithm, the Simple Evolutionary Algorithm for Multi-objective Optimization (SEAMO) [13], to automatically explore the design space of the UW_Nios II. This tool is capable of accurately estimating the area and critical path delay of different variants of the UW_Nios II on a field programmable gate array. Finally, to achieve the third objective, a set of experiments were conducted using SCBuild to explore the design space of the UW_Nios II. Different variants were compared with Altera's Nios II.

## 1.2 Thesis Organization

This thesis is organized as follows. Chapter 2 provides the reader with the background information relevant to this research. It summarizes the related previous work that has been by other researchers. Chapter 3 focuses on the design and development of our soft-core processor, the UW_Nios II. A preview of the instruction set supported by the UW_Nios II soft-core processor is first illustrated, followed by a description of the set of parameters

added to the core. The remaining part of the chapter compares the UW_Nios II's variants and Altera's Nios II. Chapter 4 discusses the results obtained from a set of experiments performed using SCBuild. This thesis is concluded in chapter 5 with suggestions for possible future work.

# Chapter 2

# *Background and Previous Work*

The concept of *reconfigurable computing* first emerged in the early 1960s [14]. In reconfigurable systems, some form of programmable hardware is used to accelerate the execution of compute-intensive algorithms. Computation-intensive parts of the algorithms are implemented in programmable hardware, while the rest of the algorithm is implemented in software that gets executed on a general-purpose processor. A lot of research has been conducted in the area of reconfigurable computing. A survey of reconfigurable systems can be found in [14]. Soft-core processors are one part of the trend in the field of reconfigurable computing. Due to recent advancements in FPGA technology, FPGA's are now a desired platform suitable for soft-core processor implementations. FPGA's can be programmed and re-programmed any number of times to reflect changes in the design architecture and parameter values, if the need arises. However, soft-core processors implemented on FPGA platforms have a lower performance than their ASIC counterparts, and consume more area and power.

In this chapter we summarize the relevant background necessary to understand this work, and also discuss the topic of soft-core processor design space exploration. This chapter starts by giving an overview of intellectual property (IP) cores, their classes and the concept of parameterization. Next, some examples of commercially available soft-core processors are given. Since Altera's Nios II soft-core processor is the focus of this research, a presentation of its architecture and its main features is provided. Then, the basic concepts

of FPGA technology and the FPGA design flow are briefly explained, followed by an overview of the FPGA CAD tool and the FPGA device used in this research. After that, an introduction to design space exploration and multi-objective optimization is provided. This chapter concludes with a presentation of previous work that's related to this research.

## 2.1 Intellectual Property (IP) Cores

Many hardware functional units tend to be repeatedly used in various embedded systems, therefore many of the developed components can be reused in different applications. Reusable hardware or software building blocks that are pre-designed and pre-tested to perform one or more tasks are referred to as *intellectual property (IP) cores* [15, 16]. Some examples of hardware IP cores include memory controllers, UARTs (Universal Asynchronous Receiver/Transmitter), timers, and even full fledged microprocessors. IP cores can be used together to form complex systems.

IP cores are classified into one of three categories: *hard cores, firm cores, and soft cores* [15, 16]. A hard core is a hardware component that is placed and routed targeting a specific IC process technology. Hard IP cores are described at the Circuit-level of abstraction, and include details about the physical layout of the core on an IC chip. Firm cores are specified as gate-level netlists, suitable for placement and routing targeting a specific process technology. A soft core is a synthesizable hardware component that is described at the Register Transfer Level using one or more hardware description languages (HDLs), such as Verilog or VHDL. Our research discusses in detail the development of soft-core processor targeting Altera FPGA platforms.

Many soft cores are parameterized, meaning that one or more of the core's features can be changed at design time prior to synthesis. A parameter is a specific aspect of the core's architecture that can be changed and assigned values from a finite set by the designer [8, 9]. Some examples of parameters include variable bus width, multiple implementations of functional units, and multiple memory sizes to name a few. Core parameterization makes soft IP cores the most flexible of the three categories of IP cores, and makes the use of soft

cores in embedded system designs attractive for a number of reasons. First, parameterized soft cores can be customized for a particular application in a relatively short time with relative ease. Second, since soft cores are described using an HDL, they are technology and platform independent. Thus, they can be fabricated into IC chips for any process technology, or they can be implemented on FPGA platforms. Finally, developing soft IP cores resembles the process of software development, which adds to the ease of developing and modifying the design.

## 2.2 Soft-core Processors

Soft-core processors are a special class of soft IP cores. Recent advancement in technology has allowed the addition of more logic capabilities to FPGA's. New FPGA's have large amounts of memory and dedicated logic. This has made FPGA's a suitable platform for implementing soft-core processors. Currently, two of the most popular commercial soft-core processors are the MicroBlaze from Xilinx Inc. [17], and the Nios II [18] from Altera Corporation. A detailed survey conducted by J. Tong et al [52] presents several commercial and open-source soft-core processors, and compares their architectural features.

MicroBlaze is a 32-bit general-purpose RISC microprocessor targeted for implementation on Xilinx FPGA's [19]. It has a register file that contains 32 32-bit general purpose registers. Instruction words are 32 bits longs, and it supports up to three operands and 2 addressing modes. The MicroBlaze family of microprocessors executes their instructions using a 3-stage pipelined datapath. Memory can be implemented using on-chip memory modules or as an off-chip external peripheral. It supports the addition of instruction and data caches, and their sizes are configurable. Depending on the configuration and target device, a MicroBlaze can have a clock frequency ranging from 65 to 150 Mhz [17]. Xilinx also offers PicoBlaze, which is an 8-bit microcontroller targeting applications requiring implementation of complex state machines.

In addition to the MicroBlaze soft-core processor, Xilinx provides a variety of soft IP cores that can be used in the development of a complete system on programmable chip (SOPC). IP cores include memory controllers, Ethernet controllers, UARTs (Universal Asynchronous Receiver/Transmitter), timers, buses, etc.

### 2.2.1 Altera's Nios II Soft-core Processor

Since the Nios II soft-core processor is the focus of this research, it will be discussed in more detail. Altera Corporation released its first commercial soft-core processor, the Nios [20], in 2000. Due to the increased popularity of soft-core processors, Altera released its next generation of soft-core processors, the Nios II family [18], whose architecture is significantly different from the Nios. The Nios II is smaller than the Nios, and provides better performance.

Embedded system designers can use the Quartus II CAD tool suite [21] and it's SOPC Builder [22] to instantiate any number of Nios II cores and connect them with other peripheral IP cores, such as timers and memory controllers, to build complete embedded systems. We've chosen to work with the Nios II core in this thesis to automatically explore its design space.

**Nios II Processor System Basics:**

The Nios II processor is a general-purpose RISC processor providing the following main features:

- Full 32-bit instruction set, datapath, and address space
- Thirty two 32-bit general-purpose registers
- Six 32-bit control registers
- Thirty two external interrupt sources
- Single-instruction 32X32 multiply and divide producing a 32-bit result
- Access to a variety of on-chip peripherals, and interfaces to off-chip memories and peripherals

- Hardware-assisted debug module enabling processor start, stop, step and trace under integrated development environment (IDE) control
- Instruction set architecture (ISA) compatible across all Nios II processor systems

The soft-core nature of the Nios II processor enables the user to integrate custom logic into the arithmetic and logic unit (ALU).

**Processor Architecture:**

A block diagram of the Nios II processor core is shown below in Figure 2.1 [18]. The Nios II architecture includes the following user-visible functional units:

- Register File
- Arithmetic and logic unit (ALU)
- Interface to custom instruction logic
- Exception controller
- Interrupt controller
- Instruction bus
- Data bus
- Instruction and data cache memories
- Tightly-coupled memory interfaces for instructions and data
- JTAG debug module

The Nios II processor core supports an ALU that implements an instruction set consisting of 94 instructions. The ALU operates on data stored in general-purpose registers and stores the result back in a general-purpose register. Some of the operations supported by the ALU are data transfer instructions, arithmetic and logical instructions, move instructions, comparison instructions, shift and rotate instructions, program control instructions, along with other control instructions. Users can also create their own custom instructions and incorporate them into the ALU.

Nios II cores have separate instruction and data bus masters. Either on-chip dedicated RAM memory blocks or off-chip peripheral devices can be used to implement instruction

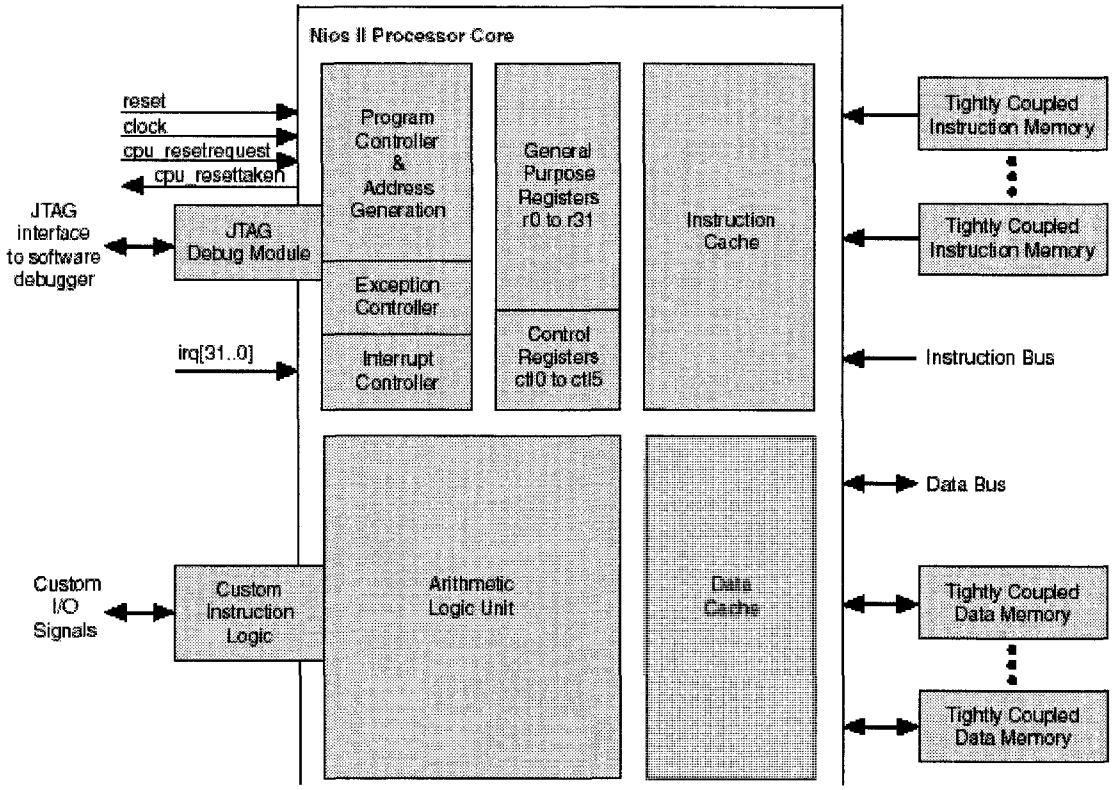



Figure 2.1: Nios II Processor Core Block Diagram [18]

and data memories. Designers using Nios II cores can debug their systems by instantiating the optional JTAG Debug Module [18]. In addition to the thirty two 32-bit general purpose registers that Nios II cores have in their register files, six control registers that are used to keep track of the status of the processor.

The Nios II processor provides an exception controller to handle all types of exceptions. All exceptions, including hardware interrupts, cause the processor to transfer execution to a single exception address. Then the cause of exception is determined and the appropriate exception routine is dispatched accordingly. The Nios II exceptions fall into one of the below-listed categories:

- Hardware interrupt
- Software interrupt
- Unimplemented instruction
- Other

Altera Corporation developed three different implementations of the Nios II processor core. These cores are called the "Fast" core, the "Standard" core and the "Economy" core. All these cores support the same instruction set.

The main objective of the fast core is to provide fast execution speed. Performance is gained at the expense of core size, making the fast core the biggest of all three cores. This core is optimal for performance-critical applications. The fast core is pipelined with a six stage pipeline depth and comes with instruction cache and optional support for data cache. It supports a 1-cycle barrel shifter/rotator, dynamic branch prediction and supports the addition of custom instructions.

The main objective of the standard core is to provide a small core size. On-chip logic and memory resources are conserved at the expense of execution performance. The standard core is designed to provide a compromise between fast processing performance and small core size. It is recommended for cost-sensitive, medium-performance applications. It is pipelined with a five stage pipeline depth and comes with instruction cache. It supports either a one-bit-per-cycle or a 3-cycle shifter/rotator, static branch prediction and supports the addition of custom instructions.

The main objective of the economy core is to provide the minimal core size. Hardware resources are conserved at the expense of execution performance. The economy core is recommended for cost-sensitive applications. It is non-pipelined and supports a one-cycle-per-bit serial shifter/rotator and supports the addition of custom instructions. See Table 2.1 gives a summary of the cores' features.

**Custom Instructions**

The custom (i.e., user-defined) instruction support that's provided by the Nios II cores allows designers to incorporate their own functional modules with a Nios II processor core. The source operands of custom instructions can be operands stored in the register file if required by the design. Custom instructions can also connect to signals outside the processor. A Nios II core can support up to 256 custom instructions.

Table 2.1: Nios II Processor Core Features

| Feature | Core | | |
|---|---|---|---|
| | Nios II/e | Nios II/s | Nios II/f |
| Objective | Minimal core size | Small core size | Fast execution speed |
| Pipeline | 1 Stage | 5 Stages | 6 Stages |
| Shifter/Rotator Implementation | 1 bit-per-cycle | 1 bit-per-cycle or 3-cycle shift | 1-cycle barrel shifter/rotator |
| Instruction Cache | No | Yes | Yes |
| Data Cache | No | No | Optional |
| Branch Prediction | No | Static | Dynamic |
| JTAG Debug Module | Optional | Optional | Optional |
| Custom Instruction Support | Yes | Yes | Yes |

**Peripheral Devices**

Peripheral IP cores, provided by Altera, can connect to Nios II cores via the Avalon Switch Fabric [51], which is a collection of point-to-point master to slave connections. A master can be connected to multiple slaves, and a slave can connect to multiple masters. Altera's SOPC Builder [22] automatically generates arbitration logic to organize the selection process when multiple masters attempt to drive a slave at the same time.

## 2.3 FPGA Technology

FPGA's are a special class of programmable logic devices that can be programmed and re-programmed any number of times to act virtually like any digital circuit, subject to the logic capacity of the FPGA. FPGA's have become an attractive medium for implementing embedded systems. FPGA's are constructed using three major types of resources: logic

blocks, I/O blocks, and programmable interconnections (also referred to as routing resources). In general, FPGA's are an array of programmable logic blocks, sometimes referred to as logic elements (LE's), connected together using a network of programmable switching boxes.

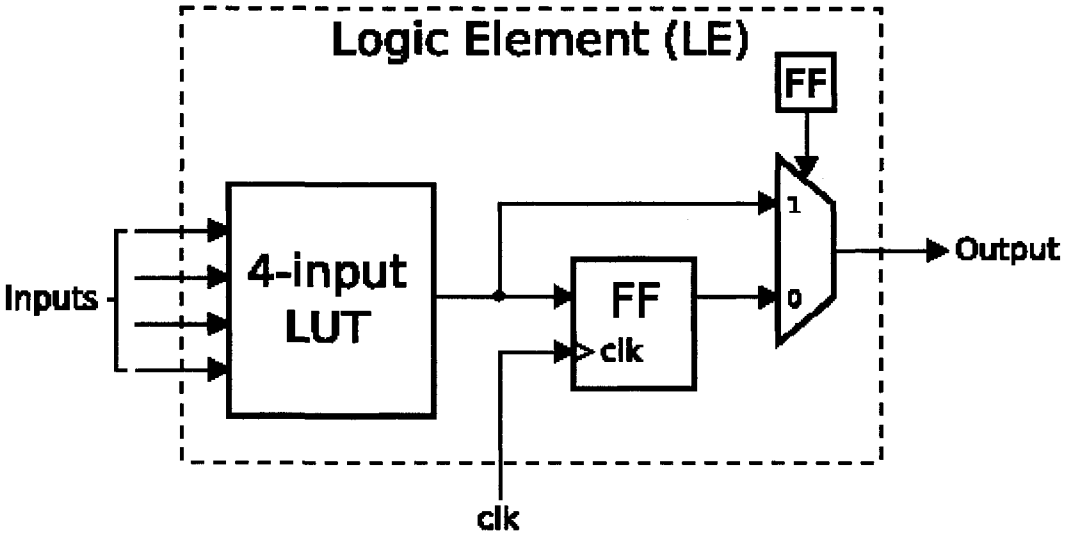



Figure 2.2: Simplified illustration of a Logic Element (LE) [53]

Logic blocks of some FPGA's are made up of a lookup table (LUT) and a flip flop. The flip flop allows the logic block to implement sequential logic. A multiplexer is used to select between the LUT and the flip flop output, as illustrated by Figure 2.2 [53]. An n-input lookup table can implement any logic function with n inputs. Previous research showed that 4-input LUT's are optimal for FPGA platforms [24]. More powerful FPGA's have logic blocks that are more complex than the one just presented [25]. Moreover, FPGA architectures differ across device families and across vendors.

While logic blocks implement logic functions, programmable interconnections (i.e., routing) are used to connect logic blocks together. By programming the logic blocks and the programmable interconnections, designers can implement virtually any digital hardware circuit's functionality. Routing in FPGA's consumes most of the chip area, and it's attributed for most of the circuit delay [24].

I/O blocks are used as a medium that connects the FPGA's internal logic with the outside pins. Often, FPGA pins can be configured as input, output or bidirectional [25].

Recent FPGA designs incorporate on-chip memory blocks, and dedicated DSP blocks to perform multiplication more efficiently. Also, due to technology advancement, recent FPGAs provide an increasingly larger number of logic blocks, memory blocks and more I/O pins. In addition to their ability to implement larger circuits, some FPGA vendors incorporate built-in hardcore processors in their FPGA chips. For example, both Altera and Xilinx provide FPGA's with built-in hardcore processors. Altera provides the Excalibur devices [26] which include the ARM922T core; the IBM PowerPC core is integrated in the Virtex-4 family of FPGA's [27] provided by Xilinx.

### 2.3.1 FPGA Design Flow

CAD tools are an essential part of circuit design targeting FPGA platforms. CAD tools are used to convert the user's specification of the digital circuit (i.e., source code describing the circuit's functionality) into a logic netlist during synthesis that can be later downloaded and programmed onto the FPGA fabric. Recent CAD tools can be used to optimize the circuit for area, speed or power consumption to meet design requirements. Figure 2.3 shows the typical steps in the design flow used by CAD tools to map the design specification into a netlist downloadable onto an FPGA [24].

Input into a CAD tool is a source code that describes the functionality of the circuit at the Register Transfer Level (RTL description). The source code is usually written using a hardware description language such as Verilog or VHDL. The synthesis process converts the source code into a netlist of basic logic gates that implement the functionality of the circuit. The netlist can then be optimized using suitable algorithms to meet design requirements.

Next, a placement algorithm is used to map each logic block from the netlist to a physical location on the FPGA fabric. Placement is an important process since it has a

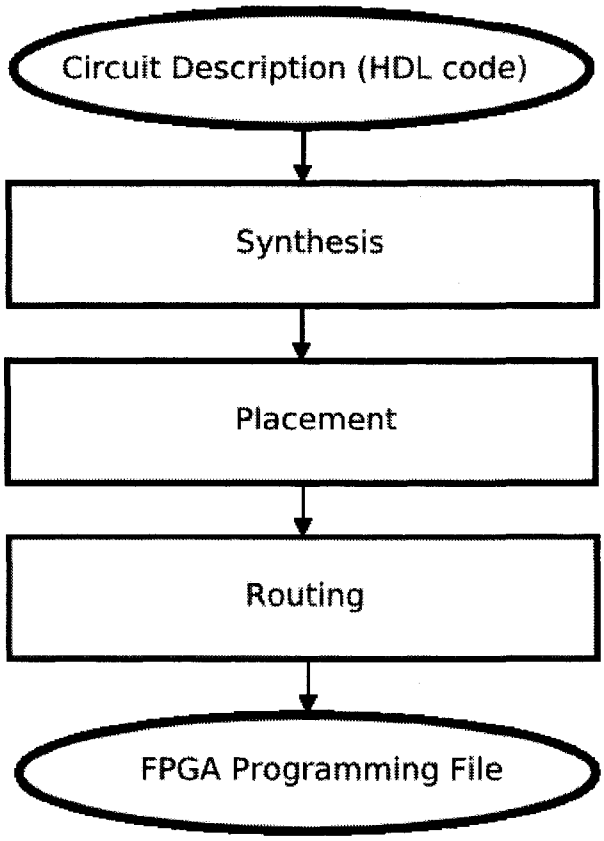



Figure 2.3: FPGA design flow

direct influence on the amount and complexity of routing performed in the next step, and as a result, placement directly influences the critical path delay of the implemented circuit. Once placement is performed, a routing algorithm is used to interconnect the placed logic blocks. The routing process is even more important than placement because of the effect it has on the critical path delay of the circuit. Routing in FPGA's consumes most of the chip area, and it's attributed for most of the circuit delay [24].

The output from the routing process is a bit stream stored in a programming file that's used to specify the state of every programmable element inside the FPGA. The entire design flow process, including synthesis, placement and routing, is referred to as *design compilation* or just *synthesis* (not to be confused with the synthesis step from the design flow). The next section will discuss the CAD tool and the FPGA device used in this research.

## 2.4 Stratix FPGA Device and the Quartus II CAD Tool

The Altera Stratix EP1S40F780C5 FPGA device was the chosen to be the target FPGA device in this research [25]. Logic blocks within the Stratix family of FPGA's are referred to as *logic elements* (LE's) in the Stratix documentation [25].

In addition to logic elements, Stratix FPGA devices contain DSP blocks (used for dedicated multiplication), phase-locked loops (PLL's), and memory blocks. Stratix devices have three different sizes of memory blocks: M512 (512 bits), M4K (4096 bits), and Mega-RAM (65,536 bytes). The blocks with the fastest speed are the M512, followed by the M4K followed by Mega-RAM. Stratix devices have anywhere between 920,448 and 7,427,520 on-chip memory bits.

Quartus II version 7.2 [28] is the CAD tool used in this research. It is provided by Altera Corporation to provide the necessary tools for circuit designs targeting Altera FPGA's. Quartus II includes a *library of parameterizable megafunctions* (LPM functions), which implement some standard building blocks commonly used by digital circuit designers. Megafunctions are often implemented more efficiently in the target FPGA than the custom design, although this is not always the case [28].

In addition to the design flow steps discussed in section 2.3.1, Quartus II uses two optional steps in its design flow: *timing analysis* and *simulation*. Timing analysis analyzes the logic netlist to locate and approximate its critical path delay. Simulation is used for design verification by comparing the expected outputs with the output of the design simulation. Quartus II provides two simulation modes: a functional simulation, and a timing simulation. Functional simulation is used to verify the functionality of the logic netlist. Timing information is separate from functional simulation. It simulates the design functionality including timing relations among signals. Therefore, timing simulation gives more accurate information about the system behaviour.

## 2.5 Design Space Exploration (DSE)

The *design space* of a digital embedded system is the complete set of all possible hardware system design configurations that can be used to achieve the system's functionality. Since embedded systems are required to perform an increasing number of tasks, the complexity of embedded systems is increasing; embedded systems are becoming more parameterized and taking on more system parameters especially with the development of FPGA platforms. Thus, the design space of embedded systems is getting extremely large (i.e., the number of possible hardware configurations that can perform a system's functionality is increasing).

Every configuration within the design space has a set of K *objectives*, and K objective functions, $F_k(p_i)$, where $p_i$ represents the parameters of the system and $k \in \{1, 2, \ldots, K\}$. Objective functions are used to measure how well a configuration from the design space meets the objectives of maximizing performance, minimizing chip area, reducing power consumption, etc. However, not all of the configurations in a design space are optimal. In fact, the majority of configurations within a design space are sub-optimal for any given application. Therefore, it's crucial that embedded system designers isolate and identify optimal configurations from a design space, since they play a key role in maximizing the system's performance and reducing its cost. This is the main objective of design space exploration.

