1-1-2007

A CAD tool for design space exploration of embedded CPU cores for FPGAs.

Ian D. L. Anderson
University of Windsor

A CAD Tool for Design Space Exploration of Embedded CPU Cores for FPGAs

by

Ian D. L. Anderson

A Thesis
Submitted to the Faculty of Graduate Studies and Research
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
2007

Your file Votre référence
ISBN: 978-0-494-42311-0
Our file Notre référence
ISBN: 978-0-494-42311-0

Abstract

In this thesis, a genetic algorithm based design space exploration technique using parameterized cores is examined. A case study was first conducted to investigate the feasibility of applying a genetic based approach to a parameterized core. Next, a computer aided design tool called SCBuild was developed which utilizes the investigated approach. This tool is capable of applying a genetic algorithm to a core's parameters, and generating hardware description language models of core variants. The tool can also compute estimates of a variant's area and critical path delay on a field programmable gate array. Using this tool, several experiments were conducted using a soft-core processor with a large design space. It was concluded from these experiments that using a genetic algorithm to explore the design space of a parameterized core can help a designer make intelligent decisions regarding the assignment of values to the parameters of an embedded hardware platform.

To Mom, Dad, Karis and Hil.

Acknowledgments

I thank the Lord Jesus for the way in which He pulled me through this degree and gave me the determination to see it to completion. I find myself standing here at the end of this task, not entirely sure about how I got here. But here I am nonetheless, and for that I am thankful.

I would like to express my sincere thanks to Dr. Khalid for all of the advice and guidance he provided over the course of this research. His assistance to me was invaluable as I worked my way through the often-confusing world of research work. My appreciation also goes out to Dr. Kobti and Dr. Wu, for taking the time to sit on my committee and to review my thesis, and to Dr. Tepe for sitting in as the Chair of Defense.

Thanks to my family for all of their support, patience and prayers as I completed this thesis. Thanks Mom for all your love, encouragement and wise advice, from which I have always benefited immensely. I know you've been through a lot over the past year, but I still appreciate the ways that you were there for me as much as you could. Dad, I am grateful for the ways in which you helped me get through this degree; for cooking dinner, for picking me up on those late nights at the office, and just being around when I needed someone to pester when my thesis was driving me

nuts. It looks like I have officially finished my thesis before you could get yours done, so I guess you owe me a trip to Timmy's. Karis, my brilliant sister; thanks to you I never forgot that my thesis was "taking forever". But at least now it's done, so you can find something new to tease me about. Thanks for keeping me humble and giving me some good laughs along the way.

To my wonderful girlfriend, Hilary: you've always been a steadfast support to me in all my endeavours. It is always great spending time with you, eating lunch, going for a run, or just sitting around chatting and laughing about anything and everything together. Your visits were always a very welcome relief from my thesis work. Thank you also for helping me to edit my thesis and even learning what "FPGA" stands for. And to the rest of the Leslie family: thanks for the ways that you've welcomed me into your lives. Also, thank you very much for lending me your car; it was *very* helpful to have it these past few weeks. And thank you Brian for taking the time to read and edit my entire thesis. Your revisions were very good and helped to make this work that much better.

Finally, I would like to acknowledge my friends and fellow graduate students at the University of Windsor. Jay, thanks for always keeping us amused with your various accents and impressions. We had some great times, from our all-day "programming parties" to just hanging out day-to-day in the office. Marwan, I will always remember all of the great conversations we've had on a whole host of topics ranging from religion, ethics and philosophy to the weather, food, politics and many other things that were (thankfully) unrelated to the field of engineering. Seldom have I met anybody who can talk intelligently on as many different subjects as you. Thanks to Amir for his willingness to give generously of his time in all of the ways that he has helped me during the course of this degree, and even before that. Thanks to Frank for his assistance on many occasions; for providing parts, equipment and advice for my various projects, and for just being available to sit and chat when I needed a break.

Lastly, thanks to the rest of my engineering colleagues as well; to Ray, Omar, Aws, Junsong, Hongmei, Kevin, Matt, Andrew, Harb, Mahzad, Ashkan and everyone else who have made this time in my life more enjoyable.

Contents

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Definition
ADL	Architecture Description Language
ALU	Arithmetic Logic Unit
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction-set Processor
CAD	Computer Aided Design
CPU	Central Processing Unit
DOF	Decode and Operand Fetch
DSE	Design Space Exploration
DSP	Digital Signal Processing
EA	Evolutionary Algorithm
EX	Execute
FF	Flip-Flop
FPGA	Field Programmable Gate Array
GA	Genetic Algorithm
GUI	Graphical User Interface
HDL	Hardware Description Language
IEEE	Institute of Electrical and Electronics Engineers
I/O	Input/Output
IC	Integrated Circuit
IF	Instruction Fetch
IOE	Input/Output Element
IP	Intellectual Property
LAB	Logic Array Block
LE	Logic Element

LUT	Lookup Table
MUX	Multiplexer
PLD	Programmable Logic Device
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
SEAMO	Simple Evolutionary Algorithm for Multi-objective Optimization
SoC	System on a Chip
SOPC	System on a Programmable Chip
UART	Universal Asynchronous Receiver/Transmitter
UML	Unified Modeling Language
VHDL	Very High Speed Integrated Circuit Hardware Description Language
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.
V_i	Set of possible values for the i_{th} parameter.
D	Design space.
$\|\cdot\|$	Cardinality operator.
N	Size of genetic population.
G	Number of generations.
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

In our modern digital age, devices utilizing embedded systems have become very common and enjoy widespread use in our daily lives. Examples of these systems are abundant and include cellular phones, digital cameras, appliances, automobiles, airplanes, and manufacturing systems. All of these rely on embedded electronic systems to carry out the task for which they were designed. Essentially, an embedded system is an electronic sub-system that utilizes computational hardware to perform a small set of tasks that are specific to a particular application [45]. They can be logically broken down into two major components: the embedded software (sometimes referred to as *firmware*) and the digital hardware, as shown in Figure 1.1 below.

The hardware component usually consists of one or more embedded central processing units (CPUs) and their associated application-specific hardware. These components communicate with one another and with embedded memory and Input/Output (I/O) components over a common bus. The software component is a program written and compiled specifically to run on the embedded processors. In contrast

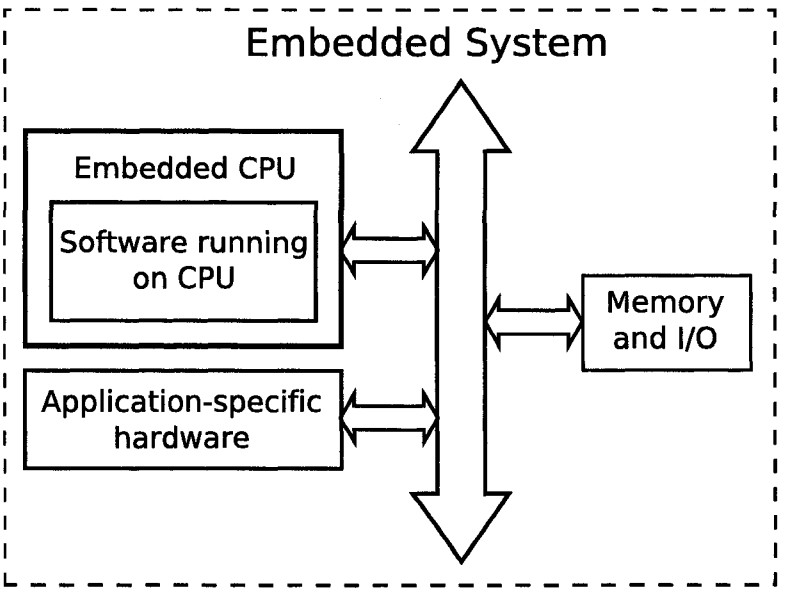


Figure 1.1: Block Diagram of an Embedded System

with general-purpose personal computer systems, which are designed to run a virtually infinite variety of different software programs, an embedded processor is only required to execute one piece of software continuously. Therefore, the processor can be optimized to run that piece of software as efficiently as possible; resulting in what is known as an Application Specific Instruction-Set Processor (ASIP). Since embedded systems are utilized in an extremely wide array of applications, it is no surprise that the market for embedded systems today is far larger than that of general-purpose personal computer systems. Therefore, it is quite clear that the field of embedded system design is an important and substantial area of study.

With the continual improvement of integrated circuit (IC) process technology, complete embedded systems can be built onto a single chip. This trend has come to be known as the "system-on-a-chip" (SoC) paradigm. As a result, ever more complex circuits can be designed and implemented on a single IC chip. The widespread use and growing complexity of embedded system designs has led to many important

innovations, but has also introduced many major design challenges. Most importantly, the challenge of developing a complex system within the constraints of a reasonable budget and time-frame is a constant consideration for all embedded systems engineers.

When designing an embedded system, there are several major approaches that have been taken. The *traditional* approach [27, 23] involves first designing the hardware portion of the embedded system, including the microprocessor and associated application-specific circuitry and then writing the software to run on the microprocessor after the hardware design phase has been completed. However, it was observed by many designers that by using this approach they often missed out on many potential optimizations that could be exploited if the design of the hardware and software portions of the system were considered together. Therefore, a second approach, known as the *hardware/software co-design* approach [52, 33, 24, 23], took shape. In this approach, the design of the hardware and software parts of the system are designed *concurrently*, allowing the designer to explore the tradeoffs between hardware and software implementations of the various system tasks.

As the complexity of embedded systems designs increased over time, designing each and every hardware component of the system from scratch soon became far too impractical and expensive for many designers. Therefore, a third approach, known as the *platform-based design* approach [13, 49, 61], emerged. Platform-based design entails the the idea of using pre-designed and pre-tested hardware components known as intellectual property (IP) cores as a *platform* upon which to build complete systems. This approach heavily emphasizes design *reuse*. Using IP cores, a designer can be confident that the building blocks he or she is using in the design will function as expected. As a result, the designer is subsequently freed from designing the components from scratch, which naturally leads to a much shorter design cycle.

Soft-cores are a particular class of hardware IP cores that are often used by designers to build their systems. Essentially soft-cores are hardware components that

are described using a hardware description language (HDL). In order to increase the reusability of soft IP cores across a wider range of application domains, many of them are *parameterized*, meaning that the core's architecture features a number of configurable options or *parameters* that can be set by the engineer at design-time. These options are built into the core's architecture by its creators and allow the designer of an embedded system to tailor the core to closely match the requirements of the system's intended application.

The recent development of field programmable gate arrays (FPGAs) and other programmable logic devices (PLDs) has introduced designers to a new type of flexible prototyping and implementation medium for embedded systems designs that utilize soft-core components. FPGAs are programmable IC chips that can be configured to function like virtually any digital circuit that can be conceived, subject to the limitations imposed by the logic capacity of the device. Soft-core descriptions of hardware components can be translated into a logic circuit which can then be mapped directly onto the programmable fabric of an FPGA. This allows a designer to test the functionality of a logic circuit in real-time without having to fabricate a custom chip. Using FPGAs, different design tradeoffs can be rapidly explored, allowing better design decisions to be made and reducing the overall development time of a system.

When designing an embedded system for any application, it is important that designers come up with a hardware platform that is well suited for their purposes. If this is not done well, the result may be a system that is over-designed or sub-optimal for the intended application, which would almost certainly incur the unnecessary expenditure of additional time and financial resources. Therefore, it is crucial that a good hardware design be selected early in the design process. This endeavour is complicated by the fact that there almost always exists a very large set of possible hardware designs to choose from. This set of all possible hardware design configurations is known as the *design space*, and the task of selecting the best design from that

set is commonly referred to as *design space exploration* (DSE) [32].

As the complexity of the system being designed increases and the number of parameters rises, the design space for that system expands. As a result, exploring the design space in search of the best system configuration for a given application can be a difficult and tedious task. An exhaustive exploration approach [31] is often infeasible, therefore many designers rely on past experience to narrow down the number of possible design configurations. Although designer experience is always a very valuable asset to any design project, this approach is considered by some to be too *ad hoc* [32], and may sometimes yield sub-optimal designs. Therefore, much research has been conducted into automating the process of design space exploration.

This thesis is primarily concerned with the question of how to derive a "good" hardware platform for a given embedded system constructed from a set of parameterized soft-core components. The emphasis of this research is on the design of embedded microprocessors targeted for implementation on FPGAs specifically, since FPGAs are a relatively new technology, and as such, microprocessor design targeting these devices is not yet well understood. In this work, the results of a preliminary investigation into an automated DSE approach involving parameterized cores [9] are presented, the design of a software-based Computer Aided Design (CAD) tool that utilizes this approach is described, and the results obtained from experimentation with this CAD tool are discussed.

1.1 Thesis Objectives

The main goal of this research is to contribute toward the enhancement of the collective understanding of how embedded microprocessor design targeting FPGAs is unique in terms of processor architectures, CAD tools, and design techniques and methodologies. In order to achieve this goal, an exploration of the design space of soft-core processors targeting FPGAs should be conducted. To this end, this research

has several major objectives:

1. Investigate the feasibility of applying an automated design space exploration algorithm to a parameterized soft-core with a sizable design space.

2. Develop a software-based "processor builder" CAD tool that is capable of exploring the design space of a parameterized soft-core using an automated DSE approach. This tool should also be able to generate HDL descriptions for "variants" of a core, given a set of parameter values.

3. Perform an exploration of the design space of a parameterized soft-core processor using the processor builder tool, use the tool to generate a set of variant cores, and evaluate the variants in terms of performance and area utilization on an FPGA.

To address the first objective, a preliminary case study was conducted in which an automated DSE approach was applied to the Altera Nios [17] parameterized soft-core processor. In this study, the design space of the Nios core was explored using the Simple Evolutionary Algorithm for Multi-objective Optimization (SEAMO) [69], a genetic algorithm (GA) based approach. For the second thesis objective, a software-based CAD tool called *SCBuild* ("Soft-Core Build") was developed which utilizes the SEAMO algorithm to explore the design space of a user-supplied parameterized core. SCBuild is also capable of estimating a core's area utilization and performance on an FPGA, and can generate structural Very High Speed Integrated Circuit Hardware Description Language (VHDL) [54] descriptions of a core given a specific set of parameter values and a library of soft-core building-block components. Finally, to satisfy the third objective, several design space exploration experiments were conducted using SCBuild and a simple parameterized Reduced Instruction Set Computer (RISC) soft-core microprocessor [48].

1.2 Thesis Organization

The outline of this thesis is as follows. Chapter 2 introduces the reader to the greater context of this research by providing the relevant background information and a summary of some of the previous work that has been done by other researchers in this area of study. Chapter 3 discusses a preliminary case study involving the design space exploration of the Altera Nios soft-core processor using the SEAMO algorithm, which was carried out in order to lead directly into the core of this research. In Chapter 4, the design and implementation of SCBuild are discussed in detail. Chapter 5 presents the results obtained through experimentation using SCBuild and a simple parameterized RISC processor. Finally, Chapter 6 concludes this thesis and discusses possible future work in this area.

Chapter 2

Background and Previous Work

In this chapter the background material that is relevant to this research is presented, followed by a brief summary of previous work that has been done in this area. This chapter begins with a discussion of the different classes of intellectual property cores that exist at the various levels of abstraction. Then the parameterization of IP cores is defined, followed by a discussion of several prominent examples of parameterized soft-core processors from the industrial and open-source communities. Next, the basic concepts of FPGA technology are briefly explained, followed by an introduction to design space exploration and multi-objective optimization. Finally, this chapter concludes with a presentation of previous work that is closely related to this research.

2.1 Intellectual Property (IP) Cores

The definition of the term "intellectual property" is a broad one and covers a wide range of products and ideas across numerous fields of research. However, in the

context of this research, the term refers to reusable hardware or software building blocks that have been pre-designed and pre-tested prior to deployment in a design [64, 34]. These building blocks are usually the property of a particular individual or organization who licenses designers to use their hardware and software blocks. This idea of reuse is certainly not a new one; relying on past knowledge and experience has led to virtually all of the great discoveries and advancements over the course of history. For example, in the area of software development, programmers and software engineers have been collecting useful functions together into libraries for a long time. In the context of hardware design for embedded systems, the term "IP core" refers to reusable hardware components that are ready to be placed into a design with little or no modification made to them. Hardware IP cores can be any of a number of different types of digital components, including full microprocessors. These cores can come in the form of descriptions of hardware at the various levels of abstraction in the digital abstraction hierarchy, as will be discussed in the next section.

2.1.1 The Digital Abstraction Hierarchy

Any digital system can be described at different levels or layers of abstraction. Bell and Newell [11] were some of the first writers to formally discuss the hierarchy of abstraction levels in the context of digital system design. This hierarchy also appears in updated form in later literature [50, 65, 61]. The hierarchy includes the five major levels of abstraction shown in Table 2.1, and each level is characterized by a distinct class of languages that are used to represent the *behaviour* and *structure* of the system. Behaviour refers to the way that the system or its components interact with their environment, while structure refers to the set of interconnected components that make up the system. A description of the behaviour and structure of the system can be made at any one of these abstraction levels, and the process of translating a higher-level description to a lower-level description is generally known as *synthesis*

Table 2.1: The Digital Design Hierarchy [11, 50, 65, 61]

Level	Behaviour	Structure
System (Architecture)	Communicating Processes	Processors, Memories
Algorithm (Program)	Programming Languages	Data Structures
Register Transfer (RTL)	Register Transfers	Registers, ALUs, MUXes
Logic (Gate)	Boolean Equations	Logic gates, Flip-flops
Circuit (Layout)	Circuit Equations	Interconnected Transistors

[50].

Starting from the highest, the five levels of system abstraction are the System level (also known as the Architecture level), the Algorithm (or Program) level, the Register Transfer level (RTL), the Logic (Gate) level, and the Circuit (or Layout) level.

At the Circuit Level, the system is viewed as a circuit consisting of a collection of interconnected transistors and their physical layout on an IC chip. All signals in the system at this level are continuously varying quantities, so the behaviour of the system can be described using the fundamental equations of circuit analysis. The system's structure may be described symbolically using a schematic or layout diagram.

At the Logic Level, it is assumed that all signals in the system are discrete variables that can take on one of two values: 1 or 0 (or alternatively, high or low). This allows the system to be described in terms of its logical behaviour using Boolean equations. The structure of the system is described as a combinational or sequential circuit consisting of primitive logic components such as gates and flip-flops and their associated set of interconnections.

As its name implies, at the Register Transfer Level the behaviour of the system is expressed as a set of data transfers between storage registers. The system performs a set of discrete *micro-operations*, in which the data stored in the registers

are manipulated or combined with other data and then stored in another register. Structurally, the system consists of any number of registers and functional units such as arithmetic logic units (ALUs) connected together by buses. In general, hardware description languages such as VHDL, Verilog [55] and others can be used to describe the structure and behaviour of digital hardware components at this level.

When viewed at the Algorithm Level, the system is seen as a collection of data structures such as variables stored within a memory block and the instructions that operate on those variables. Instructions are formed when micro-operations at the Register Transfer Level are combined to form complete operations, such as the addition of two numbers or the transfer of data to and from memory. A collection of these instructions form a complete *instruction set*, which is the base language for describing system behaviour at this level. The instructions are executed sequentially, which is unique to this level, since at all levels below the Algorithm level, the behaviour of the system is expressed as a set of events occurring in parallel. Within this level, there is a sub-hierarchy of languages that are used to express the functionality of a program. High-level languages such as C/C++ [43], Visual Basic [21], etc. are translated down to assembly language by a compiler, which is, in turn, translated to machine code by an assembler.

Finally, at the System Level, the structure of the system is depicted as a set of abstract processing elements interacting with one another and the external environment. At this level, a designer would be concerned with overall system architecture and information flow between the processors, the memory and their interface to the surrounding environment (i.e. I/O).

2.1.2 Classes of Hardware IP Cores

In general, there are three major classes of hardware IP cores available: hard-cores, firm-cores and soft-cores [64, 34]. These different classes represent descriptions of

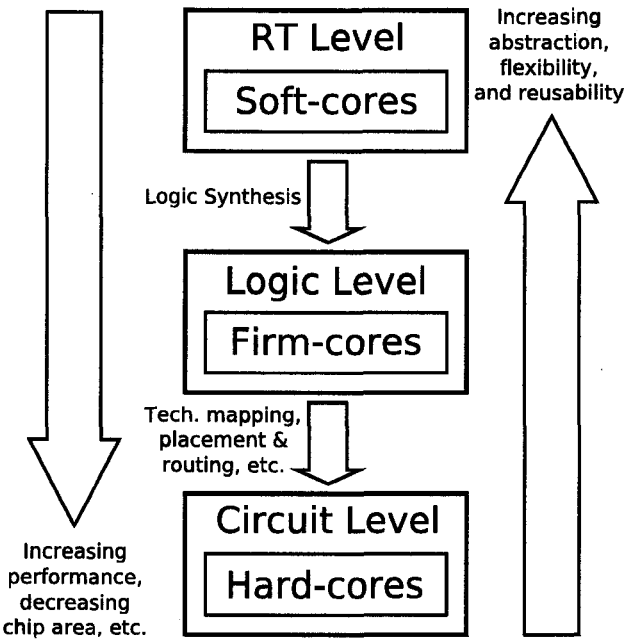


Figure 2.1: The Three Classes of Hardware IP Cores

hardware components at different levels of abstraction. Hard-cores are Circuit-level descriptions of components and are optimized for a particular target IC technology and include information regarding the physical layout of the core on a chip. Firm-cores are pre-synthesized Logic-level netlist descriptions of digital components that are ready for technology mapping, placement and routing on a given target IC process. Finally, soft-cores are components that are described at the Register Transfer Level using a synthesizable subset of a hardware description language. A soft-core description of a component can translated into a firm-core using a hardware synthesis tool such as Synopsys Design Compiler [39], Altera's Quartus II software [18], Xilinx Synthesis Technology (XST) [41], or the like. Subsequently, a firm-core description can then be translated into a hard-core through the process of technology mapping, which maps generic logic primitives such as gates and flip-flops onto particular physical implementations of those primitives. This process of translating a higher-level

description to a lower level description is illustrated in Figure 2.1.

This research focuses exclusively on the development of soft-core hardware components. Using soft-core components in designs holds a number of distinctive advantages to the designer. First, soft-cores are flexible and can be customized for a specific application with relative ease. Second, they are technology independent in that they can be synthesized for virtually any desired Application Specific Integrated Circuit (ASIC) or FPGA technology. Third, due to their technology independence, they are more immune to becoming obsolete as technology changes when compared with Circuit- or Logic-level descriptions of a component. Fourth, since a soft-core's structure and behaviour are described at a higher abstraction level using an HDL, it becomes much easier to understand the overall design of the component. Fifth, since they are written using an HDL, designing them often resembles the process of software development. Also, as an added benefit, software tools can be created to automatically generate the HDL code of a soft-core component. Finally, due to their flexibility, they can be easily parameterized, thus greatly enhancing their reusability and applicability across a wider range of designs and applications.

2.2 Parameterization: Increasing the Reusability of an IP Core

A parameterized core is a hardware component whose architectural features can be varied to a certain extent. A *parameter* is a particular feature or aspect of the component's architecture that can be changed and assigned particular values from a finite set by the embedded system designer [76, 29]. Examples of parameters include variable bus widths, functional unit implementations, hardware algorithms, memory sizes, pipeline depth, etc. Assigning values to all of the parameters of a component produces one *configuration.* Parameterizing a core greatly expands its versatility and

usefulness in a wider range of applications.

There are two different types of parameters: *static* parameters and *dynamic* parameters [29]. Static parameters must be set prior to the fabrication of the chip and often take the form of "generic" or statements within a VHDL description [54] of a soft-core component, or "parameter" statements in Verilog [55]. In contrast, dynamic parameters are those that can be set after the chip is fabricated, provided the chip has the facilities for supporting various parameter settings. Dynamic parameters are especially useful for parameterizing hard and firm-core components. For this research only static parameters of soft-core components will be considered.

Many of the parameters of a core often share interdependencies with one another. Assigning a value to one parameter will affect the choice of value assignments for other parameters, therefore the value assignments of interdependent parameters should be considered simultaneously. These interdependencies can either be *soft* or *hard* dependencies [29]. Soft interdependencies dictate that the value assignments of dependent parameters should be done at the same time in order to achieve optimal system performance, power consumption and IC area utilization. On the other hand, hard interdependencies require simultaneous parameter assignments if a valid and feasible design configuration is to be chosen.

Parameters of a soft-core hardware component are especially valuable to an embedded systems designer, since they give them the flexibility to customize the core as they desire in order to better fit the target application without having to manually rewrite large portions of the core's underlying HDL source code. Often, if the parameters of a core significantly affect its underlying structure, then it may be necessary to use a software-based HDL code generator program to customize the code automatically.

2.3 Soft-core Processors

A soft-core can be a description of virtually any digital hardware component, including a full microprocessor. For many of the reasons mentioned above, soft-core processors are a popular choice for embedded systems designers. Several examples of commercial and open-source soft-core processors will be discussed below [67].

2.3.1 Examples of Soft-core Processors

Altera Corporation [70] is an industry leader in programmable logic technology, specializing in FPGAs and other programmable logic devices. They are the makers of the Stratix and Cyclone Series of FPGAs [22]. They also provide numerous soft IP cores that are specifically designed to target their devices. Their flagship IP core is the Nios II soft-core processor [71], which is a general-purpose RISC processor that is optimized for embedded applications. This core consists of three processor variants that can be selected based on a designer's specific needs: the Nios II/f fast core, which is designed for maximum performance, the Nios II/e economy core, which is the smallest processor core, and the Nios II/s standard core, which is a tradeoff between the fast core and the economy core. These cores each feature their own set of configurable options, and all of them provide support for up to 256 custom instructions and interfacing to peripheral devices using the automatically-generated Avalon bus [3]. The Nios II processor is the successor to the original Nios [17], and features improvements over the original design that are aimed at providing better performance and FPGA area utilization. Designers working with the Nios II processor can use the Quartus II CAD tool suite [7] with System on a Programmable Chip (SOPC) Builder [19] to instantiate one or more processor cores into an embedded system design and connect them to other peripheral components, such as timers, universal asynchronous receiver/transmitters (UARTs) and memories.

MicroBlaze [73] is a 32-bit parameterized soft-core RISC processor provided by Xilinx Incorporated [40] that is targeted for Xilinx FPGAs and optimized for embedded applications. Its fixed features include 32-bit instructions, a 5-stage single-issue pipeline, a thirty-two general-purpose registers and a 32-bit address bus for data and instruction memories. The latest version of MicroBlaze (v5.00a at the time of this writing) also includes a large number of parameters, including an optional hardware barrel shifter, multiplier, divider, floating point unit (FPU), and others. Memory can reside on-chip or as an external peripheral. On-chip memory can be accessed by MicroBlaze using a Local Memory Bus (LMB), which provides single-cycle access to the memory. Also, the a general purpose interface known as the On-chip Peripheral Bus (OPB) can be used to interface MicroBlaze with memories and other peripheral components.

In addition to commercially available soft-core processors, there are numerous cores available from various open-source communities on the internet. Many of these cores can be downloaded, modified, and used in designs free of charge. Opencores.org [56] contains a large number of soft-core hardware components that have been developed by people all across the world. A number of open-source microprocessors are available, including the OpenRISC 1200 processor [2], which is a 32-bit RISC processor with a 5-stage pipeline and basic digital signal processing (DSP) functionality. Other examples of open-source soft-core processors include Qrisc [62], and the LEON3 processor by Gaisler Research [63].

2.4 FPGA Technology

Field programmable gate arrays are a specific class of programmable logic device that are designed to be programmed and reprogrammed to act like virtually any digital circuit that can be conceived, subject to logic capacity limitations. They are becoming an increasingly popular choice for embedded systems designers who want a

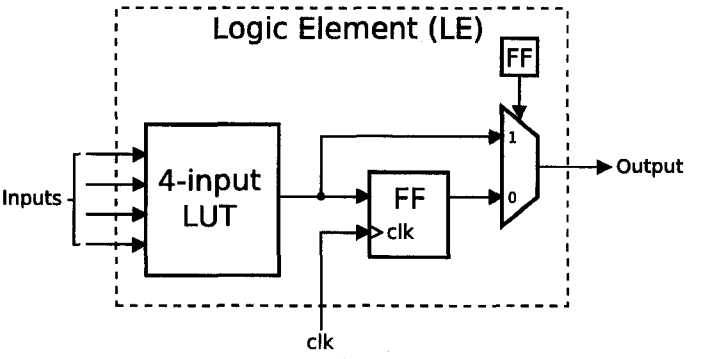


Figure 2.2: Schematic of a Generic FPGA Logic Element (LE) [42]

medium for prototyping and implementing soft-core hardware components. Soft-cores and FPGAs often go hand-in-hand. In fact, it is quite common for companies who manufacture FPGAs to also provide their own soft-cores that target their devices as well, with Altera Corporation [70] and Xilinx Incorporated [40] being the two largest and best-known examples.

The design of an FPGA differs between various manufacturers and also between different device families. However, in general, an FPGA is an IC chip that consists of an array of programmable blocks, often referred to as Logic Elements (LEs), which are connected to each other by a programmable interconnection network. A basic schematic diagram of an idealized LE is depicted in Figure 2.2 [42]. Although the LEs in the current generation of devices are much more sophisticated, this idealized LE does serve to illustrate the basic idea of how an FPGA works.

At the core of each LE is a block of programmable memory called a Lookup Table (LUT). The diagram in Figure 2.3 illustrates the essential functionality of an LUT. The LUT consists of an array of 1-bit memories connected to a multiplexed output pin. If the LUT has n inputs, then the memory array will have 2^n bits. This array can be programmed with the truth table of any possible n-input Boolean logic function, and the n multiplexer (MUX) select inputs decide which of the 2^n memory array bits appears at the LUT output. For example, to implement the logic function of a

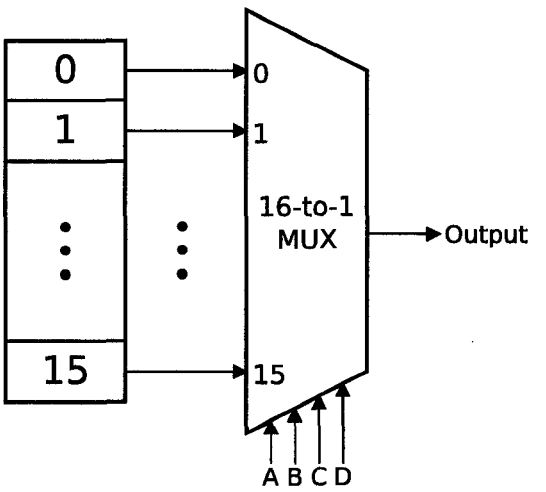


Figure 2.3: Schematic of a Lookup Table (LUT)

4-input "AND" gate, bits 0 to 14 are programmed with 0's and bit 15 is set to 1. Using this memory array, a total of 2^{2^n} logic functions can be implemented using an n-input LUT.