### 2.5.1 Multi-objective Optimization

Embedded system designers are usually concerned with balancing a set of competing *objectives*. Most often, these objectives include maximizing the system's processing speed performance, and minimizing the system's chip area and power consumption. This makes the DSE process a multi-objective optimization problem, where design configurations are required to balance between the set of competing objectives. Most often, there exists an inter-dependency relationship between the set of competing objectives, meaning that improving one objective will most likely mean sacrificing another.

In multi-objective optimization problems there is not one single optimal configuration, but rather a set of optimal configurations known as the *Pareto-optimal set*. A configuration becomes part of the Pareto-optimal set if one objective cannot be improved without sacrificing another.

Embedded system designers explore the design spaces of their systems to approximate the Pareto-optimal set by eliminating all sub-optimal configurations. Unlike the design space, the Pareto-optimal set is limited in size, allowing designers to choose a suitable configuration for their system from a small and finite set of configurations.

### 2.5.2 DSE of Parameterized Cores

Embedded system designers explore the design spaces of their parameterized cores in search of a hardware platform configuration suitable for their applications. This suitable configuration is often required to balance between each of the objectives without violating any of the requirements. As the complexity of embedded systems increases, their design spaces expand. Soon, it becomes impractical to evaluate every possible configuration in the design space to come up with a suitable platform configuration, as concluded by Givargis et al [29]. Therefore, the process of DSE needs to be automated; to this end many approaches have been proposed including the use of genetic-based algorithms. A good summary of the proposed approaches can be found in the literature [10, 30, 31].

For this thesis work, a genetic-based algorithm was chosen to automate the DSE process as will be detailed in the following sections.

**Genetic-based Algorithms Approach**

The concept of genetic-based algorithms, also known as evolutionary algorithms, was developed in 1975 by Holland [32]. It proved to be effective in solving multi-objective optimization problems, like the one we face in the DSE process of parameterized soft-core processors.

In a way, genetic algorithms try to imitate the biological process of natural selection; genes from two parents are combined and passed along to their offspring. Only strong

members of a population survive and reproduce, while weak members are eliminated. Many versions of genetic algorithms have been proposed; a summary of genetic algorithms for multi-objective optimization is given in the literature [33, 34].

The genetic algorithm chosen in this research was the Simple Evolutionary Algorithm for Multi-objective Optimization (SEAMO), proposed by Valenzuela [13]. It accepts a set of design configurations, generated by the user, as input. This set has a fixed size $N$; the set is referred to as a *population*. Each member of the population is known as a *chromosome*. In our case, a chromosome represents a unique design configuration. A chromosome is composed of a collection of *genes*; in our case, a gene represents a parameter of the system. Each parameter (i.e., gene) can be assigned a value from a finite set of possible values that the parameter can take.

After receiving the input initial population, each chromosome gets evaluated separately in terms of its objectives, which are the FPGA area utilization and critical path delay in our research. The algorithm runs for a number of iterations; an iteration is referred to as a *generation*. During each iteration, chromosomes within a population are randomly grouped into pairs (i.e., *parents*); each pair is allowed to reproduce to generate an *offspring* chromosome. Two operators control the operation of the genetic algorithm: the *crossover* and the *mutation* operators.

During reproduction, genes from both parents are combined to generate an offspring chromosome according to the crossover genetic operator. A cut-point is selected randomly by the crossover operator, and the left half of one parent in the pair is combined with the right half of the other parent. The crossover operator is only applied a certain percentage of the time; this percentage is specified by the *crossover rate,* $r_c$. Next, a certain percentage of the offspring is mutated; this percentage is specified by the *mutation rate,* $r_m$. Offspring mutation involves randomly selecting one gene from the offspring and changing it to another value.

At the end of each generation the performance of offspring chromosomes gets evaluated in terms of their objectives. If an offspring chromosome performs better than its parent chromosomes, the offspring chromosome replaces one of the parent chromosomes selected at random. Otherwise, the offspring chromosome is discarded.

The genetic algorithm is allowed to run for a number of generations, G, at the end of which the final population converges toward an optimal configuration set, the Pareto-optimal set. The SEAMO algorithm has four parameters: the crossover rate ($r_c$), the mutation rate ($r_m$), the population size (N) and the number of generations (G).

I. Anderson et al. [35] conducted a case study involving a parameterized Altera Nios soft-core processor to approximate its Pareto-optimal set of design configurations. The SEAMO genetic algorithm was employed to perform an automatic exploration of the processor's design space. It was concluded that the SEAMO algorithm proved to be useful in providing a good approximation of the Pareto-optimal set of design configurations, from which designers can easily choose a suitable hardware platform design for their application.

## 2.6 Closely Related Work

P. Yiannacouras [8, 36, 37] developed a CAD, tool named SPREE (Soft Processor Rapid Exploration Environment) that was used to automatically generate soft-core processors targeted for implementation on FPGA platforms, and explore their design spaces. SPREE has two main modules, an RTL generator and a library that stores the hardware modules used to build his soft-core processor. The RTL generator is responsible for instantiating the necessary hardware component modules from the library to build a datapath according to an input description of the architecture. The RTL generator also generates the necessary control logic.

SPREE is capable of generating both pipelined and un-pipelined soft-core processors. The soft-core processors that SPREE was used to generate are based on the MIPS-I instruction set architecture [38]. Yiannacouras investigated the performance versus area

tradeoffs of various functional unit implementations (shifters and multipliers) and different pipeline depths, along with other architectural optimizations. He determined that customizing processors with the recommended features showed an improvement in performance-per-area over general purpose processors.

The main difference between this work and the SPREE system is the exploration procedure used. The SPREE system utilizes a manual design space exploration approach, where the user is to use SPREE to generate different architectural variations of the soft-core in order to compare the various design tradeoffs. On the other hand, this work uses an automatic design space exploration approach, based on a genetic algorithm, to explore the design space of the target soft-core.

B. Fort et al. [39] developed a 4-way interleaved multithreaded soft-core processor that's instruction-set compatible with Altera's Nios II soft-core processor. The authors compared the area and performance of the multithreaded soft-core processor versus two chip multiprocessors (CMP) systems, one of which is developed using Altera's Nios II soft-core processor. They concluded that using multithreaded processors in FPGA environments can result in significant area savings with comparable performance to a CMP system. This work differs from Fort's in that our processor does not support multithreading capabilities; Fort's work does not include an automatic scheme for the design space exploration.

Plavec [40] developed a methodology for efficient soft-core processor design. He generated a parameterized processor that supports a compatible instruction set as Altera's Nios soft-core processor, and compared its performance with commercial soft cores. He also investigated his processor's performance dependence on various architectural parameters. His processor's performance was on average slightly better than Altera's Nios, but occupied a larger area on FPGAs. The major difference between his work and the present work is that he did not develop a CAD tool for the automatic generation and design space exploration of soft-core processors.

The PEAS-III system [41] developed by M. Itoh followed a hardware software co-design approach that is capable of generating synthesizable RTL descriptions of pipelined processors. He developed pre-designed stage models of each pipeline stage and stored them in a library. The PEAS-III system generates the datapath of the processor core by instantiating the stage models from the library, and then cascading them in series. It enables a wide range of explorations, but in order to make a small architectural change, significant changes to its description are required.

Changing the multiply/divide unit to sequential was explored, and a multiply-accumulate (MAC) instruction was added. Several processor cores were developed using the PEAS-III system and then evaluated, including a MIPS R3000 processor, a DLX processor [42], and a simple RISC controller. In the results, area and clock speed as reported by the synthesis tool were compared. However, the PEAS-III system does not support automatic design space exploration of soft-core processors, which is what distinguishes it from this work.

SCBuild [43, 44] developed by Ian Anderson is a CAD tool developed for automated design space exploration of parameterized CPU soft-cores targeting FPGA platforms. This tool takes a template description of the core, containing information about the core's parameters and architecture, as input. It employs a genetic algorithm based design space exploration methodology to automatically explore the core's design space and returns an approximation of its Pareto-optimal set of configurations, along with an approximation of each configuration's area utilization and critical path delay on an FPGA. When prompted, this tool can also generate a synthesizable VHDL description of the core with the selected parameter values by instantiating ready made components from a library of synthesizable VHDL components that can be used to build the core. If a copy of Altera's Quartus II CAD tool is installed, SCBuild can also be used to automatically generate a Quartus II project file and compile the generated VHDL description.

It was concluded from experimental results that using this tool, designers can make intelligent decisions regarding the assignment of values to the parameters of an embedded hardware platform. SCBuild was designed to be general enough to accept any parameterized soft-core given, provided that the user supplies a template description of the core that follows proper syntax. The initial version of SCBuild, developed by I. Anderson, supports a simple RISC processor CPU design. The work in this thesis is an extension of the work initiated by I. Anderson to enable SCBuild to support and explore the design space of a widely deployed commercial soft-core processor, Altera's Nios II.

## 2.7 Summary

In this chapter we presented the background necessary to understand this research work. We started with a discussion of intellectual property (IP) cores, their classification and the concept of parameterization. Then, examples of some of the most popular commercially available soft-core processors were given. A detailed overview of Altera's Nios II soft-core processor was presented since it's the focus of this research. Next, the basic concepts of FPGA technology and the FPGA design flow were briefly explained, followed by an overview of the FPGA CAD tool and the FPGA device used in this research. After that, an introduction to design space exploration and multi-objective optimization was provided. This chapter was concluded with a presentation of previous work that is closely related to this research. In Chapter 3, a detailed discussion of the design of UW_Nios II, a soft-core processor that supports the same instruction set as Altera's Nios II, is presented.

# Chapter 3

# *UW_Nios II*

The parameterized UW_Nios II processor developed in this research is our own implementation of the Nios II standard core. UW_Nios II resembles Altera's Nios II soft-core processor and supports the same instruction set. It was developed to enable us to use it with the SCBuild CAD tool to perform DSE of Nios II processor. We now present a description of its key features.

## 3.1 Instruction Set

The UW_Nios II core supports the same instruction set as Altera's Nios II cores [18]. It supports three types of instruction word formats: I-type, R-type, and J-type.

### 3.1.1 I-Type Instructions

The main characteristic of the I-type instruction-word format is that it contains an immediate value embedded within the instruction word. I-type instructions are composed of three components:

- A 6-bit opcode field (**OP**)
- Two 5-bit register fields (**A**, **B**)
- A 16-bit immediate field (**IMM16**)

In most cases, fields A and IMM16 specify the source operands, and field B specifies the destination register. IMM16 is considered signed except for logical operations and unsigned comparisons. Figure 3.1 illustrates the format of I-type instructions.

| 31 30 29 28 27 | 26 25 24 23 22 | 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | |
|---|---|---|---|
| A | B | IMM16 | OP |

Figure 3.1: I-type instruction format

### 3.1.2 R-type Instructions

In R-type instruction-word formats all arguments and results are specified as registers. R-type instructions are made up of 3 components:

- A 6-bit opcode field (**OP**)
- Three 5-bit register fields (**A, B, C**)
- An 11-bit opcode-extension field (**OPX**)

In the majority of cases, fields A and B specify the sources operands. The destination register is specified within field C. Certain R-type instructions have a small immediate value embedded in the low-order bits of the OPX field. Figure 3.2 illustrates the format of R-type instructions.

| 31 30 29 28 27 | 26 25 24 23 22 | 21 20 19 18 17 | 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | |
|---|---|---|---|---|
| A | B | C | OPX | OP |

Figure 3.2: R-type instruction format

### 3.1.3 J-type Instructions

J-type instructions have two components:

- A 6-bit opcode field (**OP**)
- A 26-bit immediate data field (**IMM26**)

The only J-type instruction is the "call" instruction. Figure 3.3 illustrates the format of J-type instructions.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | |
|---|---|
| IMMED26 | OP |

Figure 3.3: J-type instruction format

The UW_Nios II core supports an instruction set with a total of 94 instructions including data transfer instructions, arithmetic and logical instructions, move instructions, comparison instructions, shift and rotate instructions, program control instructions, along with other control instructions. The OP field in the instruction word specifies the class of an opcode. The majority of the OP field values are for I-type instructions. For the single J-type instruction OP = 0x00. OP = 0x3a is used for all R-type instructions, in which case, the OPX field differentiates the instructions.

### 3.2 Structure

Figure 3.4 shows the design hierarchy of the UW_Nios II core. The UW_Nios II core has two main modules, the datapath and the control unit. The datapath is further divided into 4 main components: the Instruction Fetch Stage (IF), the Decode and Operand Fetch Stage (DOF), the Execute Stage (EX) and the Write Back Stage (WB).

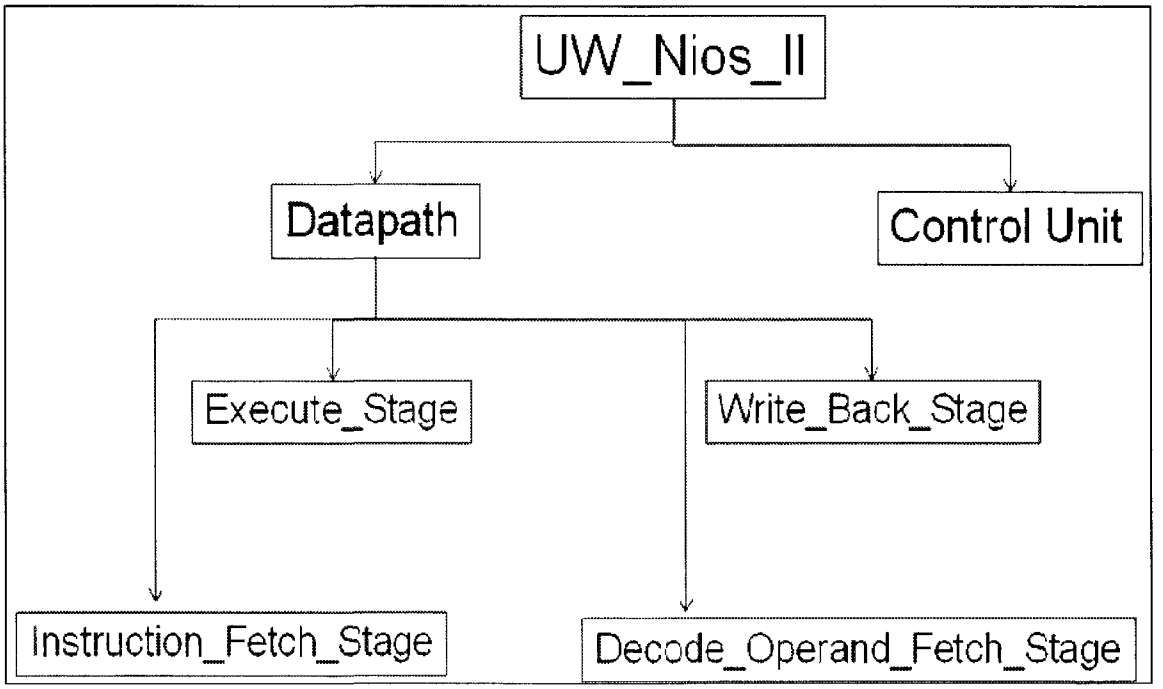



Figure 3.4: UW_Nios II Design Hierarchy

Recent work conducted by Peter Yiannacouras from the University of Toronto compared the impact of different pipeline depths (2-stage to 7-stage pipeline depths) on the performance of soft-core processors. It was concluded that both 3 and 4 stage pipelined

soft-core processors are optimal in terms of area and performance [8]. As a result, the UW_Nios II core was designed to be a four-stage pipelined RISC processor core.

In the first pipeline stage, the Instruction Fetch Stage (IF), instructions are fetched from the instruction memory. They are later decoded and operands are fetched from the Register File during the second stage, the Decode and Operand Fetch Stage (DOF). The Program Counter is incremented in this stage. The operands are then passed on to the third stage, the Execute Stage (EX), where instructions are executed by the ALU. Branch and Jump instructions are resolved in this stage and the Control Registers are read or written if necessary. Finally, the result is written back to either the register file or the data memory during the last pipeline stage, the Write Back Stage (WB). Figure 3.5 shows a simplified block diagram of the UW_Nios II's datapath core illustrating the four pipeline stages.

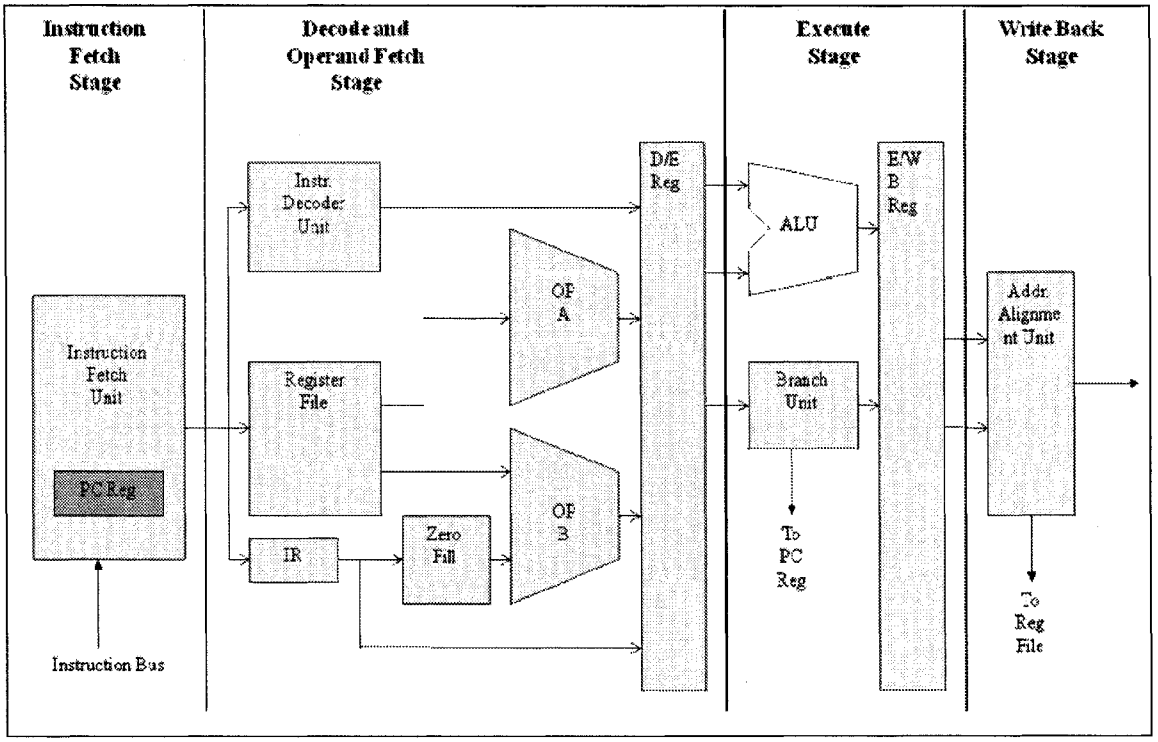



Figure 3.5: Simplified block diagram of the UW_Nios II's datapath

Results from one pipeline stage are temporarily stored in the pipeline registers before they're passed on to the next stage. The result of the Instruction_Fetch_Stage is a fetched instruction, which is temporarily stored in the Instruction Register (IR). The results from

the Decode_Operand_Fetch_Stage and Execute_Stage are stored in the pipeline registers D/E and E/WB, respectively.

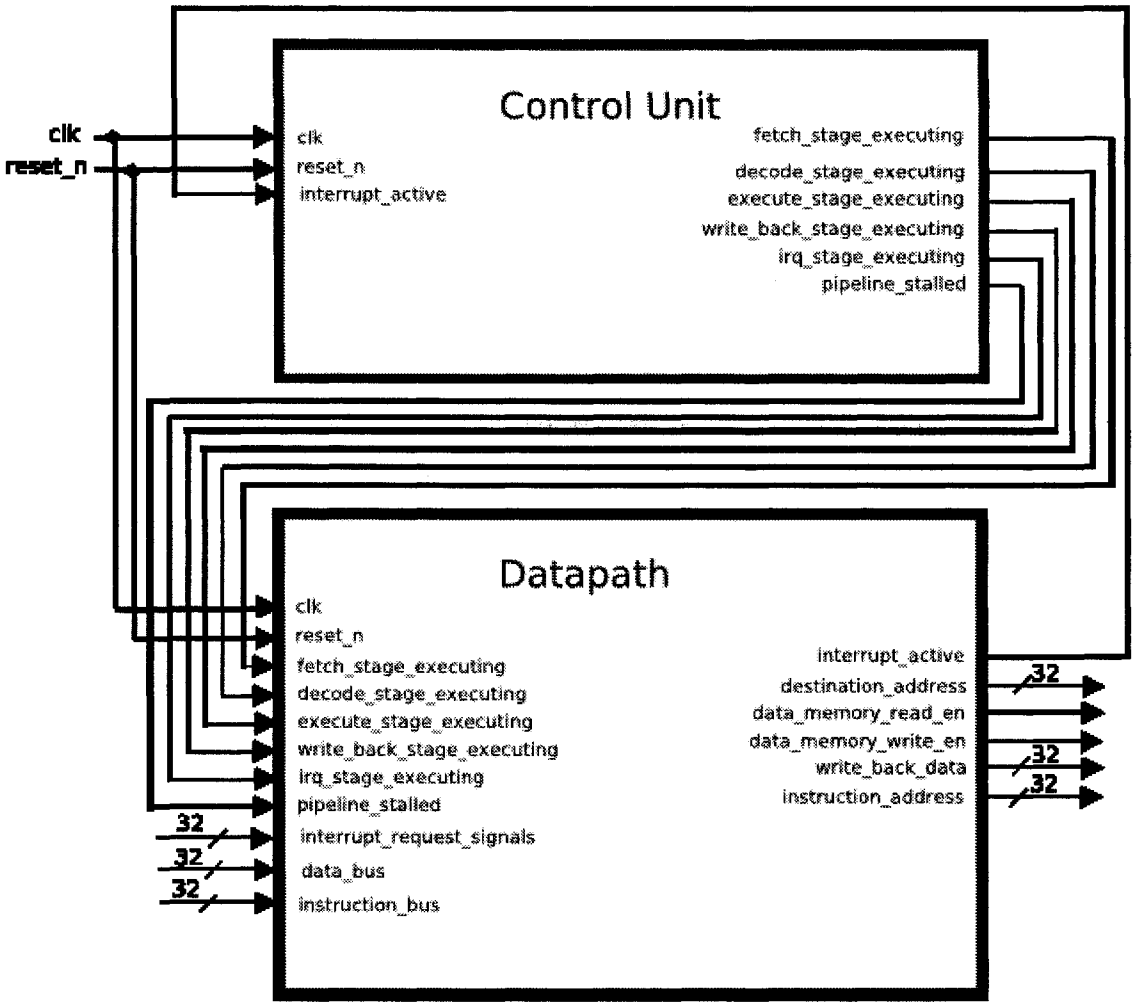



Figure 3.6: UW_Nios II block diagram with interfaces

Figure 3.6 displays a simplified block diagram of the UW_Nios II soft-core processor with the core's inputs and outputs. The datapath receives 32 "interrupt_request_signals", a 32-bit "data_bus" and a 32-bit "instruction_bus" signals from external sources, along with the "clk" and "reset_n" signals. Six control signals generated within the control unit are also passed on to the datapath to control its operation, and a 1-bit signal, "interrupt_active" is a feedback signal from the datapath to the control unit. The outputs from the datapath include a 32-bit "destination_address" signal, which specifies the address of the destination

memory word in the data memory; it's used for store operations. A 1-bit write-enable, "data_memory_write_en", and a 1-bit read-enable, "data_memory_read_en", signals are also produced by the datapath and supplied to the data memory to control the flow of information to and from the data memory module. The "write_back_data" output signal is used to transfer a 32-bit word to the data memory for store operations. Finally, the instruction memory receives a 32-bit "instruction_address" signal from the datapath; it contains the address of the instruction to be fetched.

The current version of the UW_Nios II core does not contain additional hardware for handling data and control hazards in the pipeline. Therefore, hazards must be handled in software by inserting NOPs in between instructions in a program.