In order to make the creation of sequential logic circuits possible, the output of an LUT in a Logic Element is connected to a flip-flop (FF). Then the registered and unregistered outputs of the LE are both made available through a 2-to-1 MUX whose select line value is determined by the value stored in a second flip-flop, which is set by a bit-stream when the FPGA device is configured.

On an FPGA, a large number of LEs are connected together using a network of programmable interconnects, also known as "routing". There are many different types of routing architectures available, but they all have one thing in common: programmability. As depicted in Figure 2.4 [25], the LEs are grouped together into clusters called Logic Blocks (L in the figure) which are surrounded by horizontal and vertical wires on all sides, and special I/O blocks are arranged around the perimeter of the FPGA chip. The output of each Logic Block can be programmed to connect to a set of horizontal and vertical wire segments, and each wire segment can be

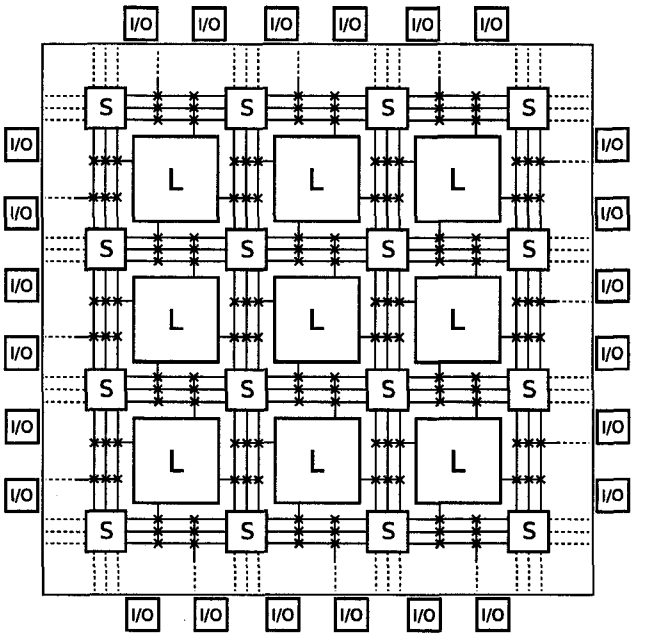


Figure 2.4: Generic FPGA Routing Architecture (adapted from [25])

programmed to connect to other wire segments through a Switch Block (S).

By programming both the contents of the Logic Blocks on the FPGA as well as the routing connecting the blocks together, a designer can implement a virtually limitless number of digital hardware circuits. Using an FPGA as an implementation medium offers the distinct advantage of flexibility—if the designer needs to change the circuit on the FPGA, it is merely a matter of reprogramming it. However, this flexibility does come with a cost. Circuits that are implemented on an FPGA will generally take up more chip area, consume more power, and run slower than they would if they were implemented using an ASIC technology. Nevertheless, since FPGAs are a relatively new technology, research is being conducted that aims at closing the performance gap between FPGA and ASIC technologies.

The first FPGAs that were introduced featured relatively small logic capacities and offered only a few basic features. However, as the technology improved, the devices were able to hold increasingly larger circuits and began including a number of

more advanced features. Currently, on top of an ever-increasing number of LEs and I/O pins, the latest cutting-edge devices also feature large amounts of on-chip memory and other special-purpose blocks such as dedicated multipliers. In addition to having configurable elements, some FPGAs also feature one or more hard-core processors built right into the device. For instance, the Xilinx Virtex-4 family of FPGAs [57] features two built-in hard-core embedded IBM PowerPC^{TM} 405 processors [16], which can be used in any number of embedded applications. These new features serve to substantially improve the performance, area utilization, and power consumption of systems implemented on FPGAs.

The Altera Stratix EP1S40F780C5 FPGA has been selected as the target device used for this research, therefore a brief description of the Stratix architecture [8] is necessary. All of the devices in the Stratix family contain six different types of logic resources: Logic Array Blocks (LABs), M512, M4K, and M-RAM memory blocks, DSP blocks, and I/O Elements (IOEs). LABs are blocks which consist of 10 LEs each and are used to implement user-defined logic functions. The M512 blocks, the smallest memory blocks, each contain 512 bits of memory, plus parity bits, and can be used to provide single-port or simple dual-port memory operation. The M4K blocks are larger than the M512s and feature 4 kilobits of memory each, plus parity. These blocks can be used in single-port, simple dual-port or true dual-port mode. The M-RAM blocks are significantly larger than both the M512 and M4K blocks, containing 512 kilobits of memory each (plus parity). Like the M4K blocks, the M-RAM blocks can be used in single-port, simple dual-port or true dual-port mode. The DSP blocks are special-purpose resources on the device, and can each be used to implement eight 9×9-bit multipliers, four 18×18-bit multipliers, or one 36×36-bit multiplier. Lastly, the IOE elements are connected to the Stratix device pins, and support a number of different I/O standards. All of these resources are arranged in a 2-dimensional row- and column-based structure on the Stratix device. The EP1S40F780C5 device, one

of several in the Stratix family, contains exactly 4,125 LABs (or 41,250 LEs), 384 M512s, 183 M4Ks, 4 M-RAMs (for a total of 3,423,744 memory bits), 14 DSP blocks (for a total of 112 9 × 9-bit multipliers, 56 18 × 18-bit multipliers or 14 36 × 36-bit multipliers) and 616 I/O pins [8].

2.5 Design Space Exploration (DSE)

The term *design space* in the context of digital embedded systems generally refers to the set of all possible system designs; that is, the complete collection of all possible digital hardware and software configurations that will achieve the functionality required to perform the system's intended tasks. When dealing with a complex system like those commonly encountered in the area of embedded systems, the design space is extremely vast and contains a large number of configurations that are sub-optimal for any given application. Therefore, it is essential to the success of any embedded systems design project that the design space be traversed to determine the system design configuration that best suits the intended application. This is, in essence, the main goal of design space exploration.

2.5.1 Multi-objective Optimization

The problem of DSE is essentially a multi-objective optimization problem in which design configurations are chosen so that they provide the best balance between a set of competing *objectives*. Most commonly, these objectives include minimizing IC chip area, reducing power consumption and maximizing system performance. These objectives cannot be optimized independently, since improving one objective will almost always mean sacrificing another. Since several competing objectives are being optimized at once, there is seldom one single "optimal" design configuration for any given multi-objective optimization problem. Instead, there exists a set of configura-

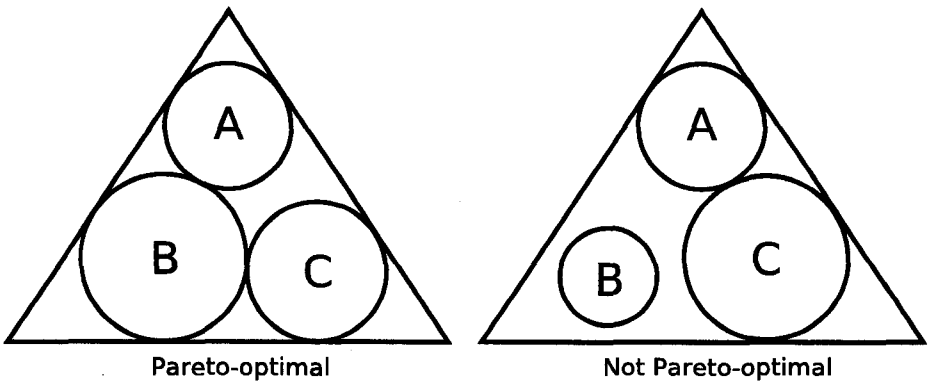


Figure 2.5: The Concept of Pareto-optimality Illustrated (adapted from [60])

tions known as the *Pareto-optimal set* that is a subset of the complete design space. A configuration is said to be Pareto-optimal if you cannot improve one objective without sacrificing another. The concept of Pareto-optimality is illustrated with a geometric example in Figure 2.5 (adapted from [60]).

Suppose in this hypothetical multi-objective optimization problem [60] that the goal was to simultaneously maximize the areas of circles A, B and C within the area of the triangle, with the constraint that the circles must not overlap or pass the boundary of the sides of the triangle. In this case there cannot be just one solution, but rather a multitude. The configuration on the left-hand side of the figure is an example of a Pareto-optimal solution, because you cannot increase the area of any of the circles without decreasing the area of the other two. In contrast, the triangle on the right-hand side is a non Pareto-optimal configuration, because the area of circle B can be increased without affecting the areas of circles A or C.

Finding the Pareto-optimal set drastically reduces the size of the design space by eliminating all sub-optimal configurations, allowing the designer to select a single design configuration from the Pareto-optimal set that is well-suited to the intended application. If one were to plot one objective against another on a graph, the result would be something that looks similar to the graph shown in Figure 2.6.

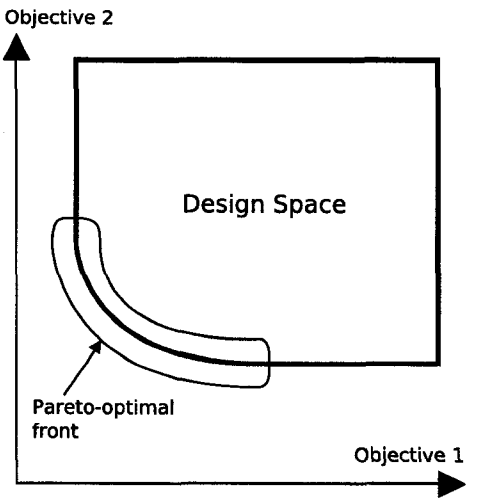


Figure 2.6: Illustration of a 2-Dimensional Design Space

All of the possible design configurations are located within the Design Space region of the graph. Outside of this region, no design configurations can exist. The boundary of the Design Space region at the lower left-hand side nearest to the origin of the graph is referred to as the *Pareto-optimal front.* Design configurations that lie along this boundary line represent the very best configurations of the design space in terms of the objectives across which the graph is plotted. It is this set of configurations that is being sought when Pareto-based design space exploration is performed. Conversely, it is clear that there is also a large space of sub-optimal configurations that exists within the boundaries of the Design Space. Ideally, these configurations can be safely eliminated, or "pruned", from consideration, thus drastically reducing the size of the space that must be explored.

2.5.2 DSE Using Parameterized Cores

In the context of platform-based embedded systems design involving parameterized cores, the goal of DSE is to prune the design space in search of a suitable combination of parameter values for the hardware platform that provides a good balance between

the each of the objectives while satisfying the design constraints imposed by the requirements of a particular application. Many approaches for handling this task have been proposed.

The simplest and most straightforward approach to design space pruning is to exhaustively visit and evaluate each and every combination of parameters in the design space and remove the worst configurations from consideration. Some research has been done on the application of this approach [31, 28]. The obvious drawback to this approach is the fact that it rapidly becomes infeasible to evaluate every configuration as the design space grows. Therefore, this approach is seldom practical on its own, except for very small systems with relatively few parameters, as concluded by Givargis *et al* [28].

Since the exhaustive approach is so often infeasible, there are a multitude of approaches that have been developed that help to automate the process of DSE of embedded hardware platforms. A good summary of some of these approaches, such as the use of Architectural Description Languages (ADLs), can be found in the literature [32, 66, 53]. For this work one particular approach will be examined: the use of genetic-based algorithms with parameterized cores, as will be discussed in the following sections.

Approaches Based on Genetic Algorithms

The idea of a genetic algorithm (GA, sometimes referred to an an evolutionary algorithm or EA) was first proposed by Holland in 1975 [37] and was used primarily in the field of artificial intelligence. However, it was later applied in a wider range of applications and was found to be very effective in solving multi-objective optimization problems, including the problems posed by DSE using parameterized cores. Genetic algorithms are a class of optimization problems that gain their inspiration from the field of biological sciences. There are many variations of the algorithm, but they

all work by attempting to emulate the biological process of natural selection, where stronger members of a population survive and pass on their genes while the weaker members gradually die off. A good overview of genetic and evolutionary algorithms for multi-objective optimization can be found in the literature [26, 14].

The genetic algorithm starts with and maintains a *set* of design configurations rather than just a single configuration. This set of configurations is called the *population* and has a fixed size N. Each member of the population represents one unique design configuration, and is referred to as a *chromosome.* Each chromosome in the population is made up of a string of symbols that represent the system's parameters. A symbol in the chromosome is called a *gene.*

During an iteration or *generation* of the algorithm, each chromosome is evaluated according to its "fitness", where fitness is a measure of how well the configuration meets the problem objectives. Pairs of chromosomes are selected to become *parents* of *offspring* through *reproduction.* During this process, features from both parents are combined to form the offspring using the genetic operators: *crossover and mutation.* During crossover, the genes of the parents are combined to form a new chromosome, called the *offspring.* The mutation operator produces random changes in a single chromosome without producing offspring.

A new generation of chromosomes is formed through the production of numerous offspring from the set of parents. Since the population has a fixed size, N chromosomes of the total number of parent and offspring chromosomes are selected to survive to the next generation; the rest are discarded. The N surviving chromosomes can either be selected randomly from the full set, or selected based on fitness. After several generations of the algorithm, the population of configurations should converge toward an optimal configuration set.

2.6 Closely Related Work

Yiannacouras [75] developed SPREE, the Soft Processor Rapid Exploration Environment, in order to facilitate the exploration of the design space for soft-core processors targeted for implementation on an FPGA. SPREE consists of a hardware Component Library and an RTL Generator. The RTL Generator fetches hardware components from the library and builds a datapath according to a special Architectural Description which is given to the RTL Generator as an input. The RTL Generator then creates the corresponding control logic, either pipelined or unpipelined, yielding a complete soft-core processor. The generated processors were based on the MIPS-I [35] instruction set architecture. The SPREE system was used to investigate several soft-core processor architectural alternatives including hardware versus software multiplication, different shifter implementations, varying pipeline depths, as well as some other interesting architectural tradeoffs. One major difference between the SPREE system and this present work is the exploration methodology. The SPREE system uses an exhaustive exploration strategy, in which the user must manually explore the various design tradeoffs by developing different architectural descriptions for each processor variant. In contrast, this research applies an automated approach based on a genetic algorithm to explore the design space of a heavily parameterized soft-core description.

The Platune system [30] is an environment that allows an embedded system designer to tune the parameters of a parameterized hardware platform. The system provides a set of simulation and power models for the components of a parameterized system consisting of a MIPS R3000 processor [35] with instruction and data caches, on-chip memory, and a set of interconnecting buses. The simulation models are used to compute the execution time of a specific program running on the processor and to gather information on the power consumption of the system. Platune also features a design space exploration framework that uses a parameter interdependency model

and an exhaustive approach to determine the Pareto-optimal set of configurations. A graph of parameter interdependencies is created, and interdependent parameters are gathered together into clusters. Then an exhaustive exploration approach is applied to each cluster to determine its local Pareto-optimal set. Once each cluster has been searched, pairs of clusters are merged together, and the exhaustive search is applied to the merged clusters. This process continues until a single cluster remains, and the Pareto-optimal set of configurations is determined. In contrast to the Platune system, this work features a GA-based approach to search the design space for the Pareto-optimal set. Additionally, the Platune system is directed specifically toward the use of the MIPS R3000 processor model, which is not designed specifically for FPGA implementation. In this research, a general framework for the design space exploration of any parameterized core has been established, with emphasis on soft-cores targeted for implementation on an FPGA.

Palesi and Givargis [59] present an approach to explore the design space of heavily parameterized systems using a genetic algorithm, namely the Strength Pareto Evolutionary Algorithm 2 (SPEA2) [77]. The approach combines the parameter dependency clustering and exhaustive search method used by Platune with the genetic-based SPEA2 algorithm to reduce the time needed to find the Pareto-optimal set of configurations. Their results indicate that an approximated Pareto-optimal set that is within 1% of the actual set can be obtained using the combined approach while reducing the amount of simulation time required to determine the set by 80%. Ascia *et al* [10] later use the SPEA2 algorithm directly with the Platune system to search for the Pareto-optimal set of configurations. These researchers provide some useful conclusions about the use of genetic-based algorithms in platform-based design problems. However, they do not focus specifically on the exploration of the design space of soft IP cores targeted for FPGAs.

The PEAS-III system by M. Itoh *et al* [44] is a System-level design environment

that enables designers to quickly explore the design space of pipelined embedded processors. The system is based on the micro-operation description of instructions, which allows designers to concentrate on the design of a processor's instruction set. A pipelined processor is built from a series of pipeline stage models. Each stage model represents a single stage in the pipeline and consists of pipeline resources such as ALUs and other functional units, inter-stage pipeline registers, a stage controller and the interconnections between them. The PEAS-III system creates a datapath and associated control logic by cascading the stage models in series. Two VHDL descriptions of the processor are generated by the system: a non-synthesizable model used purely for simulation and a version intended for synthesis. In order to evaluate the effectiveness of PEAS-III, several processors were built using the system, including a MIPS R3000 processor, a DLX processor [36], and a simple RISC controller. The PEAS-III system does not utilize any form of automated design space exploration, thus distinguishing it from this present research.

2.7 Summary

The relevant background material and related previous work was presented in this chapter. First, Intellectual Property cores, and specifically soft IP cores, were introduced. Next, the parameterization of IP cores was discussed, followed by a presentation of some examples of soft-core processors that are available from industrial vendors and open-source communities. Then the basic concepts of Field Programmable Gate Array technology were introduced, summing up with a discussion of the architecture of the Altera Stratix FPGA. The ideas of design space exploration and multi-objective optimization were presented next, leading into the main thrust of this chapter: the DSE of parameterized cores using genetic algorithms. Finally, this chapter concluded with a discussion of some of the previous research that is closely related to the work presented in this thesis. In Chapter 3, the results of a preliminary case study involv-

ing the DSE of a parameterized soft-core processor using a genetic-based approach are presented.

Chapter 3

Design Space Exploration of Embedded CPU Cores for FPGAs

Any given parameterized soft-core component of even modest complexity may have numerous parameters, and each of those parameters may have a large number of possible values to choose from. As a result, the total number of possible combinations of these parameter values may be exceedingly large, often into the thousands, millions or more. In addition, each parameter can potentially have an impact on the cost and performance of the resulting system. Since the set of possible configurations can be so large, one major question that arises is this: how does a designer go about selecting a combination of parameter values that yields a system that has the lowest cost and the highest performance for a particular application? It is this main question that the study of DSE using parameterized cores sets out to address.

In this chapter, the results of a preliminary case study are presented. This study was conducted in order to investigate the feasibility of applying a genetic algorithm-

based approach to a parameterized core with a sizable design space to determine an approximation of its Pareto-optimal set [9]. The core that was chosen was Altera's Nios [17] soft-core processor and the Simple Evolutionary Algorithm for Multiobjective Optimization (SEAMO) [69] was selected as the engine for exploration.

3.1 Problem Statement

The problem of DSE using parameterized cores is as follows [10]: a parameterized system has a set of P parameters, p_1, p_2, ..., p_P. Each of these parameters can be assigned a *value* from a finite ordered set of possible values, V_i, $i \in \{1, 2, \ldots, P\}$. The design space, D is defined as the Cartesian product of all of the sets of possible values:

$$D = V_1 \times V_2 \times \ldots \times V_P \tag{3.1}$$

Assigning particular values to all parameters of each component of the system produces one design *configuration*. If dependencies exist between the parameters of the system, then not every configuration in the design space will be feasible and physically realizable. Subsequently, the total number of configurations in the design space is the product of the cardinalities of each of the sets of values for each parameter:

$$|D| = |V_1| \times |V_2| \times \ldots \times |V_P| \tag{3.2}$$

Every configuration has a set of K *objective functions*, $F_k(p_1, p_2, \ldots, p_P)$, where $k \in \{1, 2, \ldots, K\}$, which are measures of how well or how poorly the configuration meets the objectives of chip area minimization, power consumption reduction, performance maximization, etc. The set of all possible configurations makes up the design space for the system.

The objective of DSE in this case is to determine the Pareto-optimal set of configurations from the complete design space. However, it is clear that as the number of components, parameters and values per parameter grow larger, the design space expands enormously. Therefore, to achieve the goal of finding the Pareto-optimal set, two related tasks need to be performed. The first is to *prune* the design space down to a manageable size by eliminating all sub-optimal configurations. The second is to *evaluate* design configurations by estimating their area usage, power consumption, performance values, etc. to see how well each configuration meets the objectives.

3.2 The Altera Nios Soft-core Processor

The Nios processor is a "pipelined general-purpose RISC microprocessor" [17] designed by Altera Corporation [70]. It is a flexible processing core that features numerous parameters as well as support for custom instructions. These parameters are summarized in Table 3.1.

The width of the datapath is configurable, supporting either 16 or 32-bit variants [6, 5] . However, both the 16 and 32-bit architectures use a 16-bit instruction set. The datapath has a five-stage pipeline, and a large, windowed register file, which can be configured to include either 128, 256, or 512 registers.

The ALU is configurable and supports the inclusion of up to five custom instructions integrated directly into the unit. Five user opcodes are provided so that software can make use of these custom instructions directly. Nios also provides a number of options for integer multiplication support. The 32-bit Nios variant can optionally be configured to include a full 16 $\times$ 16-bit integer multiplier (MUL instruction), a partial hardware multiplier (MSTEP instruction), or no hardware multiplier at all (multiplication is done using software routines). The designer is also given the option of implementing the processor's instruction decoder as a Read Only Memory (ROM) unit using on-chip memory resources or directly in logic using the FPGA's logic ele-

Table 3.1: Altera Nios Hardware Parameters

Parameter	**Possible Values**
Datapath width	16 or 32 bits
Instruction decoder (p_1)	Logic Elements or ROM
Register file size (p_2)	128, 256 or 512 registers
WVALID register (p_3)	Read-only or read/write
Instruction cache size (p_4)	Off, 1, 2, 4, 8 or 16 kB
Data cache size (p_5)	Off, 1, 2, 4, 8, or 16 kB
Integer multiplication (p_6)	Software, MSTEP, MUL
Pipeline optimization (p_7)	More stalls/Fewer LEs, Fewer stalls/More LEs
Support RLC/RRC (p_8)	Yes or no
Support interrupts/traps (p_9)	Yes or no
Support OCI module (p_{10})	Yes or no

ments. The core provides the designer with the option of optimizing the pipeline for fewer stalls at the expense of requiring additional LEs. Finally, the WVALID register, which stores the high and low limits of the register file window, can be set to either read-only or read/write.

Nios features a Harvard memory architecture, with separate instruction and data bus masters. Memory can reside either on-chip or as an off-chip peripheral. Direct-mapped instruction and data cache memories of various sizes can be optionally included in the 32-bit Nios variant. The Nios processor can connect to any number of on-chip and off-chip peripherals using the automatically-generated Avalon bus [3]. Nios also has an optional on-chip instrumentation (OCI) debug module [17] which connects directly to signals internal to the Nios processor that allows the user to perform various debug operations. Finally, Nios provides optional support for hardware and software interrupts and internal exceptions, as well as left and right rotate-through-

carry (RLC and RRC) instructions.

Some of these parameters share hard interdependencies with one another. For example, caches and hardware multiplication can only be included on the 32-bit Nios variant. Similarly, the OCI debug module can only be added if support for interrupts and traps has been enabled. Considering just the Nios core with the parameters mentioned above, this gives us a total of exactly 10,512 different feasible Nios configurations, with 10,368 configurations for the 32-bit Nios, and 144 configurations for the 16-bit Nios. This represents a substantial design space, which should be traversed in order to effectively select a suitable configuration for the intended application.

3.2.1 Target System

For this case study, a simple Nios system was created using Altera's Quartus II Version 4.2 and SOPC Builder [19] software packages. The target FPGA and associated hardware was the Altera Stratix EP1S40F780C5 FPGA [8] present on the Nios Development Board Stratix Professional Edition [4]. The system consisted of a 32-bit Nios core along with an Avalon Tri-state Bridge [3] used to connect the processor to an 8 MB off-chip memory located on the development board. The decision was made to restrict the case study to the 32-bit Nios variant in order to avoid complicating it with parameter interdependencies, and also because the 32-bit variant represents the significant majority of the design space.

3.3 The Simple Evolutionary Algorithm for Multi-Objective Optimization (SEAMO)

The algorithm chosen for this exploration was the Simple Evolutionary Algorithm for Multi-objective Optimization (SEAMO) proposed by Valenzuela [69]. This algorithm was chosen because of its relative simplicity and easy applicability to this particular

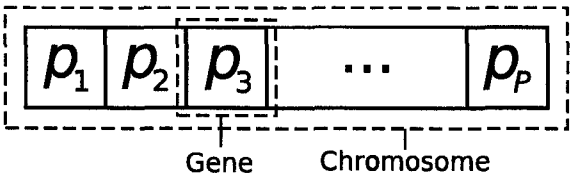


Figure 3.1: The Chromosome Used in the SEAMO Algorithm

case study. This algorithm was originally applied to the 0-1 multiple knapsack problem, a well known member of the class of NP-hard problems. However, the setup of the algorithm was modified to suit the purposes of this case study. The algorithm as it was applied to this case study is briefly summarized as follows: Each Nios parameter is represented as a *gene*—a discrete variable with a finite set of possible integer values. The integer values correspond to the possible parameter values of the Nios processor (for example parameter p_2 is a variable which represents the register file size, where 1 is a register file size of 128, 2 for 256 and 3 for 512 registers). The chromosome is given as a string of these discrete variables—genes—as illustrated in Figure 3.1.

The population is made up of a collection of N of these chromosomes. The first step of the algorithm is to generate an initial population of chromosomes randomly. After this is done, each chromosome is evaluated individually one-by-one in terms of its objectives—in this case, FPGA area utilization and critical path delay. The method used for determining these values will be discussed below. The estimated objective values for area and delay are stored separately in an "objectives vector" for each chromosome. Once every chromosome has been evaluated, the "best-so-far" values for area and for delay are recorded. This is illustrated in Figure 3.2.

After this has been done, the algorithm proceeds through every member of the population in order one-by-one. Each member is paired with another, randomly selected member from the population and is given a chance to produce an offspring using the crossover operator. The crossover operator selects a cut-point at random

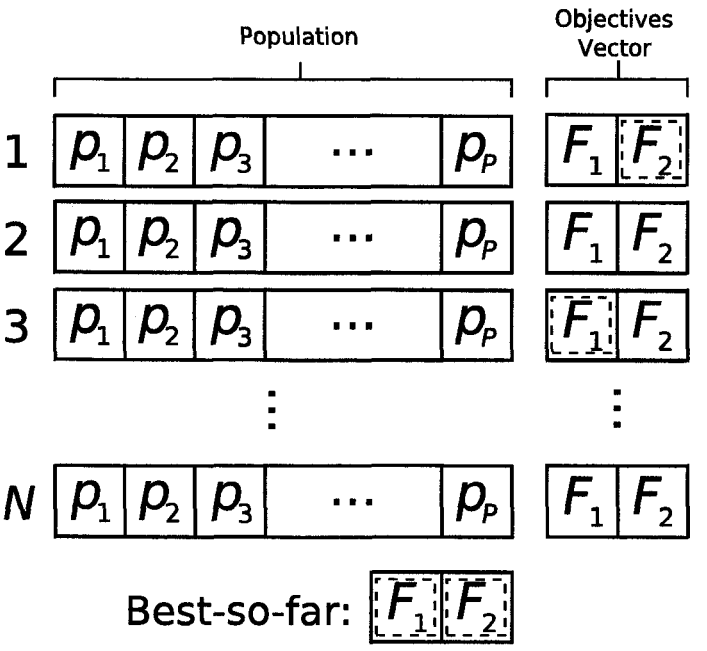
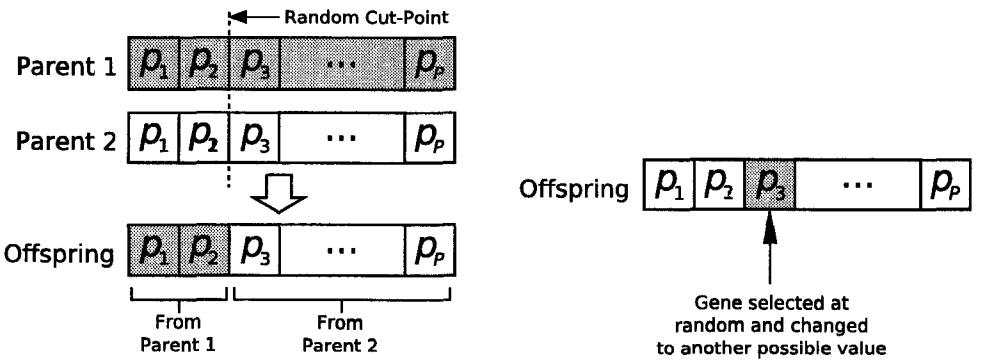


Figure 3.2: A Population of Chromosomes

and combines the left half of one parent with the right half of the other. Which half comes from which parent is also decided randomly. The crossover operator is usually only applied a certain percentage of the time, according to the specified *crossover rate*, r_c. The offspring is then mutated, which involves randomly selecting one gene within the offspring chromosome and changing it to another possible value, again selected at random. Normally, only a certain percentage of the offspring produced are mutated, the proportion of which is determined by the *mutation rate*, r_m. Crossover and mutation are illustrated in Figure 3.3.

At this point, the mutated offspring is evaluated in terms of its objectives and replaces one of the parents if one of several conditions are met. If one of the parents is dominated by the offspring (i.e. is inferior to the offspring across all of the objectives), then the dominated parent is replaced by the offspring. If the offspring improves on one or more of the best-so-far values, then the offspring replaces one of its parents



(a) Crossover (b) Mutation

Figure 3.3: The Crossover and Mutation Operators

(the parent to be replaced is randomly chosen). Finally, if an identical copy of the offspring already exists within the population, then the offspring is discarded.

After the algorithm has visited each member of the population and paired it with another to produce offspring, one *generation* has passed. The algorithm will generally iterate through a number of generations before the population converges toward an approximation of the Pareto-optimal set of configurations. The size of the population, N, and the number of generations, G, the crossover rate r_c, and the mutation rate, r_m, constitute the parameters of the algorithm and appropriate values for them are often determined through experimentation.

3.3.1 Evaluation of Configurations: The Objective Functions

Any multi-objective DSE procedure requires that individual configurations be evaluated in terms of their objectives. The simplest and most straightforward approach is to synthesize each and every possible configuration using a CAD tool and store the generated area and delay values in a library. When values for a particular configuration are needed, they are simply fetched from the library. However, the obvious

drawback of this approach is that, for larger systems, data for an exceedingly large number of configurations would have to be stored. Another option is to synthesize configurations as they are generated by the DSE algorithm; however this is often impractical due to the long run-times that are usually required to synthesize a design. A third possible option is to model the area and delay characteristics mathematically, by trying to establish equations that can be used to predict these numbers given the values for all of the parameters.