Variants of the UW_Nios II core were generated and compiled using Altera's Quartus II design software version 7.2. In order to test the functionality of different variants to ensure that they functioned as expected, a number of instructions and operands were applied to the inputs of different variants of the core and the outputs were observed using the Quartus II's Simulator Tool [28]. In this way, the processor's instructions were verified to be functioning correctly.

### 3.2.1 Datapath

Data processing operations performed by the processor are handled by the datapath module. Figure 3.6 is a simplified block diagram representation of the datapath module. The four major components of the datapath are the Instruction Fetch Stage, the Decode and Operand Fetch Stage, the Execute Stage and the Write Back Stage.

The Instruction Fetch Stage module contains the Program Counter (PC) register along with associated logic. The Decode and Operand Fetch Stage module contains the instruction decoder unit, the instruction register, the DOF/EX pipeline registers, the register file and the logic necessary to fetch the appropriate operands. The register file contains thirty two 32-bit general purpose registers. The first register, R0, always contains a value of 0; writes to this register are invalid.

The Execute Stage module contains the arithmetic and logic unit (ALU), the branch unit, the control registers and the EX/WB pipeline registers. The ALU module contains the logic necessary to perform arithmetic, logical and shift operations on data stored in the register file. The ALU can be configured with or without hardware multiplication using the *Include Multiplier* parameter. The *Include Divider* parameter is used to either emulate division operations in software or implement them in hardware.

The shifter unit module can be configured to optionally handle the arithmetic, logical, shift and rotate operations. The *Arithmetic Shifter Implementation*, *Logical Shifter Implementation* and *Rotator Implementation* parameters control which shifters are included for the shifter unit module, and whether their implementations will be "basic" or "barrel".

The write back stage module controls whether data is written back to the register file or to the data memory. In the case of memory access instructions, the write back stage module performs the necessary alignment of the memory addresses and the data to be written back to the data memory, and generates the necessary enable signals. Figure 3.7 shows a more detailed block diagram illustrating the inputs and outputs of each pipeline stage in the datapath module.

### 3.2.2 Control Unit

The control unit controls the flow of information within the datapath module and the transition between the pipeline stages. In other words, the control unit determines when the pipeline stalls, and when to transfer the execution of an instruction from one pipeline stage to the next. The control unit is also responsible for taking the appropriate action in case the "reset" signal or any of the external interrupt signals are triggered.

## 3.3 Parameters

The UW_Nios II is a parameterized soft-core with a total of ten parameters listed in Table 3.1. The table below displays each parameter along with its parameter's set of possible values. Three different types of shifters are available: an arithmetic shifter, a

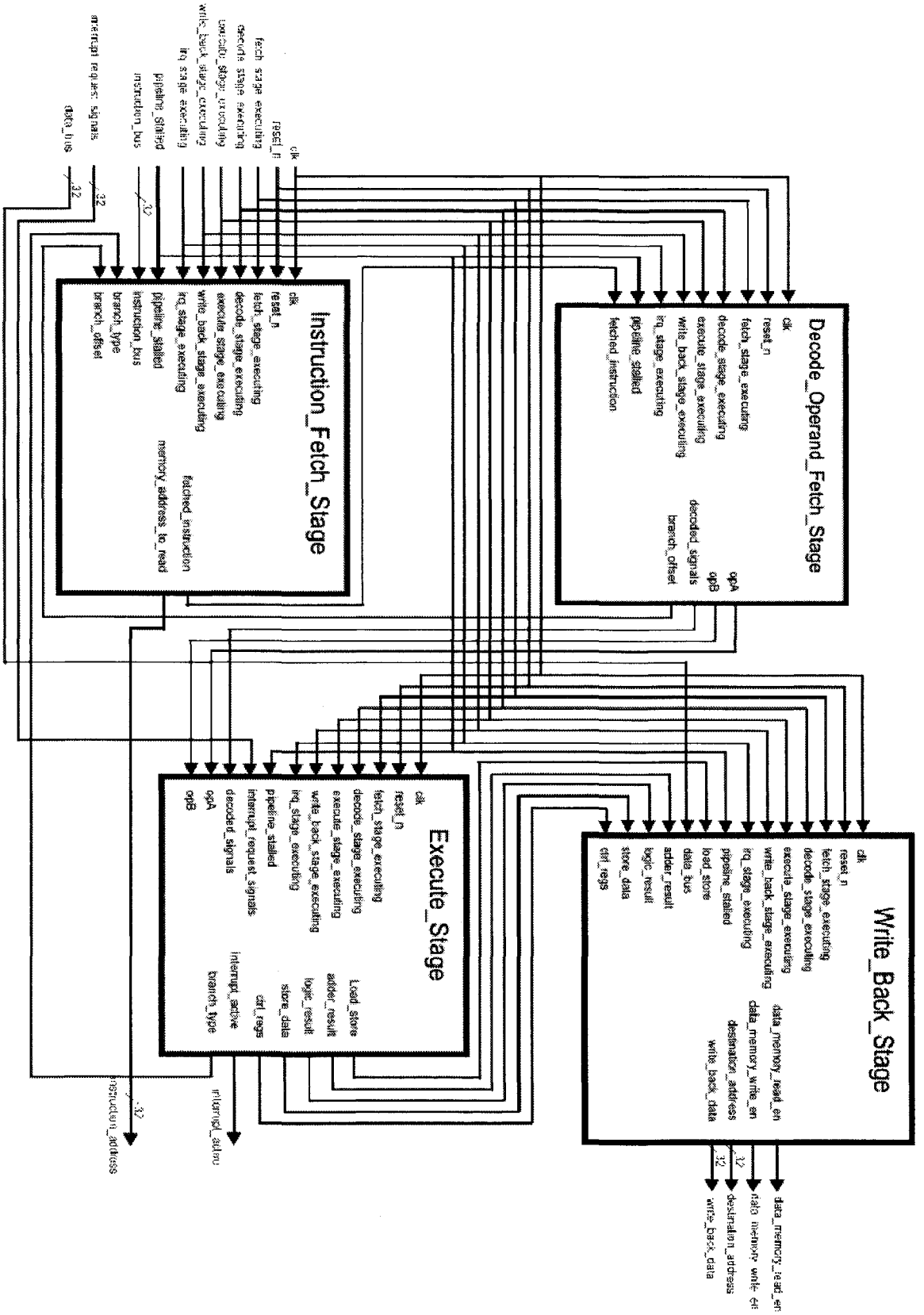



Figure 3.7: Inputs/Outputs of each pipeline stage in the datapath module

logical shifter and a rotator. The user is given the option of removing or including any or all of these types of shifters. Each of these shifter types can be emulated in software (p1 = 1; p2 = 1; p3 = 1), implemented in hardware as a "basic" shifter causing a one-bit-per-clock-cycle shift, or as a "barrel" shifter allowing shifting of multiple bit positions in a single clock cycle. The core can have either a signed or unsigned hardware multiplier module. If no multiplier implementation is chosen, multiplication will be emulated in software (an exception will be triggered upon a multiplication instruction). The multiplier can be implemented using logic element (LE's) resources within the FPGA or, to achieve a better performance, the multiplier can be implemented using dedicated DSP multiplication blocks. The ALU can be configured with or without a hardware divider module using either the *Include Signed Divider* parameter or the *Include Unsigned Divide* parameter. In case no hardware division is picked, division operations will be emulated in software (an exception will be triggered upon a division operation). The designer can choose to implement the instruction decoder, register file and pipeline registers using LE's or, if LE resources are more critical, they can be implemented using dedicated memory blocks. The output from the instruction decoder is a set of control signals that make up the control word, which will later be used to define the operations that need to be performed to implement the decoded instruction.

## 3.4 Comparison of UW_Nios II and Altera Nios II

After the design of the UW_Nios II was complete, it was necessary to see how well its variants performed when compared with Altera's Nios II variant cores. This section presents the results of comparison between the UW_Nios II variants against Altera's Nios II variant cores. Note that each variant is obtained using a specific set of parameter values.

### 3.4.1 FPGA Device and CAD Tools

While the VHDL source code description of the UW_Nios II soft-core processor is independent of the target FPGA architecture, a particular FPGA device was targeted for

performing our FPGA-based exploration. The targeted device is Altera's Stratix EP1S40F780C5 FPGA device [25], which is a mid-sized device in the Stratix family with

Table 3.1: UW_Nios II Processor Hardware Parameters

| Parameter | Possible Values |
|---|---|
| Arithmetic Shifter Implementation (p1) | (1) None, (2) Basic, (3) Barrel |
| Logical Shifter Implementation (p2) | (1) None, (2) Basic, (3) Barrel |
| Rotator Implementation (p3) | (1) None, (2) Basic, (3) Barrel |
| Include Signed Multiplier (p4) | (1) No (i.e., emulated in SW), (2) Using LE's & area optimization, (3) LE's & speed optimization, (4) Using DSP blocks |
| Include Unsigned Multiplier (p5) | (1) No (i.e., emulated in SW), (2) Using LE's, (3) Using DSP blocks |
| Include Unsigned Divider (p6) | (1) No (i.e., emulated in SW), (2) Using LE's |
| Include Signed Divider (p7) | (1) No (i.e., emulated in SW), (2) Using LE's |
| Instruction Decoder Implementation (p8) | (1) Using LE's, (2) Using RAM memory blocks |
| Register File Implementation (p9) | (1) Using LE's, (2) Using Memory blocks |
| Pipeline Register Implementation (p10) | (1) Using LE's, (2) Using Memory blocks |

the fastest speed grade. It has a total LE capacity of 41,250 LE's, a total of 3,423,744 RAM memory bits, and a total of 14 DSP blocks. In addition, Altera's Quartus II v7.2 [28] CAD software was used for the synthesis, technology mapping, placement and routing of all designs to the targeted FPGA device.

Quartus II gives its users the option of choosing between a speed, a balanced, or an area optimization option. With a speed optimization technique the design is synthesized so that speed performance is maximized at the expense of extra utilization of the LE resources of the FPGA. When the area optimization technique is chosen, the design is synthesized so that LE resource utilization is minimized at the expense of slower processing speed performance. The balanced optimization technique provides a balance between high speed performance and minimal LE resource utilization.

### 3.4.2 Metrics for Evaluating Soft-core Processors

In order to measure the speed performance and area utilization of the different variants of the UW_Nios II soft-core processor, an appropriate set of measurement metrics is required. For an FPGA device, area utilization is measured by counting the number of equivalent resources consumed. In the Stratix family of FPGAs, the main resource is the Logic Element (LE), where a LE is composed of a 4-input lookup table (LUT) and a flip flop. Thus, area is given in terms of the equivalent number of LEs consumed.

For now, speed performance is measured in terms of the maximum clock frequency (in Mhz) achieved by the processor (based on the critical path delay), as reported by Quartus II's Timing Analyzer Tool, after placement and routing.

### 3.4.3 Comparison with Altera's Nios II Cores

To ensure that our comparisons with Altera's Nios II cores were as fair as possible, several measures were taken. Comparison with the Nios II Economy core was omitted because it is an un-pipelined soft-core processor while the UW_Nios II is a four stage pipelined core. Thus comparison is performed against the Standard and the Fast cores only. Each of the two Nios II cores was generated with memory systems identical to those used in our designs: two 8KB blocks of RAM for separate instruction and data memory. Caches were not accounted for in our measurements, though extra logic to support the caches will inevitably count towards the Nios II areas. Nios II cores support operating systems (OS) instructions, which are not yet supported by the UW_Nios II variants. Despite the previously mentioned differences, we still believe that comparisons between Altera's Nios II cores and the UW_Nios II variants are fair.

When Altera's Nios II Standard Core was synthesized, placed and routed, with serial shifters and software emulation of multiplication and division, a maximum clock frequency of 222 Mhz was achieved. This core consumed the equivalent of 1290 logic elements. When a similar UW_Nios II core was synthesized, place and routed with a speed optimization option, a maximum clock frequency of 205 Mhz was achieved; which is

within 7% of Altera's Standard Core. In this core, the register file was implemented using dedicated on-chip RAM memory blocks and the pipeline registers were implemented using logic elements. And when a similar UW_Nios II core was synthesized with an area optimization option, up to a 47% saving in area compared to Altera's Standard core was achieved. This large saving in area was countered by a 60% drop in clock frequency. In this core, the register file and the pipeline registers were both implemented using on-chip RAM memory blocks. Table 3.2 illustrates these results along with other similar results.

Table 3.2: Comparison with Altera's Nios II Standard Core

| Shifters | Multiplier/ Divider | Instruction Decoder Impl. | Register File Impl. | Pipeline Register Impl. | Optim. Option | Clk (Mhz) | Eq. LE's | % Decrease in freq. | % Reduction in LE usage |
|---|---|---|---|---|---|---|---|---|---|
| Serial | Software Emulation | LE-based | RAM-based | LE-based | Speed | 205 | 935 | 7.6 | 27.5 |
| Serial | Software Emulation | RAM-based | RAM-based | LE-based | Speed | 176.41 | 875 | 20.5 | 32.1 |
| Serial | Software Emulation | RAM-based | RAM-based | RAM-based | Area | 90.01 | 677 | 60 | 47 |
| Serial | Software Emulation | LE-based | RAM-based | RAM-based | Area | 78.21 | 729 | 65 | 43 |
| Serial | Software Emulation | RAM-based | RAM-based | LE-based | Area | 109.68 | 738 | 51 | 42 |
| Serial | Software Emulation | LE-based | RAM-based | LE-based | Area | 98.9 | 800 | 55 | 38 |

When Altera's Nios II Fast Core was synthesized, place and routed, with barrel shifters and hardware multiplication using dedicated on-chip DSP blocks, a maximum clock frequency of 200 Mhz was achieved, with the equivalent of 1715 logic elements consumed. A similar UW_Nios II core, with LUT-based barrel shifters, synthesized with a speed optimization option achieved a maximum clock frequency of 125 Mhz; which is about 37% less than Altera's Fast Core. This core included a register file implemented using dedicated on-chip RAM memory blocks and the pipeline registers were implemented using logic elements. The reason for this big gap in clock frequency is because Altera's Fast Core is hand-optimized to provide the fastest execution speed. When a similar UW_Nios II core

was synthesized with an area optimization option, up to a 30% saving in area compared to Altera's Standard core was achieved. This saving in area was countered by a 59% drop in clock frequency. In this core, the register file and the pipeline registers were both implemented using on-chip RAM memory blocks. Table 3.3 illustrates these results along with other similar results.

Table 3.3: Comparison with Altera's Nios II Fast Core

| Shifters | Multiplier/ Divider | Instruction Decoder Impl. | Register File Impl. | Pipeline Register Impl. | Optim. Option | Clk (Mhz) | Eq. LE's | % Decrease in freq. | % Reduction in LE usage |
|---|---|---|---|---|---|---|---|---|---|
| Barrel | DSP blocks | LE-based | RAM-based | LE-based | Speed | 125 | 1554 | 37.5 | 9.4 |
| Barrel | DSP blocks | RAM-based | RAM-based | LE-based | Speed | 123.53 | 1445 | 38.2 | 15.7 |
| Barrel | DSP blocks | RAM-based | RAM-based | RAM-based | Area | 82.3 | 1202 | 59 | 30 |
| Barrel | DSP blocks | LE-based | RAM-based | RAM-based | Area | 89 | 1249 | 55 | 27 |
| Barrel | DSP blocks | RAM-based | RAM-based | LE-based | Area | 113.62 | 1273 | 43 | 25 |
| Barrel | DSP blocks | LE-based | RAM-based | LE-based | Area | 104.53 | 1320 | 48 | 23 |

Bearing in mind the design differences between Altera's Nios II cores and our UW_Nios II variants, it is not our goal to draw architectural conclusions from comparisons with Altera's cores, since we do not have access to Altera's Nios II architectures. The main reason for presenting performance comparisons between Altera' cores and our variants is to show that our design is relatively competitive when compared with commercial, hand-optimized soft-core processors.

### 3.4.4 Hardware vs. Software Multiplication Support

Whether multiplication is implemented in hardware or emulated in software has a large impact on the speed performance and area of soft-core processors. Hardware multipliers occupy a large area on FPGA platforms but provide better processing performance. Hence, Altera's Nios II/e core does not support hardware multiplication, while it is available for the other two cores (Nios II/s and Nios II/f). Many variations of hardware multipliers are available, variations that trade off area for performance. One example is a multiplier that

uses a software multiplication routine in which hardware performs a portion of the multiplication operation. This multiplier is much faster than the typical software version, which uses a series of shift and add operations. In this work, we do not consider such hybrid implementations; instead we focus only on either full or no hardware multiplication support.

New FPGAs have dedicated on-chip DSP blocks that are capable of supporting full hardware multiplications. We conducted an experiment on our UW_Nios II to compare its performance when hardware multiplication was implemented using the DSP blocks one time, and using logic elements (LEs) the second time. As shown in Figures 3.8 and 3.9, in the case of LE-based hardware multiplication, the UW_Nios II core used 37% more area, and had a clock frequency that was 43% slower than a similar core with DSP-based hardware multiplication. From this experiment, we conclude that DSP-based hardware multipliers are a better choice than the LE-based version.

Research conducted by Yiannacouras et al [37] showed similar results. They generated different variations of a RISC soft-core processor that supports a MIPS I instruction set architecture (ISA). Some of those variants supported full hardware multiplication and in the rest, multiplication was emulated in software. A set of benchmark circuits were run on their variants and their performance was compared. In terms of the number of cycles required to execute the benchmark circuits, it was found that some applications were minimally sped up while others benefited up to 8X from a hardware multiplier. Thus it was concluded that multiplication support is an application-specific design decision. In general, especially for multiply-intensive applications, hardware multiplication consumes more area but provides better processing performance.

### 3.4.5 Register File Implementation

New FPGAs have dedicated on-chip RAM memory blocks that can be used as storage elements. Whether the register file is implemented using logic elements (LEs) or using dedicated on-chip RAM memory blocks has a large impact on the speed performance and

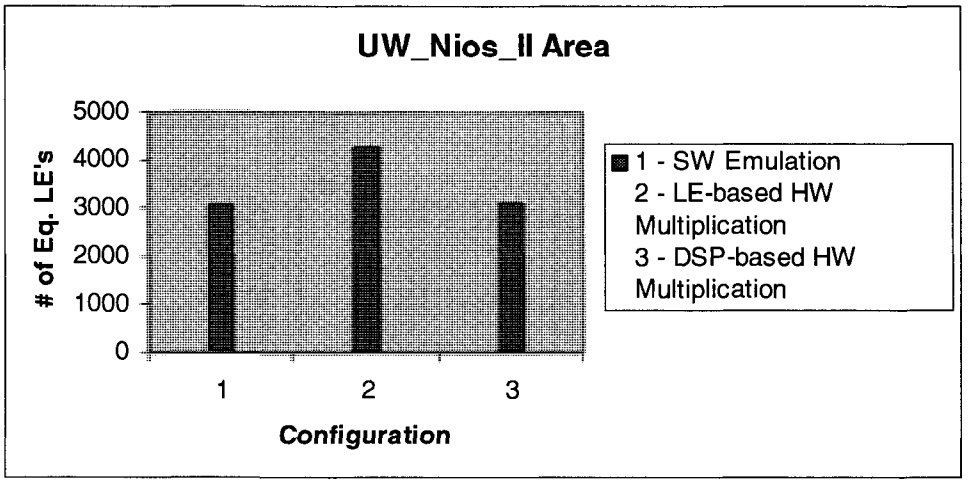



Figure 3.8: UW_Nios II Area

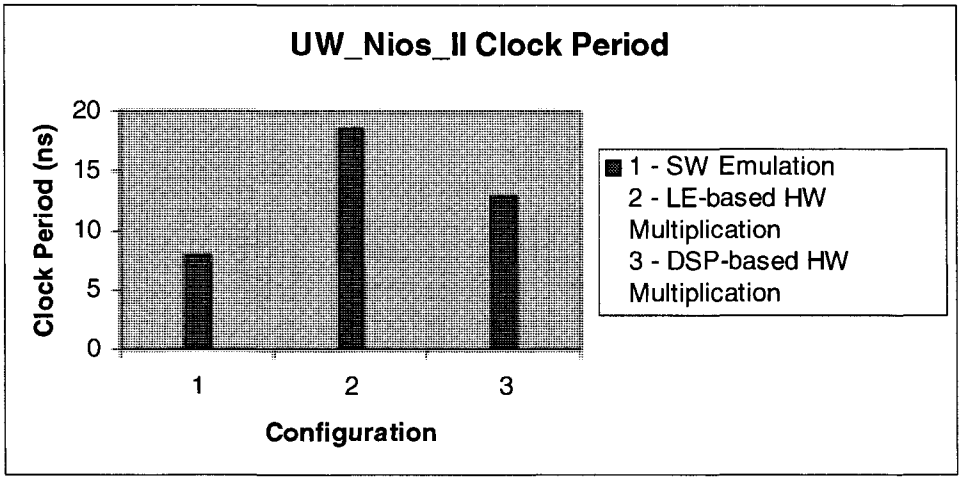



Figure 3.9: UW_Nios II Clock Period

area of soft-core processors. A very important observation can be made from Figures 3.10 and 3.11. During the course of our research, we compared two similar variations of the UW_Nios II soft-core processor. In the first variant, the register file was implemented using LE's, and in the second one, the register file was implemented using RAM memory blocks. It was found that the first variant occupied 400% more logic elements and had a clock frequency that was 37% smaller when compared with the second variant. In other words, LE-based implementation of the register file not only occupies an extremely large area on FPGA platforms, but also degrades speed performance significantly.

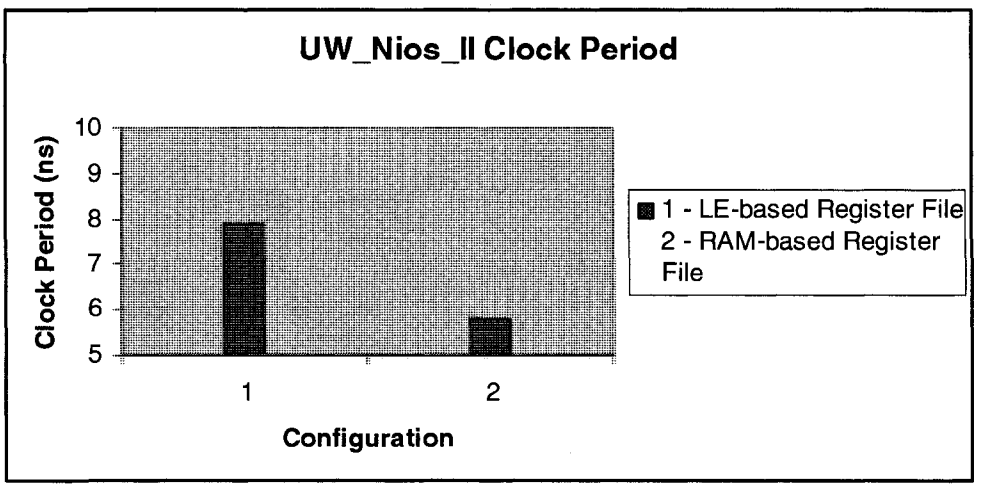



Figure 3.10: Clock Period for Register File Implementation

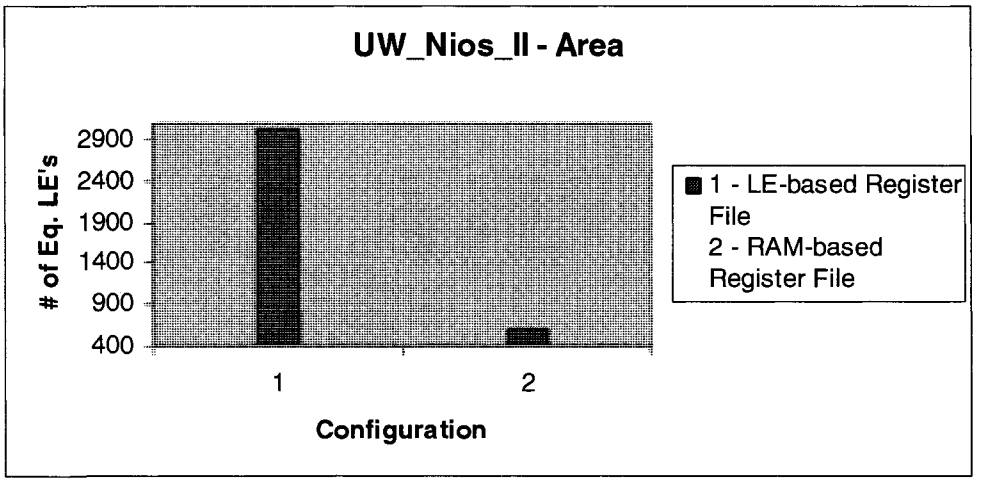



Figure 3.11: LE Utilization for Register File Implementation

When the register file is implemented using LE's, a large area is consumed because one lookup table (LUT) is required to store 1 bit (a LUT is composed of a 4-input lookup table and a flip flop). Therefore, a 32-bit register requires at least 32 LUT's to implement it. The reason for the significant rise in clock frequency (in the cased of a RAM-based register file over the LE-based version) is that the RAM blocks are optimized memory components, and thus access times are shorter. Also, the LE's used to implement the register file (in an LE-based register file) could be scattered throughout the FPGA fabric after placement, thus complicating routing process and resulting in longer routes. This in turn increases the critical path delay and translates into a smaller clock frequency. From this experiment, we conclude that RAM-based register files are a better choice than the LE-based version.