The approach used in this case study is a compromise between the first and third of these approaches. For this case study, the objective estimation approach proposed by Jha and Dutt [46] was utilized. This approach involves establishing fast and accurate equations for estimating area and critical path delay delay using least-squares regression analysis on actual synthesis data for a number of representative configurations. These equations relate the area and delay objectives to the P (the total number of parameters; in this case P=10) different parameter variables, p_1, p_2, ..., p_P and have the general form:

$$F_k(p_1, p_2, \ldots, p_P) = a_{0,k} + \sum_{i=1}^{P}(a_{i,k} \cdot f_{i,k}(p_i)) \tag{3.3}$$

Where $a_{0,k}, a_{1,k}, \ldots, a_{P,k}$ are the constant coefficients determined using regression analysis. The form of functions $f_{i,k}(p_i)$ can be determined by studying the relationships between the parameters p_i and the area and delay values. For this case study, the area of the circuit is given in terms of *equivalent LEs* used on the Stratix EP1S40F780C5 FPGA. This value is determined by summing the number of LEs, DSP blocks (9 × 9-bit), M512s, M4Ks and M-RAM memory blocks multiplied by their relative sizes on the Stratix FPGA. These values are given in Table 3.2 (obtained from [75]). *Delay* is a measure of the critical path delay of the circuit in nanoseconds (ns) reported by the Quartus II Timing Analyzer [7].

For this approach, a "parameter sweep" was performed, in which each of of the

Table 3.2: Relative Sizes of Stratix Components (from [75])

Stratix Block	Relative Size
Logic Element (LE)	1
DSP Block (9 × 9 bit)	23.37
M512 RAM	20.5
M4K RAM	47.8
M-RAM	1550.3

Table 3.3: Regression Coefficients – Nios Processor

Parameter	i	$a_{i,1}$ (Area)	$a_{i,2}$ (Delay)	$f_{i,1}(p_i)$ (Area)	$f_{i,2}(p_i)$ (Delay)
–	0	77.3	13.6	-	-
Instruction Decoder	1	31.9	-0.205	p_1	p_1
Register File Size	2	264.3	-0.205	p_2	$\log_{10}(p_2)$
WVALID Register	3	71.9	-0.092	p_3	p_3
Instruction Cache Size	4	62.6	6.22	${p_4}^2$	$\log_{10}(p_4)$
Data Cache Size	5	53.7	2.19	${p_5}^2$	$\log_{10}(p_5)$
Integer Multiplication	6	79.5	-0.16	p_6	$\log_{10}(p_6)$
Pipeline Optimization	7	87.9	0.44	p_7	p_7
Support RLC/RRC	8	74.9	-0.76	p_8	p_8
Support Interrupts/traps	9	202.1	-1.22	p_9	p_9
Support OCI Module	10	476.6	1.21	p_{10}	p_{10}

10 Nios parameters was varied across its entire range of values while the others were held constant. This resulted in 47 different Nios "sweep" configurations, each of which was generated using SOPC Builder and synthesized with Quartus II Version 5.0 using the default synthesis, fitter and timing analysis settings. After synthesis, the FPGA resource utilization and delay data were collected from the reports generated by Quartus II. Using these data as a basis, the $a_{i,k}$ coefficients in equation (3.3) were determined using least-squares regression techniques. These are listed in Table 3.3 along with the function forms of $f_{i,k}(p_i)$ used in the equations.

3.4 Experimental Results

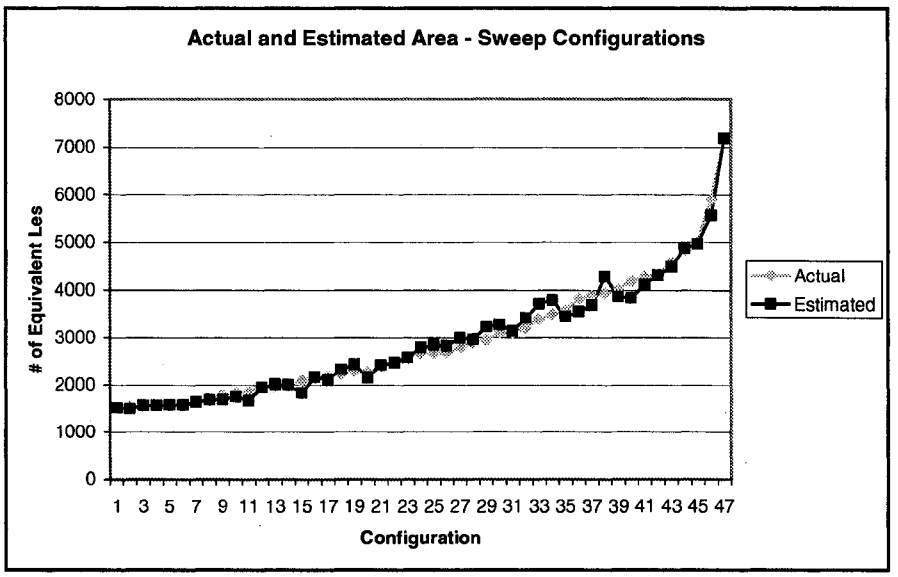
3.4.1 Testing of Objective Functions

In order to test the mathematical objective functions for accuracy, the actual and estimated area and delay values were first compared for the 47 Nios sweep configurations that were used to establish the objective estimation equations. See Figure 3.4 for graphical comparisons of these two sets of data. As can be seen in the figure, the estimated values for both area and delay tracked the actual values quite closely. The average percentage error was 3.93% for the area estimates, and 4.75% for the delay estimates.

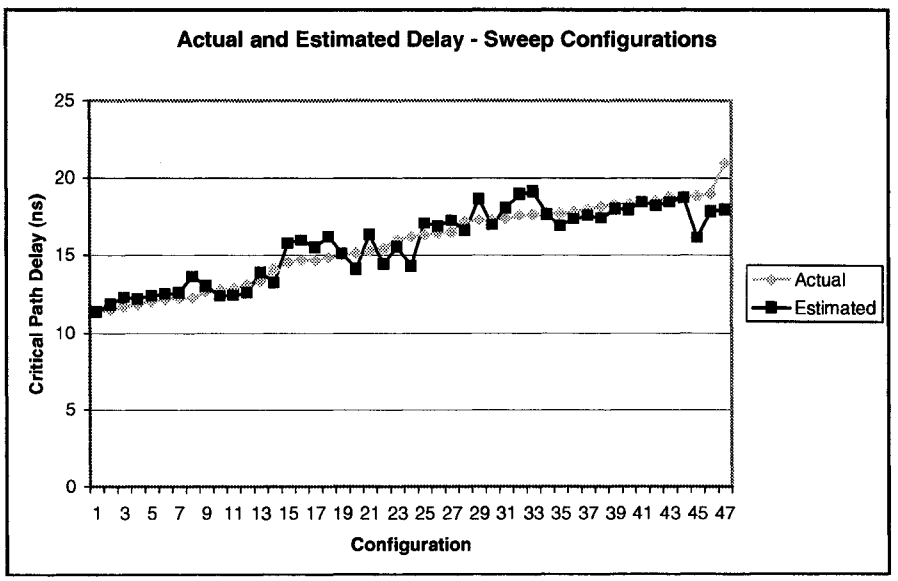
To test the accuracy of the equations for any arbitrary Nios configuration, 20 unique configurations were generated randomly and then synthesized using Quartus II. Again, the area and delay data were collected and compared against the area and delay values that were predicted by the established objective functions. Graphs of these comparisons are shown in Figure 3.5. For these 20 test cases, it was found that, on average, the estimated area values were within 7.22% of the actual values and the delay values were within an average of 7.58%.

3.4.2 Experimental Determination of Algorithm Parameters

A number of experiments were performed with the SEAMO algorithm in order to determine suitable values for the algorithm parameters—the population size, N, and the number of generations, G. To this end, the SEAMO algorithm was run with different population sizes for 50 generations each. It was determined that the algorithm converges to within 10% of its final value in about 20 generations, and that a population size of 50 provided a good diversity of configurations. For this simple case study, both the crossover and mutation rates were assumed to be 1.0, indicating that crossover happens for every pair of parent chromosomes and that and every offspring

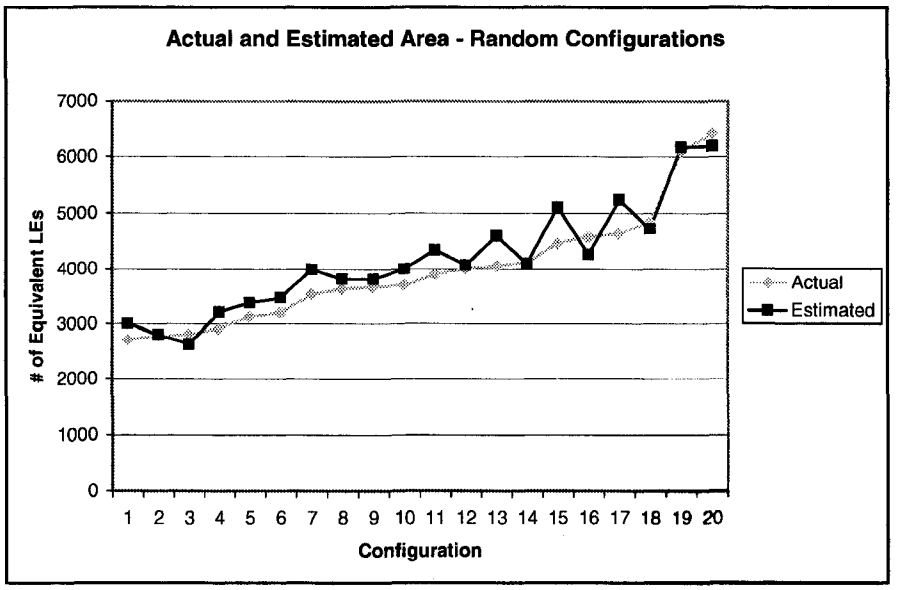


(a) Area

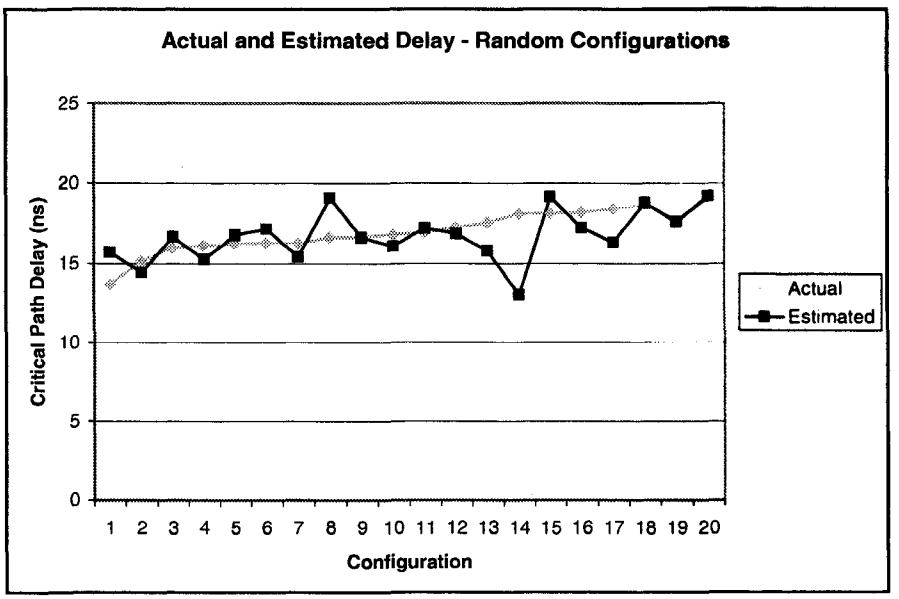


(b) Delay

Figure 3.4: Actual and Estimated Values for Nios Sweep Configurations



(a) Area



(b) Delay

Figure 3.5: Actual and Estimated Values for Nios Random Configurations

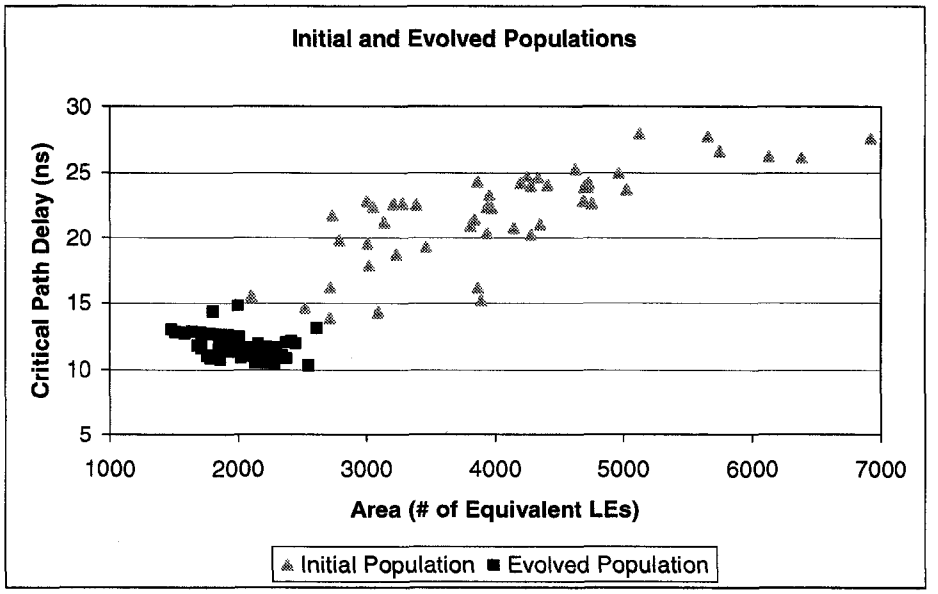


Figure 3.6: Initial and Evolved Populations (Using Estimated Values)

is mutated.

3.4.3 Comparison of SEAMO Results Vs. Randomly Generated Configurations

For this experimental case study, an initial random population of 50 chromosomes was first generated and the estimated area and delay values for each individual in the population were gathered. Then the SEAMO algorithm was run on this initial population for 20 generations, and the data for resulting population was collected. For comparison, the results are shown graphically in Figure 3.6 along with the initial data points (using the estimated objective values).

The graph shows significant overall improvement of the entire population after 20 generations over and above the initial population. The data points for the "evolved" population crowd around close to the origin of the graph. These points approximate the Pareto-optimal front, the boundary of the design space beyond which no con-

figurations can exist. The configurations on this boundary are non-dominated and represent the set of the best configurations from the design space in terms of the area and delay objectives examined.

3.5 Conclusions Drawn from this Case Study

One point that was observed during the course of experimentation was that the types of Nios configurations that tended to exist within the approximated Pareto-optimal were the more minimal configurations. All three types of multipliers were present within the configurations of this set, as were the rest of the options. However, these configurations always excluded any type of caching and often had smaller register file sizes. Also, the SEAMO algorithm almost always eliminated the OCI Debug Module.

It has been concluded from these experiments that caches, larger register files, and the OCI Debug Module add significant area and delay to the Nios processor, therefore they were eliminated by the SEAMO algorithm. However, the benefit derived from including these components in the Nios processor is not always accurately reflected in the delay and area measures that are utilized by the algorithm. Therefore, the decision about whether or not to include these components is application-specific in nature.

The approach that was utilized during this case study can be of greatest use when deriving application-specific hardware components from parameterized cores. The designer may require certain features for their intended application, therefore they could constrain those parameters to their suitable values. The rest of the parameters may be "free" and a genetic-based approach can be used to determine appropriate values for them.

3.6 Summary

In this chapter the results of a preliminary DSE case study were presented. The SEAMO algorithm was applied to the 10 parameters of the Altera Nios soft-core processor and an approximation of the Pareto-optimal set of configurations was determined. Then the "evolved" population generated by the algorithm was compared with an initial, randomly generated population of configurations and a substantial improvement was seen in the evolved population in terms of both FPGA area utilization and critical path delay. In the next chapter, the design and implementation of a CAD tool that applies the genetic-based approach presented in this chapter will be discussed in detail.

Chapter 4

SCBuild - A CAD Tool for the DSE of Embedded CPU Cores

In this chapter, the design and implementation of SCBuild (Soft-Core Build) is presented. SCBuild is a software-based CAD tool that has been developed over the course of this research in order to facilitate the rapid exploration of the design space of parameterized soft-core hardware components in general, and embedded processor cores in particular. This tool accepts a *template description* of a parameterized core as an input, which is essentially a blueprint for the core that contains information on its parameters and tells the tool how to generate HDL code for the core given certain parameter values. It uses the automated DSE approach that was presented in Chapter 3 to prune the design space of the parameterized core and determine an approximation of its Pareto-optimal set of configurations. Once one configuration from that set is selected and values for each of the core's parameters have been chosen, SCBuild generates a structural VHDL description of the core with the selected

features by instantiating components from a library of synthesizable VHDL components. If SCBuild is running on a machine that has a version of Altera's Quartus II [18] software installed, it is also able to generate a Tool Command Language (Tcl) [72] script file and invoke Quartus to run the script, creating a new Quartus Project File (.qpf), compiling the generated VHDL code and saving the synthesis results in a text file for later processing. Each of these steps will be described in detail in the sections that follow.

4.1 Major Problems Addressed by SCBuild

During the development of SCBuild, several major design problems needed to be addressed. Each of these problems has been handled by some aspect of the SCBuild software tool, as will be discussed in the following sections.

4.1.1 Representing a Parameterized Core in Software

In order for a piece of software to be able to work with parameterized cores, a precise description or representation of these cores which the software understands needs to be available. In other words, the problem that the software is trying to solve needs to be modeled or described using some type of "language". The software needs to be able to read this representation and work with it so that it can map it onto software data structures, manipulate it internally, and ultimately translate it into a viable output, which in this case is structural VHDL code. This representation should be as simple as possible and should provide just enough of the required information for the software to do its work. It should then be able to figure out the rest of the details on its own.

Ideally, when a parameterized core is viewed at its highest level of abstraction, nothing should been seen except a black box with a set of parameters that allow

users to control what features the core has without having to worry about how those features are implemented inside the box. One of the main goals of SCBuild is to hide the implementation details inside the box so that all the end-users of the software have to concern themselves with are the core's parameters. However, from the perspective of the software, more information about the core needs to be known in order for it to successfully generate an HDL description. One of the main questions that was considered during the design of SCBuild is this: how much information should the software need to be provided with ahead of time, and how much of it should it be able to figure out on its own?

These two things represent a major design tradeoff. At one extreme, the software is provided with an input description that precisely defines and describes every detail of the core, in which case there is not much left for the software to do. At the other extreme, the software is simply given a set of parameter values and is expected to generate the remaining information itself, in which case there may be insufficient information provided to perform the task. Between these two impossible extremes lies a spectrum of software designs that require varying levels of input information and which include varying capabilities for information synthesis. At the higher end of this spectrum are tools that include substantial *high-level synthesis* [50] support, meaning that the program accepts a System- or Algorithm-level description of the core and generates an RTL description with minimal guidance. For the design of SCBuild, this option was rejected, since it would inevitably involve developing some sort of high-level synthesis compiler, a difficult task in itself, and one that is well beyond the scope of this research.

Since including high-level synthesis in SCBuild is currently out of the question, it was determined that there were several aspects of parameterized cores that needed to be represented in the input description of a core that is provided to the software. These are:

1. The core's parameters. Parameters are essentially discrete variables that can be set to any value from a finite set. The input description should provide information about each of the core's parameters, including the set of all possible values that the parameter can take.

2. The ways in which the core's parameters affect its architecture. These parameters may affect the underlying architecture of the core in a variety of ways. They may be able to change the bit-width of components in the core, specify different physical implementations for various functional units, alter the number of instances of a component that are included in the core (the number of registers in a register file for example), and even dictate which components are instantiated in the core and how they are connected together. Some of these parameters may drastically affect the resulting core, so the input description needs to be able to adequately describe the ways in which each parameter changes the core's underlying structure.

3. The hierarchy of sub-components that make up the core. Building a complex core from a number of smaller sub-components makes the process of designing that core much simpler. Each sub-component can itself be made up of other sub-components and so on, and each sub-component may have its own set of parameters which can be set to certain values. This hierarchy of components should be specified in the input description.

4. The connectivity of the core's sub-components. Information should be provided on how the sub-components are connected together.

5. The set of possible physical implementations that a sub-component can have. Many components often have several functionally-equivalent implementations that differ only with regards to performance, area utilization, power consumption or some other objective. For example, a digital adder may be either ripple-

carry, carry-lookahead, or some other functionally-equivalent implementation. A separate VHDL description for each of these implementations may be created and stored in a library, so the input description should provide a list of the equivalent implementations that can be used for each component.

4.1.2 Exploring the Design Space of a Parameterized Core

SCBuild should be able to assist users in exploring the design space of the parameterized core with which they are working. The preliminary case study presented in Chapter 3 was useful for determining an approach for doing this. This approach was subsequently utilized in SCBuild as the main engine of automated exploration, as will be discussed later in this chapter.

4.1.3 Generating HDL Descriptions of Core Instances

The final major problem that was addressed by SCBuild was the question of how to generate final HDL descriptions of instances of a parameterized core given a set of parameter values and the input description. This process will be discussed in detail in Section 4.3.

4.2 SCBuild System Environment

A diagram of the SCBuild system environment is shown in Figure 4.1. SCBuild accepts a special input *template description* that describes all of the aspects of a given parameterized core that were discussed in Section 4.1.1. Ultimately, this description will be a set of files generated by another software program called the *Template Architect Tool*. This tool will be a design environment complete with a graphical user interface (GUI) that will allow a designer to drag and drop a set of template components from a Template Component Library to create a template description

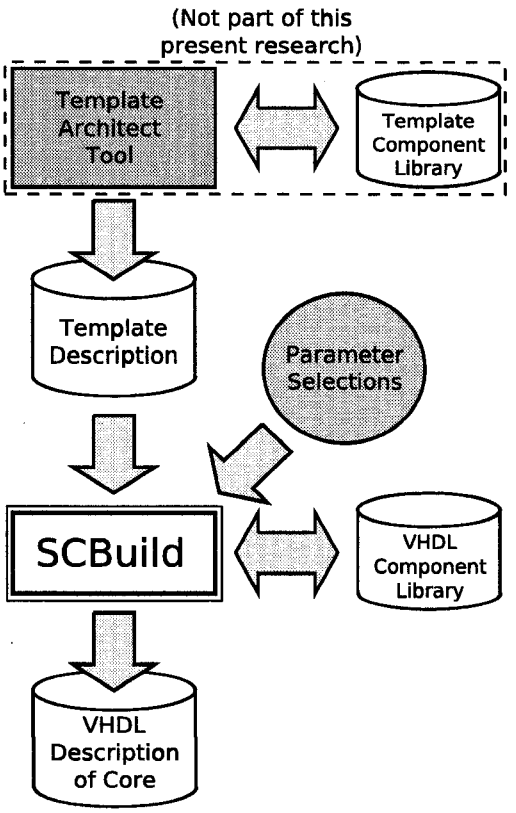


Figure 4.1: The SCBuild System Environment

for virtually any parameterized core that can be envisioned. The development of this tool and the accompanying Template Component Library were deemed to be beyond the scope of this work and were relegated to future research projects. At present, the template descriptions used with SCBuild are created manually. The format of these template descriptions is covered in greater detail in Section 4.3.1 and in Appendix A.

Once the SCBuild software has a template description of a parameterized core to work with, it processes the description, and based on a set user-selected parameter values, it builds a structural VHDL description of a core variant by instantiating ready-made components from the *VHDL Component Library.* It also applies the SEAMO algorithm to the core's parameters in order to explore its design space and

return an approximation of the Pareto-optimal set of configurations. Users can select one of these automatically-generated configurations for final implementation, or they can manually set a core's parameter values. An estimate of each configuration's objective values, such as its FPGA logic resource utilization and critical path delay, are also computed by the program. Many different variant cores can be potentially generated from a single template description simply by specifying different values for the core's parameters. In addition, SCBuild is not locked into using one particular template description. It has been designed to be general enough to accept virtually any template description with which it is provided, as long as that description represents a functionally correct design and follows proper syntax.

4.3 CAD Flow for SCBuild

The SCBuild CAD flow is depicted as a flowchart in Figure 4.2. Each step in the flowchart will be discussed in detail in the sections that follow.

4.3.1 Design Entry and Template Description

The first step in the CAD flow for SCBuild is Design Entry. At this initial stage, a template description for a parameterized core is created by the end-user so that it can be given to SCBuild as an input. At present, this template description is created manually by the template designer, although in future research work, the Template Architect Tool will assist the designer in creating this description.

As its name implies, the SCBuild template description serves as a *template* or *blueprint* for a parameterized core from which many different variant cores can be generated, given a set of parameter values. The complete template description for a parameterized core is made up of a set of text files containing *Extensible Markup Language* (XML) [74] code. XML was chosen as the template description language

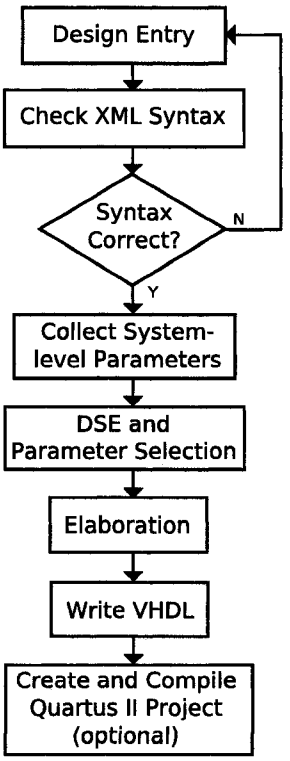


Figure 4.2: The SCBuild CAD Flow

for a number of reasons:

1. It is a well-defined and widely-used data exchange format with simple but strict syntax rules.

2. Since authors of XML documents "invent" their own tags, they are not restricted to using a set of predefined tags or keywords, thus making it ideal for virtually any application.

3. It is relatively easy to write software code that can parse XML.

A detailed description of the SCBuild template description file format can be found in Appendix A. To summarize, the template description is made up of a set of

template components, which are "abstract" or "virtual" hardware components that define a *class* of real hardware modules rather than just a single component. Each and every template component description is stored in a single XML file. The following pieces of information are common to all template component description files:

- The name of the component: Each component must be provided with a unique name before it can be instantiated to build larger cores.
- The component's parameters: Each component can have any number of parameters which can be used to alter the underlying structure of the component in some way. In addition to having a specific name, each parameter is classified as a certain type, which can be either "scalable", "implementation", or "general". Scalable type parameters represent numerical quantities such as bit-widths and correspond directly to "generic" statements in VHDL [54]. Implementation type parameters are used to specify the physical implementation of the component. Finally, General type parameters are open-ended, and are often used to make various changes to a component's structure. Each parameter has a list of possible values that it can take, as well as a default value in case a value is not explicitly assigned to the parameter.

Just as with conventional hardware design, several smaller template components can be used to build larger, more complex components. A component that is constructed from one or more sub-components is referred to as an *aggregate* component, while those components that do not have any sub-components are called *primitive* components. Since many components can have multiple, functionally-equivalent implementations, descriptions of primitive components also contain a list of all of the possible *implementations* from the VHDL Component Library that the component can have. An Implementation type parameter can be created to control which implementation is used for the component in the final structural VHDL description of

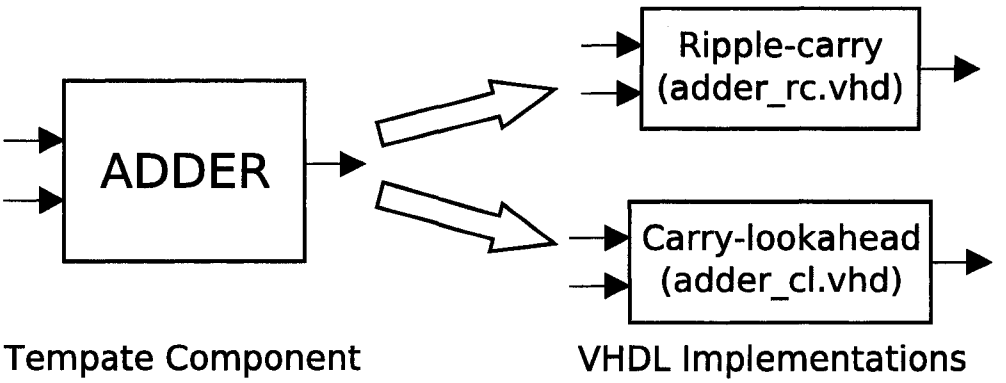


Figure 4.3: A Primitive Template Component with Multiple VHDL Implementations

the core. Figure 4.3 illustrates an example of a primitive template component with multiple implementations.

Descriptions of aggregate components do not contain a list of possible implementations. Instead, they include a list of ports and sub-components. Ports define the interface of the component to the outside world. These can be connected to the ports of other components to form circuits. The name, mode (either "in", or "out") and bit-width of each port are specified in the port listing. Sub-components are instances of other template components that are used to construct the component. In a template component description, the sub-components section includes a listing of which template components are instantiated, as well as information on how the sub-components' ports are connected to each other, and what values their parameters are mapped to. Most importantly, the listing can contain conditional statements which work just like "if" statements in a computer programming language. These statements specify which sub-components are instantiated, which ports are connected to each other, and which parameters are mapped when specific parameters are certain values. Using these statements, template components can be created that have enormous variability in terms of their internal structure, the extent of which is controllable simply by assigning values to their parameters.

The complete template description of a parameterized core consists of a set of XML template component description files. SCBuild can read and use any template description that it is provided with as long as all of the proper information is provided in the description and it is syntactically correct. This means that SCBuild is not locked into working with only one particular hardware template; it is general enough to be able to work with a multitude of different template designs.

The template description also includes several special-purpose files. These are the *Parameter Dependencies file*, the *Objectives file*, and the *System file*. The Parameter Dependencies file contains a set of dependency rules that are used to model the hard interdependencies between various parameters. The Objectives file contains information on the relationships between each parameter of the system and the values of the objectives. The Parameter Dependencies file and the Objectives file will be covered in greater detail later in this chapter. Finally, the System file conveniently stores the names of all of the template component files, the Parameter Dependencies file, and the Objectives file in a single location so that SCBuild has easy access to these file names.

4.3.2 XML Syntax Checking

Once the template description files have been created, they are opened and checked to ensure that they follow correct XML syntax. If they do not, then the user must go back to the Design Entry step and correct any errors that are present. Ultimately, this step will not be necessary, since the Template Architect Tool, discussed in Section 4.2, will automatically generate error-free XML files, thus eliminating the need to verify the syntax.