### 3.4.6 Pipeline Register Implementation

Finally, an experiment was conducted to study the impact of pipeline register implementation on the overall performance of the processor. Figures 3.12 and 3.13 illustrate that impact on the processor's clock period and equivalent area respectively. In this experiment, two similar UW_Nios II variants were compared; in the first variant the pipeline registers were implemented using logic elements (LE's) and, in the second one, they were implemented using RAM memory blocks.

It was found that the first variant had a clock frequency was about 27% larger than the second variant, but it consumed 55 more LE's. This increase in area is relatively small compared to the gain achieved for the clock frequency. This increase in clock frequency can be attributed to the fact that using LE's to implement pipeline register allows them to be placed closer to the logic of the next stage, resulting in shorter routes. That in turn translates into a shorter critical path delay resulting in a shorter clock period (i.e., higher clock frequency). From this experiment, we conclude that LE-based pipeline registers are a better choice than the RAM-based version.

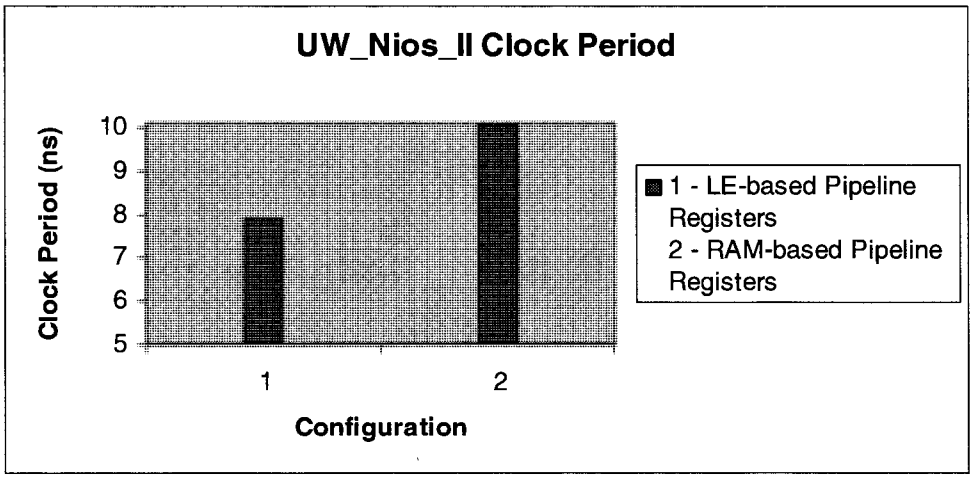



Figure 3.12: Clock Period for Pipeline Register Implementation

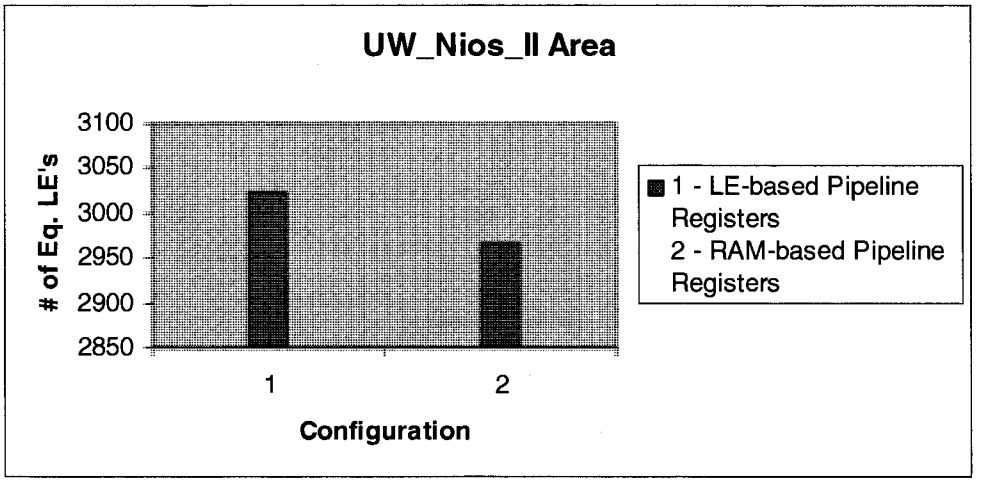



Figure 3.13: LE Utilization for Pipeline Register Implementation

### 3.5 Summary

This chapter started by presenting the design and implementation of the UW_Nios II soft-core processor. A review of the instruction set supported by the UW_Nios II soft-core processor was first illustrated, followed by a description of the datapath and the control unit, respectively. Next, the set of parameters for the core were summarized. The remaining part of the chapter discussed the experiments conducted to evaluate the performance of the UW_Nios II soft-core processor along with the proposed metrics of evaluation. A comparison between the UW_Nios II's variants and Altera's Standard and Fast cores was presented. It was found that, in the best, the UW_Nios II was 47% smaller and had a critical path delay that was only 7.6% larger than Altera's Standard core. Finally, a study of the effects that some parameters have on the core's performance when varied across their range of possible values was presented. It was concluded that a RAM-based implementation of the register file and an LE-based implementation of the pipeline registers resulted in a better overall performance.

In the next chapter, a discussion of the design and implementation details of the SCBuild CAD tool is provided along with an overview of the results of some experimental studies that were conducted using SCBuild and the UW_Nios II soft-core processor.

# Chapter 4

# *Design Space Exploration of UW_Nios II*

This chapter starts by presenting the design and implementation of SCBuild (Soft-Core Build). SCBuild is a CAD tool developed to explore the design space of a given parameterized soft-core processor. A description of the target core, containing some of its major features, is supplied to SCBuild as input. Later in the chapter, the design space exploration experiments conducted throughout the course of this research are presented and the results are analyzed. In these experiments, SCBuild was supplied with an input template description of the UW_Nios II parameterized soft-core processor. Next, SCBuild was used to apply the SEAMO genetic algorithm to the supplied core to approximate its Pareto-optimal set.

## 4.1 SCBuild – a CAD Tool for the DSE of the UW_Nios II

SCBuild (Soft-Core Build) is a CAD tool that was designed to perform an automated exploration of the design space of a parameterized RISC soft-core. This tool was developed by Ian Anderson during his master's program at the University of Windsor. Figure 4.1 [43] illustrates a simplified diagram of SCBuild's system environment.

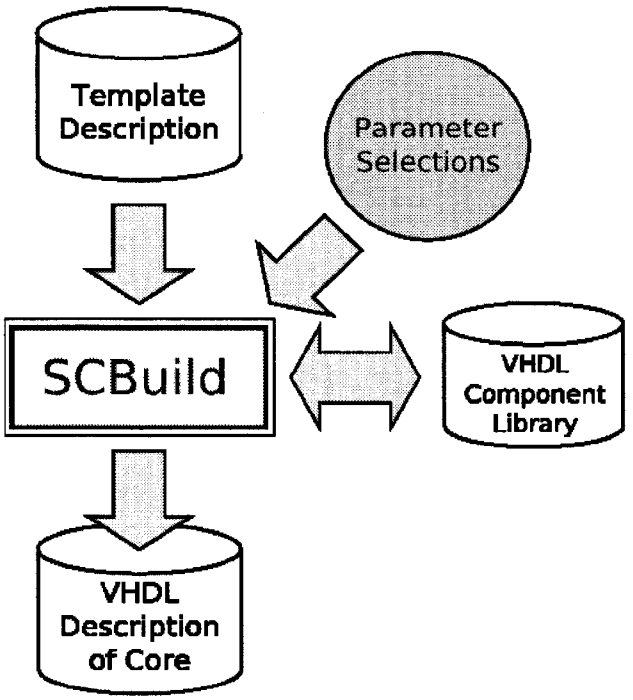



Figure 4.1: SCBuild System Environment [43]

SCBuild takes a *template description* of the target core as input. The template description contains details about the hierarchy of sub-components that make up the core, and also contains information about the parameters of the core. After supplying the template description, SCBuild uses the SEAMO [13] genetic algorithm to explore the core's design space and approximate its Pareto-optimal set of configurations. SCBuild provides an approximation of each configuration's area (i.e., number of equivalent logic elements consumed) and critical path delay (reported in nanoseconds).

After assigning a value to each parameter, this tool is capable of generating structural VHDL description of optimized variants of the target core, with the user-selected parameter values, by instantiating components from a library of synthesizable VHDL components, the *VHDL Component Library*. This library contains modules that are the building blocks for the soft-core. If a version of Altera's Quartus II software [28] is installed on the machine, when prompted, SCBuild can create a Tool Command Language (Tcl) [45] script file that's used by Quartus II to create a new Quartus Project File (.qpf), compile the generated VHDL code and save the synthesis results in a text file.

SCBuild is not restricted to using a single template description. Instead, it is general enough that it's able to accept and work with any template description of any core, provided that this description complies with the syntax required by SCBuild.

The initial version of SCBuild used a RISC processor core whose architecture is presented in [46]. This soft-core has a simple architecture and is not commercially used. During the course of this research, SCBuild was enhanced to accommodate the UW_Nios II soft-core processor. The UW_Nios II supports the same instruction set as Altera's Nios II soft-core processor [18], which is a widely deployed commercial soft-core processor. This chapter presents a brief overview of the design and implementation of SCBuild.

### 4.1.1 The Core's Template Description

SCBuild is a CAD tool that's was developed using the C++ programming language. In order for it to be able to explore the design space of a parameterized soft-core, a template description of the core needs to be supplied. This template description is a collection of files that describes certain features about the target core, such as its parameters and architecture design hierarchy, that the software tool can read, properly translate and map onto data structures. This allows the tool to manipulate the input description to produce the desired output, which in this case is the Pareto-optimal set of configurations. The format of the template description files will be briefly presented later in section 4.1.2.1; refer to section 4.3.1 and Appendix A in [43] for more details on the format of these files.

SCBuild was designed to hide as much of the implementation details of the target soft-core as possible so that end-users do not have to concern themselves with many of the core's design details. The following is a list of the core's features that the input template description is required to have:

1. The core's parameters: The template description must contain a list of the core's parameters along with the set of possible values that each parameter can be assigned. Each sub-component within the core can have its own set of parameters that can be

assigned certain values. The input template description should define each parameter along with their set of possible values.

2. The effects each parameter has on the core's architecture: Often, varying a parameter's value changes the underlying architecture of the core. For example, some parameters are responsible for indicating the physical implementation of some of the functional units used within the core. Varying this kind of parameters changes the physical implementation of the functional unit, and therefore changes the physical implementation of the core as a whole. Other parameters control the instantiation or elimination of complete functional units within the core (eg., include hardware support for multiplication or emulate it in software). This kind of parameters has a substantial impact on the resulting core. Therefore, the input template description should include details about the ways each parameter can change the core's architecture.

3. The set of possible physical implementations that a sub-component can have: Some components have multiple physical implementations that are functionally equivalent, but differ in the way they manipulate input data to produce the output result (i.e., they are structurally not the same). This difference often translates into varying performance levels, area utilization, power consumption and/or other objectives. For instance, a shifter can be implemented as a serial shifter, barrel shifter, or it can have some other functionally equivalent implementation. Each implementation has its own VHDL file that describes it; these files are stored in a library. The input template description should specify all the possible physical implementations that a sub-component may have.

4. The design hierarchy of sub-components that make up the core: The design of a soft-core processor is a complicated task. Describing the behaviour of an entire core using a single module (i.e., a single VHDL entity) is challenging. This task is drastically simplified by breaking the design into a number of smaller sub-components that collectively define the core's behaviour. Every sub-component can itself be built

using any number of smaller sub-components and so on. The core's template description should show its design hierarchy.

5. The connectivity of the core's sub-components: This contains information about the interface that each sub-component has with other sub-components and modules.

### 4.1.2 SCBuild CAD Flow

SCBuild performs its tasks by executing a series of steps. These steps are better illustrated using the flowchart in Figure 4.2 [43]. These steps define the CAD flow for SCBuild. The following sections will discuss each step briefly.

#### 4.1.2.1 Design Entry

This is the initial step in the CAD flow. The user supplies the input template description of the target parameterized core at this stage. In this research, the template description was developed manually. In future work, this step can be automated by creating a GUI tool that can be used to develop the template description.

As mentioned in previous sections, a template description is a collection of files that contain certain details about the target parameterized core that are required by SCBuild. These files contain *Extensible Markup Language* (XML) code [47]. A more detailed description of the format of the template files is provided in Appendix A of [43]. To summarize, each module in the VHDL Component Library is represented in the template description using a *template Component*; the description of each template component is stored separately in an XML file. Every template component file must contain the name of the component and a list of the names of the component's parameters. Each parameter is assigned a list of possible values that it can take, as well as a default value. Every parameter is further classified as a "scalable", "implementation", or "general" type parameter.

Scalable type parameters are assigned numerical values; they are used to represent bit-widths or any type of numerical quantities (i.e., parameters that are represented using numerical values). They are represented using "generic" statements in VHDL [48].

Modules that have multiple possible physical implementations are represented using the implementation type parameters. These parameters are used to indicate which physical implementation of the functional unit is used (i.e., they are used to control the VHDL implementation of the module in the VHDL code produced by SCBuild). General type parameters are used to indicate possible changes in the component's architecture.

In addition to the template component name and parameter list, for a component that is constructed using one or more sub-components, the template component description contains a list of ports and sub-components used to construct it. Ports define the component's interface with other components.

The template description should also contain a *Parameter Dependencies file,* an *Objectives file,* a *Top-Level Entity File,* and a *System file*. The Parameter Dependencies file serves to define any hard interdependencies between various parameters. No hard interdependencies currently exist between any of the parameters used in the UW_Nios II core. The Objectives file contains the equations that approximate how each parameter affects the FPGA area utilization (defined as the equivalent number of logic elements utilized) and the core's critical path delay (reported in nanoseconds). The Top-Level Entity File contains a summary of all the core's parameter names, their possible values, their type and their default values. The System file stores the names of the Parameter Dependencies file, the Objectives file, along with the names of the template component files. More details about the input template description and the content of the template component files can be found in [43].

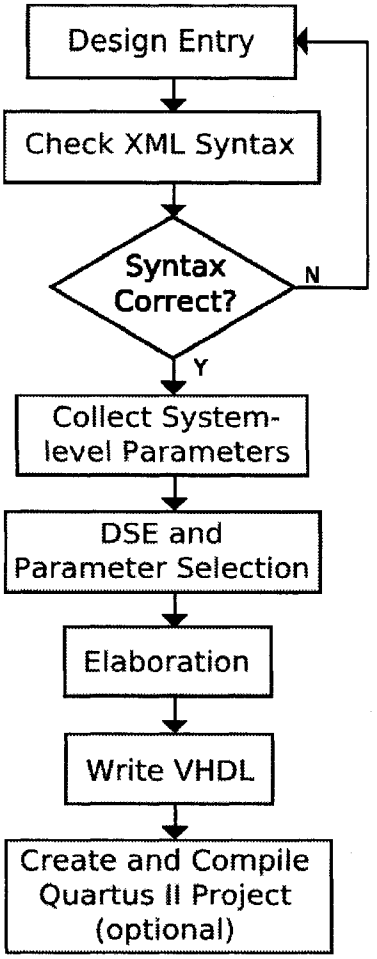


Figure 4.2: SCBuild CAD Flow [43]

#### 4.1.2.2 XML Syntax Checking

Once provided with the template description files, SCBuild proceeds to check these files for any possible errors that may exist. This step ensures that these files follow proper XML syntax required by SCBuild. Any errors should be fixed for execution to continue.

#### 4.1.2.3 Collect System Level Parameters

During this stage, SCBuild reads the Top-Level Entity template component file. Information about the core's parameters provided in this file is stored. At this stage, users are free to lock any or all the parameters to certain values, or keep them free to be used in the design space exploration process of the core; locked parameters will not be changed during this process.

#### 4.1.2.4 DSE and Parameter Selection

Once the core's parameters are obtained, SCBuild prompts the user to supply the SEAMO algorithm parameters, which are the population size, the number of generations for which to run the algorithm, the crossover and the mutation rates. Then, SCBuild explores the design space of the soft-core by applying the SEAMO genetic algorithm to the free parameters of the system. If there are any hard parameter interdependencies rules specified in the Parameter Dependencies file, SCBuild makes sure that none of these rules are violated during the DSE process (Refer to [43] for more on hard parameter interdependencies).

Any parameterized core supplied to SCBuild is allowed to have K objectives with their corresponding K estimation equations. Some of the objectives can be FPGA area utilization, critical path delay, power consumption along with others. In order to develop the forms of the objective estimation equations, a set of configurations representative of the core are synthesized using Quartus II. Information about the FPGA resource utilization and critical path delay are gathered from reports provided by Quartus II at the end of each configuration's synthesis. The forms of the objective estimation equations, $f_{i,k}(p_i)$ in equation 4.1 (discussed later in section 4.2.2), are determined by studying the relationships between each parameter value and the corresponding objective values. Once the form of each objective estimation equation is obtained, P-dimensional regression analysis can be applied to the collected data to determine the values of the regression coefficients $a_{0,k}$, $a_{1,k}$, ..., $a_{p,k}$. The objective estimation equations should provide estimations with acceptable degree of accuracy.

The Pareto-optimal set of configurations is the outcome of the DSE process. SCBuild uses the equations included in the Objectives file to calculate approximating values for the area and critical path delay. SCBuild displays each configuration's parameter values, along with its estimated area and critical path delay values. At this point, the user can select a configuration from the Pareto-optimal set to lock all the parameters to specific values.

#### 4.1.2.5 Elaboration

After locking all the parameters of the core to certain values, SCBuild proceeds to generate the VHDL structural code for the core with the selected features and parameter values specified previously during the elaboration stage. To achieve this goal, SCBuild constructs two intermediate representations of the system using data obtained from the input template description files.

The first representation is the *System-level* description of the *hierarchy of template components*. As the name implies, this representation uses the template description files to gather information about every component in the system, starting with the top level entity, and which sub-components are instantiated under it. In this representation, SCBuild forms a hierarchical representation of component parameters by linking each sub-component's parameter(s) to parameters of their parent component, and so on up the hierarchy up to the top level entity of the system. The second representation is the *Register Transfer Level* representation. This representation describes the system at the RTL level of abstraction; this description can directly be used to generate the VHDL code of the core. More specifically, it lists the ports the each sub-component and ways in which its ports are interconnected with the ports of other sub-components (refer to [43] for more details on each representation). Once SCBuild has finished forming the two representations, it proceeds to form the final structural VHDL description of the system.

#### 4.1.2.6 Creating Quartus II Project File and Compilation

If a copy of Altera's Quartus II software is installed on the machine then, when prompted by the user, SCBuild can generate a Tool Command Language (for short Tcl) script file [45]. Quartus II uses this file to create a new project file, include the generated VHDL files in the project, perform a complete synthesis of the entire design and store the synthesis results reported by Quartus II in a text file.

## 4.2 Enhancements to SCBuild

The template description contains a set of XML files that contain certain details about the processor core required by SCBuild to perform DSE. Every file in the template description describes one template component using XML. This section explains some of the key enhancements made to SCBuild to enable it to work with the UW_Nios II core.

Every template component description file lists the component's name and parameters. For examples that illustrate the exact syntax, see Appendix A in [43]. One of the files that has been modified was the *Objectives File*. Varying each parameter has a unique effect on the area and delay of the resulting core. These effects are modeled using mathematical equations. The *Objectives File* contains all the objective estimation equations that are used by SCBuild to estimate the core's area and delay during design space exploration. Another file that's been significantly modified is the "risc_cpu.xml" file. Part of this file contains a complete list of all the parameters of the system, each parameter's type and the set of possible values, and a default value. This file has been modified to reflect the parameters of the UW_Nios II system and their possible values. For more details on the content and format of each file, refer to Appendix A in [43]. Simple modifications were also added to SCBuild to enable it to tokenize equations with negative terms.

## 4.3 Experimental Framework and Results

Two sets of experiments were performed on a number of variants of the UW_Nios II core and the results from those experiments will be presented in this section. For these experiments, Altera's Quartus II 7.2 design software was used to generate and compile the different variant implementations. The purpose of the first set of experiments was to generate enough real synthesis data in order to establish estimation equations that provided reasonable estimates of FPGA logic element (LE) utilization and critical path delay for any arbitrary processor configuration. This helped draw conclusions and lead to a better understanding of processor design targeting FPGAs. The purpose of the second set of

experiments was to perform a comparison between Altera's Nios II and the UW_Nios II cores in terms of logic element utilization and processing speed performance.

### 4.3.1 Target Processor Core

The processor core targeted in this research is the UW_Nios II parameterized RISC soft-core processor core. Chapter 3 provides a detailed description of this processor core. To summarize, the parameterized UW_Nios II soft-core processor core developed in this research is a modified version of the Nios II standard core and supports the same instruction set as Altera's Nios II cores. It consists of a datapath module and a control unit module with no data or instruction memories.

The UW_Nios II core has a 4-stage pipelined datapath. Instructions are fetched in the Instruction Fetch Stage (IF). During the second stage, the decode and operand fetch stage (DOF), fetched instructions are decoded and proper operands are fetched from the register file. Instruction execution is done within the third stage, the execute stage (EX). Finally, results are written back to either the register file or the data memory during the last pipeline stage, the write back stage (WB).

The integer operations supported by the UW_Nios II soft core are data transfer, arithmetic, logical, comparison, shift and rotate, program control, along with other instructions. Table 3.1 (see section 3.4.1) lists the parameters for this core. Calculations show that UW_Nios II core has a total of 10,313 possible configurations.

### 4.3.2 Evaluation of Configurations: The Objective Functions

As the complexity of embedded systems and the number of system parameters they take increase, the design space expands. In any multi-objective DSE procedure, designers are required to evaluate individual configurations within the design space in terms of their objectives. Synthesizing each and every configuration within the design space is impractical, due to the increased sizes of design spaces. One possible option to solve this problem is to develop a mathematical model that estimates the effects of each parameter on the objectives. In order to achieve this, the objective estimation approach proposed by Jha

and Dutt [49] was adopted during this research. This approach suggests developing mathematical equations to accurately estimate the area and critical path delay using least-squares regression analysis on actual synthesis data for a number of representative configurations. These equations will be a function of the total number of parameters used in the system, P, and they have the following general form:

$$F_k(p_1, p_2, \ldots, p_P) = a_{0,k} + \sum_{i=1}^{P} (a_{i,k} * f_{i,k}(p_i)) \tag{4.1}$$

Where $a_{0,k}$, $a_{1,k}$, ..., $a_{P,k}$ are constant coefficients determined using a regression analysis procedure. The exact form of functions $f_{i,k}(p_i)$ can be determined by studying the relationship between each parameter and the area and delay values, as will be detailed in the following section.