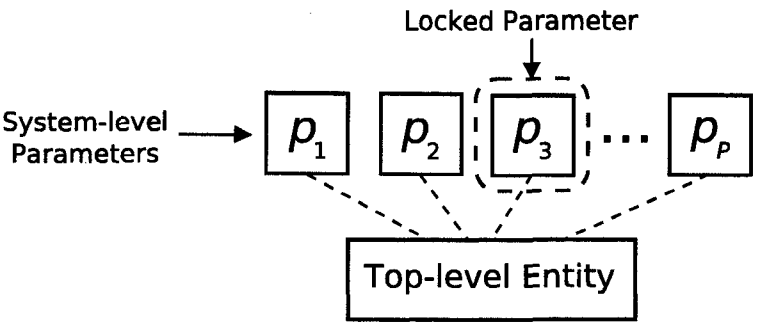


Figure 4.4: Top-level Entity with System-level Parameters

4.3.3 Collect System-level Parameters

At this step, one of the template component files is chosen as the top-level entity of the core, and is denoted the "System" component. Then the template component file containing the description of the System component is read and all of the parameter information is collected and stored in a data structure internal to SCBuild. All of these parameters then become visible to the user. If a core with a specific set of features is required, then any or all of the System-level parameters can be *set and locked* to their desired values. Parameters that are locked will not be changed during the process of automated DSE and are considered constant. Parameters that are not locked are considered "free" and will be used to explore the design space of the parameterized core. This concept is illustrated in Figure 4.4.

4.3.4 DSE and Parameter Selection

Once SCBuild has finished collecting the System-level parameters, the SEAMO genetic algorithm is applied to the free parameters of the system, resulting in a set of configurations that approximates the Pareto-optimal set. The algorithm is implemented in SCBuild as described in Section 3.3 of Chapter 3. Prior to running the algorithm, the user must specify the population size to use, the number of genera-

tions for which to run the algorithm and the crossover and mutation rates. Once the algorithm has run its course and the final set of configurations has been determined, SCBuild will display a listing of these configurations along with their respective objective values. The user selects one of these configurations as the final configuration, and all of the System-level parameters are set and locked to these values.

When applying the SEAMO algorithm to the problem of DSE using parameterized cores, there are two major issues that need to be considered:

1. Hard parameter interdependencies: SCBuild should ensure that only valid configurations are generated by the SEAMO algorithm during the process of exploration.

2. Evaluation of configurations: Configurations need to be quickly evaluated in terms of their objectives during the course of the algorithm.

The ways in which SCBuild handles these two issues will be discussed in the next sections.

Handling Hard Parameter Interdependencies

It is very common for two or more parameters of a parameterized system to share hard interdependencies with one another. Hard interdependencies dictate that value assignments of interdependent parameters must be done simultaneously in order to generate valid configurations. For example, the Nios soft-core processor core has parameters for datapath bus width, instruction and data cache size and hardware multiplication support. Instruction and data caches and hardware multiplication can only be included in the processor if its datapath width is set to 32 bits; any other configuration is considered invalid. Therefore, the data and instruction cache and hardware multiplication parameters each have a hard interdependency relationship with the datapath bus-width parameter.

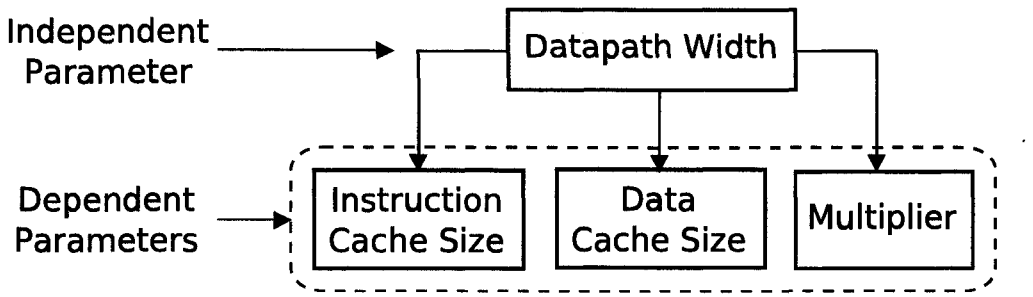


Figure 4.5: Interdependency Relationships Between System-level Parameters

In SCBuild, there are two situations in which parameter interdependencies need to be considered. First, when values are manually assigned to the System-level parameters, the program should ensure that the combination of parameter values chosen does not violate the interdependency rules. Second, when SCBuild is running the SEAMO algorithm, the program should make sure that only valid configurations are created when applying the genetic crossover and mutation operators.

SCBuild handles both of these situations in a similar manner. Between any pair of parameters in the System, a dependency relationship can exist. There are two major rules that govern the dependency relationships between pairs of parameters:

1. In any given dependency relationship, one parameter is considered independent and the other is dependent.

2. Any given parameter can be directly dependent on one and only one other parameter, although many parameters can be dependent on it.

This concept is illustrated in Figure 4.5.

In the figure, the Datapath Width parameter is the independent parameter, and the Instruction and Data Cache Sizes and the Multiplier parameters are the dependent parameters. The arrows in the figure denote dependency relationships, with the arrowhead pointing to the dependent parameter in each case. A dependency relationship in which the parameter is the independent parameter is called an *independent*

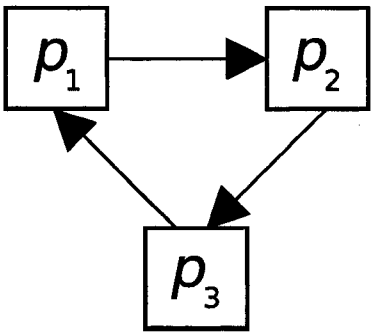


Figure 4.6: Cyclic Dependency Loop

relationship, and conversely, a relationship in which the parameter is dependent is referred to as a *dependent relationship*. As the rules above dictate, any given parameter can have many independent relationships, but only one dependent relationship. These rules ensure that no cyclic dependency loops are created, as illustrated in Figure 4.6.

The parameter pairs in each dependency relationship share a dependency table which stores the valid values of the first parameter for all the values of the second parameter and vise versa. An example of the dependency table between the Datapath Width and Multiplier parameters of the Nios processor core is shown in Table 4.1.

As can be seen from the table, when Datapath Width is 16 bits, then the only value that Multiplier can take is "Software". However, when Data Width is 32 bits then the Multiplier parameter can be Software, MSTEP (partial hardware multiplication) or MUL (full hardware multiplier).

All of the information pertaining to each dependency relationship for a given parameterized core is stored in the Parameter Dependencies file that was introduced in Section 4.3.1. All of the dependency relationships between the System-level parameters must be defined in this file in order for them to be enforced by SCBuild.

The procedure for assigning values to parameters in a dependency relationship is as follows. If the parameter being set is independent in a relationship, then all parameters which are dependent on that parameter are forced to take valid values.

Table 4.1: An Example of a Dependency Table

Datapath Width	Multiplier
16-bit	Software
32-bit	Software, MSTEP, MUL

For example, if Datapath Width is set to 16 bits, then SCBuild will look up the 16-bit entry in the first column of the Datapath Width parameter's dependency table. In the second column beside that entry are all of the possible values that Multiplier can take when Datapath Width is 16 bits. In this case, "Software" is the only valid possible value, so the Multiplier parameter will be forced to take that value. If Datapath Width is *set and locked* to 16 bits, then Multiplier will also be locked to "Software" as well, since it is the only valid possible value that Multiplier can take when Datapath Width is 16 bits. However, if Datapath Width is set to 32 bits, then Multiplier will be forced to be either Software, MSTEP or MUL, selected randomly. If Datapath Width is locked to 32 bits, Multiplier will remain unlocked, because there is more than one valid possible value that it can take when Datapath Width is 32 bits. If the parameter that is being assigned a value is a dependent parameter, then the value that it is assigned must be valid and must not violate any of its dependency relationships. If the Multiplier parameter is being set to MSTEP for example, then the Datapath Width parameter must be 32-bit, otherwise the assignment will not be allowed.

Evaluation of Configurations

As discussed in Section 3.3.1 in Chapter 3, individual configurations are rapidly evaluated by using fast objective estimation equations that relate the values of each parameter to the objective values, the form of which is given in equation (3.3). Any parameterized system that SCBuild works with can have K objective estimation

equations. These objective equations will most often directly represent physically quantifiable aspects of the system such as FPGA circuit area utilization, critical path delay and power consumption, although the user is not restricted to these objectives alone. In order for these equations to be used during the automated DSE process, they must be known to SCBuild ahead of time. Therefore, they are stored in the Objectives file in the template description that was discussed briefly in Section 4.3.1.

The constant coefficient and the information on each term are determined using a set of real-world synthesis data for the core [46]. A small set of "representative" configurations for the core are produced and synthesized using a logic synthesis tool such as Quartus II, which then reports the results of the synthesis, including information such as the FPGA resource utilization and critical path delay. In order to determine the form of the functions $f_{i,k}(p_i)$ in each term of a equation (3.3), it is necessary to study the relationships between each parameter and the values of the respective objectives. For example, the equivalent LE utilization on an FPGA may be found to increase linearly as the datapath bus width increases, so the form of $f_{i,k}(p_i)$ for the datapath bus width will be linear. Once the form of each term has been determined and the set of synthesis data has been obtained for a number of different synthesized configurations, then P-dimensional regression analysis can be applied to the collected data in order to determine the values of the regression coefficients, $a_{0,k}, a_{1,k}, \ldots, a_{P,k}$. The designer of the estimation equations should endeavour to produce just enough configurations in order to provide sufficient definition to the design space so that the objective values for any arbitrary configuration can be computed with a reasonable degree of accuracy.

Once the objective estimation equations have been established and the information regarding these equations is stored in the Objectives File, then it is a relatively simple matter for SCBuild to read the file and compute the objective values for any given combination of parameter values. Each of the possible values of every parameter

is assigned a corresponding integer, and these integer values are plugged into the estimation equations when SCBuild calculates the objectives.

4.3.5 Elaboration

Once all of the values of the System-level parameters are known, SCBuild proceeds with the most important step of the CAD flow: elaboration. At this stage, a final VHDL model of a given core with the set of features specified by the user-selected parameter values is constructed. At this step, SCBuild processes the input template description files and uses the assigned parameter values to build two intermediate representations of the system before writing the final VHDL code. In order to clearly explain what follows, a distinction must be made between these two major internal representations.

The first representation is the *System-level* description of the *hierarchy of template components*. This representation is derived directly from the input template description files and the set of selected parameter values, and can be depicted as a tree graph as shown in Figure 4.7.

Each node in the figure is a template component that is used to build the system. The node at the top is the System, or top-level entity node, and each solid line connecting the nodes represents a parent/sub-component relationship. The System can have any number of sub-components, and any given sub-component can have any number of sub-sub-components, etc. Also, any given component in the hierarchy can have any number of parameters. An important feature of this hierarchy representation is that there also exists a hierarchy of component parameters. The parameters of sub-components can be linked to parameters of their parent components and so on up the hierarchy until the System component is reached, at which point the parameter becomes a System-level parameter. For example, in the figure p_1 of sub-sub-component 1 is linked with p_1 of sub-component 1, which is, in turn,

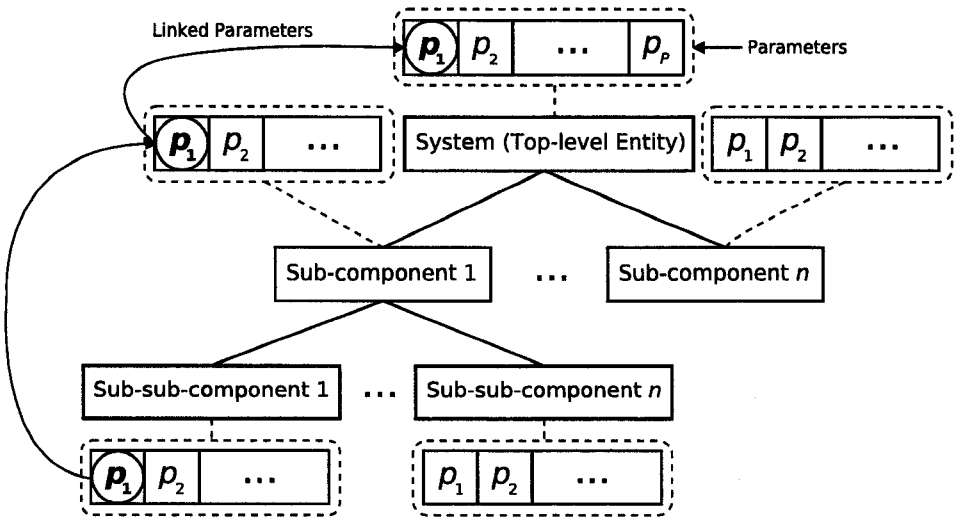


Figure 4.7: System-level Description: Elaboration Hierarchy

linked with p_1 of the System component. Linking component parameters together until the parameter chain reaches all the way to the top-level node allows System-level parameters to directly affect components that are buried deep within the hierarchy.

The second internal representation is the *Register Transfer Level* representation. This description stores the information on the system at the RT-level of abstraction and is directly translatable into VHDL or some other hardware description language. This representation features more detailed information on the structure of each sub-component of the system including what ports and "generic" constants the component has, what sub-components are instantiated under it, and how the sub-components' ports are connected to one another using interconnecting nets. This representation is not parameterized to any extent beyond what is possible to represent using the "generic" constructs within VHDL (or conversely, the "parameter" constructs in Verilog).

SCBuild translates the information in the input template description files into an internal System-level hierarchy description, then into an internal Register Transfer Level representation before finally writing a set of VHDL files describing the resulting

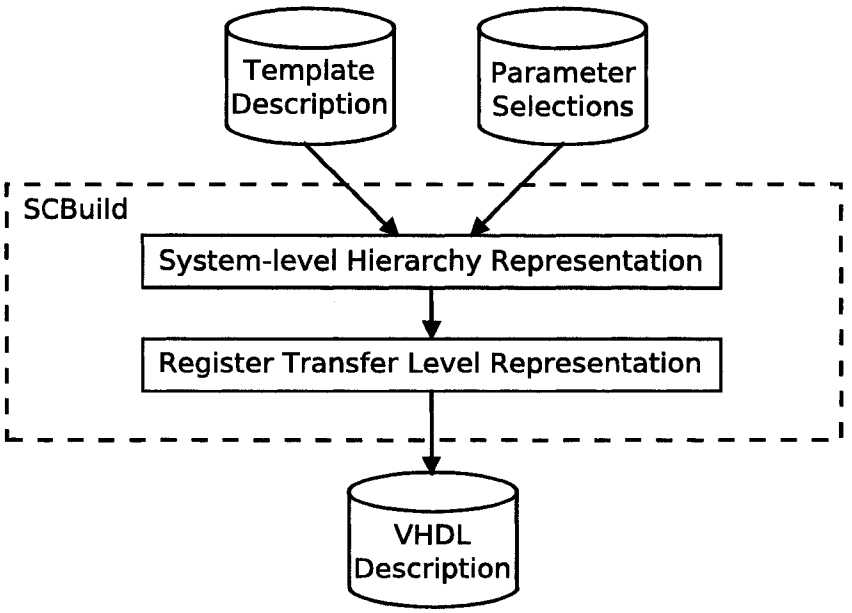


Figure 4.8: Translation of Representations

core variant. This process of translation is illustrated in Figure 4.8.

The process of translating the template description into VHDL code involves a number of steps. The flowchart for the SCBuild Elaboration algorithm is shown in Figure 4.9. This flowchart depicts how a single template component is elaborated. The very first step in the process is to read the name of the template component from its template description file. SCBuild then performs a check to see whether the component is "primitive" or "aggregate" by searching for an `<implementations>` section in the file. If a set of implementation definitions is found in the file, then it is assumed that the template component is a primitive component with one or more implementations in the VHDL Component Library. If there is more than one VHDL implementation to choose from, SCBuild then selects the appropriate one by checking the value that is assigned to the "implementation" type parameter that controls the template component's final implementation. Once this is done, SCBuild then reads the selected VHDL file from the Component Library and constructs an

internal Register Transfer Level representation of the component for later use.

If no implementation definitions are found in the template description file, then the component is assumed to be an aggregate of one or more sub-components. In this case, the process is more complex. First, a brand new RTL component model is created with the name specified. Then generic constants are added to the newly-created RTL component that correspond directly to the "scalable" type parameters that belong to its associated System-level component. Then a set of new ports are added to the component. Next, SCBuild proceeds to read the `<sub_components>` section of the template description file where the current component's sub-component instantiations are located. All of the sub-components that lie outside of the conditional blocks are created first and added to the System-level hierarchy. Then any conditional statements that present within the `<sub_components>` section are evaluated and any conditions that evaluate to "true" are recorded of for later use. Then the sub-components contained within the true conditional blocks are created and subsequently added to the hierarchy as well. At this point, the current component should have all of its sub-components stored in the hierarchy, but these sub-components are missing parameter information, therefore parameters are added to these sub-components by reading their respective template description files. Then these sub-component parameters are either mapped to specific values or linked to parameters of their parent components. Finally, each sub-component is itself elaborated using the exact same algorithm just described. In this way, the entire System-level hierarchy is built by recursively calling the Elaborate algorithm for each component in the hierarchy. Once the sub-components have been successfully elaborated, their ports are connected together in the RTL description using a set of intermediate nets, and the internal RTL model of the system is complete. Finally, a structural VHDL description of the component is written.

By recursively calling the Elaborate algorithm for each component in the System-

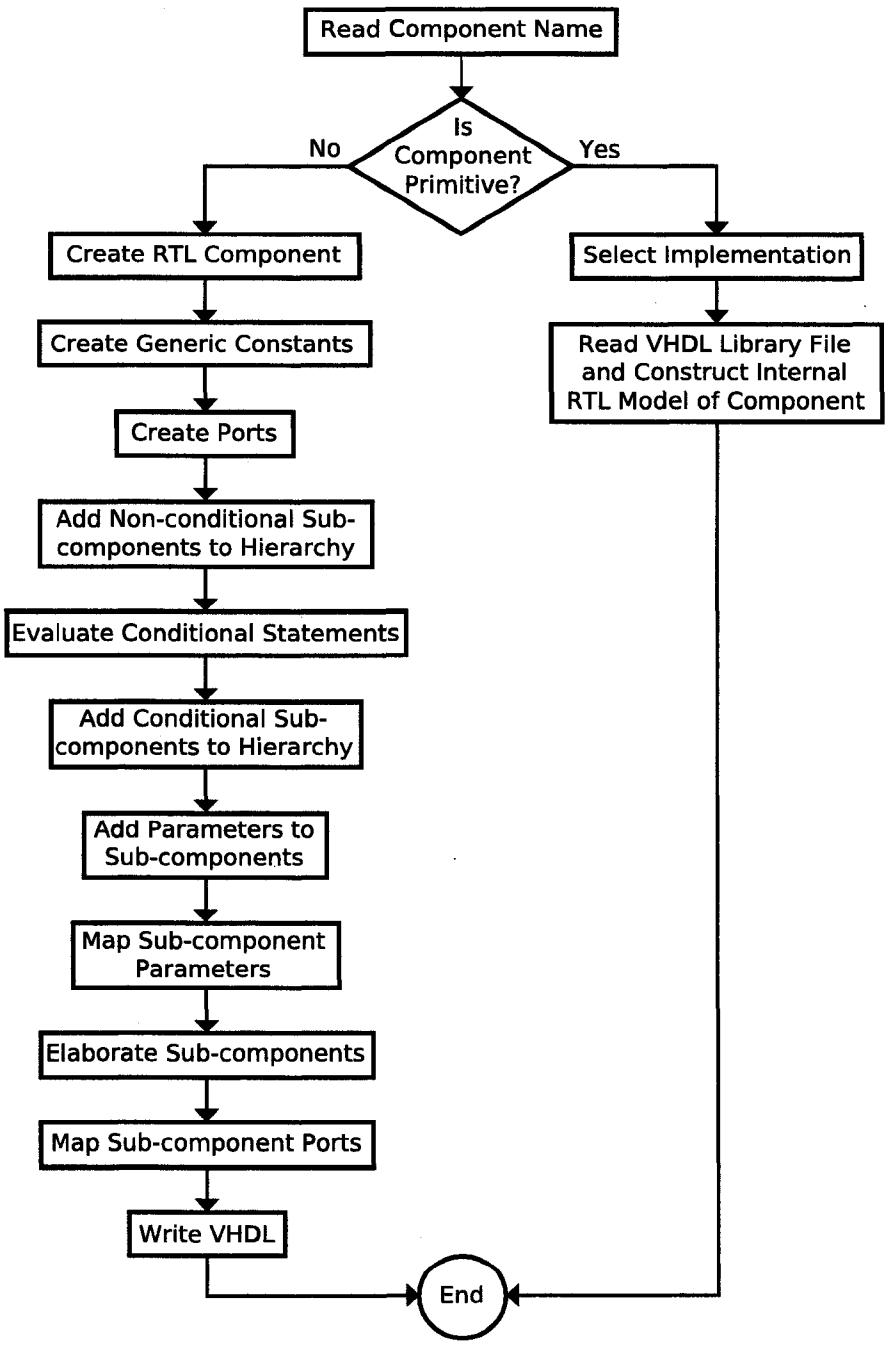


Figure 4.9: Flowchart for the SCBuild Elaboration Algorithm

level hierarchy, a complete description of a core variant, consisting of a set of VHDL files, is created. The very last step in the elaboration process is to copy the generated files into a specified project directory along with the set of VHDL files used from the Component Library.

4.3.6 Quartus II Project Creation and Compilation

If SCBuild is running on a computer that has a copy of Altera's Quartus II software installed, then it can optionally generate a simple Tool Command Language (abbreviated Tcl, and pronounced "tickle") script file [72] that Quartus II's Tcl interpreter [7] can subsequently read and execute. Quartus II has a set of Application Programming Interface (API) functions [7] that can be called from a Tcl script to automate a large number of tasks, including creating new projects, compiling designs, making device pin assignments and creating custom report files. SCBuild can generate a simple Tcl script file that directs Quartus II to create a new project file, take the newly-generated set of VHDL files and include them in the project, perform a complete compilation including analysis and synthesis, fitting, assembly and timing analysis and write the pertinent compilation report information into a text file that can be subsequently read back by SCBuild for later use. SCBuild calls Quartus II in batch mode directly so that all of these functions are handled automatically without requiring the intervention of the user. Once Quartus II has finished its work, it returns control back to SCBuild.

4.4 The VHDL Component Library

The SCBuild VHDL Component Library is an open-ended collection of pre-designed and pre-tested primitive soft-core components described in VHDL that can be used to build more complex designs using SCBuild. The Library can be expanded limitlessly

by adding new hardware components to it. For the purposes of this research, the VHDL Component Library consists of the hardware modules listed in Table 4.2.

Table 4.2: Components in the VHDL Component Library

Component	Description
adder_cl	Generic carry-lookahead adder. Taken from [20].
adder_rc	Generic ripple-carry adder.
b_input_logic	Controls which arithmetic operations are performed by an adder.
barrel_arith_shifter	Performs arithmetic left and right shifts any number of places.
barrel_logical_shifter	Performs logical left and right shifts any number of places.
barrel_rotator	Rotates a bit-field left or right any number of places.
basic_arith_shifter	Performs arithmetic shifts left or right one bit-position at a time.
basic_logical_shifter	Performs logical shifts left or right one bit-position at a time.
basic_rotator	Rotates a bit-field left or right one position at a time.
branch_resolve	Computes the branch targets for branch and jump instructions in a RISC processor.
bus_interface	Interfaces two buses of unequal bit-width together. If the bit-width of the output bus is greater than the input bus, then the extra bits are filled with 0's. If the width of the input bus is greater, then the extra bits are left "open".
constant_unit	Performs either zero-fill or sign extension, based on the value of the select signal.
data_RAM	Single-port RAM block with variable word size and address width. Created using Altera's "altsyncram" megafunction [51].
incrementer	Increments the input by a given value.
instr_memory	ROM module with a variable word size and address width. Created using Altera's "altyncram" megafunction [51].
logic_cct	Logic circuit that performs one of four bitwise logical operations: AND, OR, XOR, NOT.
mem_controller	A data memory controller.

Continued on next page ...

Table 4.2 – continued from previous page

Component	Description
multiplier	A full combinational $n \times n \rightarrow 2n$ bit multiplier.
mux_2_to_1	Generic 2-to-1 multiplexer.
mux_4_to_1	Generic 4-to-1 multiplexer.
reg	Generic register with synchronous load and asynchronous reset.
register_file	Generic register file with any number of registers.
register_file_r0	Generic register file with any number of registers. R0 is always 0.
risc_instr_dec	Instruction decoder for a simple RISC processor.
useful_functions	VHDL package containing useful functions such as the $\log_2()$ function.
zero_detect	Outputs a logical '1' when the input is all 0's, and a '0' when otherwise.
zero_fill	Generic zero-fill circuit with an m-bit input and an n-bit output.
zero_fill_1_bit	Generic zero-fill circuit with a 1-bit input and an n-bit output.
zero_reg	Zero register. Stores all 0's. Writes to this register are invalid.

Each of these components was designed and coded in VHDL, and compiled using Quartus II Version 5.0 software. The data_RAM and the instr_memory components, were created using the Altera's "altsyncram" megafunction [51], and the source-code for the adder_cl component was obtained from [20]. The remaining components were designed and coded from scratch using behavioural and structural VHDL modeling techniques. They were then verified to be functionally correct using Quartus II's simulator tool [7].

4.5 Development and Implementation

The development of SCBuild was achieved by applying many of the principles from the field of software engineering. In this section, the overall design methodology used

to develop and implement SCBuild is discussed in some detail.

4.5.1 General Design Priorities

In any software project development project, it is important to define a set of design priorities that will ultimately guide the development of the software from concept to completed system. During the development of SCBuild, several design priorities were set forth. These are listed in order of decreasing importance below:

1. **Synthesizable output**. It is absolutely essential that the VHDL code generated by SCBuild be synthesizable. Similarly, it should be easy to simulate the behaviour of the resulting hardware component using any available simulator, and the results of RTL-level and gate-level simulation should match up.

2. **Generality and flexibility**. The program should possess an internal data structure that is capable of representing virtually any hardware component described using a hardware description language. This includes support for the hierarchical structural descriptions of components that are quite common in soft-core hardware component designs.

3. **Easy extensibility**. It should be relatively easy to build on the framework of earlier versions of SCBuild so that the system can be extended to include more complex features.

4. **Portability**. Ideally, SCBuild should be easily portable between different operating systems, and the source code should be easily interchangeable between different compilers.

All of these priorities were given their proper consideration during the development of SCBuild.

4.5.2 SCBuild Software Development Methodology

It was determined that an object-oriented approach was a convenient way of representing a soft-core hardware component in software. Since hardware components are made up of other objects such as sub-components, ports and nets, this paradigm provides a natural and intuitive way of representing all the necessary information about the component's structure and attributes. Each part of a component can be represented as an *object* of a class, and these classes can be created in order to represent the set of all of these objects.

For the development of SCBuild, an *iterative* design methodology [47] was applied. Using this approach, five prototype "alpha" versions of the SCBuild system were developed: Versions 0.1, 0.2, 0.3, 0.4 and 0.41. Each successive version implemented more of the final system's functionality than the one before. Version 0.1 contained a partial implementation of the facilities necessary to construct a Register Transfer Level representation of a core and to translate that representation into structural VHDL code. Version 0.2 featured the added ability of representing soft-core components that contained "generic" constants at the Register Transfer Level. In Version 0.3, classes and functions for constructing an Algorithm-level description of a soft-core processor were added. Finally, a framework for handling the System-level hierarchy representation was added in Version 0.4, and cleaned up and expanded in Version 0.41. Using this iterative strategy, the development of SCBuild was broken down into a set of manageable tasks that were completed one-by-one during the development of each prototype. The development of each version followed a software design methodology that included the following tasks:

1. **Requirements gathering**. The initial stage of the design process involved defining precisely what the system is supposed to do (functional requirements) as well as what constraints the system must meet (non-functional requirements). These requirements were gathered together and recorded in a *Requirements*

Document for later reference.

2. **Design**. After deciding what tasks the software is required to perform, the next step was to design its overall architecture by dividing it up into sub-systems and deciding which sub-systems will handle which tasks.

3. **Modeling**. At this step, the detailed design of each software sub-system was performed. Since an object-oriented approach was used to design SCBuild, the structure of each sub-system was modeled using the Unified Modeling Language (UML) [68], a visual language that was developed specifically for modeling the structure and behaviour of object-oriented software systems. Class diagrams were used to model the basic class structure of each of SCBuild's sub-systems, and package diagrams were drawn to model the relationships between each of the sub-systems. The design of each sub-system will be discussed in more detail later in this chapter.

4. **Coding**. After detailed UML models for each sub-system were constructed, they were implemented in software using the C++ programming language.

5. **Program Inspection, Testing and Debugging**. In order to ensure that each prototype version functioned as expected, each sub-system was tested using a variety of methods. The software was subjected to a series of exceptional inputs designed to uncover bugs in the implementation. For the later prototypes, a simple "test processor" template model was also developed to use as a test case to verify that SCBuild was able to correctly generate synthesizable VHDL code as an output.

4.5.3 SCBuild Software Architecture

As was discussed in Chapter 2, any given hardware core can be described at several levels or "layers" of abstraction. Based on this observation, it seemed logical to split

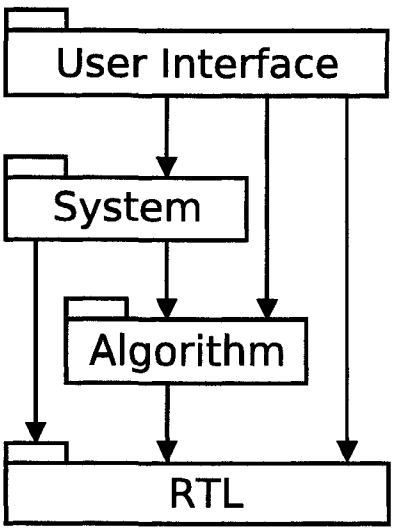


Figure 4.10: UML Package Diagram for the SCBuild Software Architecture

SCBuild up into sub-systems in a like manner. The Multi-Layer architectural pattern was taken from [47] and adapted for this purpose. A UML package diagram for the software architecture model used for SCBuild is shown in Figure 4.10.

Each layer, except for the User Interface layer, contains a data structure that represents some aspect of a core at that abstraction level. Each layer also contains an API that allows higher layers to access its functionality. Each API is a set of functions that controls the data structure at that layer. Functions at higher layers can call the API functions of lower layers in order to construct lower-level representations of a processor or fetch information from those representations. The System layer contains the System-level hierarchy representation of a component that was discussed in Section 4.3.5. Likewise, the RTL layer holds the internal Register Transfer Level description which is ultimately translated into HDL code. In between these two layers is the Algorithm layer, which can represent certain aspects of a processor's instruction set architecture. Although it was not ultimately used in the current version of SCBuild, the Algorithm layer does provide a basis for extending the software

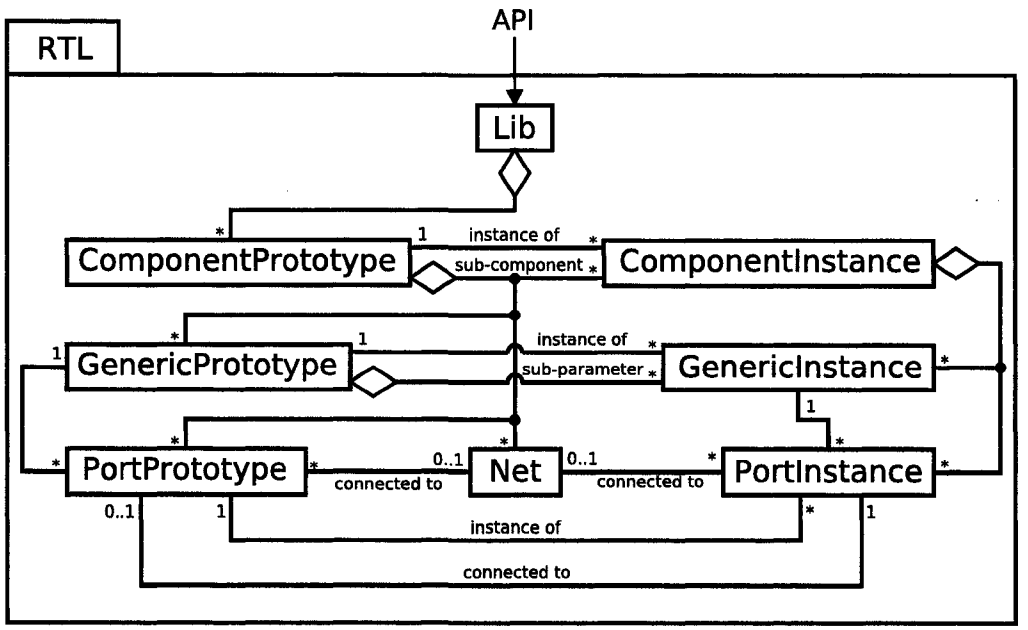


Figure 4.11: Class Model for the RTL Layer

tool to perform other useful functions such as automatic control logic generation and the removal of unused hardware. Each of these three layers will be discussed briefly in the paragraphs that follow.

The RTL Layer

The UML class diagram for the RTL layer is shown in Figure 4.11. This layer is responsible for storing all of the data necessary to construct the Register Transfer Level representation of a soft-core hardware component that was presented in Section 4.3.5.

The model consists of eight classes: Lib, ComponentPrototype, ComponentInstance, PortPrototype, PortInstance, GenericPrototype, GenericInstance, and Net. These eight classes together have the capabilities and functions necessary to construct a data structure representing a soft-core hardware component as a set of smaller, in-

terconnected sub-components. Essentially, this model states the following: any given hardware component prototype can have any number of ports, generics, and sub-components (component instances). A prototype is an abstract object from which many identical instances can be made. When an instance of a given component prototype is created under another component prototype, instances of its ports and generics are created as well. Nets are intermediate signals that connect the ports of the various sub-components together to form complete circuits.

The Lib class contains the API, a set of functions that control how the data structure is constructed. Higher layers can call these API functions in order to construct an RTL description of a component. This model, minus the GenericPrototype and GenericInstance classes, was created during the development of SCBuild Version 0.1. Support for the representation of generics was added in Version 0.2.

The Algorithm Layer

The purpose of the Algorithm layer is to represent the behaviour of all instructions in the instruction set of any given processor. This provides a basis for specifying what instructions will be implemented in a generated processor as well as how they will be implemented. A common approach used to represent the behaviour of an instruction is the *data dependence graph*, used by the author of SPREE [75] as well as others. This approach was adapted for implementation in Version 0.3. In this approach, the instruction is broken down into a series of *microoperations*. Each microoperation is a small unit of functionality which can be implemented by a single hardware module (for example, add, subtract, etc.), although multiple implementations of the same microoperation can exist. Each microoperation has a set of input and output ports. These ports represent the required inputs to that microoperation and the outputs that are generated. The ports of the various microoperations are connected together to form a complete data dependence graph which defines the order in which the

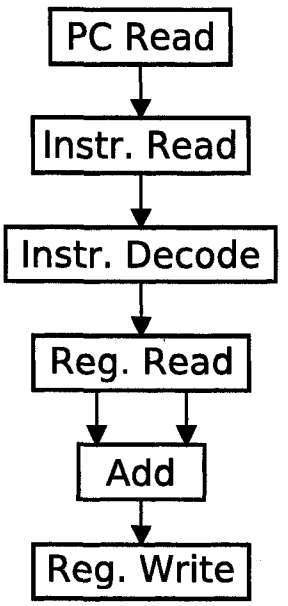


Figure 4.12: Data Dependence Graph for a Generic Add Instruction

microoperations need to execute in order to carry out an instruction. An example of a data dependence graph for for a generic "add" instruction is shown in Figure 4.12.

In order to completely define an instruction set, a data dependence graph for each instruction in that instruction set should be constructed. With this conceptual framework established, a UML class model was developed which efficiently represents data dependence graphs internally in SCBuild. The model that was developed and implemented is shown in Figure 4.13.

Essentially, the class model states the following:

1. An Instruction Set is made up of many Instructions.
2. An Instruction is made up of many Microoperations.
3. A Microoperation owns many MicroopPorts.
4. Any given MicroopPort can be connected to many other MicroopPorts.

This simple model is general enough to represent the instruction set for any processor provided that it is possible to construct a set of data dependence graphs for

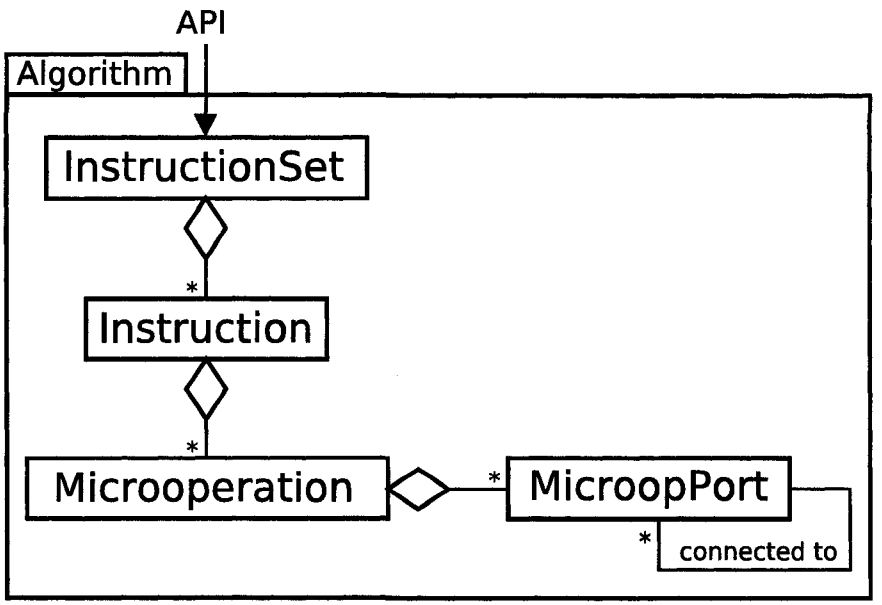


Figure 4.13: Class Diagram for the Algorithm layer

that instruction set. Also included in the model, although not shown in Figure 4.13, is the ability to represent the various instruction formats that a processor may have. Each instruction in the instruction set is assigned an instruction format based on the number and bit-widths of the operands that are needed.

The System Layer

The System layer is responsible for representing a parameterized soft-core hardware component at the System level, the highest level of abstraction. At this level, the soft-core is seen as a set of user-configurable parameters. The System layer is responsible for constructing the System-level hierarchy description that was discussed in Section 4.3.5. It also handles all of the functionality related to the design space exploration of the core's parameters using the SEAMO algorithm. The UML class diagram for the System layer is given in Figure 4.14.

The most important class in the diagram is the System class. It is derived from

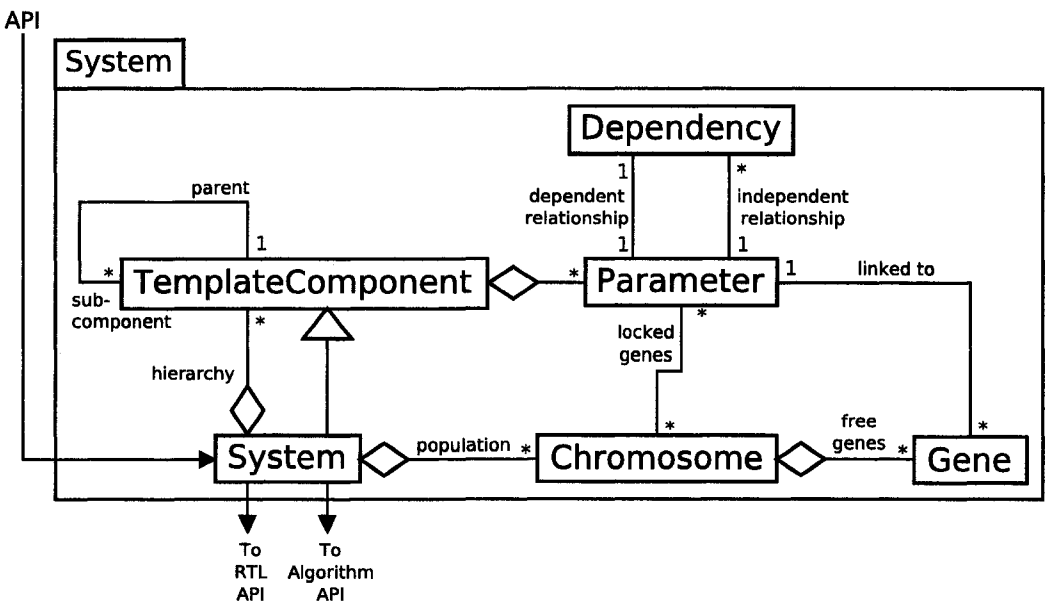


Figure 4.14: Class Diagram for the System layer

the TemplateComponent class and inherits all of its functionality and associations to other classes. It also owns multiple objects of the TemplateComponent class, which represent the components of the hierarchy, as depicted earlier in Figure 4.7. Each of these template components has a set of objects of the Parameter class which represent its parameters. Parameter objects that belong to the System class are the System-level parameters, and as such, are the ones that can be configured by the user or left free to be varied during design space exploration.

Also contained within the System class are numerous objects of the Chromosome class, which represent the population of configurations used by the SEAMO algorithm. Each Chromosome object, in turn, possesses a collection of objects of the Gene class, whose values can be varied freely during DSE. Finally, the Parameter class also has a double association with the Dependency class, which is an association class whose objects store information on the interdependency relationships between pairs of parameters in the core.

The System class contains all of the functions and associations to other classes that are necessary to perform an automated design space search using the SEAMO algorithm and to construct a System-level hierarchy model given a set of XML template description files. It communicates with classes at lower layers by calling their API functions, which enables it to direct the construction of lower-level representations of a core, and ultimately generate synthesizable VHDL code.

4.5.4 Implementation Details

All five versions of SCBuild were written completely using C++. Version 0.41, the final alpha version, is about 16,400 lines long. In order to maintain code portability, an effort was made during development to use only those library functions and classes that were recognized by a number of different compilers. Two different compilers were used to develop the software: the Microsoft Visual C++ 6.0 [12] compiler and the MinGW compiler [38] paired with the Code::Blocks [1] integrated development environment.

4.6 Summary

In this chapter, the design and implementation of the SCBuild CAD tool was presented. The major problems that were addressed by SCBuild were first presented, followed by a discussion of the environment under which SCBuild operates. An overview of the SCBuild CAD flow was then given, and a detailed explanation of each step in the flow was provided. Finally, the VHDL Component Library was briefly discussed before the details of the development and implementation of SCBuild were presented. The next chapter discusses the results of some experimental studies that were conducted using SCBuild and a simple RISC processor.

Chapter 5

Experimental Results

In this chapter the results of several design space exploration experiments are presented. For these experiments, a template description model of a simple parameterized pipelined RISC processor core was created. Then the SCBuild CAD tool was used to generate and compile a number of variant implementations of the processor core by performing a parameter sweep of all the core's parameters. The purpose of this exercise was to generate enough real synthesis data in order to establish objective estimation equations that provided reasonable estimates of the FPGA area utilization and critical path delay for any arbitrary processor configuration. Next, an initial population of 50 random configurations was generated, and each of these configurations was synthesized using Quartus II. The area and delay objective estimation equations were tested for accuracy against these 50 data points. Experiments were then conducted in order to determine suitable values for the crossover and mutation rates used in the SEAMO algorithm. Finally, using the determined crossover and mutation rates, the initial population was "evolved" for 20 generations of the SEAMO algo-

rithm, yielding an approximation of the Pareto-optimal set. VHDL implementations for these 50 evolved configurations were generated and synthesized using Quartus II, and the resulting synthesis data were compared with that of the initial population.

5.1 Target Core

The parameterized RISC processor template used in this research is a modified version of the pipelined RISC CPU presented by Mano and Kime [48]. A detailed description of the modified core can be found in Appendix B. To summarize, the core is a simple microprocessor with a load-store architecture consisting of a datapath, a control unit, and separate data and instruction memories. It features a total of 38 instructions, including instructions for performing arithmetic, logical, shift, branch and memory operations with integer data.

In order to facilitate experimentation, a set of parameters was added to the processor, a template description model of the core was created and a library of VHDL building-block components was constructed (see Section 4.4). The core features the parameters listed in Table 5.1. The numbers in parentheses are the integer values assigned to each parameter value. The data width of the processor can be either 8, 16, 32, or 64 bits. The data and instruction memory bus widths are configurable to any value between 5 and 15 bits, yielding anywhere between 32 to 32,768 words of memory each. The adder present in the ALU and the branch adder can be implemented using either a ripple-carry or a carry-lookahead structure. There are also three types of shifters available: logical, arithmetic and rotator. Each of these can be implemented either as a "basic" shifter, meaning that a value is shifted or rotated only one position per clock cycle, or as a "barrel" shifter, allowing values to be shifted or rotated more than one position per cycle at the expense of a much larger shifter. The user can also choose not to include any or all of these shifters. A combinational integer multiplier can also be optionally included. The operand width, which directly affects the reg-

ister file size, can be configured between 2 and 9 bits, providing anywhere between 4 and 512 general-purpose registers. Finally, pipelined and unpipelined versions of the processor are available, with the pipelined version featuring a 4-stage pipeline. Data and control hazards are presently handled in software by inserting NOP (no operation) instructions into a program. None of these parameters share hard interdependencies with any of the others—they are all independent. Applying equation (3.2) to the parameters listed in the table, there are a total of exactly 1,672,704 possible configurations for this core.

In order to ensure that the RISC processors generated by SCBuild functioned as expected, a simple assembler was written for the processor. This assembler translates a listing of program instructions down into a memory initialization file (.mif) [7], which is used to specify the contents of the processor's instruction memory. A sample configuration was then generated by SCBuild in order to test the processor's functionality. The configuration was a 32-bit pipelined variant with 32 general-purpose registers, 32 words of data and instruction memory each, hardware multiplication and barrel shifters for the arithmetic shifter, the logical shifter and the rotator. This configuration was compiled using Quartus II and its functionality was observed using Quartus II's Simulator Tool [7]. In this way, all of the processor's instructions were verified to be functioning correctly.

5.2 Establishing the Objective Estimation Equations

In order to establish a set of objective estimation equations (equation (3.3)) for the RISC processor core using the P-dimensional regression technique described in Chapter 3, it was first necessary to synthesize a set of configurations that are representative of core's design space. To this end, a parameter sweep was performed on each of the

Table 5.1: RISC Processor Hardware Parameters

Parameter	**Possible Values**
ALU Adder Implementation (p_1)	(1) Ripple-carry, (2) Carry-lookahead
Arithmetic Shifter Implementation (p_2)	(1) None, (2) Basic, (3) Barrel
Branch Adder Implementation (p_3)	(1) Ripple-carry, (2) Carry-lookahead
Data Address Width (p_4)	(1–11) 5 to 15 bits
Data Width (p_5)	(1) 8, (2) 16, (3) 32, (4) 64 bits
Include Multiplier (p_6)	(1) False, (2) True
Instruction Address Width (p_7)	(1–11) 5 to 15 bits
Logical Shifter Implementation (p_8)	(1) None, (2) Basic, (3) Barrel
Operand Width (p_9)	(1–8) 2 to 9 bits
Pipelined (p_{10})	(1) False, (2) True
Rotator Implementation (p_{11})	(1) None, (2) Basic, (3) Barrel

core's 11 parameters. Starting from a base configuration, (in which all of the parameter values are set to 1), each of the core's parameters were varied across their entire range of values while the other parameters were held constant at their base values. This yielded a total of 41 sweep configurations, each of which was generated by SCBuild and compiled using Quartus II [7]. All of these configurations were targeted for the Altera Stratix EP1S40F780C5 FPGA [8], and were compiled using the default compiler settings. The following pieces of information were collected from the Quartus II compilation reports for each configuration: the number of LEs, DSP blocks, M512, M4K and M-RAM memory blocks used by the configuration, and the critical path delay of the processor in nanoseconds reported by the Timing Analyzer.

A second set of sweep configurations was also produced. In order to study the relationship between the data width parameter (p_5) and the 10 remaining parameters of the core, an additional 111 sweep configurations were generated and compiled.

Table 5.2: Summary of Parameter Sweep Results

Config.	clk (ns)	LEs	M512s	M4Ks	M-RAMs	DSPs	Eq. LEs
Smallest	21.197	173	3	0	0	0	234.5
Largest	49.99	12,393	3	1	0	0	12,502.3
Max.	–	49,250	384	0	0	32	57,869.84
Fastest	9.53	181	1	1	0	0	249.3
Slowest	62.84	1,152	2	2	0	32	2036.44

This time, the data width parameter was varied at the same time as each of the other parameters. Of these 111 configurations, four were so large that the Quartus II Fitter could not successfully place them into the Stratix FPGA. Quartus II failed to compile another two configurations due to a shortage of memory in the computer on which they were compiled. In total, 146 different sweep configurations were generated and compiled successfully. These configurations served as the basis for establishing area and delay objective estimation equations for the RISC processor. The results of the parameter sweep experiments will be discussed in greater detail in the sections that follow.

5.2.1 Results of Parameter Sweep

The sweep configurations showed great variability in both FPGA area utilization and critical path delay. These results are summarized in Table 5.2. The complete table of sweep results can be found in Appendix C. In terms of area, the smallest configuration generated was the "base" configuration—an unpipelined 8-bit variant that used only 234.5 equivalent LEs. The largest successfully-compiled sweep configuration was a 32-bit variant with 256 general-purpose registers. This particular configuration used 12,502.3 equivalent LEs, although larger configurations are possible provided an FPGA with a higher logic capacity is available. In fact, the "maximum"

configuration—with all parameters set to their highest values—required a total of 49,250 LEs (119% of 41,250 LEs total), 384 M512 RAM blocks (100%), and 32 DSP blocks (29%) after fitting, equal to 57,869.84 equivalent LEs. The timing analysis results could not be obtained for this configuration. In terms of delay, the fastest sweep configuration seen was an 8-bit pipelined variation, with a critical path delay of 9.534 ns (104.9 MHz), while the slowest configuration was an unpipelined 64-bit processor with a integer multiplier, showing a delay of 62.84 ns (15.9 MHz).

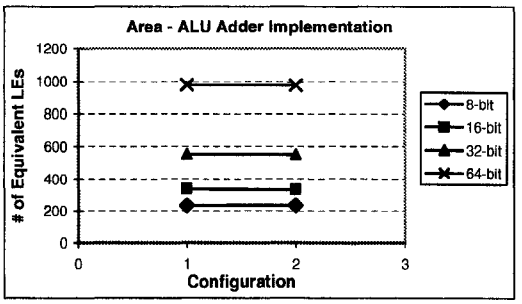
For the area objective estimation equations, the forms of the functions $f_{i,k}(p_i)$ in equation (3.3) were determined by studying the relationships between each of the RISC processor's parameters and the resulting area and critical path delay data collected from the parameter sweep results. The effects on the total equivalent LE utilization and the critical path delay of the processor for each parameter were studied individually.

Area Utilization

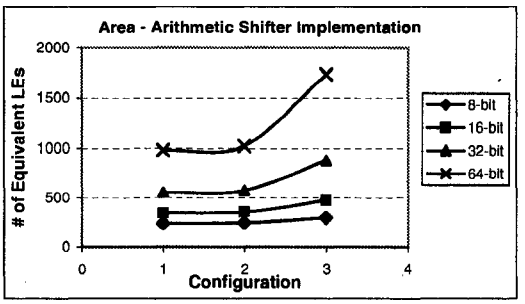
Figure 5.1 contains graphs showing the relationships between the total area of the processor and each of the 11 parameters. In general, these relationships followed fairly predictable patterns. Based on the results, the following observations were made:

- The ALU and branch adder implementation parameters had only a minor effect on the total area of the processor. At most, the carry-lookahead adder contributed only 2 extra LEs to the total count. Since there are only two possibilities for these two parameters, the relationships between them and the total FPGA area utilization were assumed to be linear.

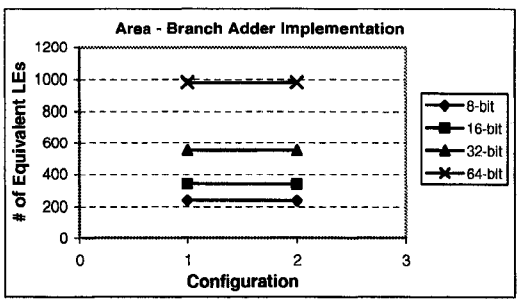
- The implementation parameters for the arithmetic shifter, the logical shifter, and the rotator affect the total area of the processor significantly. The basic shifter adds a few LEs to the processor: anywhere between 10 and 43 LEs for the arithmetic shifter, 19 to 42 LEs for the logical shifter, and 10 to 41 for



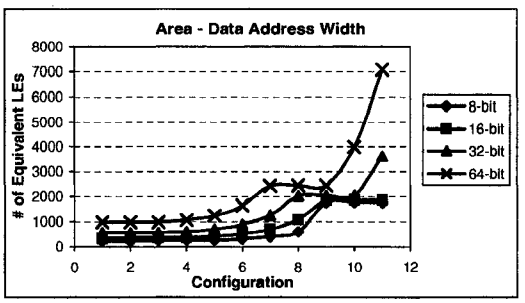
(a) ALU Adder Implementation (p_1)



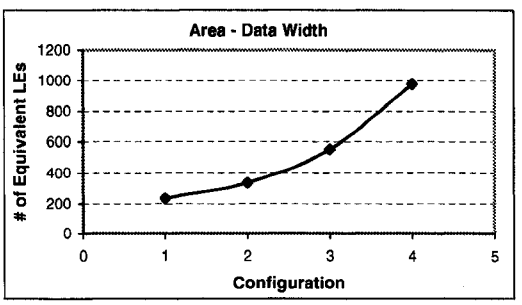
(b) Arithmetic Shifter Implementation (p_2)



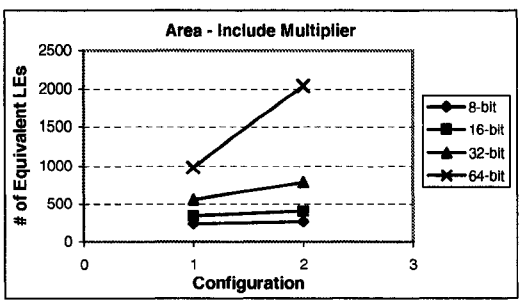
(c) Branch Adder Implementation (p_3)



(d) Data Address Width (p_4)

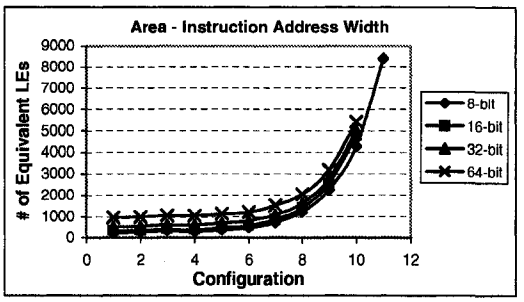


(e) Data Width (p_5)

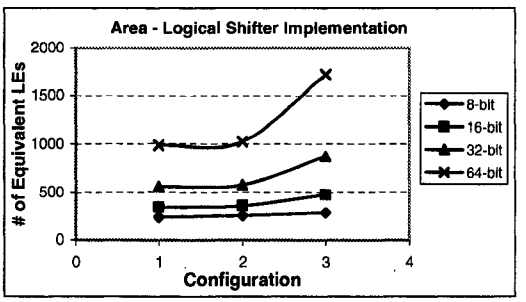


(f) Include Multiplier (p_6)

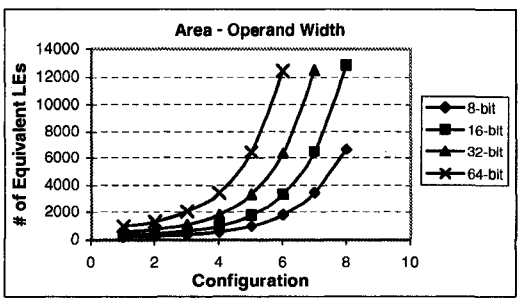
Figure 5.1: Parameter Sweep Results – Area



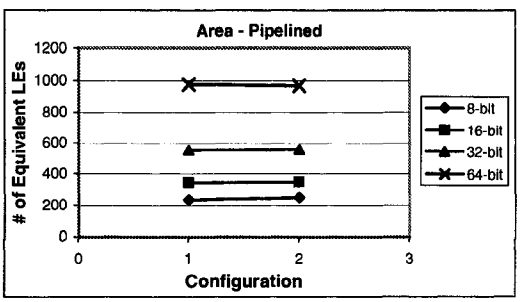
(g) Instruction Address Width (p_7)



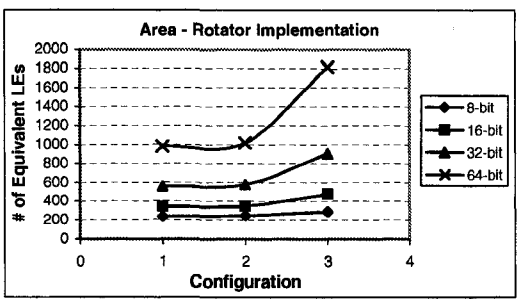
(h) Logical Shifter Implementation (p_8)



(i) Operand Width (p_9)



(j) Pipelined (p_{10})



(k) Rotator Implementation (p_{11})

Figure 5.1: Parameter Sweep Results – Area (Cont'd)

the rotator, depending on the Data Width. As can be expected, the barrel versions of each shifter are considerably larger than their "basic" counterparts. The arithmetic barrel shifter can add anywhere between 61 to 758 LEs, the logical barrel shifter adds 56 to 745 LEs, and the barrel rotator adds 58 to 838 LEs to the processor, again depending on the Data Width parameter. The relationships between these three implementation parameters and the total area were modeled using exponential functions.

- The total equivalent LE usage increases exponentially as the the bit-width of the data and instruction memory addresses increase. The "kinks" in the curves for the Data Address Width parameter are due to the fact that some of the FPGA memory resources that are occupied by the processor, such as the M4Ks and M-RAM blocks, are only partially utilized by some configurations. As the Data Address Width increases, more and more of the partially-full memory resource is filled until the whole block is occupied, making it necessary to use additional memory blocks and resulting in a "jump" in the total equivalent LE usage.

- As expected, the data width parameter causes an exponential increase in the total area of the processor as the parameter is increased.

- The integer multiplier was implemented using the Stratix DSP blocks, plus some additional LEs. Anywhere from 1 to 32 DSP blocks were utilized, and an additional 11 to 311 LEs were added by the multiplier, depending on the data width parameter. The relationship between the integer multiplier parameter and total equivalent LE count was considered to be linear.

- The Operand Width parameter, which directly affects the number of general-purpose registers included in processor, has an extremely significant impact on the total size of the processor. For instance, if the Operand Width is set to 8 bits, giving a total of 512 registers, then an additional 6,383 LEs and one extra

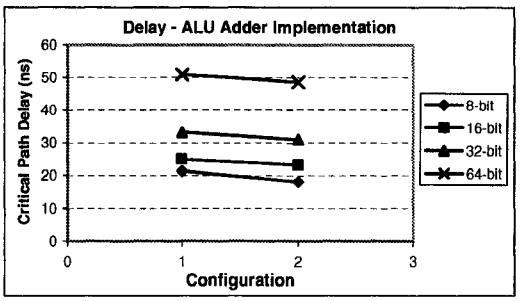
M512 memory block are added to the total resource count of the 8-bit processor variant. These numbers increase further as the size of the Data Width parameter increases. Again, the relationship between the Operand Width parameter and the total area of the processor can be approximated by an exponential function.

- The pipelined parameter has only a negligible effect on the size of the processor and can be approximated by a simple linear function. This is due to the fact that only a small amount of hardware (about 10 equivalent LEs) is added to the RISC processor when it is pipelined. However, this would not be the case with a more complex processor, which may require additional hardware for branch prediction, data forwarding, and pipeline stalling.

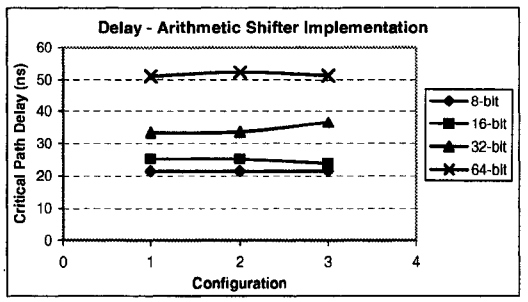
- Almost all of the parameters have a *soft* interdependency relationship with the data width parameter. The amount that a parameter affected the final area of the processor was often proportional to the data width of the processor. Therefore, in order to increase the accuracy of the objective estimation equations, several of the terms in the area objective estimation equation were scaled by the data width parameter.