### 4.3.3 Establishing the Objective Estimation Equations

In order to develop the area and delay objective estimation equations (equation 4.1) for the UW_Nios II processor core using the P-dimensional regression technique described in section 4.2.2, a set of configurations that are representative of the core's design space was synthesized. In this configuration set, a parameter sweep was performed on each of the core's ten parameters. Starting from a base configuration, in which all parameters are set to 1, each of the core's parameters were varied across their entire range of possible values while the other parameters were held constant at their base values. This produced a configuration set with a total of 17 configurations, each of which was compiled using Quartus II version 7.2 [28]. All of these configurations targeted an Altera Stratix EP1S40F780C5 FPGA device [25], and were compiled using the default compiler settings. The Stratix device used as the target FPGA has a total of 41,250 LE's, 3,423,744 RAM memory bits, and a total of 14 DSP blocks. For each configuration, the equivalent number of LE's occupied by the core, the number of DSP block elements, the total number of dedicated memory bits given by the compilation report at the end of synthesis, and the

critical path delay of the core (given in nanoseconds) as reported by the timing analyzer tool were recorded. The following is a detailed discussion of the results from the parameter sweep experiments.

Table 4.1: Summary of the Parameter Sweep Results

| Configuration | Clk (ns) | Number of LE's |
|---|---|---|
| Smallest & Fastest | 5.768 | 594 |
| Largest & Slowest | 149.443 | 4331 |

#### 4.3.3.1 Parameter Sweep Results

A large variation in both FPGA LE resource utilization and critical path delay was observed from the sweep configurations. A summary of the results from the sweep configuration is shown in Table 4.1. The complete table can be found in Appendix A. In terms of critical path delay, the fastest sweep configuration was configuration 15 with a critical path delay of 5.768 ns (173.4 Mhz). In this configuration, the register file was implemented using dedicated on-chip RAM memory bits, with multiplication and division emulated in software. The slowest configuration was configuration 13, with a critical path delay of 149.443 ns (6.7 Mhz), in which division was implemented in hardware. In terms of LE resource utilization, configuration 15 was the smallest with 594 LE's consumed, consuming less than 1.5% of the total FPGA LE capacity; the largest configuration was configuration 13 utilizing 4331 logic elements.

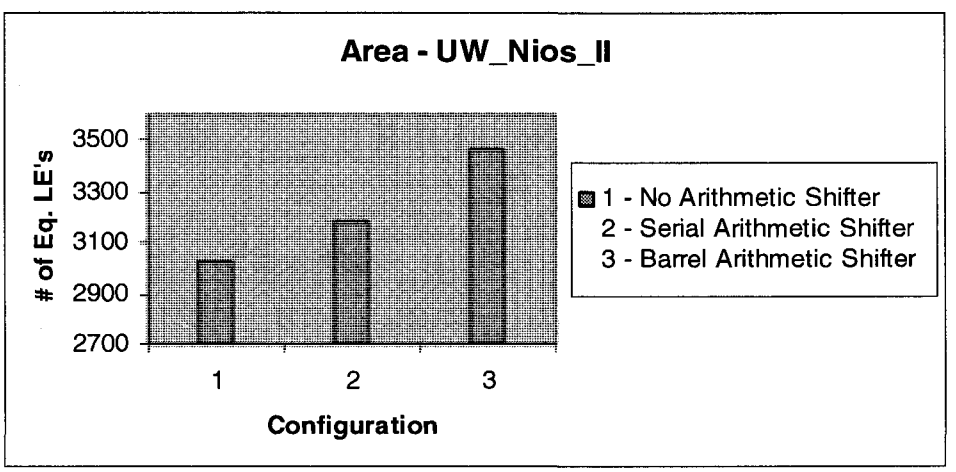
Configuration 15 was of particular importance. An important observation to be noted from this configuration is that implementing the register file using the on-chip dedicated RAM memory bits significantly improves the performance of the processor and reduces the LE resource utilization when compared with the rest of the configurations. In fact, it gives the fastest processing speed and the smallest LE usage. This observation triggered more experiments for comparison reasons between certain variants of the UW_Nios II core and Altera's Nios II cores.

In order to form the area objective estimation equations, functions $f_{i,k}(p_i)$ in equation 4.1, a study of the relationship between each of the processor's parameters and the resulting core area was conducted.

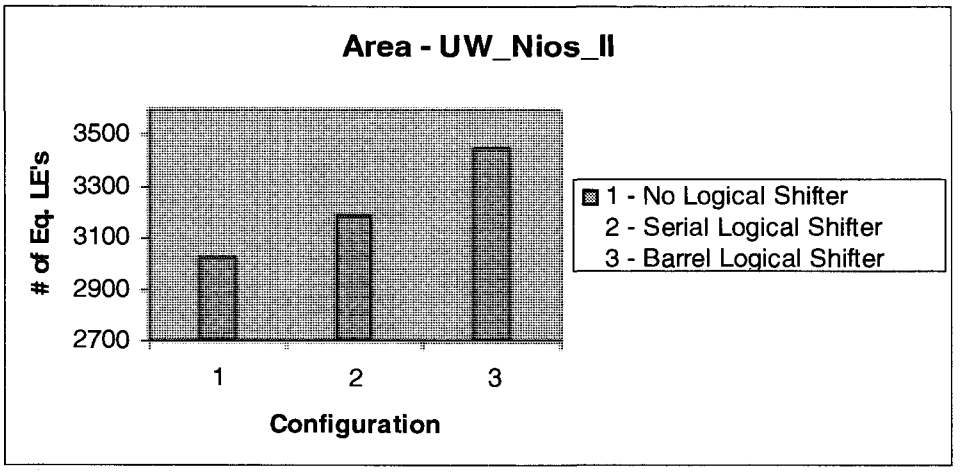
**Area Utilization**

Figure 4.3 shows a set of graphs that illustrate the relationships between each of the core's parameters and the core's total area (given as the total number equivalent LE's). The following points can be observed:

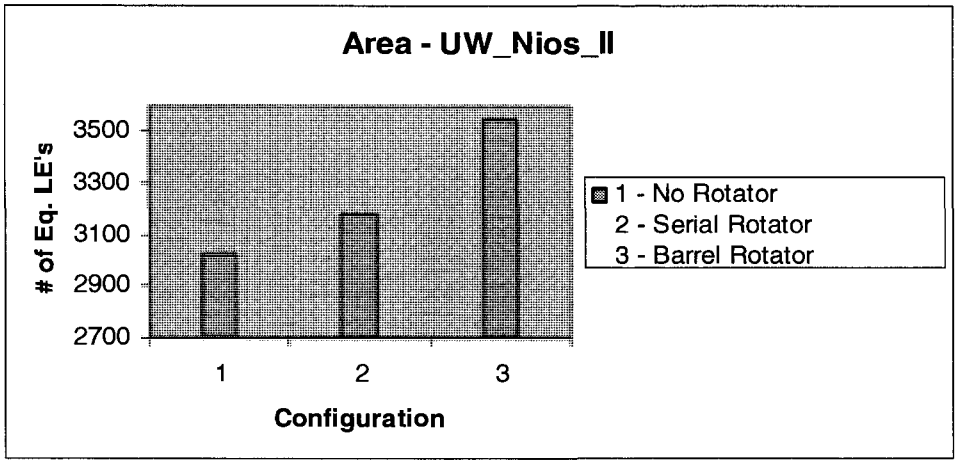
- As can be seen in Figures 4.3(a), (b) and (c), the arithmetic, logical shifters and rotator implementations have a significant impact on the processor's total area. The basic implementations of these units add 159 LE's to the processor. LUT-based barrel shifters/rotators result in a large increase in the total area. The LUT-based barrel implementations of shifters/rotator add anywhere between 440 LE's for the arithmetic and logical shifters, to 529 LE's for the barrel rotator. The relationships between the processor's total area and these parameters were modeled using a quadratic polynomial of the form: $ax^2 + bx + c$.



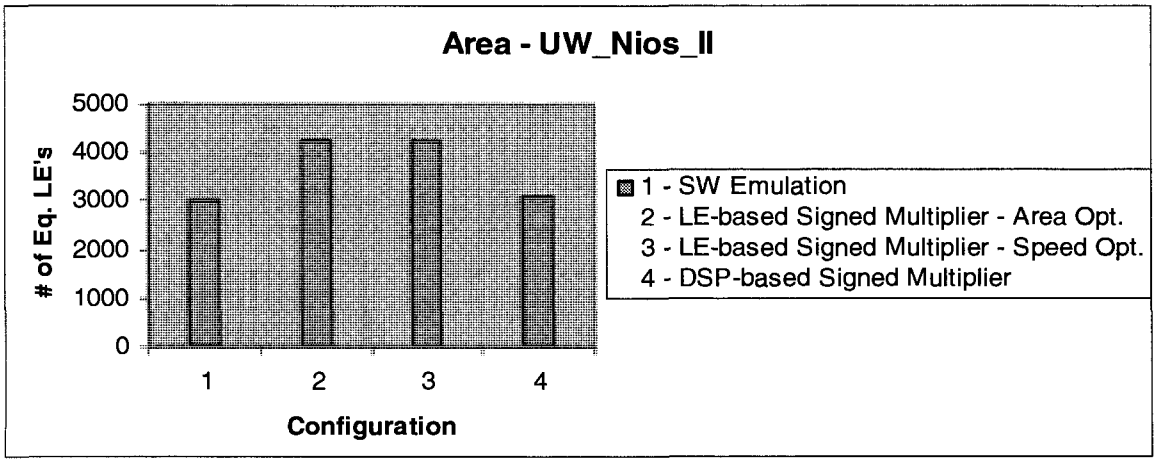

(a) Arithmetic Shifter Implementation (p1)



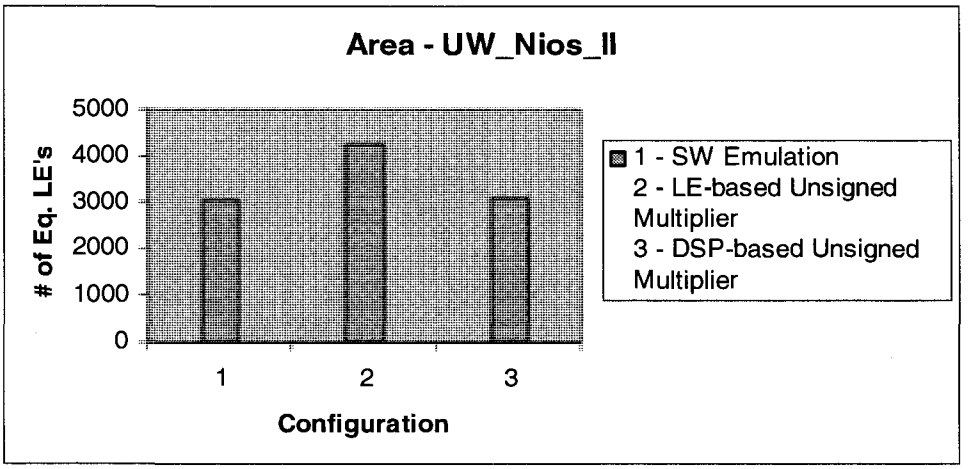

(b) Logical Shifter Implementation (p2)



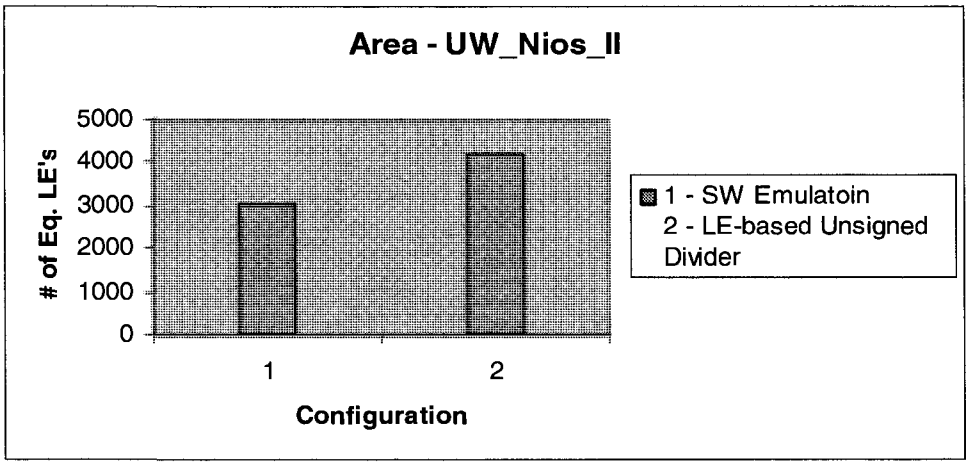

(c) Rotator Implementation (p3)



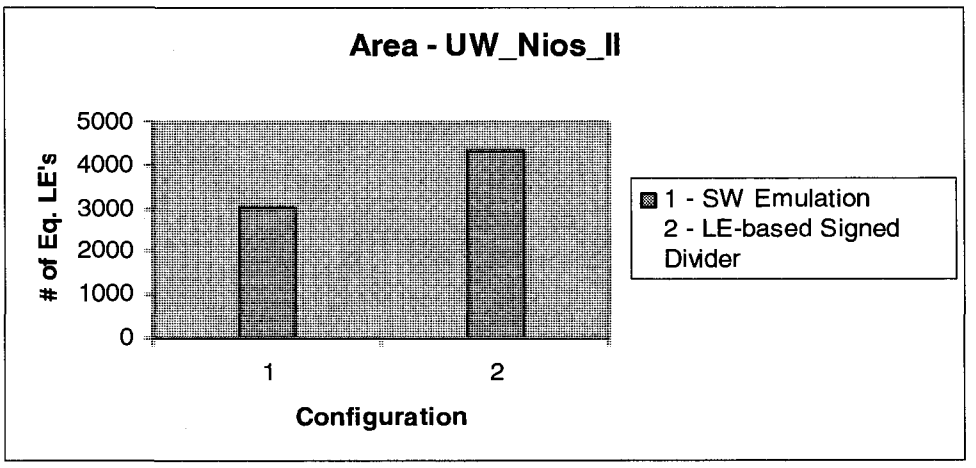

(d) Singed Multiplier Implementation (p4)



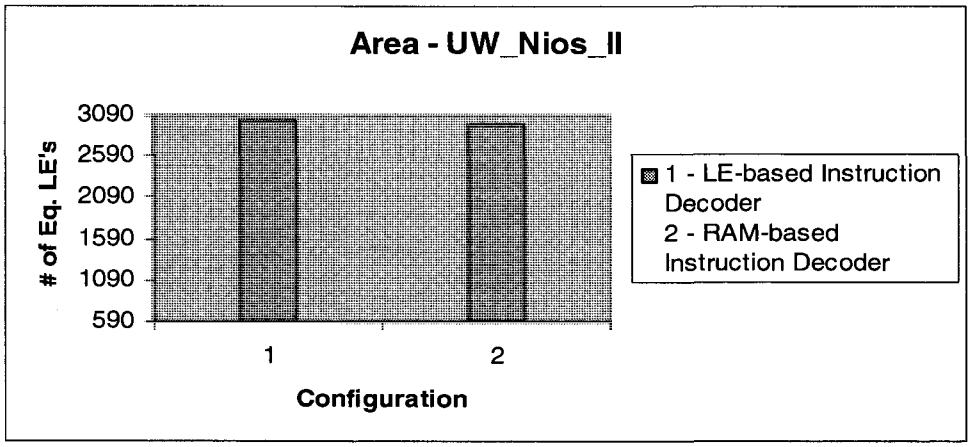

(e) Unsigned Multiplier Implementation (p5)



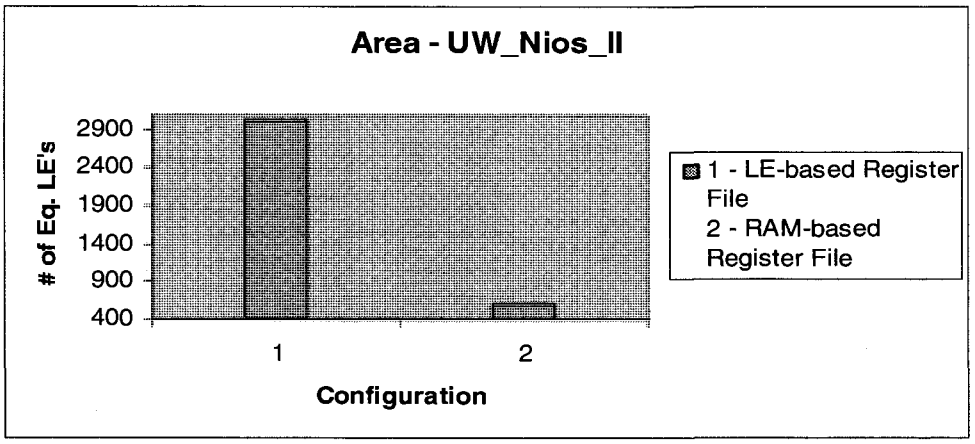

(f) Unsigned Divider Implementation (p6)



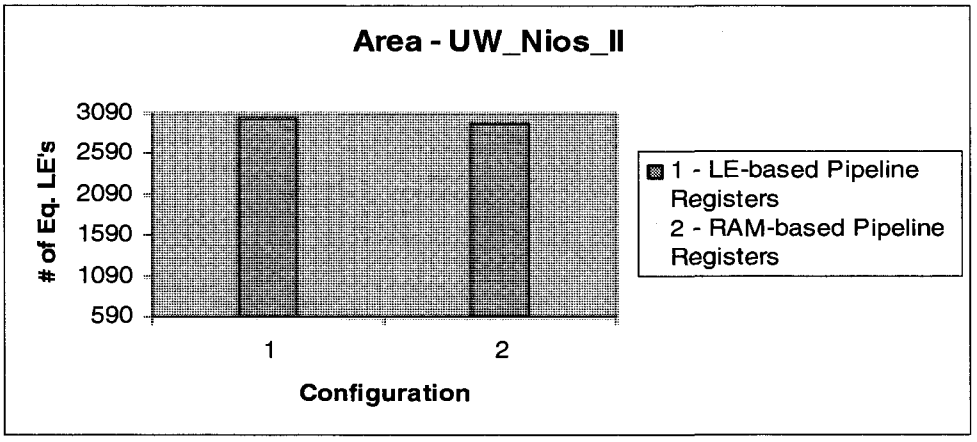

(g) Singed Divider Implementation (p7)




(h) Instruction Decoder Implementation (p8)




(i) Register File Implementation (p9)




(j) Pipeline Register Implementation (p10)

Figure 4.3: Parameter Sweep Results – Area

- Hardware multiplication modules, both singed and unsigned, consume anywhere between 1206 LE's for the LE-based singed multiplier, and 1248 LE's for the LE-based unsigned multiplier. Hardware multipliers are very expensive in terms of

LE utilization in FPGA platforms. However, when dedicated DSP blocks are used to implement hardware multipliers, they occupy 8 DSP blocks with only 56 LE's of additional logic. Signed and unsigned multiplication parameter implementations were modeled using polynomials of third and second degrees, respectively.

- Signed and unsigned hardware implementations of division are sometimes more expensive than hardware multipliers in terms of LE resource utilization on FPGA's. An unsigned LE-based divider adds 1155 LE's, while a signed LE-based divider adds 1309 LE's to the processor's total area. The relationship between the divider implementations and the processor's area was considered to be linear in both cases.

- Varying the instruction decoder implementation parameter between LE-based or RAM-based implementations has an insignificant impact on the processor's total area. The RAM-based implementation consumes 52 LE's less than the LE-based version (i.e., a saving of 52 LE's). Therefore, the relationship between the processor's area and the instruction decoder implementation parameter was assumed to be linear.

- Figure 4.3 (i) shows that the register file implementation parameter has the greatest impact on the processor's area. Implementing the register file using RAM memory blocks requires 2428 LE's less than the LE-based implementation. A first degree polynomial was chosen to model the relationship between the register file parameter and the total area of the processor.

- Finally, as illustrated by figure 4.3 (j), only 55 LE's can be saved when the pipeline registers are implemented using RAM memory blocks versus the LE-based implementation. This is not a large saving compared to the processor's total area. Thus, the relationship between the processor's total area and the pipeline register implementation was modeled by a first degree polynomial (i.e., a linear relation).

**Critical Path Delay**

The graphs in Figure 4.4 depict the relationship between the UW_Nios II's critical path delay (given in nanoseconds) and each of the processor's parameters. In general, predicting the effects of varying the parameter values on the critical path delay was harder than predicting the effects on the processor's area. Implementing division in hardware causes the greatest increase in the processor's critical path delay. The following points can be observed from the graphs in Figure 4.4:

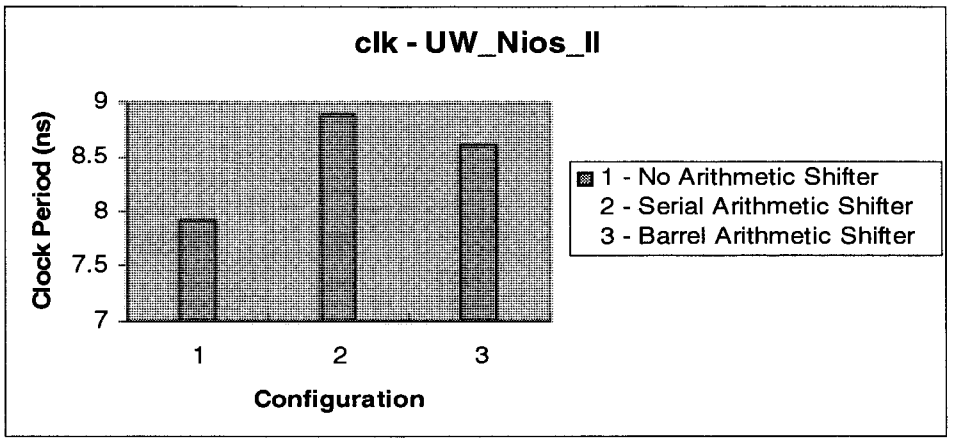
- As can be seen in Figures 4.4(a), (b) and (c), the arithmetic, logical shifters and rotator implementations have a relatively small impact of the processor's critical path delay. The basic implementations of these units add close to 1 ns of delay to the processor. The LUT-based barrel implementations of the shifters add less than 1 ns, while the barrel rotator adds a bit more than 1 ns to the clock period of the processor. The relationships between the processor's critical path delay and these parameters were modeled using a quadratic polynomial of the form: $ax^2 + bx + c$.

- Hardware multiplication units, both signed and unsigned, cause an increase in the clock period anywhere between 10.415 ns for the LE-based singed multiplier, and 10.639 ns for the LE-based unsigned multiplier, which makes the clock frequency 2.5X slower. Hardware multipliers are expensive in terms of critical path delay on FPGA platforms. However, when dedicated DSP blocks are used to implement hardware multipliers, they increase the clock period by less 5 ns. In other words, the processor's clock frequency is 2X faster with a DSP-based multiplier compared with an LE-based multiplier. Signed and unsigned multiplication parameter implementations were modeled using polynomials of third and second degrees, respectively.

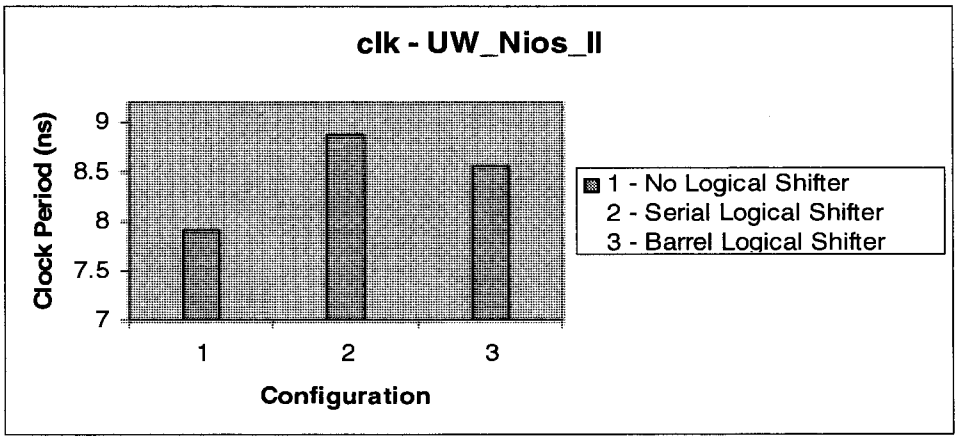
- Signed and unsigned hardware implementations of division are most expensive in terms of critical path delay on FPGA's. An unsigned LE-based divider adds 125 ns, while a signed LE-based divider adds 141 ns to the processor's clock period. In

other words, a hardware divider increases the clock period by 15 to 17 times. The relationship between the divider implementations and the processor's critical path delay was considered to be linear in both cases.