Critical Path Delay

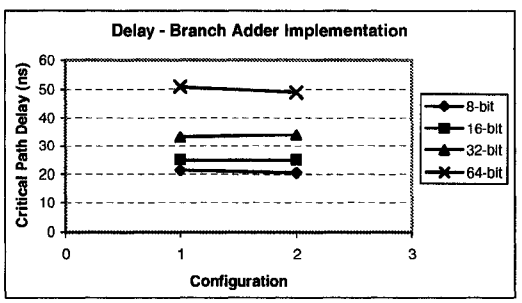
Figure 5.2 contains graphs depicting the relationships between the critical path delay and the processor parameters. As can be seen from the figures, the critical path delay generally did not follow any predictable patterns and was therefore more difficult to estimate accurately. Therefore, to increase the accuracy of the estimates, the relationships between all of the parameters, except for the branch adder implementation, the include multiplier, and the pipelined parameters, were modeled using base 10 or base 2 logarithmic functions. As would be expected, the pipelined parameter had the greatest positive effect on the critical path delay of the processor, reducing the delay by at least 10 ns in all cases.



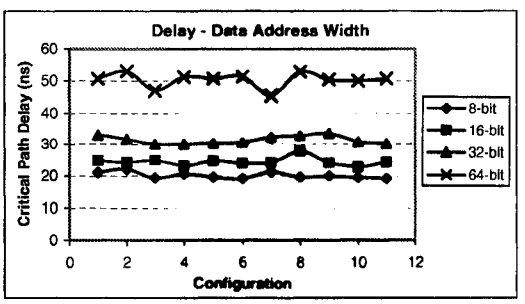
(a) ALU Adder Implementation (p_1)



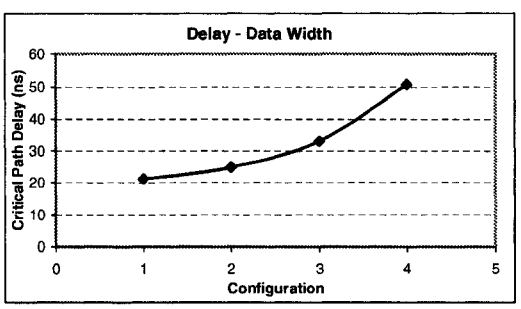
(b) Arithmetic Shifter Implementation (p_2)



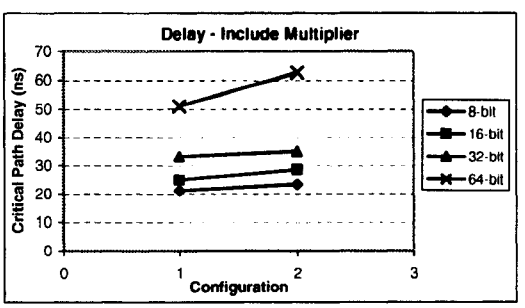
(c) Branch Adder Implementation (p_3)



(d) Data Address Width (p_4)

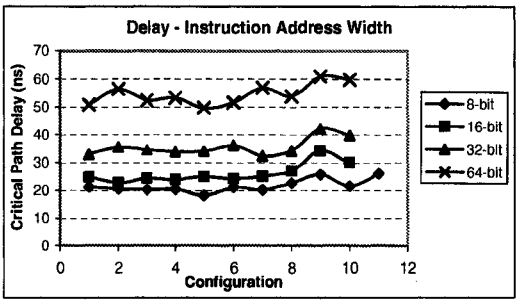


(e) Data Width (p_5)

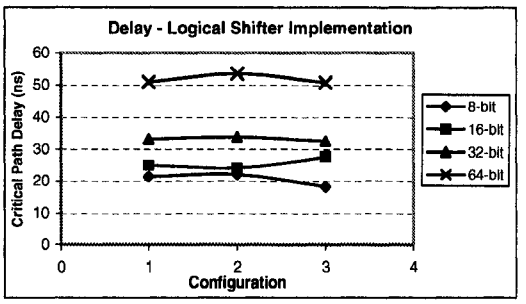


(f) Include Multiplier (p_6)

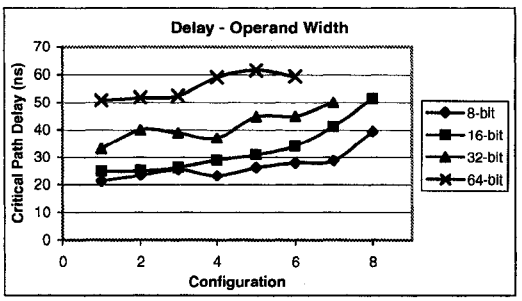
Figure 5.2: Parameter Sweep Results – Delay



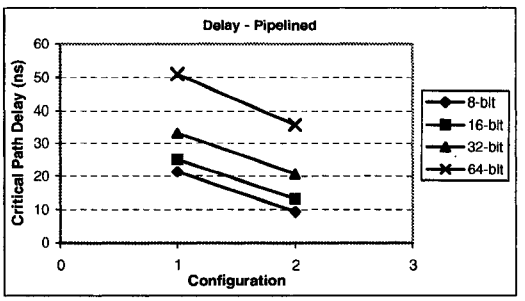
(g) Instruction Address Width (p_7)



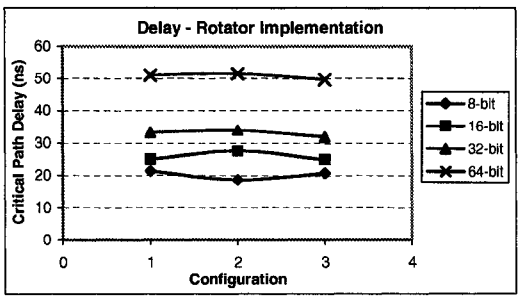
(h) Logical Shifter Implementation (p_8)



(i) Operand Width (p_9)



(j) Pipelined (p_{10})



(k) Rotator Implementation (p_{11})

Figure 5.2: Parameter Sweep Results – Delay (Cont'd)

Table 5.3: Regression Coefficients for RISC CPU

Parameter	i	$a_{i,1}$ (Area)	$a_{i,2}$ (Delay)	$f_{i,1}(p_i)$ (Area)	$f_{i,2}(p_i)$ (Delay)
–	0	198.86	17.52	-	-
ALU Adder Implementation (p_1)	1	-25.28	-1.03	p_1	$\log_{10}(p_1)$
Arithmetic Shifter Implementation (p_2)	2	$0.00014p_5$	4.59	99^{p_2}	$\log_{10}(p_2)$
Branch Adder Implementation (p_3)	3	$-80.73p_5$	0.95	p_3	p_3
Data Address Width (p_4)	4	$14.11p_5$	0.21	1.5^{p_4}	$\log_{10}(p_4)$
Data Width (p_5)	5	15.46	1.73	2.3^{p_5}	2.12^{p_5}
Include Multiplier (p_6)	6	$89.44p_5$	6.33	p_6	p_6
Instruction Address Width (p_7)	7	4.33	0.68	1.99^{p_7}	$\log_{10}(p_7)$
Logical Shifter Implementation (p_8)	8	$0.00014p_5$	3.74	99^{p_8}	$\log_{10}(p_8)$
Operand Width (p_9)	9	12.22×1.98^{p_5}	2.55	2^{p_9}	$\log_2(p_9)$
Pipelined (p_{10})	10	-52.86×1.1^{p_5}	-11.44	p_{10}	p_{10}
Rotator Implementation (p_{11})	11	$0.00020p_5$	2.30	$90^{p_{11}}$	$\log_{10}(p_{11})$

5.2.2 Determining the Final Objective Estimation Equations

Some trial and error was necessary to determine the exact forms for all of the functions $f_{i,k}(p_i)$ for each parameter. After these functions were determined, P-dimensional regression analysis was applied to the parameter sweep data in order to compute the $a_{i,k}$ coefficients in equation (3.3). The final functions and coefficients used in the area and delay estimation equations are listed in Table 5.3.

5.2.3 Testing the Objective Estimation Equations

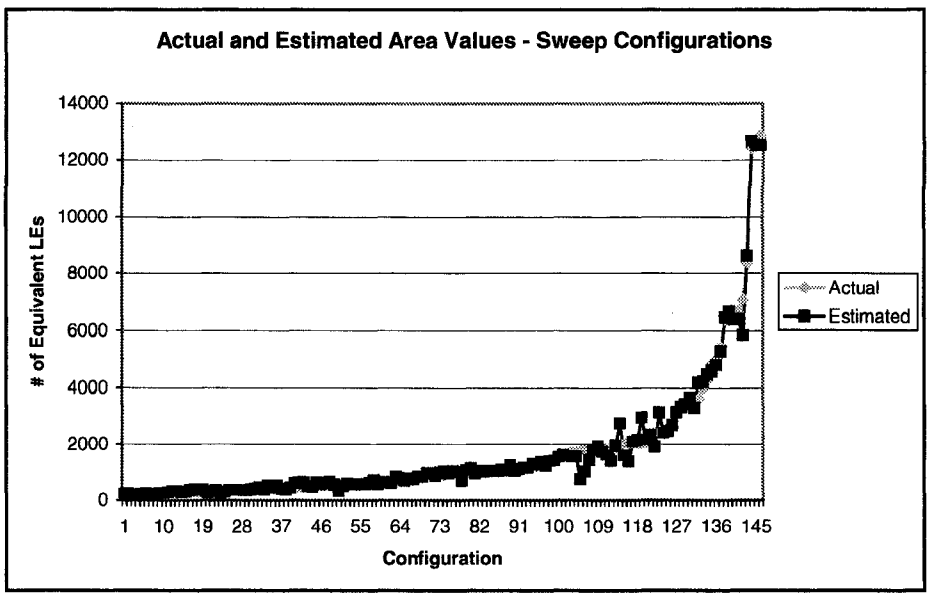
The accuracy of the area and delay estimation equations was first tested using the 146 parameter sweep configurations used to establish the equations. The "actual" values obtained from compilation data collected from Quartus II were compared with

the estimated values that were computed using the established objective estimation equations. Graphs comparing the actual values to the estimated values of area and critical path delay for the 146 sweep configurations is shown in Figure 5.3. As can be seen in Figure 5.3(a), the estimated area values correlate well with the actual values, showing an average percentage error of 10.1%. The delay estimates also track the actual values reasonably well, with an average percentage error of 11.7%, as shown in Figure 5.3(b).

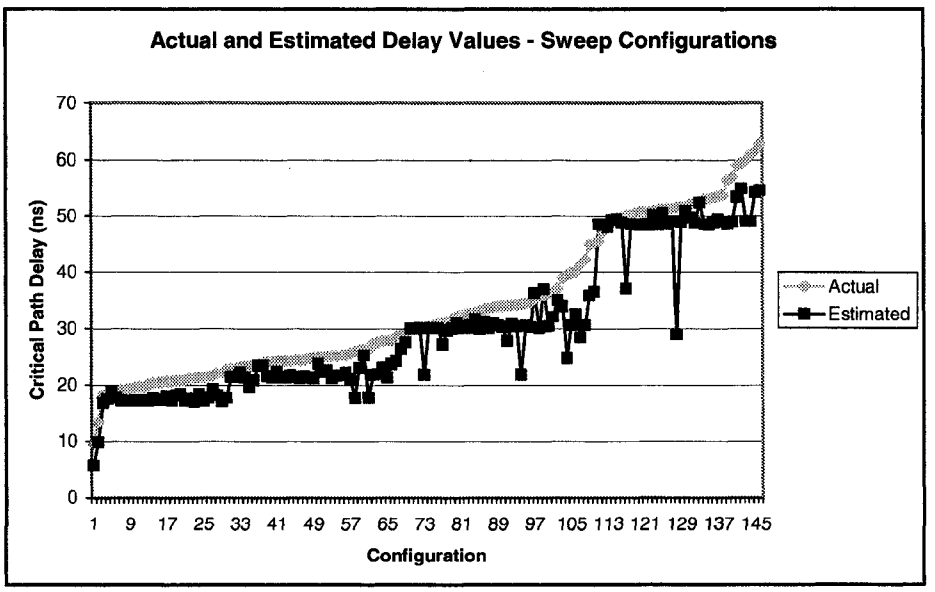
Next, the equations were tested to determine their accuracy for any arbitrary configuration. For this test, a set of 50 individuals was randomly generated and compiled with Quartus II. Nine of these configurations failed to compile, so the remaining 41 configurations were used as test points. Again, the compilation data obtained for these 41 configurations were compared to the approximated values that were determined using the estimation equations. Graphs comparing the actual and estimated values for area and delay are shown in Figure 5.4. As can be seen from the figure, the area equation provides better overall estimates than does the delay equation; however they are both still within reasonable tolerances. The average error was 13.3% for the area estimates and 16.4% for the delay estimates.

These experiments serve to demonstrate the inherent difficulty with estimating the critical path delay of an arbitrary parameterized core. This difficulty is due to the fact that a number of different factors affect the critical path delay of a core, including the implementation of the core's underlying components, the placement of the circuit on the FPGA, the routing running between the various parts of the core, etc. By contrast, the FPGA area utilization of a core is easier to estimate accurately, because many of the parameters affect the total area of a core in a regular and predictable manner.

Another comment that can be made about these results is that a tradeoff exists between the accuracy of the estimated objective values and the amount of computation

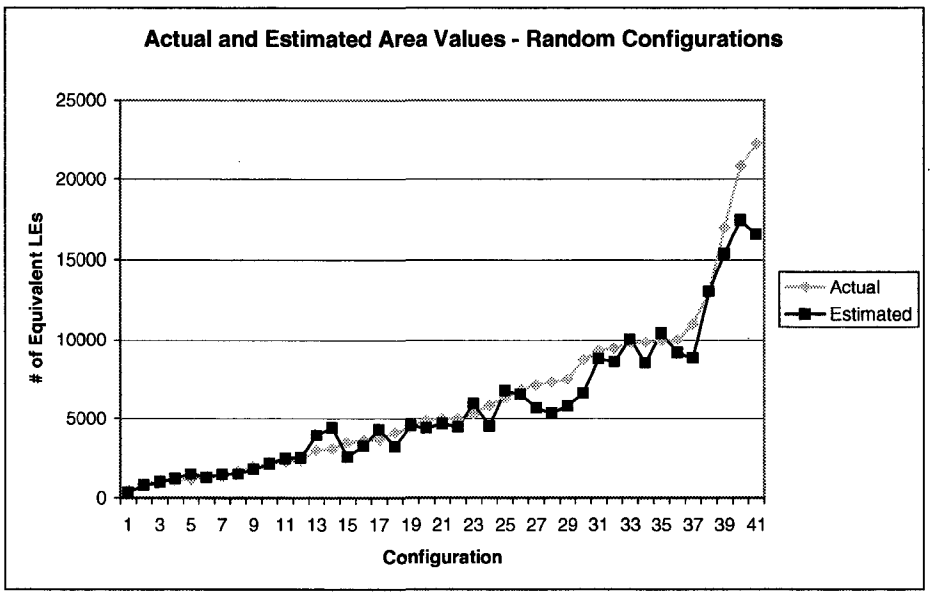


(a) Area

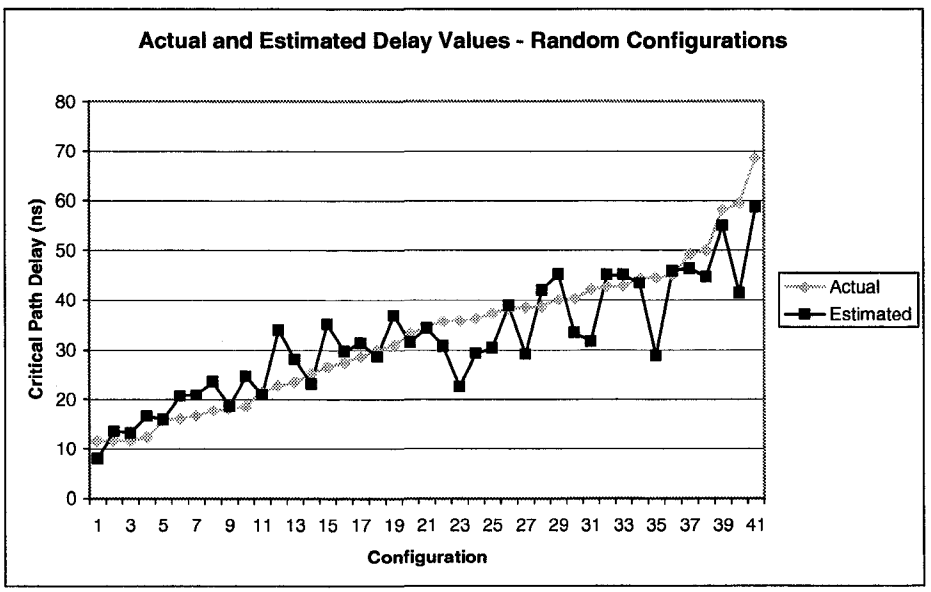


(b) Delay

Figure 5.3: Actual and Estimated Values for Sweep Configurations



(a) Area



(b) Delay

Figure 5.4: Actual and Estimated Values for Random Configurations

required to obtain those values. In general, more accurate estimations can be made at the expense of longer computation times. The goal of the regression-based objective estimation technique used in this research is to produce a set of estimation equations that can be evaluated quickly and easily. However, the light computation workload required by this method does come at the cost of reduced estimation accuracy. In addition, an up-front investment of time and effort is also necessary to produce enough real sythesis data in order to establish these equations. In future work, different objective estimation techniques may be applied in order to increase the accuracy of the estimates and remove the need to generate a set of sweep configurations.

5.3 Design Space Exploration

The SEAMO algorithm was applied to a population of randomly-generated configurations in order to determine an approximation of the Pareto-optimal set. In this section, the results of this experiment are presented.

5.3.1 Algorithm Parameters

The set of 50 configurations (of which 9 failed to compile) that was generated previously to test the objective estimation equations was also used as the initial population for this exploration experiment. Suitable values of the crossover and mutation rates were determined experimentally. The crossover and mutation rates were both varied between 0.1 and 1.0, and the resulting evolved population was observed. It was determined that a value of 0.1 for the crossover rate and 0.2 for the mutation rate provided the greatest diversity of configurations in the evolved population. Finally, the number of generations of the algorithm was set to 20.

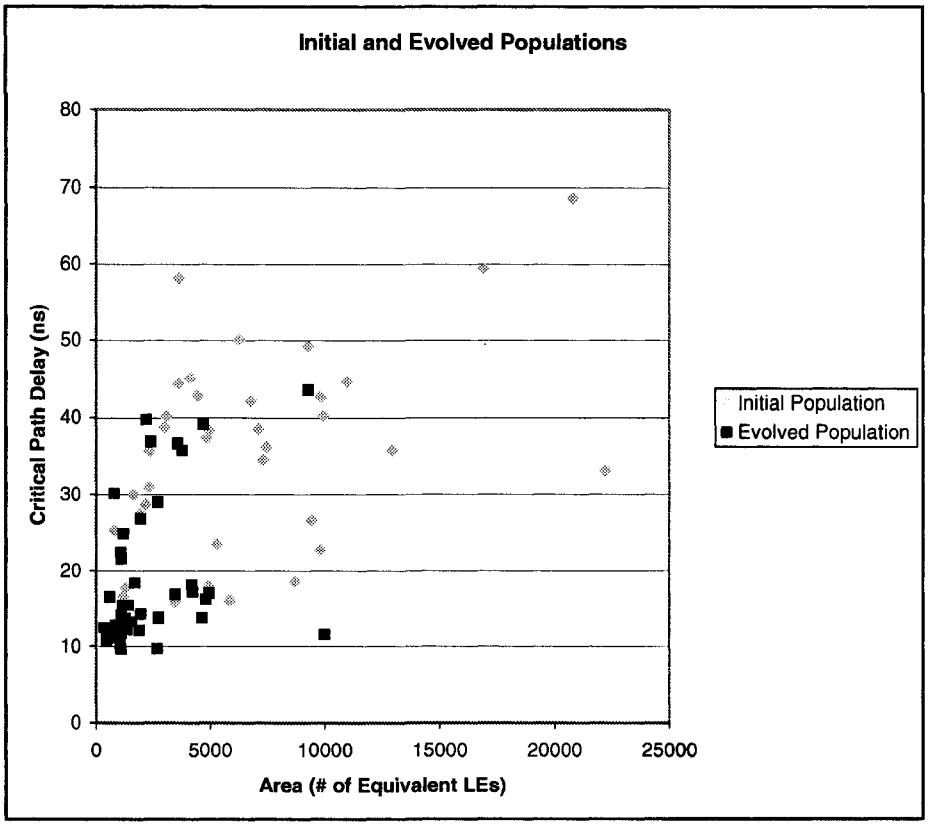


Figure 5.5: Initial and Evolved Populations

5.3.2 Results

Using the algorithm parameters mentioned above, the SEAMO algorithm was applied to the initial population of 50 configurations (including the nine configurations that failed to compile). After 20 generations of the algorithm, the population began to converge toward an approximation of the Pareto-optimal set. Each of the configurations in the resulting evolved population was compiled using Quartus II (see Appendix C for the table of results). A visual comparison of the initial and the evolved populations (using the actual values collected from the compilation reports), is shown in Figure 5.5.

As can be seen in the figure, the majority of the configurations in the evolved

Table 5.4: Number of Occurrences of Each Parameter Value in the Evolved Population

Value	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}
1	23	25	40	7	26	39	7	21	19	3	21
2	27	8	10	9	15	11	6	17	15	47	16
3	-	17	-	2	3	-	2	12	4	-	13
4	-	-	-	0	6	-	4	-	5	-	-
5	-	-	-	3	-	-	6	-	3	-	-
6	-	-	-	1	-	-	2	-	4	-	-
7	-	-	-	9	-	-	13	-	0	-	-
8	-	-	-	8	-	-	5	-	0	-	-
9	-	-	-	7	-	-	3	-	-	-	-
10	-	-	-	2	-	-	1	-	-	-	-
11	-	-	-	2	-	-	1	-	-	-	-

population tend to cluster around the lower left corner of the design space, approximating the Pareto-optimal front. The variability in the evolved population is due to the inaccuracies present in the objective estimation equations used to evaluate each configuration. If more accurate estimation methods were used, then the evolved population would form a smoother curve along the lower-left boundary of the design space.

5.4 Conclusions Drawn From Results

In Table 5.4, the number of occurrences of each parameter value in the evolved population are listed. A number of observations can be made from this table:

- There are roughly the same number of configurations with the ALU adder implemented using a ripple-carry structure as with a carry-lookahead implementa-

tion. However, the SEAMO algorithm tended to heavily favour the ripple-carry implementation for the branch adder.

- In roughly half the configurations, the arithmetic and logical shifters and the rotator were completely eliminated. For the remaining configurations, the barrel implementation was the favoured choice for the arithmetic shifter, and there was a slight bias toward the basic implementation for the logical shifter and the rotator.

- There appears to be a roughly even distribution of data and instruction address widths across the whole population.

- The integer multiplier was removed in the majority of the configurations. Only 11 configurations featured hardware multiplication support.

- The SEAMO algorithm tended to eliminate configurations that had longer operand widths (and therefore larger register files). This is unsurprising, considering the fact that the operand width parameter contributes significant area to the processor when its value is large.

- As would be expected, 47 out of 50 configurations were pipelined. This is because of the dramatic improvement that the pipelined parameter makes to the critical path delay of the processor with only a minimal increase in area.

These experiments have shown that utilizing a genetic-based approach to prune the design space of a parameterized core can be helpful in assisting a designer making decisions regarding the selection of parameter values for a parameterized hardware platform. Coupled with an accurate configuration evaluation methodology, the genetic algorithm helps to eliminate the sub-optimal configurations from consideration, yielding a much smaller set of configurations from which to choose. Designers can then

choose one configuration from the pruned set that satisfies their application-specific design constraints.

5.5 Summary

In this chapter, the results of several experiments involving the SCBuild CAD tool and a simple RISC processor were presented. First, a total of 146 different "parameter sweep" configurations were generated and compiled in order to provide sufficient definition to the processor's design space so that equations could be established that provided reasonably accurate estimations of the FPGA area utilization and critical path delay of any arbitrary configuration. After the exact forms of the area and delay estimation equations were determined, their accuracy was tested using a set of 41 randomly-generated configurations. It was discovered that the equations provided reasonably accurate estimations of the area and delay for the configurations tested—within 13.3% for area and 16.4% for delay. Next, the SEAMO algorithm was applied to a population of 50 random configurations for 20 generations. The evolved population showed significant overall improvement in both the area and delay objectives.

The next chapter concludes this thesis by providing a summary of the research contributions made by this work. In addition, a discussion of some of the future work that remains to be done in this area is also presented.

Chapter 6

Conclusions and Future Work

As long as the rate of growth in the complexity of embedded systems continues to increase at its present pace, new technologies, design techniques and methodologies will continue to be developed to meet the challenges that this growth in complexity presents. An emerging technique known as platform-based design has generated a good deal of interest in recent years, due mainly to its emphasis on the use of pre-designed and pre-tested IP cores as hardware platforms upon which to build designs. This thesis presented an investigation of one particular platform-based design technique: genetic algorithm-based design space exploration using parameterized soft-cores. After providing the relevant background material, the results of a preliminary case study involving the SEAMO genetic algorithm and the Altera Nios parameterized soft-core processor were presented. From this study, it was concluded that applying a genetic-based algorithm to a parameterized core with a sizable design space can be helpful in narrowing down the number of design configurations that must be considered by the designer when selecting one for a particular application.

In Chapter 4, the design and implementation of SCBuild, a CAD tool which incorporates the techniques investigated during the case study, were discussed in some detail. In Chapter 5, the results of a set of experiments carried out using SCBuild and simple parameterized RISC processor were presented. From these results, a number of observations were made. First, it is comparatively easier to predict the final implemented area of a given core than it is to estimate its critical path delay. This is due to the numerous factors that affect the critical path delay of a circuit implemented on an FPGA. Second, it was concluded that utilizing a genetic-based approach, coupled with accurate objective estimation models, can help an embedded systems designer to make intelligent decisions regarding the assignment of values to the parameters of an embedded hardware platform. It does this by pruning uninteresting and sub-optimal configurations from the design space and returning the set of Pareto-optimal configurations, allowing the designer to select one configuration from that set that satisfies the requirements of the intended application. Finally, some observations were made regarding the exploration of the design space of the simple RISC processor.

6.1 Summary of Research Contributions

The following contributions were made over the course of this research:

1. A preliminary case study was conducted in which the feasibility of applying a genetic algorithm-based approach to parameterized soft-core hardware components was investigated.

2. A technique for estimating the objective values (i.e. FPGA area utilization and critical path delay) given a set of parameter values was examined and applied to several soft-core components. Using this technique, reasonably accurate estimations were made for both area and delay of the cores that were tested.

3. SCBuild, a software-based CAD tool, was developed which utilizes the exploration and objective estimation approaches that were investigated during the preliminary case study. The tool is capable of exploring the design space of a parameterized soft-core using the SEAMO algorithm, it can generate structural VHDL descriptions of core variants given a user-selected set of parameter values, and it is able to provide estimates of the FPGA area utilization and critical path delay of the final logic circuit. SCBuild helps to lay the groundwork for a more thorough and detailed investigation of the soft-core processor design space targeting FPGAs to be conducted. During the the development of SCBuild, a number of contributions were made:

   (a) A unique text-based file format was created in conjunction with the development of SCBuild, which allows descriptions of "template" cores to be created. SCBuild can potentially generate thousands, millions or more of different variant VHDL implementations of a soft-core from a single template description.

   (b) UML models were created to represent the structure of soft-core hardware components at the System, Algorithm and RTL levels of abstraction.

4. A parameterized template description of a simple pipelined RISC processor was created, and several design space exploration experiments were carried out with it using SCBuild.

6.2 Future Work

The research work presented in this thesis can be extended in a variety of ways. First, a number of new features could be added to SCBuild that would enable it to automate a greater number of design tasks. For instance, the tool could be extended so that

it is able to automatically generate the necessary control logic for a processor given a datapath template description and a set of parameter values. A greater number of tasks can also be automated using Tcl scripts—automatic compilation of the cores generated by SCBuild and reporting of the synthesis results are just the beginning of what is possible to accomplish using this method.

Adding more accurate objective estimation techniques would also be another way of improving the quality of the results generated by SCBuild. At its present state, a fair number of "representative" variants of a processor core must be generated by SCBuild and synthesized in order to provide enough definition to the design space to make accurate estimations of a core's FPGA area utilization, critical path delay, etc, for any arbitrary configuration. These estimations provide some useful information that helps to determine which configurations are "good" universally speaking; however they make no indication of how well or how poorly a specific software application would run on a given processor variant. Later versions of SCBuild could include a simulation framework, similar to that found in the Platune system [30] for example, that could compute estimates of the run-times of software applications on particular processor configurations.

New tools could be developed that would help to supplement the work performed by SCBuild. As was discussed in Chapter 4, a Template Architect Tool could be developed that would allow a designer to quickly and easily create parameterized Template Descriptions of soft-cores by dragging and dropping Template Components from a library. Another interesting avenue to explore would be the development of a software code profiling tool that can analyze a given software application written in C/C++ or some other high-level language and call SCBuild to automatically create a customized processor that is optimized to run that software program [58, 15]. SCBuild could also generate a compiler and assembler that is customized for the particular processor variant for which the software application is targeted.

One major addition to SCBuild could be in the area of high-level synthesis [50]. Since the tool manipulates descriptions of a core at the various levels of abstraction, it may be possible to add functionality that allows it to automatically translate a higher-level description (System or Algorithm) into an RTL description without the aid of a pre-existing template. Other exploration algorithms may also be investigated and compared with one another to see which one proves the most useful in assisting in the process of deriving an application-specific processor given a parameterized processor core.

Finally, a broader exploration of the soft-core processor design space targeting FPGAs can be conducted. A wider variety of architectural features can be examined, including cache sub-systems, branch prediction schemes, floating-point support, various pipeline architectures, custom instruction support, functional unit implementations, interrupt and exception handling and others. Also, parameterized multi-processor hardware platforms and the various accompanying implementation issues would prove to be a useful course of study.

Appendix A

Details of the SCBuild Template Description File Format

Each file in the template description contains an XML [74] description of one template component. Template components are abstract hardware components that describe a class of hardware modules rather than just a single concrete component. Template components fall into two categories: primitive and aggregate components. Figures A.1 and A.2 show example skeleton XML descriptions for each of these two categories.

Aggregate components are those components that are made up of one or more sub-components, while primitive components are singular modules that do not have sub-components. Every template component description, regardless of whether that component is aggregate or primitive, contains information on the component's name and parameters. The `<parameters>` declaration section can contain numerous parameter definitions, each of which has the form shown in Figure A.3.