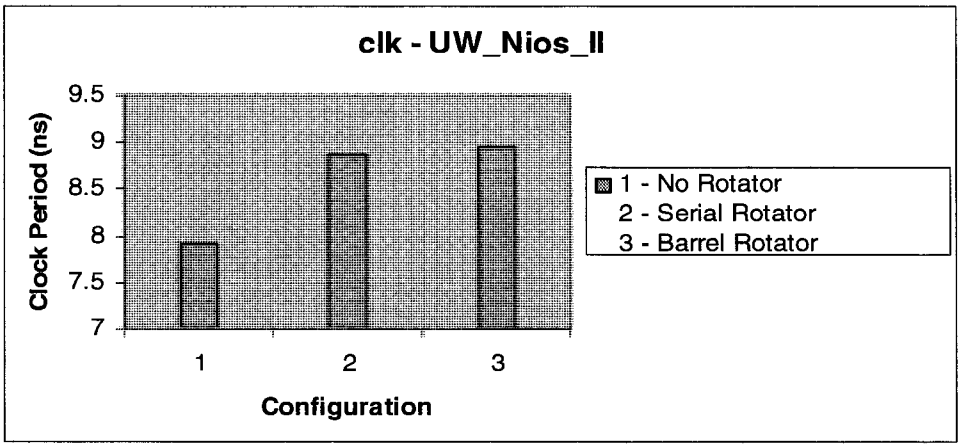
- Varying the instruction decoder implementation parameter between LE-based or RAM-based implementations has an insignificant impact on the processor's total area. The RAM-based implementation requires a clock period that is 0.05 ns less than the LE-based version. Therefore, the relationship between the processor's critical path delay and the instruction decoder implementation parameter was assumed to be linear.



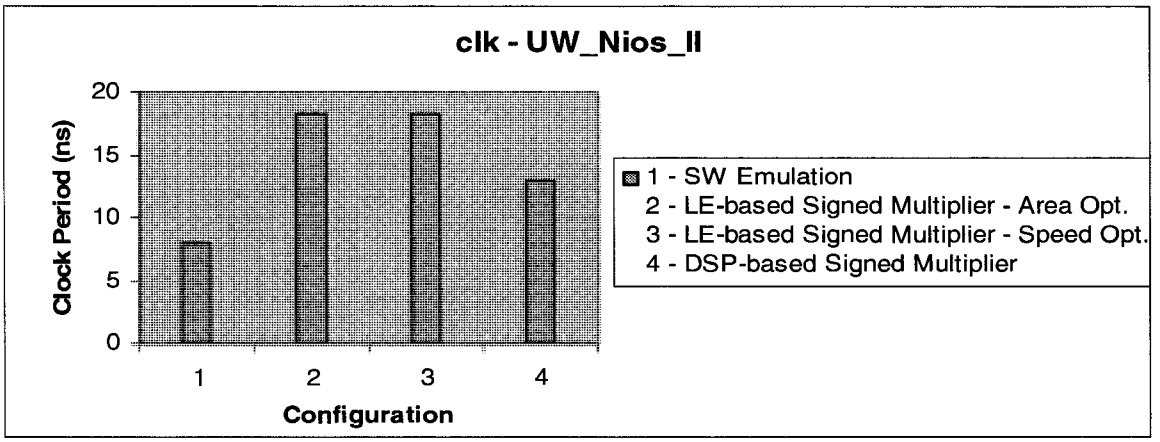

(a) Arithmetic Shifter Implementation (p1)



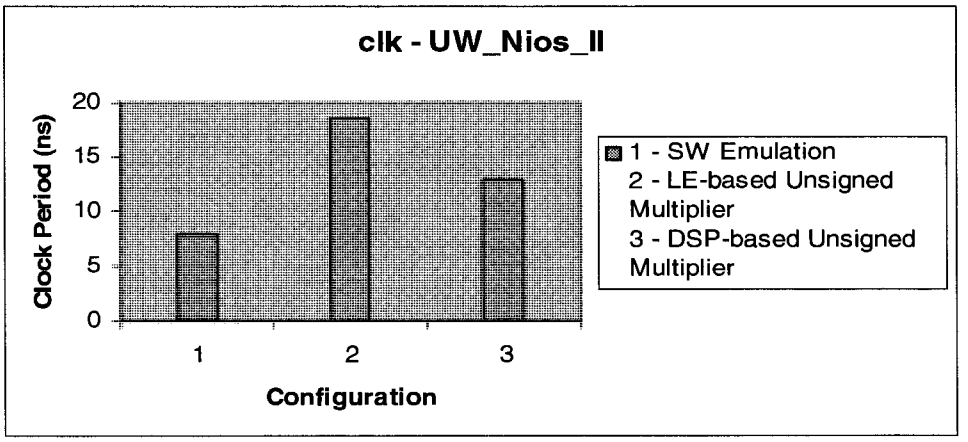

(b) Logical Shifter Implementation (p2)



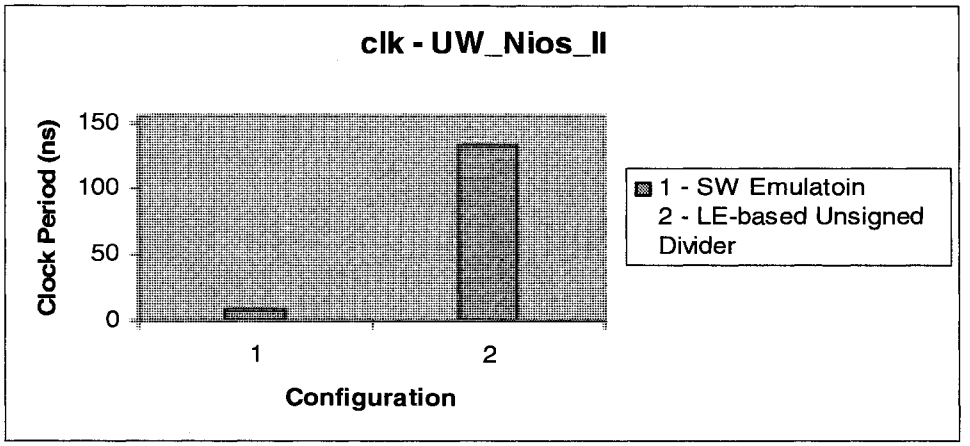

(c) Rotator Implementation (p3)



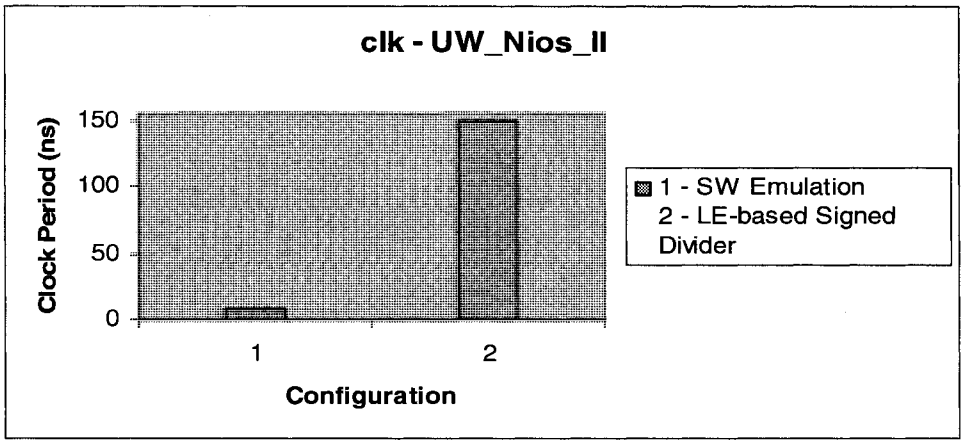

(d) Signed Multiplier Implementation (p4)



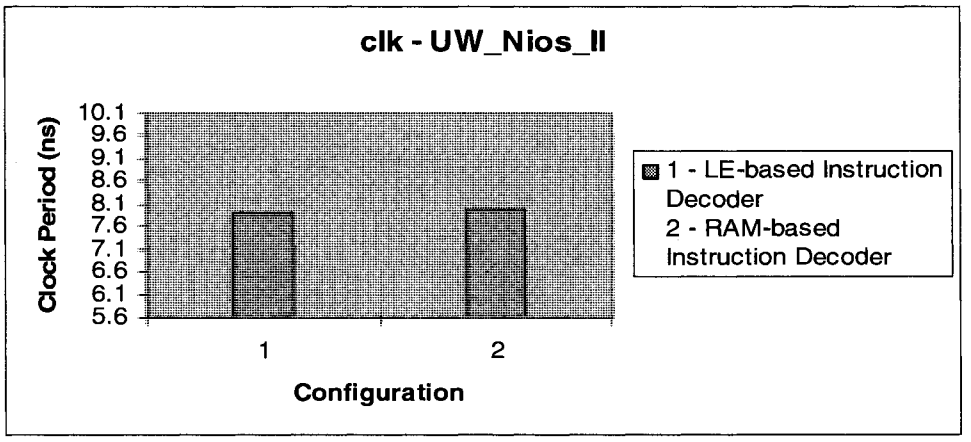

(e) Unsigned Multiplier Implementation (p5)



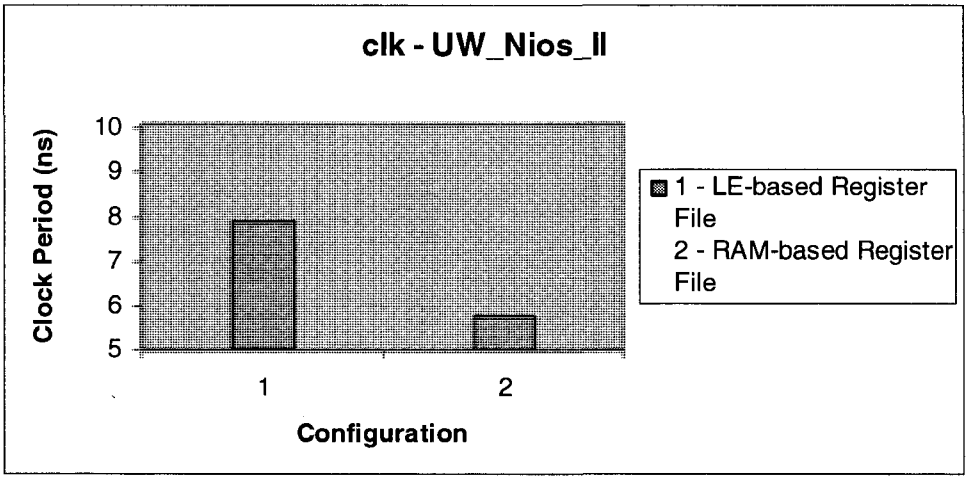

(f) Unsigned Divider Implementation (p6)



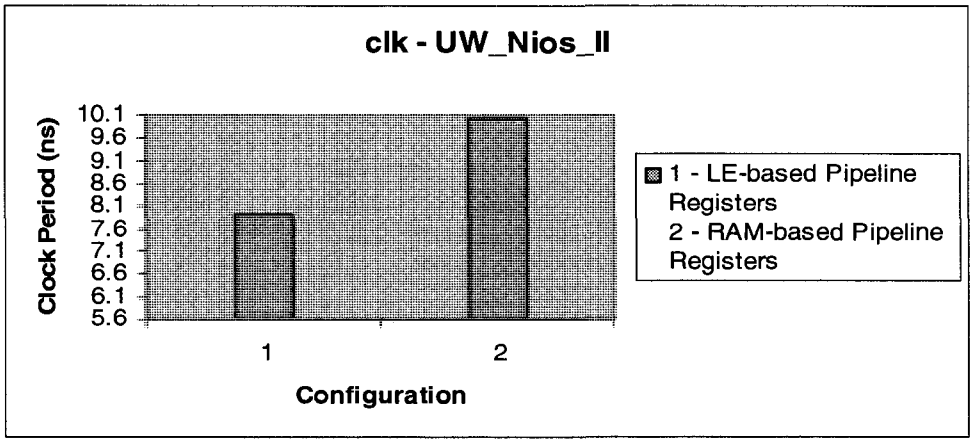

(g) Signed Divider Implementation (p7)




(h) Instruction Decoder Implementation (p8)




(i) Register File Implementation (p9)




(j) Pipeline Register Implementation (p10)

Figure 4.4: Parameter Sweep Results – Critical Path Delay

- Figure 4.4 (i) shows that the register file implementation parameter has a significant impact on the processor's critical path delay. Implementing the register file using RAM memory blocks causes a decrease of 2.139 ns in the processor's clock period compared with the LE-based implementation (i.e., a 27% improvement). A first degree polynomial was chosen to model the relationship between the register file parameter and the total area of the processor.

- Finally, as illustrated by Figure 4.4 (j), a 2.125 ns increase is added to the processor's clock period when the pipeline registers are implemented using RAM memory blocks versus the LE-based implementation. In other words, implementing the pipeline registers using LE's improves the clock period by about 27% compared

with the RAM-based implementation. The relationship between the processor's critical path delay and the pipeline register implementation was modeled by a first degree polynomial (i.e., a linear relation).

#### 4.3.3.2 Objective Estimation Equations

The Curve Fitting Tool provided by MATLAB [50] was used to determine the exact forms for all of the functions $f_{i,k}(p_i)$ for each parameter. A plot was generated to model each parameter's effect on the processor's area and critical path delay. The Curve Fitting Tool uses a library of parametric models, including polynomials, exponentials, rationals and others to determine the function that best fits the plot. The tool was then used to perform regression analysis on each plot to compute the $a_{i,k}$ coefficients in equation 4.1. The final functions along with their coefficients used to approximate the processor's area and critical path delay are listed in Table 4.2.

#### 4.3.3.3 Testing the Accuracy of the Objective Estimation Equations

Having developed the objective estimation equations for the delay and area as discussed in the previous section, we next test the accuracy of these equations. The area and delay results for the 17 parameter sweep configurations, used to establish the objective estimation equations, as reported by Quartus II were compared with the results produced using the objective estimation equations. The two graphs in Figure 4.5 illustrate this comparison and show that the estimated values for delay and area match up with the actual values almost perfectly. The percentage error between the "actual" versus the "estimated" values for the parameter sweep configurations is negligible.

Next, a set of 20 random configurations were developed. They were compiled in Quartus II; the delay and area values were collected from the compilation reports. These results were compared with the estimated values for area and delay obtained using the objective estimation equations. Figures 4.6(a) and 4.6(b) illustrate how close the estimated values trace the actual values for area and delay, respectively. The average percentage error

for area estimates was 0.59%, and 6.56% for delay estimates. This step serves as an accuracy test of the objective estimation equations for any arbitrary configuration. As shown by the figures, it was easier to estimate area with greater precision than delay, however they are both still within a tolerable margin of error.

Table 4.2: Regression Coefficients for UW_Nios II

| Parameter | i | $a_{i,1}$ (Area) | $a_{i,2}$ (Delay) | $f_{i,1}(p_i)$ (Area) | $f_{i,1}(p_i)$ (Delay) |
|---|---|---|---|---|---|
| - | 0 | 3022 | 7.907 | - | - |
| Arithmetic Shifter Implementation (p1) | 1 | 61 | -0.6245 | $P_1^2 - 0.393\ p_1 - 0.607$ | $P_1^2 - 4.549\ p_1 + 3.55$ |
| Logical Shifter Implementation (p2) | 2 | 56 | -0.6445 | $P_2^2 - 0.161\ p_2 - 0.839$ | $P_2^2 - 4.503\ p_2 + 3.502$ |
| Rotator Implementation (p3) | 3 | 105.5 | -0.445 | $P_3^2 - 1.493\ p_3 + 0.493$ | $P_3^2 - 5.175\ p_3 + 4.175$ |
| Include Signed Multiplier (p4) | 4 | 9.333 | 0.8217 | $P_4^3 - 70.610\ p_4^2 + 334.083 - 264.438$ | $P_4^3 - 12.340\ p_4^2 + 42.692\ p_4 - 31.350$ |
| Include Unsigned Multiplier (p5) | 5 | -1220 | -8.174 | $P_5^2 - 4.023\ p_5 + 3.023$ | $P_5^2 - 4.301\ p_5 + 3.302$ |
| Include Unsigned Divider (p6) | 6 | 1155 | 125.4 | $P_6 - 1$ | $P_6 - 1$ |
| Include Signed Divider (p7) | 7 | 1309 | 141.5 | $P_7 - 1$ | $P_7 - 1$ |
| Instruction Decoder Implementation (p8) | 8 | -52 | 0.046 | $P_8 + 1$ | $P_8 - 1$ |
| Register File Implementation (p9) | 9 | -2428 | -2.139 | $P_9 - 1$ | $P_9 - 1$ |
| Pipeline Register Implementation (p10) | 10 | -55 | 2.125 | $P_{10} - 1$ | $P_{10} - 1$ |

The above mentioned experiments demonstrate the difficulty in estimating the critical path delay of parameterized soft-core processors compared with their area estimates. This difficulty is a result of the high complexity of the placement and routing processes performed by CAD tools, such as Quartus II. The core's critical path delay is highly sensitive to changes in the implementation and placement of the circuit on the FPGA and the routing between the various components of the core. The impact that such changes have on the core's critical path delay is hard to predict with great precision. By contrast, the area utilized by a core is easier to predict more accurately because the effects of varying the core's parameters on the synthesis results are fairly fixed and predictable.

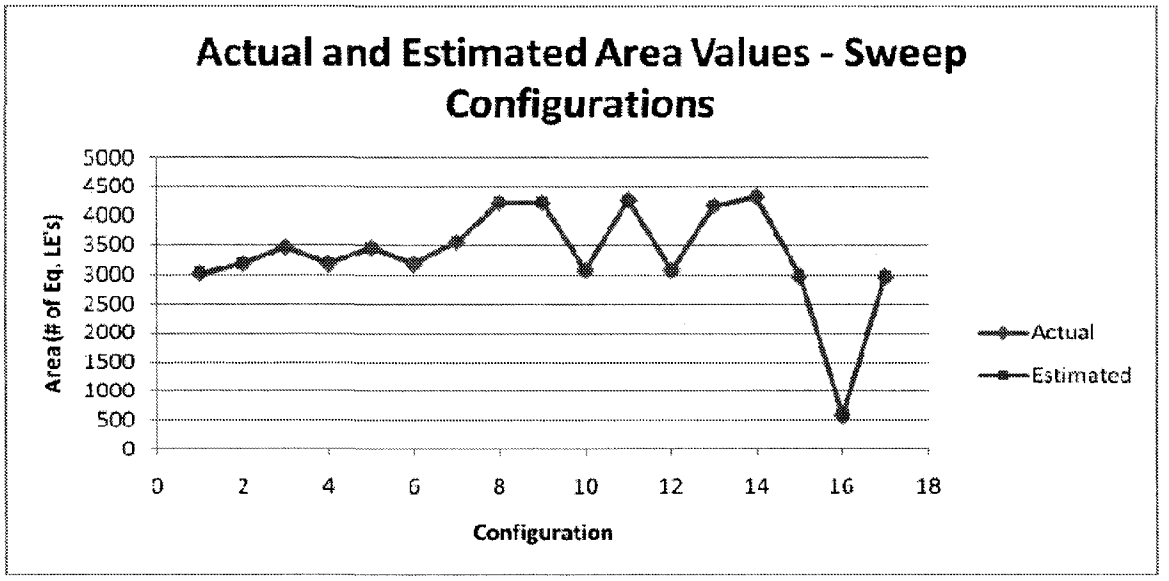
Another outcome that can be inferred from these results is that a tradeoff relationship exists between the precision of the estimated objective values and the amount of computation required to obtain those values. CAD tools, such as Quartus II, are able to report the exact delay and area because they utilize information about the implementation, placement and routing of the core in their delay and area computations. Utilizing such information requires a significant amount of complex computations. On the other hand, the goal of the regression-based objective estimation technique used in this research is to provide reasonably close estimations that can be evaluated quickly and easily. In general, more accurate estimations can be made at the expense of longer computation times; faster and simplified computations can be utilized at the expense of reduced estimation accuracy. In future work, increased accuracy of the estimates may be achieved and the need to generate a set of sweep configurations may be removed by employing different objective estimation techniques.

### 4.3.4 Design Space Exploration (DSE)

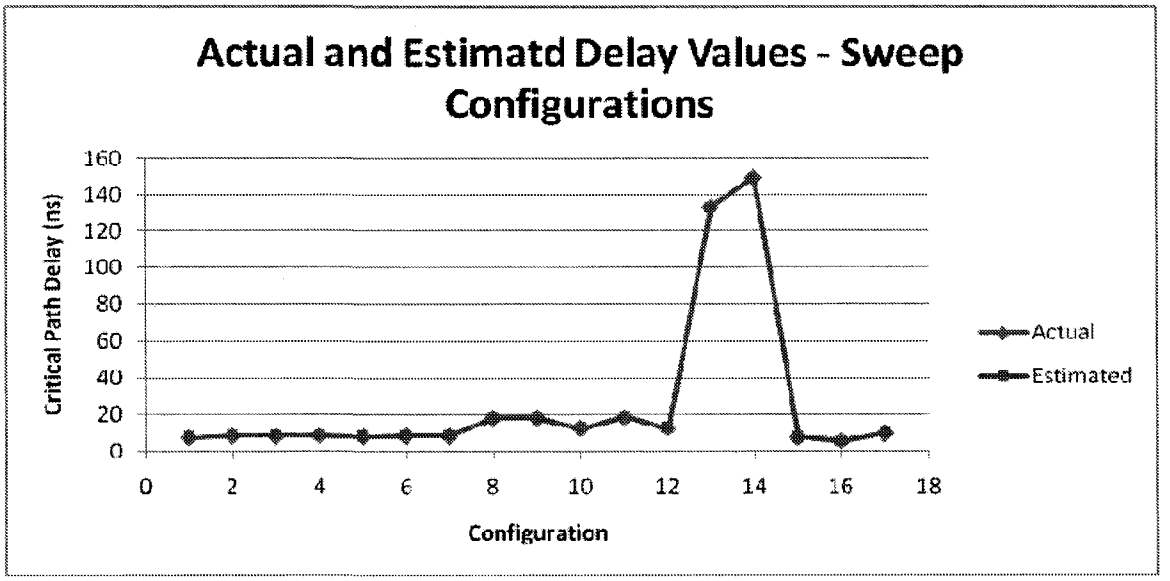
Now that we determined the objective estimation equations and verified their accuracy, SCBuild CAD tool was used to apply the SEAMO algorithm to a population of randomly-generated configurations in order to approximate the Pareto-optimal set. This section presents the results from this experiment.