The parameter type can be either "scalable", "implementation" or "general".

```
<component>
   <name>...</name>
   <ports>
      ...
   </ports>
   <parameters>
      ...
   </parameters>
   <sub_components>
      ...
   </sub_components>
</component>
```

Figure A.1: Aggregate XML Template Component Descriptions

```
<component>
   <name>...</name>
   <parameters>
      ...
   </parameters>
   <implementations>
      ...
   </implementations>
</component>
```

Figure A.2: Primitive XML Template Component Descriptions

```
<parameters>
   <parameter>
      <name>parameter_name</name>
      <type>parameter_type</mode>
      <values>v1, v2, ...</values>
      <default_value>def_value</default_value>
   </parameter>
   ...
</parameters>
```

Figure A.3: An Example Parameter Declaration

Scalable type parameters are numerical parameters that are used to represent variable bit-widths within a component and correspond directly to the "generic" constants used in VHDL entity declarations [54], or "parameter" statements in Verilog [55]. Implementation type parameters are used to select which component from the VHDL Component Library to instantiate in the final VHDL description of the core. Lastly, General type parameters are an open-ended parameter category that can be used for various purposes to alter the underlying structure of the parameterized core.

Each parameter must have a set of possible values specified within its declaration. These can be given either as a set of discrete comma-separated values, or, if the parameter is a Scalable type, a range of values can be given (for example, the values can be specified as "1 to 10"). Finally, each parameter must have a default value that is used when a value is not explicitly assigned to the parameter when the template component is instantiated.

A.1 Primitive Template Component Descriptions

Each primitive template component can be linked to any number of "physical" implementations from the VHDL Component Library. As shown in Figure A.2, primitive template components have a special section called `<implementations>`. In this section, the various VHDL implementations that the template component has are listed. An example `<implementations>` section is shown in Figure A.4.

If there is more than one possible implementation for a given template component, then these implementations should be linked with one of the Implementation type parameters declared in the `<parameters>` section discussed above. The `<parameter>` field in the `<implementations>` section specifies the parameter to which the implementation of that component is linked. If this field is missing, SCBuild will simply use the first implementation that it finds in the section. The `<implementations>` section also has many `<implementation>` fields, and each of these fields provides the

```
<implementations>

   <parameter>parameter_name</parameter>

   <implementation>
      <name>implementation_name</name>
      <file>file_name.vhd</file>
   </implementation>
   ...
</implementations>
```

Figure A.4: An Example Implementation Declaration

```
<ports>
   <port>
      <name>port_name</name>
      <mode>port_mode</mode>
      <width>port_bit_width</width>
   </port>
   ...
</ports>
```

Figure A.5: An Example Port Declaration

implementation with a name and the name of the VHDL file in which the implementation can be found. In order to properly link an Implementation type parameter to the listed implementations in the section, that parameter must have the names of all listed implementations in its `<values>` field.

A.2 Aggregate Template Component Descriptions

Aggregate component descriptions contain information on the component's ports and sub-components. A component's `<ports>` declaration section can contain as many ports as needed. These ports define the interface to the component. An example port declaration section is given in Figure A.5.

```
<sub_components>
   <sub_component>
      <name>sub_cmpt1</name>
      <file>sub_cmpt_file_name.xml</file>
   </sub_component>
   ...
   <parameter_map>sub_cmpt1.p1 = v1</parameter_map>
   <parameter_map>sub_cmpt2.p2 = parent_cmpt.p2</parameter_map>
   ...
   <port_map>sub_cmpt1.port1 = sub_cmpt2.port2</port_map>
   <port_map>sub_cmpt1.port2 = parent_cmpt.port2</port_map>
   ...
   <if condition = "expression">
      ...
   </if>
</sub_components>
```

Figure A.6: An Example of a Sub-Components Section

Each port in the port declaration section must include a name for the port, the port's mode (which can either be "in" or "out"), and the port's bit-width, which can either be an integer or an expression containing the names of one or more "scalable" type parameters.

Template components can be constructed by instantiating smaller template components as sub-components and specifying the connections between the ports of the sub-components. As shown in Figure A.1, aggregate template component descriptions have a special `<sub_components>` section where the sub-components are instantiated and connected together. In Figure A.6, an example `<sub_components>` section is given.

Individual sub-components are instantiated in the `<sub_component>` field, where the name of the sub-component instance and the XML file name of the instantiated template component are provided. The `<parameter_map>` statements are used to map each sub-components' parameters to specific values or to link the parameter with one

of the parent component's parameters. If a sub-component's parameter is linked to one of the parent's parameters, then whatever value the parent parameter receives will automatically be mapped to the sub-component's parameter as well. In this way, a parameter that may be buried deep within the hierarchy of sub-components may be made visible to a higher level component. The `<port_map>` statements are used to connect the ports of each of the sub-components together to create a complete circuit. A sub-component's port may be mapped to one or more ports of other sub-components, or to one of the parent component's ports.

One of the most significant features of the sub-components section are the conditional `<if>` statements. These statements can be used to make potentially drastic changes to the underlying structure of an aggregate component based on the values given to particular parameters. The "condition" attribute in the opening `<if>` tag is a Boolean expression containing the names of one or more of the the parent component's parameters. If the condition evaluates to "true" then all of the statements present in between the opening and closing `<if>` tags are executed. All three types of statements mentioned above—`<sub_component>`, `<port_map>`, and `<parameter_map>`—can be included in an `<if>` block, thereby allowing the structure of the aggregate component to be significantly different depending on the values of the parameters named in the condition. There is no limit on the number of `<if>` blocks that can be present under the `<sub_components>` section.

A.3 The Parameter Dependencies File

It is quite common for parameters of a core to share hard interdependencies with one another. SCBuild can be provided with a *Parameter Dependencies file* that contains the necessary information on each parameter interdependency. In this file, dependency relationships between pairs of parameters are declared, and their dependency tables are defined. An example dependency relationship definition is shown in Figure

```
<dependency>
   <dependent_parameter>multiplier</dependent_parameter>
   <independent_parameter>data_width</independent_parameter>
   <if data_width = "-">multiplier = -</if>
   <if data_width = "16">multiplier = software</if>
   <if multiplier = "software">data_width = 32</if>
   <if multiplier = "MSTEP">data_width = 32</if>
   <if multiplier = "MUL">data_width = 32</if>
</dependency>
```

Figure A.7: Example of a Dependency Relationship Definition

Table A.1: Dependency Lookup Tables for Data Width and Multiplier Parameters

Data Width	Multiplier
–	–
16-bit	Software
32-bit	Software, MSTEP, MUL

Multiplier	Data Width
–	–
Software	16, 32-bit
MSTEP	32-bit
MUL	32-bit

A.7.

In this example, the dependent and independent parameters are both listed. The `<if>` statements define the elements of the dependency table. Internal to SCBuild, the dependency table is actually stored as two separate one-way lookup tables—one for the dependent parameter and the other for the independent parameter. These are depicted in Table A.1.

The first entry in each of these tables is the default null '–' value. Each System-level parameter, regardless of whether or not it has any dependency relationships, has this value as one of its possible values. When a parameter is set to the null value, it essentially means that the parameter is "unset". The remaining entries in the tables contain the valid values of the second parameter for each possible value of the first parameter.

```
<objective>
   <name>area</name>
   <improvement>smaller_better</improvement>
   <const_coef>10.1</const_coef>

   <term>
      <parameter>data_width</parameter>
      <function>data_width**2</function>
      <coefficient>2.5</coefficient>
   </term>
   ...
</objective>
```

Figure A.8: An Example of an Objective Estimation Equation Definition in the Objectives File

A.4 The Objectives File

The Objectives file stores the declarations for each objective estimation equation that will be evaluated during design space exploration. Expressed in its most general form, each objective estimation equation has the form given in equation (3.3). A declaration in the Objectives file includes information on the functional form $f_{i,k}(p_i)$ and the regression coefficient $a_{i,k}$ for each term of every objective function. An example of an objective estimation equation definition is shown in Figure A.8.

Each objective is given a unique name (e.g. "area"). The type of improvement is specified (either "smaller_better" or "larger_better"), because some objectives improve when their values decrease, while others improve when they increase. The `<const_coef>` element stores the value of the $a_{0,k}$ coefficient from equation (3.3). Information on each term of the objective estimation equation is stored in the `<term>` sections of the file. The `<parameter>` field specifies which System-level parameter the variable p_i in the term corresponds to. The `<function>` field provides the form of function $f_{i,k}(p_i)$ in equation (3.3) and the `<coefficient>` element stores the $a_{i,k}$ coefficient for each term. Each objective estimation equation can have any number

```
<system>
   <template_files>
      <file>file1.xml</file>
      <file>file2.xml</file>
      ...
   </template_files>
   <objectives_file>objectives.xml</objectives_file>
   <dependencies_file>dependencies.xml</dependencies_file>
</system>
```

Figure A.9: An Example of an System File Listing

of terms as long as the system has the corresponding set of parameters.

A.5 The System File

The System file stores the names of all of the template component files, as well as the names of the Parameter Dependencies file and the Objectives file for a given core in a single location. When users of SCBuild wish to load a particular template description, then they must provide the program with the name of the System file. An example System file listing is shown in Figure A.9. Under the `<template_files>` section, the names of all of the XML template component files are listed. All of the template component files must be listed in this section in order for SCBuild to recognize them. The `<objectives_file>` element stores the Objectives file name, and the `<dependencies_file>` element contains the name of the Parameter Dependencies file.

Appendix B

Description of the RISC Processor Template

The parameterized RISC processor template used in this research is a modified version of the pipelined RISC CPU presented by Mano and Kime [48]. The following is a brief description of the processor template model used during this research. See Mano and Kime's book for a more detailed discussion of the processor's design.

B.1 Parameters

The RISC processor template features the parameters listed in Table B.1. The Data Width of the processor can be set to either 8, 16, 32 or 64 bits and both the Data and Instruction Address Widths can be varied between 5 and 15 bits, giving anywhere between 32 and 32,768 words of data and instruction memory, respectively. The ALU can be configured with or without a full combinational hardware multiplier using the

Table B.1: RISC Processor Hardware Parameters

Parameter	**Possible Values**
ALU Adder Implementation (p_1)	(1) Ripple-carry, (2) Carry-lookahead
Arithmetic Shifter Implementation (p_2)	(1) None, (2) Basic, (3) Barrel
Branch Adder Implementation (p_3)	(1) Ripple-carry, (2) Carry-lookahead
Data Address Width (p_4)	(1–11) 5 to 15 bits
Data Width (p_5)	(1) 8, (2) 16, (3) 32, (4) 64 bits
Include Multiplier (p_6)	(1) False, (2) True
Instruction Address Width (p_7)	(1–11) 5 to 15 bits
Logical Shifter Implementation (p_8)	(1) None, (2) Basic, (3) Barrel
Operand Width (p_9)	(1–8) 2 to 9 bits
Pipelined (p_{10})	(1) False, (2) True
Rotator Implementation (p_{11})	(1) None, (2) Basic, (3) Barrel

Include Multiplier parameter. Both the and the ALU adder and the branch adder can be implemented using either a ripple-carry or a carry-lookahead structure. Three different types of shifters are also available: an arithmetic shifter, a logical shifter, and a rotator. Any or all of these shifters can be optionally included with the processor, and each can be implemented as "basic" shifter (allowing a shift of only one position per clock cycle), or as a barrel shifter (enabling the shifting of operands multiple positions in a single clock cycle). Finally, pipelined and unpipelined variants of the processor can be generated by setting the value of the Pipelined parameter to "true" and "false" respectively.

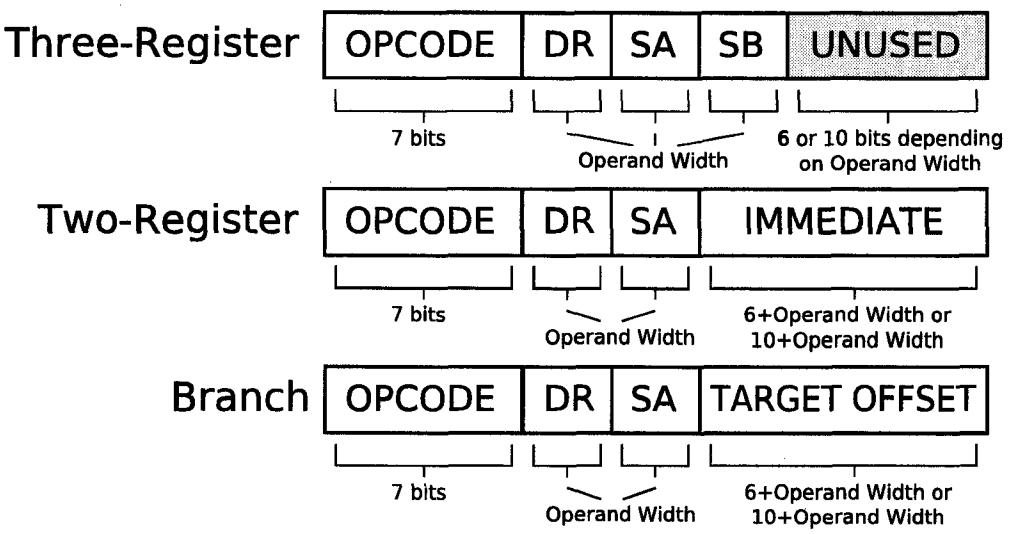


Figure B.1: Instruction Formats for the RISC Processor

B.2 Instruction Set

The RISC processor has three different instruction formats: three-register, two-register, and branch. These formats are illustrated in Figure B.1. In all cases, the opcode is 7 bits wide. The widths of the operand fields, DR, SA and SB, all depend directly on the value of the Operand Width parameter. Additionally, the widths of the Unused, Immediate and Target Offset fields depend on the Operand Width parameter as well, only indirectly. If the Operand Width is less than or equal to 7 bits, then the Unused field is 10 bits wide, and if it is greater than 7 bits, then the Unused field is only 6 bits wide. The core was designed this way so that configurations utilizing large amounts of instruction memory with wide operand widths would fit onto a Stratix EP1S40F780C5 FPGA [8].

The instruction set of the RISC processor features 38 different instructions for performing arithmetic, logic, shift, branch, and memory operations. It also features one "trap" instruction, which asserts a trap output signal, which can be used to indicate the end of program execution or other exceptional conditions. Table B.2

lists the instructions supported by the RISC processor.

Table B.2: RISC Processor Instructions

Opcode	Symbol	Format	Description
0000000	NOP	3-Reg	No Operation
0000001	ADD	3-Reg	Add register
0000010	SUB	3-Reg	Subtract Register
0000011	SLT	3-Reg	Set Less Than
0000100	MLL	3-Reg	Multiply Low
0000100	MLH	3-Reg	Multiply High
0000101	MLI	2-Reg	Multiply Low Immediate
0000110	MHI	2-Reg	Multiply High Immediate
0001000	INC	3-Reg	Increment
0001001	DEC	3-Reg	Decrement
0001010	LSL	2-Reg	Logical Shift Left
0001011	LSR	2-Reg	Logical Shift Right
0001100	ASL	2-Reg	Arithmetic Shift Left
0001101	ASR	2-Reg	Arithmetic Shift Right
0001110	ROL	2-Reg	Rotate Left
0001111	ROR	2-Reg	Rotate Right
0010000	AND	3-Reg	Bitwise AND
0010001	OR	3-Reg	Bitwise OR
0010010	XOR	3-Reg	Bitwise XOR
0010011	NOT	3-Reg	Bitwise Complement
0010100	ST	3-Reg	Data Memory Store
0010101	LD	3-Reg	Data Memory Load
0010110	ADI	2-Reg	Add Signed Immediate
0010111	SBI	2-Reg	Subtract Signed Immediate
0011000	ANI	2-Reg	AND Immediate
0011001	ORI	2-Reg	OR Immediate
0011010	XOR	2-Reg	XOR Immediate
0011011	AIU	2-Reg	Add Unsigned Immediate
0011100	SIU	2-Reg	Subtract Unsigned Immediate
0011101	MOV	3-Reg	Move
0011110	JMR	Branch	Jump Register
0011111	BZ	Branch	Branch on Zero
0100000	BNZ	Branch	Branch on Not Zero
0100001	JMP	Branch	Jump

Continued on next page ...

Table B.2 – continued from previous page

Opcode	Symbol	Format	Description
0100010	JML	Branch	Jump and Link
0100011	IMP	3-Reg	Increment Memory Page
0100100	DMP	3-Reg	Decrement Memory Page
0100101	TRP	3-Reg	Trap

B.3 Structure

A block diagram of the RISC processor is shown in Figure B.2. A brief description of each of the processor's components is provided here. The processor consists of four major components: the Datapath, Control Unit, Instruction Memory and Data Random Access Memory (RAM). The word size of the processor can be set to 8, 16, 32 or 64 bits using the Data Width parameter, and the number of addressable locations in the Data RAM and Instruction Memory can be set to any value between 32 to 32,768 words by setting the Data Address Width and Instruction Address Width parameters respectively.

If the Data Width is equal to the Data Address Width, then the Datapath's "Data Address Out" port is connected directly to the Data RAM's "Address" port. However, if the values of these two parameters are unequal, then one of two different components may be instantiated in order to interface the Datapath with the Data RAM. If the value of the Data Width parameter is larger than that of the Data Address Width parameter, then a Bus Interface component is used to connect the "Data Address Out" port on the Datapath with the "Address" port on the Data RAM. The Bus Interface component simply connects two buses of unequal width together, leaving the extra input signals open. If the Data Width is less than the Data Address Width, then a Memory Controller component is instantiated instead. The purpose of this component is to make all of the space in the Data RAM addressable through the use

of "memory pages". Since the width of the Data Address Out port on the Datapath is less than the width of the Address port on the Data RAM, not all of the words in the Data RAM are addressable directly. Therefore, the Data RAM is divided into $2^{(data\_address\_width - data\_width)}$ memory *pages*, with each page containing 2^{data_width} words. Only one memory page is addressable at a time. The Memory Controller contains a *page register* which points to the current page of memory being accessed. The IMP and DMP instructions are used to increment and decrement the page register so that the entire space of data memory can be addressed.

Pipelined and unpipelined variants of the processor can be generated by setting the Pipelined parameter to "true" and "false" respectively. The pipelined version of the processor contains the following four pipeline stages: Instruction Fetch (IF), Decode and Operand Fetch (DOF), Execute (EX), and Write Back (WB). Currently, there is no additional hardware for handling data and control hazards in the pipeline. Hazards must be handled in software by inserting NOPs in between instructions in a program.

B.3.1 Datapath

The Datapath component handles all of the data processing operations performed by the processor. See Figure B.3 for a block diagram of the Datapath. The two major components of the Datapath are the Register File and the Function Unit. If the Pipelined parameter is set to a value of "true", then two pipeline registers are created as well: DOF/EX and EX/WB.

The number of general-purpose registers in the Register File can be controlled using the Operand Width parameter and is equal to $2^{operand\_width}$. The first register, R0, always contains a value of 0, and writes to this register are invalid. The remaining registers can be used for any purpose.

The Function Unit contains the logic necessary to perform arithmetic, logical and

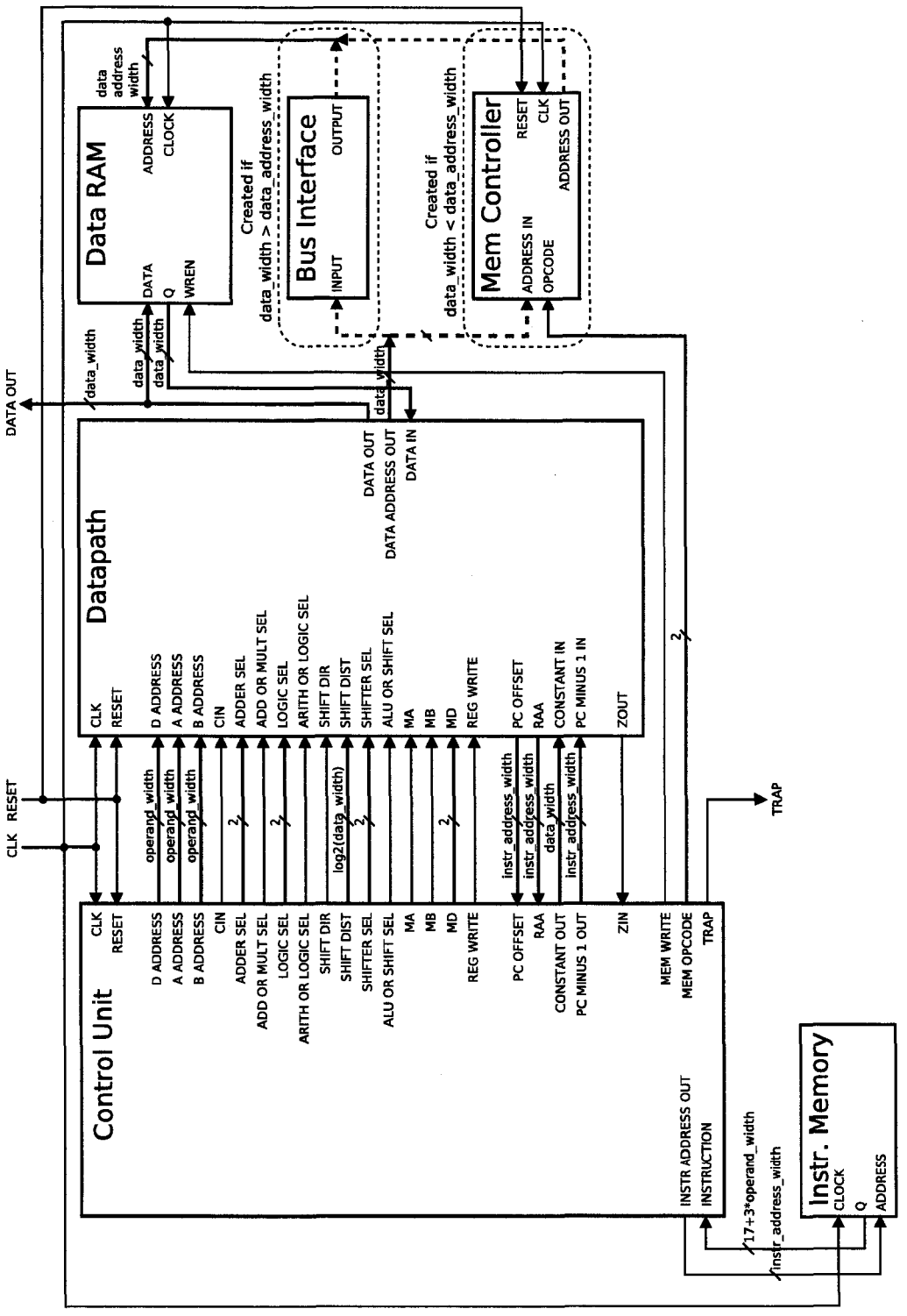


Figure B.2: RISC Processor Block Diagram

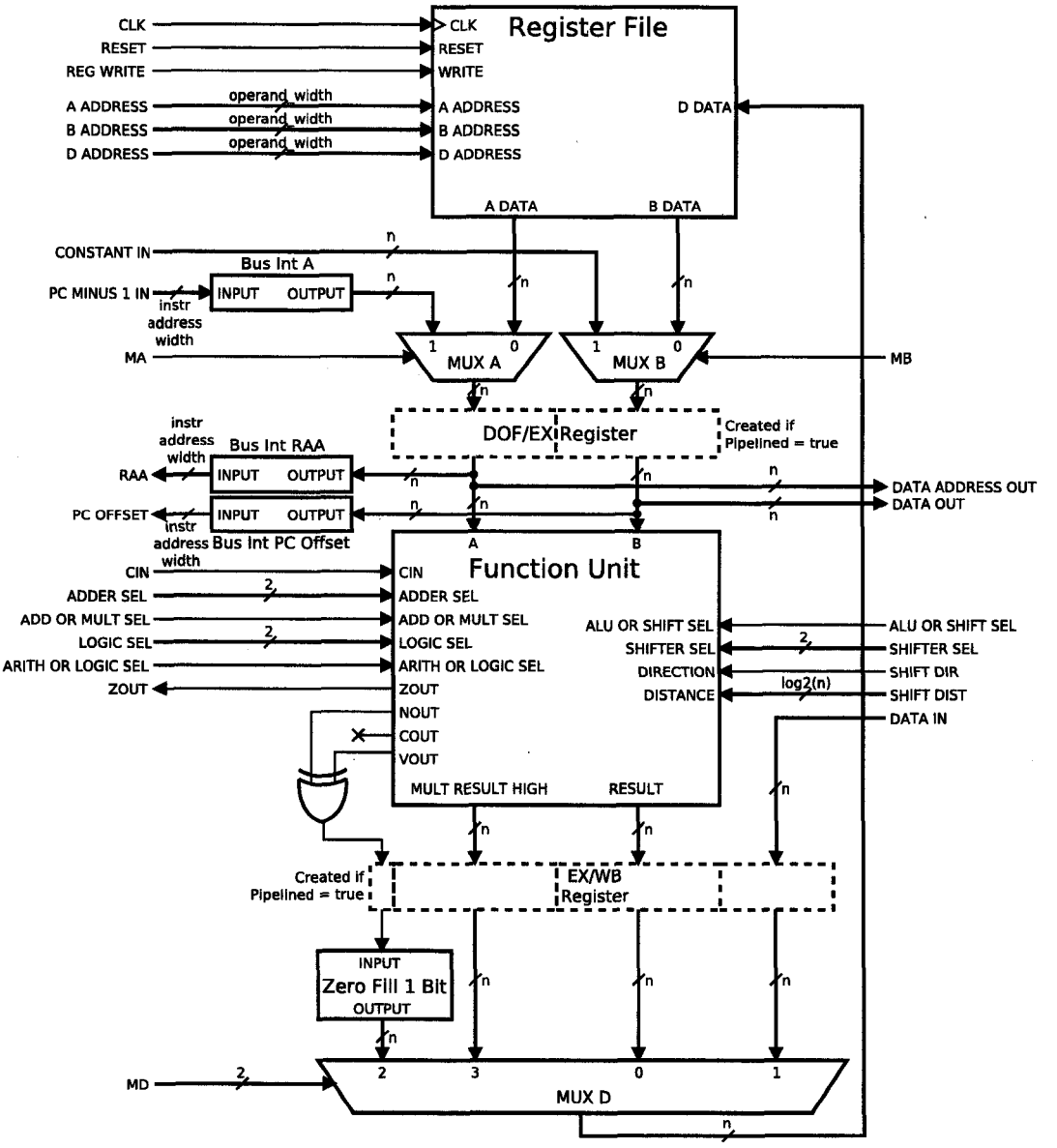


Figure B.3: Datapath Block Diagram

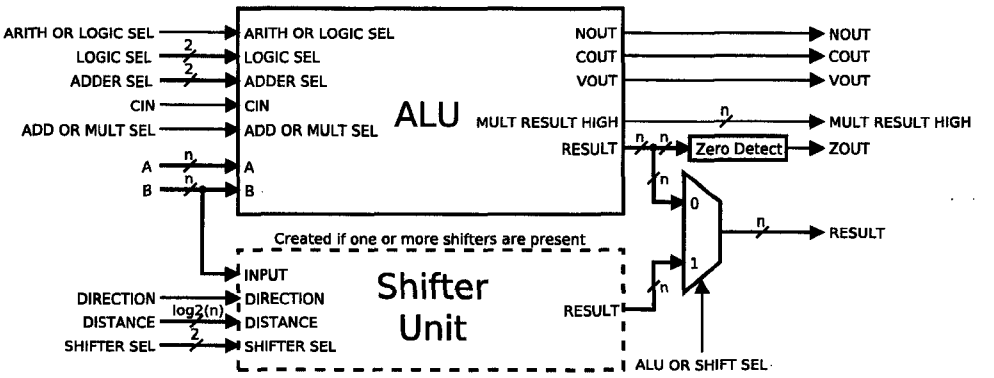


Figure B.4: Function Unit Block Diagram

shift operations on data stored in the general-purpose registers. A logic block diagram for the Function Unit is given in Figure B.4. The Function Unit consists of the ALU, the Shifter Unit and a Zero Detect circuit. The ALU performs arithmetic and logical operations on integer data, and can be configured with or without hardware multiplication using the Include Multiplier parameter. In addition, the ALU's adder can be implemented using a either a ripple-carry or a carry-lookahead structure by setting the value of the ALU Adder Implementation parameter.

The Shifter Unit can be configured to optionally handle the arithmetic shift, logical shift and rotation operations. The Arithmetic Shifter Implementation, Logical Shifter Implementation and Rotator Implementation parameters control which shifters are included in the Shifter Unit, and whether their implementations will be "basic" or "barrel".

B.3.2 Control Unit

The Control Unit determines which operations the Datapath will perform by fetching instructions from the Instruction Memory and decoding them. The block diagram of the Control Unit is shown in Figure B.5. There are two configurable features on the Control Unit. First, the IR, PC Minus 1, PC Minus 2, DOF/EX and EX/WB

pipeline registers are added to the unit when the value of the Pipelined parameter is set to "true". Second the implementation of the Adder can be set to either a ripple-carry or carry-lookahead structure by configuring the Branch Adder Implementation Parameter.

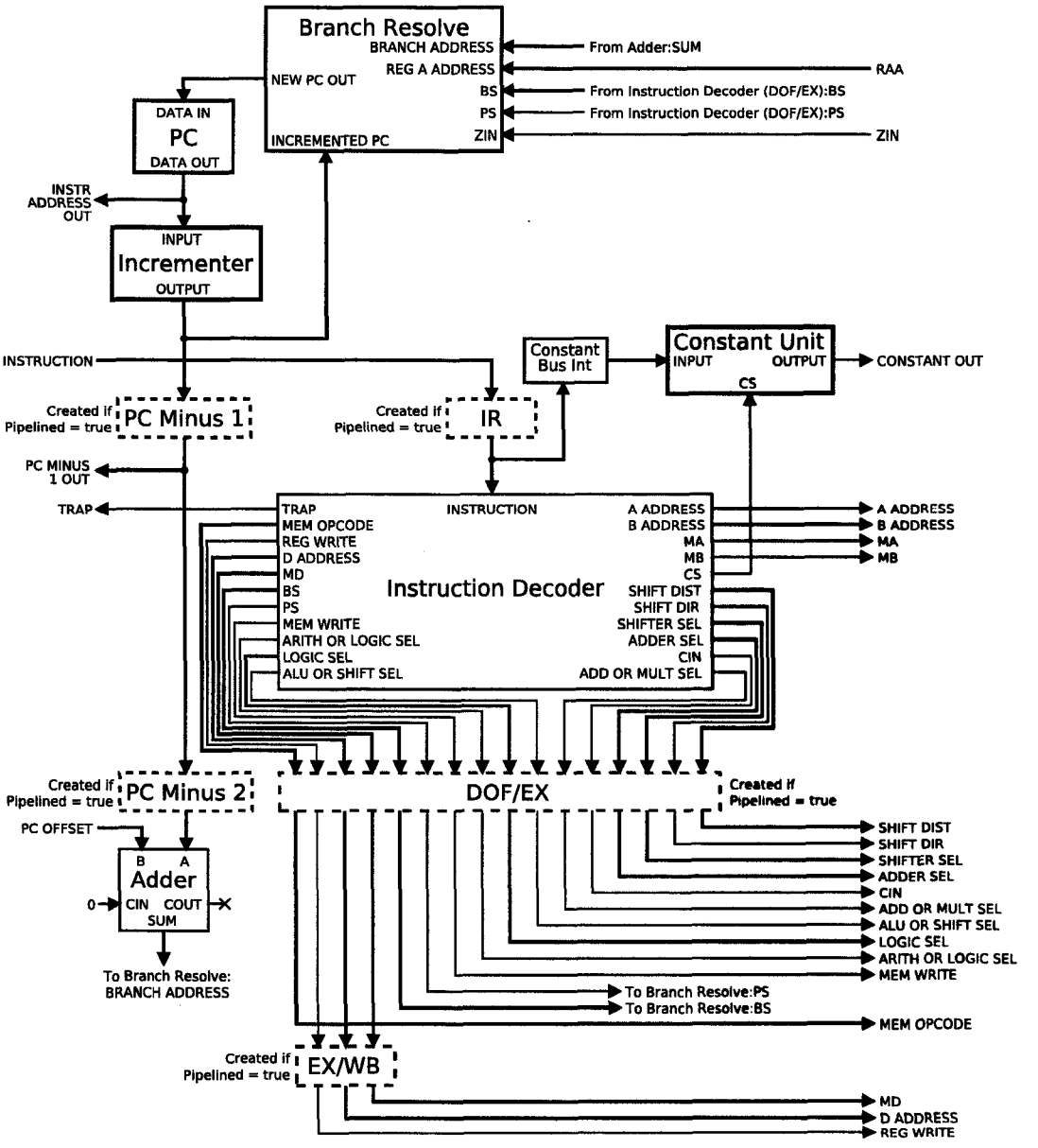


Figure B.5: Control Unit Block Diagram

Appendix C

Synthesis Results for the RISC Processor Template

C.1 Parameter Sweep Results

Table C.1: Parameter Sweep Data

Config.	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	clk (ns)	Eq. LEs
0	1	1	1	1	1	1	1	1	1	1	1	21.2	234.5
1	2	1	1	1	1	1	1	1	1	1	1	18.1	236.5
2	1	2	1	1	1	1	1	1	1	1	1	21.3	244.5
3	1	3	1	1	1	1	1	1	1	1	1	21.6	295.5
4	1	1	2	1	1	1	1	1	1	1	1	20.6	235.5
5	1	1	1	2	1	1	1	1	1	1	1	22.1	234.5

Continued on next page ...

Table C.1 – continued from previous page

Config.	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	clk (ns)	Eq. LEs
6	1	1	1	3	1	1	1	1	1	1	1	19.4	261.8
7	1	1	1	4	1	1	1	1	1	1	1	20.8	261.8
8	1	1	1	5	1	1	1	1	1	1	1	19.7	261.8
9	1	1	1	6	1	1	1	1	1	1	1	19.2	311.6
10	1	1	1	7	1	1	1	1	1	1	1	21.3	408.2
11	1	1	1	8	1	1	1	1	1	1	1	19.7	601.4
12	1	1	1	9	1	1	1	1	1	1	1	20.1	1770.3
13	1	1	1	10	1	1	1	1	1	1	1	19.6	1771.3
14	1	1	1	11	1	1	1	1	1	1	1	19.2	1772.3
15	1	1	1	1	2	1	1	1	1	1	1	24.8	338.5
16	1	1	1	1	3	1	1	1	1	1	1	33.0	552.8
17	1	1	1	1	4	1	1	1	1	1	1	50.8	977.6
18	1	1	1	1	1	2	1	1	1	1	1	23.6	268.87
19	1	1	1	1	1	1	2	1	1	1	1	20.6	260
20	1	1	1	1	1	1	3	1	1	1	1	20.4	323.5
21	1	1	1	1	1	1	4	1	1	1	1	20.4	288.3
22	1	1	1	1	1	1	5	1	1	1	1	18.2	361.6
23	1	1	1	1	1	1	6	1	1	1	1	21.1	495.3
24	1	1	1	1	1	1	7	1	1	1	1	20.3	737.3
25	1	1	1	1	1	1	8	1	1	1	1	22.6	1218.3
26	1	1	1	1	1	1	9	1	1	1	1	25.8	2268.1
27	1	1	1	1	1	1	10	1	1	1	1	21.5	4282.7
28	1	1	1	1	1	1	11	1	1	1	1	26.2	8399.9

Continued on next page ...

Table C.1 – continued from previous page

Config.	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	clk (ns)	Eq. LEs
29	1	1	1	1	1	1	1	2	1	1	1	22.0	253.5
30	1	1	1	1	1	1	1	3	1	1	1	18.4	290.5
31	1	1	1	1	1	1	1	1	2	1	1	23.2	286.5
32	1	1	1	1	1	1	1	1	3	1	1	25.4	397.5
33	1	1	1	1	1	1	1	1	4	1	1	23.1	589.5
34	1	1	1	1	1	1	1	1	5	1	1	26.1	986.5
35	1	1	1	1	1	1	1	1	6	1	1	27.9	1805.5
36	1	1	1	1	1	1	1	1	7	1	1	28.7	3456
37	1	1	1	1	1	1	1	1	8	1	1	39.5	6638
38	1	1	1	1	1	1	1	1	1	2	1	9.5	249.3
39	1	1	1	1	1	1	1	1	1	1	2	18.6	244.5
40	1	1	1	1	1	1	1	1	1	1	3	20.7	292.5
41	2	1	1	1	2	1	1	1	1	1	1	23.3	336.5
42	1	2	1	1	2	1	1	1	1	1	1	25.0	352.5
43	1	3	1	1	2	1	1	1	1	1	1	23.8	477.5
44	1	1	2	1	2	1	1	1	1	1	1	25.2	339.5
45	1	1	1	2	2	1	1	1	1	1	1	24.3	366.8
46	1	1	1	3	2	1	1	1	1	1	1	25.0	366.8
47	1	1	1	4	2	1	1	1	1	1	1	23.1	366.8
48	1	1	1	5	2	1	1	1	1	1	1	24.6	414.6
49	1	1	1	6	2	1	1	1	1	1	1	24.0	510.2
50	1	1	1	7	2	1	1	1	1	1	1	24.2	701.4
51	1	1	1	8	2	1	1	1	1	1	1	27.8	1083.8

Continued on next page ...

Table C.1 – continued from previous page

Config.	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	clk (ns)	Eq. LEs
52	1	1	1	9	2	1	1	1	1	1	1	24.1	1869.3
53	1	1	1	10	2	1	1	1	1	1	1	22.8	1869.3
54	1	1	1	11	2	1	1	1	1	1	1	24.3	1869.3
55	1	1	1	1	2	2	1	1	1	1	1	28.9	407.24
56	1	1	1	1	2	1	2	1	1	1	1	22.8	360
57	1	1	1	1	2	1	3	1	1	1	1	24.4	423.5
58	1	1	1	1	2	1	4	1	1	1	1	24.0	414.8
59	1	1	1	1	2	1	5	1	1	1	1	25.1	506.6
60	1	1	1	1	2	1	6	1	1	1	1	24.2	596.3
61	1	1	1	1	2	1	7	1	1	1	1	25.1	886.1
62	1	1	1	1	2	1	8	1	1	1	1	27.1	1416.9
63	1	1	1	1	2	1	9	1	1	1	1	34.3	2554.3
64	1	1	1	1	2	1	10	1	1	1	1	30.1	4782.1
65	1	1	1	1	2	1	11	1	1	1	1	-	-
66	1	1	1	1	2	1	1	2	1	1	1	24.1	360.5
67	1	1	1	1	2	1	1	3	1	1	1	27.8	474.5
68	1	1	1	1	2	1	1	1	2	1	1	25.0	430.5
69	1	1	1	1	2	1	1	1	3	1	1	26.1	634.5
70	1	1	1	1	2	1	1	1	4	1	1	28.8	997.5
71	1	1	1	1	2	1	1	1	5	1	1	30.9	1780.5
72	1	1	1	1	2	1	1	1	6	1	1	34.1	3320
73	1	1	1	1	2	1	1	1	7	1	1	41.3	6459
74	1	1	1	1	2	1	1	1	8	1	1	51.5	12837

Continued on next page ...

Table C.1 – continued from previous page

Config.	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	clk (ns)	Eq. LEs
75	1	1	1	1	2	1	1	1	1	2	1	13.3	347.3
76	1	1	1	1	2	1	1	1	1	1	2	27.5	348.5
77	1	1	1	1	2	1	1	1	1	1	3	25.0	480.5
78	2	1	1	1	3	1	1	1	1	1	1	31.1	553.8
79	1	2	1	1	3	1	1	1	1	1	1	33.4	574.8
80	1	3	1	1	3	1	1	1	1	1	1	36.6	877.8
81	1	1	2	1	3	1	1	1	1	1	1	34.1	553.8
82	1	1	1	2	3	1	1	1	1	1	1	31.6	552.8
83	1	1	1	3	3	1	1	1	1	1	1	30.0	552.8
84	1	1	1	4	3	1	1	1	1	1	1	30.0	600.6
85	1	1	1	5	3	1	1	1	1	1	1	30.2	696.2
86	1	1	1	6	3	1	1	1	1	1	1	30.5	887.4
87	1	1	1	7	3	1	1	1	1	1	1	32.4	1269.8
88	1	1	1	8	3	1	1	1	1	1	1	32.7	2034.6
89	1	1	1	9	3	1	1	1	1	1	1	33.6	2055.3
90	1	1	1	10	3	1	1	1	1	1	1	30.7	2055.3
91	1	1	1	11	3	1	1	1	1	1	1	30.1	3605.6
92	1	1	1	1	3	2	1	1	1	1	1	35.2	788.76
93	1	1	1	1	3	1	2	1	1	1	1	35.7	574.3
94	1	1	1	1	3	1	3	1	1	1	1	34.6	638.8
95	1	1	1	1	3	1	4	1	1	1	1	34.0	628.1
96	1	1	1	1	3	1	5	1	1	1	1	34.2	718.9
97	1	1	1	1	3	1	6	1	1	1	1	36.1	811.6

Continued on next page ...

Table C.1 – continued from previous page

Config.	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	clk (ns)	Eq. LEs
98	1	1	1	1	3	1	7	1	1	1	1	32.5	1101.4
99	1	1	1	1	3	1	8	1	1	1	1	34.4	1629.2
100	1	1	1	1	3	1	9	1	1	1	1	42.2	2770.6
101	1	1	1	1	3	1	10	1	1	1	1	39.9	4995.4
102	1	1	1	1	3	1	11	1	1	1	1	-	-
103	1	1	1	1	3	1	1	2	1	1	1	33.8	574.8
104	1	1	1	1	3	1	1	3	1	1	1	32.8	874.8
105	1	1	1	1	3	1	1	1	2	1	1	40.1	728.8
106	1	1	1	1	3	1	1	1	3	1	1	39.0	1110.8
107	1	1	1	1	3	1	1	1	4	1	1	37.1	1804.8
108	1	1	1	1	3	1	1	1	5	1	1	44.9	3324.8
109	1	1	1	1	3	1	1	1	6	1	1	45.2	6350.3
110	1	1	1	1	3	1	1	1	7	1	1	50.0	12502.3
111	1	1	1	1	3	1	1	1	8	1	1	-	-
112	1	1	1	1	3	1	1	1	1	2	1	20.8	555.6
113	1	1	1	1	3	1	1	1	1	1	2	33.9	574.8
114	1	1	1	1	3	1	1	1	1	1	3	32.0	907.8
115	2	1	1	1	4	1	1	1	1	1	1	48.6	977.6
116	1	2	1	1	4	1	1	1	1	1	1	52.2	1020.6
117	1	3	1	1	4	1	1	1	1	1	1	51.3	1735.6
118	1	1	2	1	4	1	1	1	1	1	1	49.1	978.6
119	1	1	1	2	4	1	1	1	1	1	1	53.1	979.6
120	1	1	1	3	4	1	1	1	1	1	1	47.0	979.6

Continued on next page ...

Table C.1 – continued from previous page

Config.	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	clk (ns)	Eq. LEs
121	1	1	1	4	4	1	1	1	1	1	1	51.3	1075.2
122	1	1	1	5	4	1	1	1	1	1	1	50.7	1239.1
123	1	1	1	6	4	1	1	1	1	1	1	51.4	1648.8
124	1	1	1	7	4	1	1	1	1	1	1	45.6	2413.6
125	1	1	1	8	4	1	1	1	1	1	1	52.9	2435.3
126	1	1	1	9	4	1	1	1	1	1	1	50.5	2434.3
127	1	1	1	10	4	1	1	1	1	1	1	50.0	3984.6
128	1	1	1	11	4	1	1	1	1	1	1	50.7	7085.2
129	1	1	1	1	4	2	1	1	1	1	1	62.8	2036.44
130	1	1	1	1	4	1	2	1	1	1	1	56.4	997.1
131	1	1	1	1	4	1	3	1	1	1	1	52.4	1066.6
132	1	1	1	1	4	1	4	1	1	1	1	53.3	1052.9
133	1	1	1	1	4	1	5	1	1	1	1	49.8	1142.7
134	1	1	1	1	4	1	6	1	1	1	1	51.7	1234.4
135	1	1	1	1	4	1	7	1	1	1	1	56.9	1525.2
136	1	1	1	1	4	1	8	1	1	1	1	53.9	2054
137	1	1	1	1	4	1	9	1	1	1	1	60.9	3193.4
138	1	1	1	1	4	1	10	1	1	1	1	59.8	5421.2
139	1	1	1	1	4	1	11	1	1	1	1	-	-
140	1	1	1	1	4	1	1	2	1	1	1	53.6	1019.6
141	1	1	1	1	4	1	1	3	1	1	1	51.0	1722.6
142	1	1	1	1	4	1	1	1	2	1	1	51.8	1312.6
143	1	1	1	1	4	1	1	1	3	1	1	52.4	2065.6

Continued on next page ...

Table C.1 – continued from previous page

Config.	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	clk (ns)	Eq. LEs
144	1	1	1	1	4	1	1	1	4	1	1	59.0	3410.6
145	1	1	1	1	4	1	1	1	5	1	1	61.6	6454.6
146	1	1	1	1	4	1	1	1	6	1	1	59.4	12437.1
147	1	1	1	1	4	1	1	1	7	1	1	-	-
148	1	1	1	1	4	1	1	1	8	1	1	-	-
149	1	1	1	1	4	1	1	1	1	2	1	35.9	967.4
150	1	1	1	1	4	1	1	1	1	1	2	51.4	1018.6
151	1	1	1	1	4	1	1	1	1	1	3	49.7	1815.6

C.2 Initial and Evolved Populations

C.2.1 Initial Population

Table C.2: Data for Initial Population

Config.	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	clk (ns)	Eq. LEs
0	2	3	2	1	1	2	5	3	5	2	2	17.713	1324.07
1	1	3	2	5	2	2	9	2	8	1	3	59.394	16935.34
2	2	3	1	8	3	2	7	2	6	2	1	26.643	9430.76
3	2	1	1	7	4	2	7	1	7	1	1	-	-
4	1	3	1	11	4	1	7	1	3	2	3	42.58	9834.2
5	2	2	2	5	2	1	2	3	8	2	3	35.777	12971.9
6	2	1	2	4	3	2	7	1	2	2	3	27.452	1958.96
7	1	3	2	10	3	1	4	1	5	2	1	23.509	5281.9

Continued on next page ...

Table C.2 – continued from previous page

Config.	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	clk (ns)	Eq. LEs
8	1	2	1	11	2	1	8	3	7	1	3	40.189	9951.9
9	1	2	1	5	3	1	1	1	1	2	3	21.539	1125
10	1	3	1	3	2	2	9	3	5	1	2	38.381	4952.94
11	2	2	2	1	4	2	8	2	8	2	2	-	-
12	1	3	2	8	2	1	6	2	2	1	1	29.923	1675.4
13	1	2	2	2	4	2	11	3	7	1	2	-	-
14	1	2	1	5	2	1	9	1	5	1	3	37.343	4866
15	2	1	2	10	2	2	5	1	2	1	1	28.675	2153.44
16	1	3	1	2	1	2	8	2	1	2	2	12.265	1352.67
17	1	1	1	6	4	2	11	1	3	1	3	-	-
18	1	1	1	7	1	1	2	1	4	1	2	25.283	823.2
19	2	2	2	6	2	1	4	1	7	1	2	42.068	6825.3
20	1	2	1	2	3	2	8	1	4	1	1	44.284	3623.36
21	1	2	1	11	2	1	11	1	4	1	3	-	-
22	1	3	1	3	2	2	8	3	3	1	2	30.974	2319.74
23	2	2	2	11	3	2	3	1	6	2	1	22.758	9797.86
24	2	2	2	1	2	1	10	1	8	1	1	33.145	22222.5
25	1	2	1	1	1	1	9	3	7	1	3	38.477	7138.7
26	1	3	2	10	3	2	2	3	2	1	3	40.073	3100.76
27	1	1	2	10	1	1	3	2	6	2	3	15.794	3481.1
28	1	1	1	5	1	1	6	3	4	2	1	11.624	1076.2
29	2	1	2	10	3	2	7	1	5	1	3	49.979	6298.66
30	2	2	2	3	4	2	11	1	2	2	3	-	-

Continued on next page ...

Table C.2 – continued from previous page

Config.	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	clk (ns)	Eq. LEs
31	1	2	1	8	1	1	9	2	8	1	2	44.489	10996.4
32	1	1	2	4	2	2	10	1	4	1	1	34.451	7324.94
33	1	3	1	5	4	1	4	1	7	1	3	-	-
34	1	3	2	8	1	1	9	2	7	1	1	36.17	7469.6
35	2	3	2	10	3	2	5	2	1	1	3	38.641	2996.36
36	1	3	1	8	4	2	11	1	7	2	1	-	-
37	1	2	1	7	2	1	3	3	5	1	1	35.705	2344.2
38	1	1	1	2	4	1	8	3	5	2	3	49.159	9270.6
39	2	2	2	2	1	1	1	3	3	2	1	11.527	487.3
40	2	3	1	10	1	2	9	3	7	2	3	18.529	8713.87
41	2	3	1	1	2	2	7	3	1	2	2	16.577	1187.84
42	1	1	1	4	3	2	6	2	5	1	3	42.75	4454.76
43	2	1	1	9	1	1	11	3	1	2	1	11.521	9994.7
44	2	1	1	8	3	1	8	1	8	1	3	-	-
45	2	3	1	5	4	2	8	1	1	2	1	45.061	4139.34
46	2	3	2	9	2	1	9	2	4	2	3	16.1	5875.5
47	2	2	1	6	4	1	10	2	6	1	2	68.567	20821.4
48	1	2	1	9	4	1	3	1	3	1	2	58.031	3642.1
49	2	1	1	8	2	1	7	2	6	2	2	18	4968

C.2.2 Evolved Population

Table C.3: Data for Evolved Population

Config.	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	clk (ns)	Eq. LEs
0	2	3	2	1	1	2	1	1	1	2	2	12.534	370.67
1	2	3	1	6	3	2	5	2	2	2	3	26.805	1958.76
2	2	3	1	2	1	2	7	3	1	2	2	12.854	877.67
3	1	3	1	3	2	2	8	2	1	2	2	18.359	1716.94
4	2	2	1	9	4	1	8	1	1	2	1	36.655	3608.7
5	1	1	1	9	4	1	2	1	1	2	1	36.902	2428.1
6	2	2	2	9	1	1	7	2	6	2	2	18.136	4214.7
7	1	1	1	8	2	1	4	2	1	2	2	15.473	1183.4
8	2	1	1	9	2	1	3	1	2	2	1	14.332	1975.1
9	1	2	1	5	3	1	1	1	1	2	3	21.539	1125
10	1	3	1	2	1	2	8	2	1	2	2	12.265	1352.67
11	1	1	1	9	4	1	9	1	1	2	2	39.107	4709.1
12	1	1	1	2	2	1	6	2	2	1	1	30.109	799.4
13	2	3	1	2	1	1	1	3	3	2	1	10.647	480.3
14	1	1	1	7	4	1	5	2	2	2	3	35.758	3806
15	2	1	1	7	1	1	7	2	2	1	3	22.421	1098.8
16	1	3	1	2	1	2	8	2	1	2	2	12.265	1352.67
17	1	1	1	8	2	1	7	2	6	2	2	17.026	4961
18	1	1	1	7	1	1	2	1	4	2	1	11.15	805
19	2	1	1	7	1	1	7	2	2	2	3	9.606	1106.8
20	2	3	2	1	1	1	5	2	2	2	1	11.345	491.9
Continued on next page ...													

Table C.3 – continued from previous page

Config.	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	clk (ns)	Eq. LEs
21	2	1	1	7	1	1	2	1	4	2	1	11.483	807
22	1	3	2	1	1	1	5	2	2	2	1	11.564	487.9
23	2	3	2	1	1	1	5	3	5	2	2	13.736	1282.7
24	2	2	2	1	2	1	10	1	1	2	1	16.212	4823.1
25	2	1	1	8	2	1	2	1	1	2	2	11.731	1120.6
26	2	1	1	7	1	1	2	3	4	2	1	12.719	872
27	1	3	2	1	3	1	5	2	2	2	1	24.835	1226.2
28	1	1	1	5	1	1	6	3	4	2	1	11.624	1076.2
29	1	1	1	2	1	2	7	3	4	2	1	15.463	1416.87
30	1	1	1	7	1	1	7	1	2	2	3	9.783	1079.8
31	2	1	1	9	2	1	7	1	2	1	3	29.034	2717.7
32	2	2	1	11	2	1	7	2	2	2	3	13.656	2754.7
33	2	3	1	8	1	1	9	3	5	2	1	13.732	4652.2
34	1	3	2	8	1	1	9	1	1	2	1	9.66	2702
35	1	1	1	7	1	2	7	1	2	2	3	12.256	1115.17
36	2	2	1	11	2	1	7	1	2	2	3	13.829	2755.7
37	1	2	2	10	1	1	3	2	6	2	3	16.868	3483.1
38	2	1	1	2	4	1	8	3	5	2	3	43.484	9306.6
39	2	2	2	2	1	1	1	3	3	2	1	11.527	487.3
40	1	3	1	10	1	2	1	1	1	2	2	12.153	1907.47
41	2	3	1	1	2	2	1	1	1	2	2	16.538	612.04
42	2	3	1	2	1	2	1	3	3	2	1	11.136	520.67
43	2	1	1	9	1	1	11	3	1	2	1	11.521	9994.7

Continued on next page ...

Table C.3 – continued from previous page

Config.	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	clk (ns)	Eq. LEs
44	2	1	1	7	1	1	7	1	2	2	3	10.551	1080.8
45	2	3	1	5	4	1	4	1	1	2	2	39.727	2209.7
46	1	1	1	8	2	1	4	3	3	2	1	13.264	1578.4
47	1	1	1	8	2	1	2	1	1	2	2	14.161	1115.6
48	1	1	1	8	2	1	4	1	1	2	2	12.362	1175.4
49	2	1	1	3	2	1	7	2	6	2	1	17.188	4250

References

[1] Code::Blocks IDE - Open source, cross-platform free C++ IDE. http://www.codeblocks.org/, January 2007.

[2] OpenRISC 1200. http://www.opencores.org/projects.cgi/web/or1k/openrisc_1200, January 2007.

[3] Altera Corporation. *Avalon Bus Specification Reference Manual. Version 2.3*, July 2003.

[4] Altera Corporation. *Nios Development Board Reference Manual Stratix Professional Edition*, July 2003.

[5] Altera Corporation. *Nios Embedded Processor 32-bit Programmer's Reference Manual Version 3.1*, January 2003.

[6] Altera Corporation. *Nios Embedded Processor 16-bit Programmer's Reference Manual Version 3.1*, January 2004.

[7] Altera Corporation. *Quartus II Version 5.0 Handbook. Version 5.0.0.* May 2005.

[8] Altera Corporation. *Stratix Device Handbook*, July 2005.

[9] I. D. L. Anderson and M. A. S. Khalid. Design space exploration using parameterized cores: A case study. In *Proc. of the Canadian Conference on Electrical and Computer Engineering (CCECE)*, Ottawa, Ontario, Canada, May 2006.

[10] G. Ascia, V. Catania, and M. Palesi. A GA-based design space exploration framework for parameterized system-on-a-chip platforms. *IEEE Transactions on Evolutionary Computation*, 8(4):329–345, August 2004.

[11] C. G. Bell and A. Newell. *Computer Structures: Readings and Examples.* McGraw-Hill, New York, New York, USA, 1971.

[12] Visual C++ Developer Center. http://msdn.microsoft.com/visualc/, January 2007.

[13] 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.

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

[15] XPRES Compiler. http://www.tensilica.com/products/xpres.htm, January 2007.

[16] IBM Microelectronics PowerPC 405 Embedded Cores. http://www-306.ibm.com /chips/techlib/techlib.nsf/products/PowerPC_405_Embedded_Cores, January 2007.

[17] Altera Corporation. Nios 3.0 CPU datasheet, version 2.2. http://www.altera.com /literature/ds/ds_nios_cpu.pdf, October 2004.

[18] Altera Corporation. Quartus II software. http://www.altera.com/products/ software/products/quartus2/qts-index.html, January 2007.

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

[20] Altera Corporation. VHDL: Carry look-ahead adder. http://www.altera.com/ support/examples/vhdl/v_cl_addr.html, January 2007.

[21] Microsoft Corporation. Visual Basic 6.0 resource center. http://msdn2. microsoft.com/en-us/vbrun/default.aspx, January 2007.

[22] Altera Devices. http://www.altera.com/products/devices/dev-index.jsp, January 2007.

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

[24] R. Ernst, J. Henkel, and T. Benner. Hardware-software cosynthesis for microcontrollers. *IEEE Design & Test of Computers*, pages 64–75, December 1993.

[25] M. Fang, M. Jarvin, and L. Ling. http://www.eecg.toronto.edu/~aling/ece1718/ project/Main/Welcome.shtml, January 2007.

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

[27] D. W. Franke and M. K. Purvis. Hardware/software codesign: A perspective. In *Proc. of the 13th International Conference on Software Engineering*, pages 344–352, Austin, Texas, USA, May 13–16, 1991.

[28] 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.

[29] 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.

[30] T. Givargis and F. Vahid. Platune: A tuning framework for system-on-a-chip platforms. *IEEE Transactions on Computed Aided Design of Integrated Circuits and Systems*, 21(11):1317–1327, September 2002.

[31] T. Givargis, F. Vahid, and J. Henkel. System-level exploration for pareto-optimal configurations in parameterized system-on-a-chip. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 10(4):416–422, August 2002.

[32] 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.

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

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

[35] 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 2–7, 1982.

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

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

[38] MinGW Home. http://www.mingw.org/, January 2007.

[39] Synopsys Incorporated. Synopsys products: RTL synthesis. http://www.synopsys.com/products/logic/design_compiler.html, January 2007.

[40] Xilinx Incorporated. http://www.xilinx.com/, January 2007.

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

[42] EECS Instructional and & Electronics Groups Homepage at University of California Berkeley. http://inst.eecs.berkeley.edu/, January 2007.

[43] International Electrotechnical Commission (IEC) International Standardization Organization (ISO). International standard: Programming languages – C++, ISO/IEC 14882:1998, 1998.

[44] 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 12000 International Conference on Computer Design, 2000*, pages 430–436, Austin, Texas, USA, September 2000.

[45] A. A. Jerraya. Long term trends for embedded system design. In *Proc. EUROMICRO Systems Digital System Design (DSD'04)*, pages 20–26, Rennes, France, August 31–September 3, 2004.

[46] 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.

[47] T. C. Lethbridge and R. Laganière. *Object-Oriented Software Engineering.* McGraw Hill, Glasgow, Scotland, 2001.

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

[49] 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.

[50] M. C. McFarland, A. C. Parker, and R. Camposano. The high-level synthesis of digital systems. 78(2):301–318, February 1990.

[51] Altera Megafunctions. http://www.altera.com/products/ip/altera/mega.html, January 2007.

[52] G. De Micheli and R. K. Gupta. Hardware/software co-design. *Proc. of the IEEE*, 85(3):349–365, March 1997.

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

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

[55] Institute of Electrical and Electronics Engineers. IEEE standard for Verilog hardware description language, IEEE 1364-2005, 2006.

[56] Opencores.org. http://www.opencores.org/, January 2007.

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

[58] Xtensa Configurable Processors Overview. http://www.tensilica.com/products/xtensa_overview.htm, January 2007.

[59] M. Palesi and M. Givargis. Multi-objective design space exploration using genetic algorithms. In *Proc. of the Tenth International Symposium on Hardware/Software Codesign, 2002 (CODES 2002)*, pages 67–72, Estes Park, Colorado, USA, May 6–8, 2002.

[60] C. J. Petrie, T. A. Webster, and M. R. Cutkosky. Using pareto-optimality to coordinate distributed agents. http://www-cdr.stanford.edu/NextLink/papers/pareto/pareto.html, January 2007.

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

[62] XiRisc Homepage Qrisc. http://xirisc.deis.unibo.it/, January 2007.

[63] Gaisler Research. http://www.gaisler.com/, January 2007.

[64] 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.

[65] O. Tanir, V. K. Agarwal, and P. C. P. Bhatt. A specification-driven architectural design environment. *Computer*, 28(6):26–35, June 1995.

[66] 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.

[67] J. G. Tong, I. D. L. Anderson, and M. A. S. Khalid. Soft-core processors for embedded systems. In *Proc. of the 18th International Conference on Microelectronics (ICM)*, Dhahran, Saudi Arabia, December 2006.

[68] Object Management Group UML. http://www.uml.org/, January 2007.

[69] 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.

[70] Altera Corporation Website. http://www.altera.com/, January 2007.

[71] Nios II Website. http://www.altera.com/products/ip/processors/nios2/ni2-index.html, January 2007.

[72] Tcl Developer Xchange. http://www.tcl.tk/, January 2007.

[73] Xilinx Incorporated. *MicroBlaze Processor Reference Guide, UG081 (v6.0)*, June 2006.

[74] Extensible Markup Language (XML). http://www.w3.org/XML/, January 2007.

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

[76] 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.

[77] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the strength pareto evolutionary algorithm. Technical report, Computer Engineering and Communication Networks Lab, Swiss Federal Institute of Technology (ETH) Zurich Gloriastrasse, CH-8092, May 2001.

VITA AUCTORIS

Ian D. L. Anderson was born in Winnipeg, Manitoba, Canada on November 27, 1981. He received his B.A.Sc degree in electrical engineering in 2004 from 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 field-programmable technologies and platform-based design for embedded applications.