#### 4.3.4.1 Determining Algorithm Parameters

In order to Apply SEAMO to approximate the Pareto-optimal configuration set, SEAMO's parameters need to be specified first. Suitable values for these parameters were determined experimentally. A set of experiments were conducted on a configuration set with randomly-generated configurations. In these experiments, the mutation and crossover rates

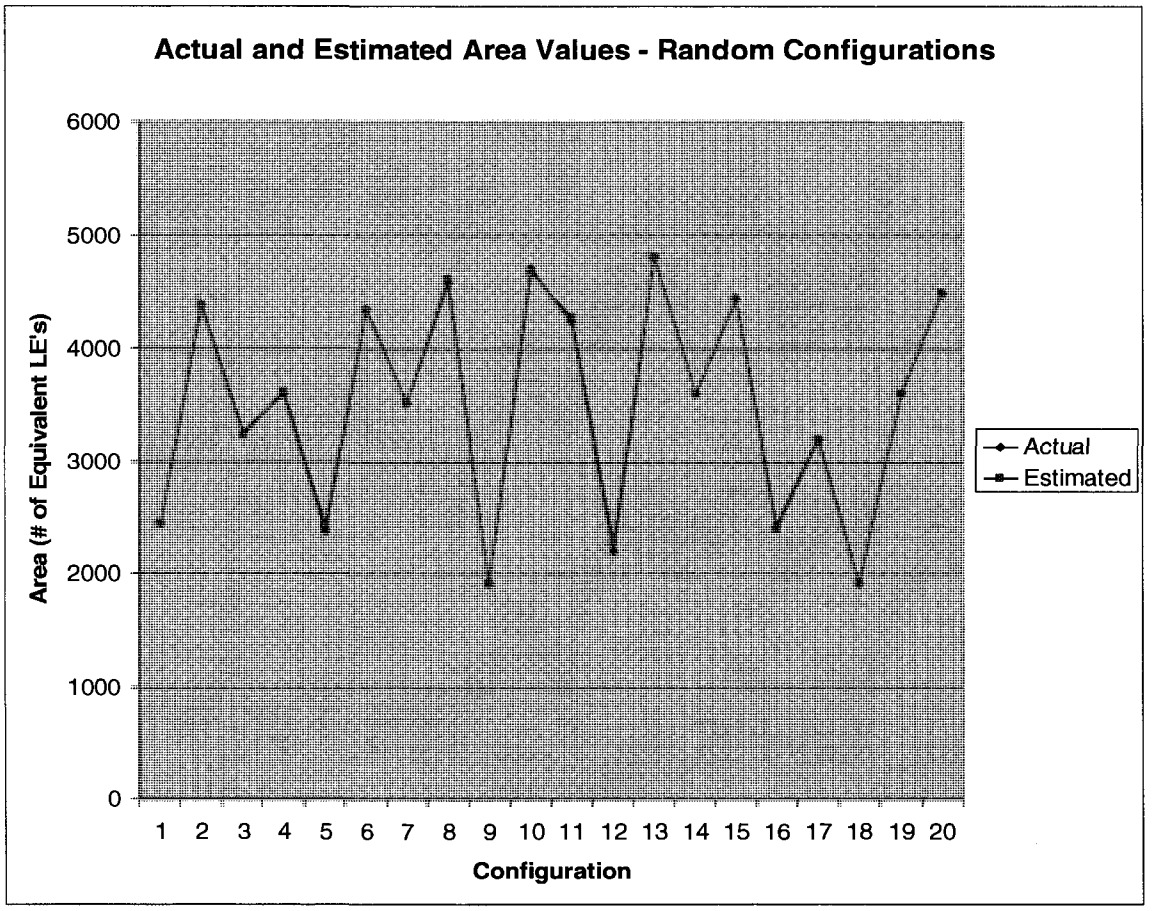



(a) Area

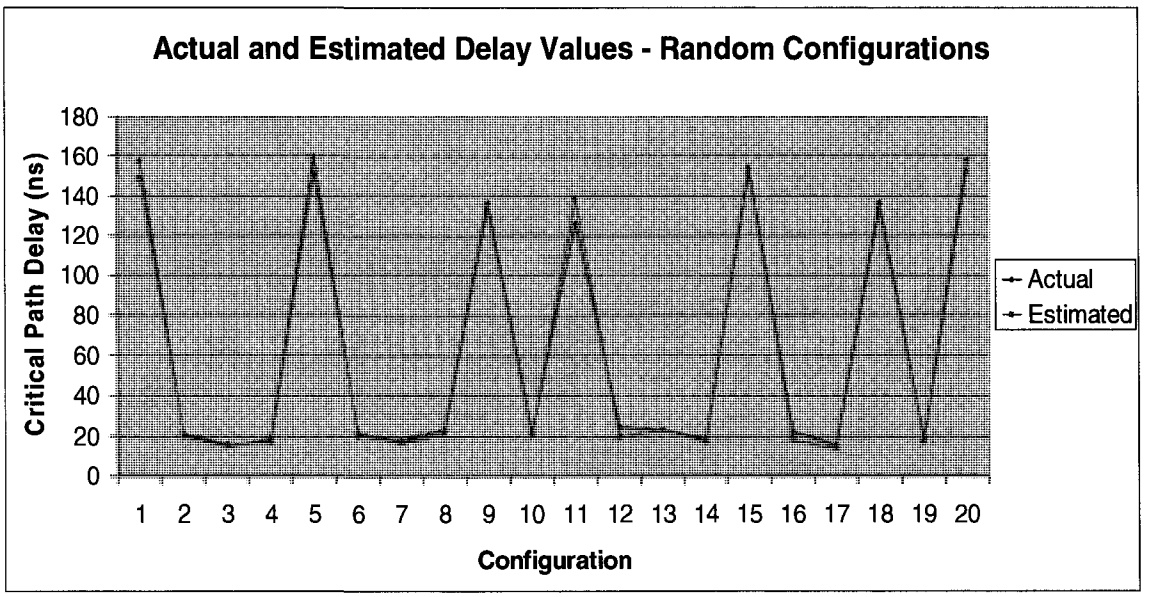



(b) Delay

Figure 4.5: Actual versus Estimated Values for the Parameter Sweep Configurations




(a) Area




(b) Delay

Figure 4.6: Actual versus Estimated Values for Random Configurations

were varied between 0.1 and 0.7, and the resulting evolved populations were observed. It was found that for a mutation rate of 0.5 and a crossover rate of 0.4, the average area and delay values were lowest. Another set of similar experiments were conducted to determine the number of generations parameter of the SEAMO algorithm. The number of generations, N, in these experiments was varied between 10 and 60. It was found that N = 40 provided a large diversity of configurations and resulted in lower average values for the area and delay.

#### 4.3.4.2 Results

The SCBuild CAD tool was used to explore the design space of the UW_Nios II soft-core processor and apply the SEAMO algorithm to an initial population of 88 randomly-generated configurations. After 40 generations, SCBuild produced an evolved population, which approximates the Pareto-optimal set of the UW_Nios II's design space. The developed objective estimation equations were used to estimate the area and delay of each configuration in the initial and evolved populations (See Appendix A for a list of the initial and evolved populations). Figure 4.7 illustrates a graphical comparison between the initial and evolved populations.

As shown in Figure 4.7, the majority of configurations in the evolved population cluster around the lower left corner of the design space, whereas configurations from the initial population tend to be scattered throughout the entire design space. It is clear that configurations from the initial population tend to occupy much more area and have a significantly larger critical path delay than those from the evolved population. More specifically, the evolved population's configurations have an average area that is about 65% smaller than the randomly generated configurations in the initial population, and a critical path delay that is more than 75% smaller. This indicates that SCBuild successfully explores the design space of the supplied soft-core processor and approximates its Pareto-optimal set. More accurate estimation equations would result in a smoother curve along the lower left boundary of the design space.

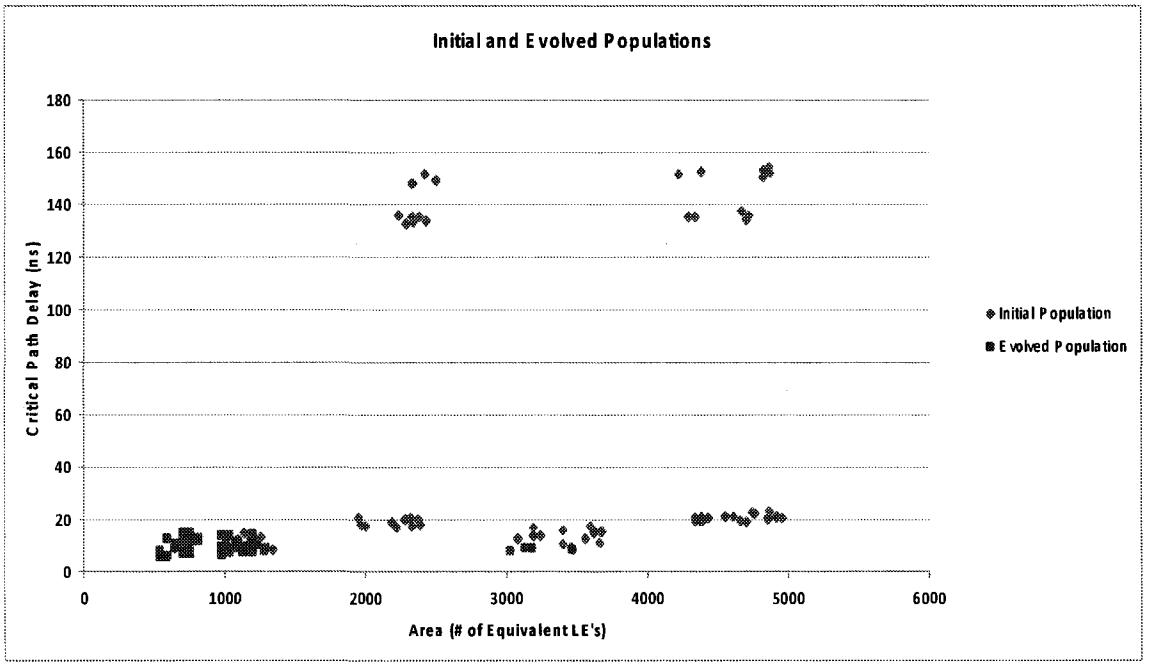



Figure 4.7: Initial and Evolved Population

### 4.3.5 Conclusions

Table 4.3 was produced after a study of the evolved population was conducted (refer to Table A.3 of Appendix A). This table lists the number of occurrences of each parameter value in the evolved population. The following observations can be made:

- In about half the configurations, the SEAMO algorithm tended to eliminate the use of hardware shifting and rotating. As for the remaining configurations, the number of occurrences of serial arithmetic shifters was almost equal to the barrel implementation, and the basic implementations of the logical shifter and rotator were favored over the barrel implementations.

- In approximately 75% of the configurations, signed multiplication was set to be emulated in software. In the remaining configurations, dedicated DSP blocks were always used to implement the hardware signed multiplier as recommended by section 3.4.4 (i.e.,. LE-based implementation of signed multiplication was never used in any of the configurations).

Table 4.3: Number of Occurrences of Each Parameter Value in the Evolved Population

| Value | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| P1 | 40 | 26 | 22 | - |
| P2 | 39 | 30 | 19 | - |
| P3 | 47 | 32 | 9 | - |
| P4 | 65 | 0 | 0 | 23 |
| P5 | 83 | 0 | 5 | - |
| P6 | 88 | 0 | - | - |
| P7 | 88 | 0 | - | - |
| P8 | 44 | 44 | - | - |
| P9 | 4 | 84 | - | - |
| P10 | 48 | 40 | - | - |

- In all but five of the evolved configurations, unsigned multiplication was set to be emulated in software. Dedicated DSP blocks were utilized to implement the hardware unsigned multipliers in the remaining five configurations. No LE-based implementations of unsigned multipliers were utilized.

- The SEAMO algorithm always favored the software emulation of signed and unsigned division in all of the evolved populations. This can be attributed to the fact that hardware dividers consume a very large area and cause a significant decrease in the processor's clock period.

- As would be expected, exactly half of the evolved configurations contained an instruction decoder that's implemented using dedicated RAM memory blocks, while the other half contained a LE-based implementation. This can be attributed to the fact that varying this parameter has a negligible effect on both area and clock period.

- Since the RAM-based implementation of the register file provides a greater advantage over the LE-based version (as explained in section 3.4.5) only 4 configurations out of 88 implemented the register file using LE's; the rest were implemented using RAM memory blocks.

- Recall that implementing the pipeline registers using LE's caused a small increase in area but resulted in a smaller clock period, as illustrated by section 3.4.6. Therefore, the SEAMO algorithm favored an LE-based implementation of the pipeline registers in 48 out of 88 configurations. In the remaining cases, a RAM-based implementation was utilized.

The experimental results show that using a genetic-based approach for exploration of the design space of a parameterized core can be helpful in assisting designers choose a well-optimized and customized hardware platform configuration for their target application, and in selecting the proper parameter values in a short amount of time. This is possible because the genetic algorithm employed within SCBuild removes the non-optimal configurations from consideration by approximating the Pareto-optimal set. This Pareto-optimal set contains a small number of optimized configurations compared with the large number of possible configurations that exists in the design space of the parameterized core. Designers can then choose a configuration from this set that satisfies their design constraints utilizing an accurate evaluation of each configuration's area and performance provided by SCBuild.

### 4.4 Summary

This chapter started by presenting the design and implementation details of the SCBuild CAD tool. The core's template description, provided to SCBuild as input, was illustrated, followed by a brief overview of SCBuild's CAD flow. The CAD flow illustrates the step by step approach utilized by SCBuild during its execution. The remaining part of the chapter discussed the set of experiments conducted on the parameterized UW_Nios II soft-core

processor using SCBuild. An initial set of 17 different "parameter sweep" configurations that represent the processor's design space were compiled. The compilation results obtained were used to establish the objective estimation equations. These equations were used to provide reasonably accurate estimates of the processor's area utilization and critical path delay on an FPGA platform for arbitrary configurations. Next, a set of 20 randomly-generated configurations were compiled to test the accuracy of the established objective estimation equations. It was found that the equations provided estimates for area that were, on average, within 0.59% of the actual values, and within 6.56% of the actual values for delay. Finally, SCBuild was used to apply the SEAMO algorithm on an initial population of 88 randomly-generated configurations for 40 generations. In general, the evolved population showed a substantial improvement in the area and delay objectives. More specifically, the evolved population, on average, utilized 65% less area and had a critical path delay that was 75% smaller than the initial population.

In the next chapter, this thesis is concluded with a summary of our research contributions, followed by a discussion of possible extensions of this research work that could be done in the future.

# Chapter 5

# *Conclusions and Future Work*

As embedded systems are becoming more complex, FPGAs provide a low cost and flexible medium for implementing and testing complete embedded systems. The platform-based design methodology of embedded systems is becoming more desirable for designers since they can build more complex systems in less time by using pre-designed and tested IP cores. This thesis presented a methodology that could help designers make intelligent decisions when they develop embedded systems using a platform-based design approach. It employs a genetic-based algorithm to automate the design space exploration process of parameterized soft-core processors. After presenting some relevant background material, the design and architecture of a parameterized soft-core processor, UW_Nios II, were discussed in detail, and the performance of different variants was compared with Altera's Nios II. It was found that, in the best case, the UW_Nios II's clock frequency was only 7% less and occupied 47% less area.

Chapter 4 starts by discussing the design and implementation details of SCBuild, a CAD tool for the design space exploration of soft-core processors. The remainder of this chapter presents the results obtained from a set of experiments carried out using SCBuild to automatically explore the design space of the UW_Nios II soft-core processor and approximate its Pareto-optimal set of configurations. It was concluded that applying a

genetic algorithm to approximate the Pareto-optimal set of an embedded system helps designers choose a well optimized hardware platform configuration for their systems.

## 5.1. Thesis Contributions

The research contributions of this thesis are:

1. The source code for a parameterized RISC soft-core processor, UW_Nios II, that supports the same instruction set as Altera's commercial Nios II was developed using VHDL, and its functionality was tested. During the development of the UW_Nios II, several contributions were made:
    a. Ten system parameters were added to the processor core.
    b. Different architectural variations were studied to find out what works best for FPGA platforms.
    c. A comparison between UW_Nios II and Altera's commercial Nios II soft-core processors was conducted
2. A method for estimating the objective values (i.e., FPGA area utilization and critical path delay) given a set of parameter values was applied to variants of the UW_Nios II. Using this method, accurate estimations were obtained.
3. A parameterized template description of the UW_Nios II soft-core processor was developed and utilized to conduct a set of design space exploration experiments on the UW_Nios II core.
4. SCBuild, a software CAD tool, was modified and used to automatically explore the design space of the UW_Nios II using the SEAMO genetic algorithm. Using SCBuild, a good approximation of the Pareto-optimal set of configurations for the UW_Nios II was obtained.

## 5.2. Future Work

In the future, this thesis work can be extended in many different ways. More parameters can be added to the developed soft-core processor. Instruction and data cache can be added to the processor core, and different experiments can be conducted to see which cache line

depth is optimal for FPGA platforms. Also the performance of cached and un-cached soft-core processors can be compared. Support for different kinds of branch predictions can be added, and the performance of different variants with different branch prediction schemes can be compared. Also, floating-point support, different pipeline depths and support for custom instructions can be added as system parameters. More implementations of functional units can be explored, including different implementations of shifters, multipliers, dividers, adders etc. More optimizations can be applied to the processor system to improve its speed performance and area utilization even further. Lastly, a better estimate of the core's performance can be achieved by running different benchmark circuits on different variants of the core.

Also, more template description files can be developed and supplied to SCBuild to enable it to automatically generate VHDL source code of different variants of the soft-core processor, and then, if a copy of Altera's Quartus II CAD tool is installed, automatically prompt it to create a project file and compile the VHDL code of the processor core. The number of objective functions estimated by SCBuild can be increased to include estimating the power consumption of different cores. Other design space exploration algorithms can be investigated and compared to see which one give the best approximation of the Pareto-optimal set of a core. More features can also be added to SCBuild, such as adding a profiling capability, to enable SCBuild to analyze different software applications and benchmarks and automatically remove un-used instructions from the instruction set of the processor core, and automatically optimize the processor core for the target application. Finally, SCBuild can be extended to enable it to explore the design space of more commercially-deployed soft-core processors.

# *References*


[1] R. Ernst. Codesign of embedded systems: Status and trends. IEEE Design & Test of Computers, pages 45-54, April-June 1998.

[2] R. Ernst, J. Henke, and T. Benner. Hardware-software cosynthesis for micro-controllers. IEEE Design & Test of Computers, pages 64-75, December 1993.

[3] R. K. Gupta and G. De Micheli. Hardware-software cosynthesis for digital systems. IEEE Design & Test of Computers, pages 29-41, September 1993.

[4] G. De Micheli and R. K. Gupta. Hardware/software co-design. Proc. of the IEEE, 85 (3): 349-356, March 1997.

[5] H. Chang, L. Cooke, M. Hunt, G. Martin, A. McNelly, and L. Todd. Surviving the SOC Revolution: A Guide to Platform-Based Design. Kluwer, Norwell, Massachusetts, USA, 1999.

[6] G. Martin and J.-Y. Brunel. Platform-based co-design and co-development: Experience, methodology and trends. In Proc. of the 9th IEEE/DATC Electronic Design Processes Workshop, April 2002.

[7] P. Pop, P. Eles, and Z. Peng. Analysis and Synthesis of Distibuted Real-Time Embedded Systems. Kluwer Academic Publishers, Boston/Dordrecht/London, 2004.

[8] P. Yiannacouras, J. Rose, and J. G. Steffan. The microarchitecture of FPGA-based soft processors. In Proc. of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'05), pages 202-212, San Francisco, California, USA, September 2005.

[9] T. Givargis and F. Vahid. Parameterized system design. In Proc. of the 8th International Workshop on Hardware/Software Codesign (CODES'00), pages 98-102, San Diego, California, USA, May 3-5, 2000.

[10] M. Gries. Methods for evaluating and covering the design space early in design development. RFC UCB/ERL MO3/32, Electronics Research Lab, University of California at Berkeley, August 2003.

[11] P. Metzgen, "A high Performance 32-bit ALU for programmable logic," in Proceeding of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays. ACM Press, 2004, p. 61-70.

[12] P. Metzgen, "Optimizing a High-Performance 32-bit Processor for Programmable Logic," in International Symposium on System-on-Chip, 2004.

[13] C. L. Valenzuela. A simple evolutionary algorithm for multi-objective optimization (SEAMO). In Proc. of the 2002 Congress on Evolutionary Computation (CEC'02), volume 1, pages 717-722, Honolulu, Hawaii, USA, May 12-17 2002 .

[14] K. Compton and S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software," ACM Computing Surveys, vol. 34, no. 2 (June 2002), pp. 171-210.

[15] R. Saleh, S. Wilton, S. Mirabbasi, A. Hu, M. Greenstreet, G. Lemieux, P. P. Pande, C. Grecu, and A. Ivanov. System-on-chip: Reuse and integration. Proc. of the IEEE, 94(6): 1050-1069, June 2006.

[16] R. K. Gupta and Y. Zorian. Introducing core-based system design. IEEE Design & Test of Computers, 14(4): 15-25, October-December 1997.

[17] Xilinx, Inc., "MicroBlaze Soft Processor," http://www.xilinx.com/xlnx/xil_prodcat/product.jsp?title=microblaze, January 2007.

[18] Altera Corporation. *Nios II Processor Reference Handbook,* May 2007.

[19] Xilinx, Inc., "MicroBlaze Processor Reference Guide," http://www. Xilinx.com/ise/embedded/mb_ref_guide.pdf, January 2007.

[20] Altera Corporation. Nios Processor Reference Handbook, May 2004. http://www.altera.com/products/ip/processors/nios/nio-index.html.

[21] Altera Corporation. Quartus II Version 7.2 Handbook, Version 7.2, May 2007.

[22] Altera Corporation. SOPC builder. http://www.altera.com/products/software/products/sopc/sop-index.html, January 2007.

[23] Xilinx Incorporated. Xilinx logic design: (XST). http://www.xilinx.com/products/design_tools/logic_design/synthesis/xst.htm, January 2007.

[24] V. Betz, J. Rose, and A. Marquardt, Architecture and CAD for Deep-Submicron FPGAs, Kluwer Academic Publishers: Norwell, MA, 1999.

[25] Altera Corporation, "Stratix Device Handbook," http://www.altera.com/literature/hb/stx/stratix_handbook.pdf, January 2007.

[26] Altera Corporation, "Excalibur Devices," http://www.altera.com/products/devices/arm/arm-index.html, January 2007.

[27] Xilinx Virtex-4 Overview. http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/overview/index.htm, January 2007.

[28] Altera Corporation, "Quartus II Development Software Handbook v7.2," http://www.altera.com/literature/hb/qts/quartusii_handbook.pdf, January 2007.

[29] T. Givargis, J. Henkel, and F. Vahid. Interface and cache power exploration for core-based embedded system design. In Proc. of the 1999 IEEE/ACM International Conference on Computer_Aided Design, pages 270-273, San Jose, California, USA, November 1999.

[30] P. Mishra and N. Dutt. Architecture description languages for programmable embedded systems. IEE Proceedings Computers & Digital Techniques, 152(3): 285-297, May 2005.

[31] H. Tomiyama, A. Halambi, P. Grun, N. Dutt and A. Nicolau. Architecture description languages for systems-on-chip design. In Proc. of the Sixth Asia Pacific Conference on Chip Design Language, pages 109-116, Fukuoka, Japan, October 1999.

[32] J. H. Holland. Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor, Michigan, USA, 1975.

[33] C. A. C. Coello. A comprehensive survey of evolutionary-based multiobjective optimization techniques. Knowledge and Information Systems, 1(3): 129-156, August 1999.

[34] C. M. Fonseca and P. J. Fleming. An overview of evolutionary algorithms in multiobjective optimization. Evolutionary Computation, 3(1): 1-16, Spring 1995.

[35] I. D. L. Anderson, M. A. S. Khalid. SCBuild: Design Space Exploration using Parameterized Cores: A Case Study", Proceedings of Canadian Conference on Electrical and Computer Engineering, 2006.

[36] P. Yiannacouras, J. Rose, and J. Gregory Steffan. Application-Specific Customization of Soft Processor Microarchitecture, Proc. of FPGA '06, February 2006. ACM Press.

[37] P. Yiannacouras. The michroarchitecture of FPGA-based soft processors. Master's thesis, University of Toronto, Toronto, Ontario, Canada, 2005.

[38] J. L. Hennessy, N. P. Jouppi, J. Gill, F. Baskett, A. Strong, T. R. Gross, C. Rowen, and J. Leonard. The MIPS machine. In COMPCON, pages 207, 1982.

[39] B. Fort, D. Capalija, Z. G. Vranesic and S. D. Brown. A Multithreaded Soft Processor for SoPC Area Reduction. Proc. of FCCM '06. IEEE Computer Society, April 2006.

[40] F. Plavec, B. Fort, Z. Vranesic, and S. Brown. Experiences with soft-core processor design. Proc. of IPDPS '05. IEEE Computer Society, April 2005.

[41] M. Itoh, S. Higaki, J. Sato, A. Shiomi, Y. Takeuchi, A. Kitajima, and M. Imai. PEAS-III: An ASIP Design Environment. In Proc. of the International Conference on Computer Design, September 2000.

[42] J. L. Hennessy and D. A. Patterson. Computer Architecture: A Quantitative Approach, Fourth Edition. Morgan Kaufmann, San Francisco, California, USA, September 2006.

[43] I. D. L. Anderson. A CAD Tool for Design Space Exploration of Embedded CPU Cores. M. A. Sc. Thesis, Department of Electrical and Computer Engineering, University of Windsor, February, 2007.

[44] I. D. L. Anderson, M. A. S. Khalid. SCBuild: A CAD Tool for Design Space Exploration of Embedded CPU Cores for FPGAs, accepted for publication, IET Computer and Digital Techniques (IET-CDT).

[45] Tcl Developer Xchange. http://www.tcl.tk/, January 2008.

[46] M. M. Mano and C. R. Kime. *Logic and Computer Design Fundamentals 2nd Edition Updated.* Prentice Hall, Upper Saddle River, New Jersey, USA, 2001.

[47] Extensible Markup Language (XML). http://www.w3.org/XML. January 2008.

[48] Institute of Electrical and Electronics Engineers. IEEE standard VHDL language reference manual, ANSI/IEEE Std 1076-1993, 1993.

[49] P. K. Jha and N. D. Dutt. Rapid estimation for parameterized components in high-level synthesis. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 1(3):296-303, September 1993.

[50] MathWorks. http://www.mathworks.com/access/helpdesk/help/toolbox/curvefit/index.html. May 2008.

[51] Altera Corporation. *Avalon Bus Specification Reference Manual, version 2.3*, July 2003.

[52] J. G. Tong, I. D. L. Anderson, M. A. S. Khalid. Soft-Core Processors for Embedded Systems. Proc. of International Conference on Microelectronics, ICM '06.

[53] EECS Instructional and Electronics Groups Homepage at University of California Berkeley. http://inst.eecs.berkeley.edu/, January 2008.

# Appendix A

# *Synthesis Results for the UW_Nios II Processor Template*

## A.1 Parameter Sweep Results

Table A.1: Parameter Sweep Data

| Config. | P1 | P2 | P3 | P4 | P5 | P6 | P7 | P8 | P9 | P10 | Clk (ns) | Eq. LE's |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 7.907 | 3022 |
| 1 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 8.875 | 3181 |
| 2 | 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 8.594 | 3462 |
| 3 | 1 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 8.875 | 3181 |
| 4 | 1 | 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 8.554 | 3452 |
| 5 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 8.875 | 3181 |
| 6 | 1 | 1 | 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 8.953 | 3551 |
| 7 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 18.322 | 4228 |
| 8 | 1 | 1 | 1 | 3 | 1 | 1 | 1 | 1 | 1 | 1 | 18.322 | 4228 |
| 9 | 1 | 1 | 1 | 4 | 1 | 1 | 1 | 1 | 1 | 1 | 12.837 | 3078 |
| 10 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 1 | 18.546 | 4270 |
| 11 | 1 | 1 | 1 | 1 | 3 | 1 | 1 | 1 | 1 | 1 | 12.837 | 3078 |
| 12 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 133.332 | 4177 |
| 13 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 149.443 | 4331 |
| 14 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 7.953 | 2970 |
| 15 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 5.768 | 594 |
| 16 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 10.032 | 2967 |

## A.2 Initial and Evolved Populations

### A.2.1 Initial Population

Table A.2: Initial Population

| Config. | P1 | P7 | P4 | P6 | P5 | P8 | P2 | P10 | P9 | P3 | Clk (ns) | Eq. LE's |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 2 | 1 | 2 | 1 | 1 | 2 | 2 | 1 | 1 | 1 | 20.2986 | 4339 |
| 1 | 3 | 1 | 3 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 21.1199 | 4614 |
| 2 | 1 | 1 | 4 | 1 | 1 | 2 | 2 | 2 | 2 | 1 | 13.8123 | 703 |
| 3 | 2 | 1 | 1 | 2 | 1 | 2 | 2 | 1 | 1 | 1 | 135.287 | 4287 |
| 4 | 2 | 1 | 1 | 1 | 2 | 1 | 3 | 1 | 1 | 1 | 20.1582 | 4859 |
| 5 | 2 | 1 | 1 | 1 | 3 | 1 | 3 | 1 | 1 | 2 | 15.4162 | 3670 |
| 6 | 3 | 1 | 1 | 1 | 1 | 2 | 3 | 1 | 2 | 2 | 8.1137 | 1192 |
| 7 | 2 | 2 | 1 | 1 | 1 | 1 | 3 | 2 | 1 | 2 | 154.113 | 4868 |
| 8 | 3 | 1 | 2 | 1 | 1 | 1 | 1 | 2 | 1 | 3 | 22.1736 | 4764 |
| 9 | 2 | 1 | 3 | 1 | 1 | 1 | 3 | 2 | 1 | 1 | 22.0494 | 4763 |
| 10 | 3 | 1 | 4 | 1 | 1 | 1 | 2 | 1 | 2 | 2 | 13.2953 | 1253 |
| 11 | 1 | 1 | 1 | 2 | 1 | 1 | 2 | 2 | 2 | 3 | 135.305 | 2382 |
| 12 | 1 | 1 | 1 | 1 | 2 | 1 | 3 | 1 | 1 | 1 | 19.1907 | 4700 |
| 13 | 1 | 1 | 1 | 1 | 3 | 2 | 1 | 1 | 1 | 2 | 13.8467 | 3185 |
| 14 | 2 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 2 | 3 | 8.7477 | 1285 |
| 15 | 3 | 2 | 1 | 1 | 1 | 2 | 3 | 1 | 2 | 2 | 149.614 | 2503 |
| 16 | 1 | 1 | 2 | 1 | 1 | 1 | 3 | 1 | 2 | 2 | 17.7936 | 2389 |
| 17 | 3 | 1 | 3 | 1 | 1 | 2 | 1 | 2 | 2 | 2 | 19.9949 | 2292 |
| 18 | 1 | 1 | 4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 12.8118 | 3079 |
| 19 | 2 | 1 | 1 | 2 | 1 | 2 | 3 | 2 | 2 | 2 | 135.92 | 2234 |
| 20 | 3 | 1 | 1 | 1 | 2 | 2 | 1 | 1 | 1 | 1 | 19.2747 | 4658 |
| 21 | 1 | 1 | 1 | 1 | 3 | 2 | 3 | 2 | 1 | 1 | 15.6517 | 3401 |
| 22 | 3 | 1 | 1 | 1 | 1 | 2 | 3 | 1 | 2 | 1 | 7.1457 | 1033 |
| 23 | 2 | 2 | 1 | 1 | 1 | 2 | 1 | 2 | 1 | 1 | 152.543 | 4383 |
| 24 | 1 | 1 | 2 | 1 | 1 | 2 | 3 | 2 | 1 | 1 | 21.1356 | 4552 |
| 25 | 2 | 1 | 3 | 1 | 1 | 1 | 2 | 2 | 1 | 3 | 23.4159 | 4865 |
| 26 | 2 | 1 | 4 | 1 | 1 | 1 | 3 | 2 | 1 | 1 | 16.5523 | 3613 |
| 27 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 1 | 3 | 134.351 | 4706 |
| 28 | 2 | 1 | 1 | 1 | 2 | 2 | 1 | 1 | 1 | 1 | 19.5562 | 4377 |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 29 | 2 | 1 | 1 | 1 | 3 | 2 | 1 | 2 | 2 | 1 | 13.8322 | 702 |
| 30 | 2 | 1 | 1 | 1 | 1 | 1 | 3 | 2 | 1 | 2 | 12.6132 | 3559 |
| 31 | 1 | 2 | 1 | 1 | 1 | 2 | 1 | 2 | 1 | 1 | 151.576 | 4224 |
| 32 | 1 | 1 | 2 | 1 | 1 | 1 | 2 | 1 | 2 | 2 | 18.1141 | 1963 |
| 33 | 2 | 1 | 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 19.2764 | 4388 |
| 34 | 2 | 1 | 4 | 1 | 1 | 2 | 3 | 1 | 1 | 2 | 15.4413 | 3619 |
| 35 | 1 | 1 | 1 | 2 | 1 | 1 | 2 | 1 | 1 | 2 | 135.241 | 4339 |
| 36 | 1 | 1 | 1 | 1 | 2 | 2 | 3 | 1 | 2 | 1 | 17.0977 | 2220 |
| 37 | 2 | 1 | 1 | 1 | 3 | 2 | 1 | 1 | 2 | 1 | 11.7072 | 757 |
| 38 | 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 8.5907 | 3462 |
| 39 | 1 | 2 | 1 | 1 | 1 | 1 | 3 | 1 | 2 | 1 | 147.914 | 2333 |
| 40 | 2 | 1 | 2 | 1 | 1 | 2 | 3 | 1 | 1 | 3 | 21.0241 | 4915 |
| 41 | 2 | 1 | 3 | 1 | 1 | 2 | 2 | 1 | 1 | 3 | 10.9327 | 3661 |
| 42 | 3 | 1 | 4 | 1 | 1 | 2 | 3 | 1 | 1 | 2 | 15.1598 | 3677 |
| 43 | 3 | 1 | 1 | 2 | 1 | 2 | 1 | 2 | 1 | 2 | 137.13 | 4669 |
| 44 | 2 | 1 | 1 | 1 | 2 | 1 | 2 | 1 | 1 | 1 | 20.4747 | 4432 |
| 45 | 1 | 1 | 1 | 1 | 3 | 1 | 1 | 1 | 1 | 2 | 13.8007 | 3237 |
| 46 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 1 | 8.9052 | 646 |
| 47 | 3 | 2 | 1 | 1 | 1 | 1 | 2 | 1 | 2 | 1 | 148.92 | 2502 |
| 48 | 3 | 1 | 2 | 1 | 1 | 2 | 3 | 2 | 2 | 3 | 20.7286 | 2318 |
| 49 | 1 | 1 | 3 | 1 | 1 | 1 | 1 | 1 | 2 | 3 | 17.2159 | 2330 |
| 50 | 3 | 1 | 4 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 14.4518 | 1195 |
| 51 | 1 | 1 | 1 | 2 | 1 | 2 | 3 | 1 | 2 | 2 | 132.828 | 2286 |
| 52 | 3 | 1 | 1 | 1 | 2 | 1 | 3 | 2 | 2 | 1 | 19.8627 | 2278 |
| 53 | 3 | 1 | 1 | 1 | 3 | 2 | 3 | 2 | 1 | 3 | 17.3837 | 3596 |
| 54 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 8.8732 | 3181 |
| 55 | 3 | 2 | 1 | 1 | 1 | 2 | 3 | 2 | 2 | 3 | 151.817 | 2421 |
| 56 | 2 | 1 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 19.2841 | 4388 |
| 57 | 3 | 1 | 3 | 1 | 1 | 2 | 3 | 2 | 1 | 3 | 22.8599 | 4747 |
| 58 | 1 | 1 | 4 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 13.7798 | 3238 |
| 59 | 2 | 1 | 1 | 2 | 1 | 2 | 3 | 1 | 2 | 3 | 133.873 | 2436 |
| 60 | 2 | 1 | 1 | 1 | 2 | 1 | 2 | 2 | 2 | 1 | 20.4647 | 1949 |
| 61 | 2 | 1 | 1 | 1 | 3 | 2 | 3 | 2 | 2 | 1 | 14.4802 | 1132 |
| 62 | 1 | 1 | 1 | 1 | 1 | 1 | 3 | 2 | 2 | 2 | 9.5067 | 1128 |
| 63 | 3 | 2 | 1 | 1 | 1 | 1 | 3 | 1 | 1 | 1 | 150.739 | 4822 |
| 64 | 2 | 1 | 2 | 1 | 1 | 1 | 2 | 1 | 1 | 2 | 21.2206 | 4382 |
| 65 | 3 | 1 | 3 | 1 | 1 | 1 | 3 | 1 | 1 | 3 | 20.6889 | 4854 |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 66 | 3 | 1 | 4 | 1 | 1 | 2 | 1 | 1 | 1 | 3 | 14.5898 | 3617 |
| 67 | 2 | 1 | 1 | 2 | 1 | 2 | 3 | 1 | 1 | 2 | 135.934 | 4717 |
| 68 | 1 | 1 | 1 | 1 | 2 | 1 | 3 | 2 | 2 | 2 | 20.1447 | 2376 |
| 69 | 3 | 1 | 1 | 1 | 3 | 2 | 2 | 1 | 2 | 2 | 13.3622 | 1200 |
| 70 | 1 | 1 | 1 | 1 | 1 | 1 | 3 | 2 | 1 | 1 | 10.6777 | 3397 |
| 71 | 2 | 2 | 1 | 1 | 1 | 2 | 3 | 1 | 1 | 2 | 152.034 | 4871 |
| 72 | 1 | 1 | 2 | 1 | 1 | 1 | 2 | 1 | 1 | 3 | 20.3311 | 4917 |
| 73 | 2 | 1 | 3 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 19.3224 | 4336 |
| 74 | 2 | 1 | 4 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 14.7333 | 758 |
| 75 | 1 | 1 | 1 | 2 | 1 | 2 | 2 | 2 | 2 | 3 | 135.351 | 2330 |
| 76 | 1 | 1 | 1 | 1 | 2 | 1 | 2 | 1 | 2 | 1 | 17.3722 | 2001 |
| 77 | 3 | 1 | 1 | 1 | 3 | 2 | 3 | 1 | 2 | 1 | 12.0737 | 1089 |
| 78 | 3 | 1 | 1 | 1 | 1 | 2 | 3 | 1 | 1 | 1 | 9.2847 | 3461 |
| 79 | 3 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 147.952 | 2343 |
| 80 | 3 | 1 | 2 | 1 | 1 | 1 | 1 | 2 | 2 | 1 | 18.9886 | 2186 |
| 81 | 3 | 1 | 3 | 1 | 1 | 1 | 1 | 1 | 2 | 3 | 17.9019 | 2391 |
| 82 | 3 | 1 | 4 | 1 | 1 | 1 | 2 | 1 | 2 | 2 | 13.2953 | 1253 |
| 83 | 3 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 2 | 3 | 132.898 | 2339 |
| 84 | 1 | 1 | 1 | 1 | 2 | 1 | 2 | 1 | 1 | 3 | 20.5572 | 4958 |
| 85 | 1 | 1 | 1 | 1 | 3 | 1 | 2 | 2 | 1 | 2 | 16.8942 | 3185 |
| 86 | 3 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 2 | 3 | 8.4662 | 1343 |
| 87 | 3 | 2 | 1 | 1 | 1 | 2 | 1 | 2 | 1 | 2 | 153.23 | 4823 |

### A.2.2 Evolved Population

Table A.3: Evolved Population

| Config. | P1 | P7 | P4 | P6 | P5 | P8 | P2 | P10 | P9 | P3 | Clk (ns) | Eq. LE's |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 2 | 2 | 3 | 1016 | 8.9827 |
| 1 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 1 | 698 | 8.8582 |
| 2 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 1 | 646 | 8.9052 |
| 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 3 | 1068 | 8.9367 |
| 4 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 1 | 698 | 8.8592 |
| 5 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 1 | 1 | 1 | 3129 | 8.9192 |
| 6 | 3 | 1 | 1 | 1 | 1 | 2 | 3 | 1 | 2 | 2 | 1192 | 8.1137 |
| 7 | 3 | 1 | 1 | 1 | 1 | 2 | 2 | 1 | 2 | 1 | 1141 | 7.4662 |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8 | 2 | 1 | 1 | 1 | 1 | 1 | 3 | 1 | 2 | 2 | 1186 | 8.3492 |
| 9 | 3 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 1 | 979 | 8.5767 |
| 10 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 701 | 9.8272 |
| 11 | 1 | 1 | 4 | 1 | 1 | 2 | 2 | 1 | 2 | 1 | 758 | 11.6873 |
| 12 | 1 | 1 | 4 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 810 | 11.6408 |
| 13 | 2 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 2 | 2 | 704 | 7.7472 |
| 14 | 2 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 2 | 3 | 1285 | 8.7477 |
| 15 | 2 | 1 | 1 | 1 | 1 | 2 | 3 | 1 | 2 | 1 | 1131 | 7.4272 |
| 16 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 2 | 1 | 753 | 6.7342 |
| 17 | 2 | 1 | 4 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 810 | 11.6403 |
| 18 | 3 | 1 | 4 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 1091 | 11.3588 |
| 19 | 1 | 1 | 1 | 1 | 1 | 2 | 3 | 1 | 2 | 1 | 972 | 6.4597 |
| 20 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 753 | 6.7332 |
| 21 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 2 | 1 | 542 | 5.8117 |
| 22 | 3 | 1 | 1 | 1 | 1 | 2 | 3 | 1 | 2 | 1 | 1033 | 7.1457 |
| 23 | 2 | 1 | 1 | 1 | 1 | 2 | 2 | 1 | 2 | 1 | 704 | 7.7477 |
| 24 | 1 | 1 | 4 | 1 | 1 | 1 | 1 | 2 | 2 | 1 | 596 | 12.7978 |
| 25 | 3 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 1 | 1138 | 9.5452 |
| 26 | 2 | 1 | 4 | 1 | 1 | 2 | 1 | 1 | 2 | 1 | 758 | 11.6863 |
| 27 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 1 | 2 | 2 | 704 | 7.7482 |
| 28 | 3 | 1 | 4 | 1 | 1 | 2 | 1 | 1 | 2 | 2 | 1198 | 12.3728 |
| 29 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 1 | 539 | 7.8907 |
| 30 | 1 | 1 | 1 | 1 | 1 | 1 | 3 | 2 | 2 | 3 | 1119 | 9.5847 |
| 31 | 2 | 1 | 1 | 1 | 3 | 1 | 1 | 1 | 2 | 1 | 809 | 11.6612 |
| 32 | 1 | 1 | 4 | 1 | 1 | 2 | 3 | 2 | 2 | 1 | 974 | 13.4918 |
| 33 | 2 | 1 | 1 | 1 | 3 | 2 | 1 | 1 | 2 | 1 | 757 | 11.7072 |
| 34 | 3 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 1 | 1086 | 9.5912 |
| 35 | 2 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 1 | 701 | 9.8267 |
| 36 | 2 | 1 | 1 | 1 | 1 | 2 | 3 | 2 | 2 | 1 | 1076 | 9.5522 |
| 37 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 1 | 2 | 1 | 701 | 6.7802 |
| 38 | 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 3462 | 8.5907 |
| 39 | 2 | 1 | 1 | 1 | 1 | 2 | 3 | 2 | 2 | 2 | 1079 | 10.5202 |
| 40 | 2 | 1 | 1 | 1 | 1 | 1 | 3 | 2 | 2 | 1 | 1128 | 9.5062 |
| 41 | 2 | 1 | 1 | 1 | 3 | 2 | 1 | 2 | 2 | 2 | 705 | 14.8002 |
| 42 | 1 | 1 | 4 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 651 | 10.6728 |
| 43 | 3 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 2 | 2 | 1141 | 7.4657 |
| 44 | 2 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 1 | 649 | 9.8727 |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 45 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 594 | 5.7657 |
| 46 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 1 | 646 | 8.9052 |
| 47 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 3 | 1175 | 9.9512 |
| 48 | 3 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 2 | 3 | 1132 | 7.5437 |
| 49 | 1 | 1 | 4 | 1 | 1 | 2 | 2 | 2 | 2 | 1 | 703 | 13.8123 |
| 50 | 3 | 1 | 4 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 1195 | 14.4518 |
| 51 | 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 1193 | 7.4197 |
| 52 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 3 | 1227 | 9.9052 |
| 53 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 3022 | 7.9047 |
| 54 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 3181 | 8.8732 |
| 55 | 1 | 1 | 1 | 1 | 1 | 2 | 3 | 1 | 2 | 2 | 1131 | 7.4277 |
| 56 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 753 | 6.7337 |
| 57 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 2 | 2 | 701 | 6.7797 |
| 58 | 1 | 1 | 4 | 1 | 1 | 2 | 3 | 1 | 2 | 2 | 1188 | 12.3348 |
| 59 | 1 | 1 | 1 | 1 | 1 | 2 | 3 | 2 | 2 | 2 | 1076 | 9.5527 |
| 60 | 3 | 1 | 1 | 1 | 1 | 2 | 3 | 1 | 2 | 1 | 1033 | 7.1457 |
| 61 | 2 | 1 | 1 | 1 | 3 | 2 | 1 | 1 | 2 | 2 | 760 | 12.6752 |
| 62 | 1 | 1 | 1 | 1 | 1 | 1 | 3 | 2 | 2 | 2 | 1128 | 9.5067 |
| 63 | 3 | 1 | 1 | 1 | 1 | 1 | 3 | 2 | 2 | 2 | 1189 | 10.1927 |
| 64 | 2 | 1 | 1 | 1 | 1 | 2 | 1 | 2 | 2 | 3 | 1175 | 9.9502 |
| 65 | 2 | 1 | 4 | 1 | 1 | 2 | 1 | 1 | 2 | 2 | 761 | 12.6543 |
| 66 | 2 | 1 | 1 | 1 | 1 | 2 | 2 | 1 | 2 | 2 | 695 | 8.7157 |
| 67 | 3 | 1 | 4 | 1 | 1 | 2 | 1 | 2 | 2 | 1 | 984 | 13.5298 |
| 68 | 1 | 1 | 1 | 1 | 1 | 1 | 3 | 2 | 2 | 2 | 1128 | 9.5067 |
| 69 | 1 | 1 | 4 | 1 | 1 | 1 | 2 | 1 | 2 | 2 | 813 | 12.6093 |
| 70 | 3 | 1 | 1 | 1 | 1 | 2 | 3 | 1 | 2 | 2 | 1192 | 8.1137 |
| 71 | 3 | 1 | 1 | 1 | 1 | 2 | 3 | 2 | 2 | 1 | 978 | 9.2707 |
| 72 | 2 | 1 | 4 | 1 | 1 | 1 | 1 | 2 | 2 | 1 | 755 | 13.7653 |
| 73 | 3 | 1 | 1 | 1 | 1 | 2 | 2 | 1 | 2 | 2 | 1144 | 8.4342 |
| 74 | 1 | 1 | 4 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 755 | 13.7658 |
| 75 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 698 | 8.8587 |
| 76 | 3 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 1138 | 9.5447 |
| 77 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 2 | 2 | 756 | 7.7022 |
| 78 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 2 | 3 | 1282 | 7.7802 |
| 79 | 3 | 1 | 4 | 1 | 1 | 1 | 1 | 2 | 2 | 1 | 1036 | 13.4838 |
| 80 | 2 | 1 | 4 | 1 | 1 | 1 | 2 | 1 | 2 | 1 | 813 | 12.6088 |
| 81 | 2 | 1 | 4 | 1 | 1 | 2 | 1 | 2 | 2 | 1 | 703 | 13.8113 |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 82 | 1 | 1 | 4 | 1 | 1 | 1 | 2 | 1 | 2 | 1 | 1012 | 11.6413 |
| 83 | 2 | 1 | 4 | 1 | 1 | 2 | 2 | 2 | 2 | 1 | 706 | 14.7798 |
| 84 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 756 | 7.7012 |
| 85 | 1 | 1 | 4 | 1 | 1 | 1 | 2 | 2 | 2 | 1 | 755 | 13.7663 |
| 86 | 3 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 2 | 1 | 982 | 6.4977 |
| 87 | 1 | 1 | 1 | 1 | 3 | 1 | 2 | 2 | 2 | 2 | 757 | 14.7552 |

# VITA AUCTORIS

Omar Al Rayahi was born in Al-Ain, United Arab Emirates on March 19,1982. He received his B. A. Sc. degree in electrical engineering in 2005 form the University of Windsor in Windsor, Ontario, Canada. He is currently a candidate in the electrical and computer engineering M. A. Sc. program at the University of Windsor. His research interests include FPGA-related technologies, embedded systems and their applications.