**Circuit synthesis evolution using a hardware-based
genetic algorithm**

by

Rami Abielmona

B.A.Sc., University of Ottawa

A thesis submitted to the
Faculty of Graduate and Postdoctoral Studies of the
University of Ottawa in partial fulfillment
of the requirements for the degree of
Master's of Applied Science
Ottawa-Carleton Institute of Electrical and Computer Engineering
School of Information Technology and Engineering
©Rami Abielmona, Ottawa, Canada

2002

**National Library
of Canada**

**Acquisitions and
Bibliographic Services**

**395 Wellington Street
Ottawa ON K1A 0N4
Canada**

**Bibliothèque nationale
du Canada**

**Acquisitions et
services bibliographiques**

**395, rue Wellington
Ottawa ON K1A 0N4
Canada**

Your file Votre référence

Our file Notre référence

0-612-65995-X

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

This thesis entitled:
Circuit synthesis evolution using a hardware-based genetic algorithm
written by Rami Abielmona
has been approved for the Ottawa-Carleton Institute of Electrical and Computer
Engineering
School of Information Technology and Engineering
©Rami Abielmona, Ottawa, Canada

Voicu Z. Groza

Nicolas D. Georganas

Dorina C. Petriu

Date _____

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

ABSTRACT

Genetic algorithm Synthesis (GaS) is presented in this thesis. GaS is based on a hardware implementation of a genetic algorithm (GA), aimed at evolving the logic circuit of a defined input function, while minimizing the total number of resources utilized on the underlying hardware platform. A GA bases its operation on society itself, attempting to imitate natural selection in computing systems [1]. It has been found that GAs are very good search techniques, to be used when either the search space is vast or the present deterministic techniques are too restrictive. The field of logic synthesis, as well as technology mapping onto a field-programmable gate array (FPGA), contain both of the aforementioned obstacles, and thus a new method must be realized to overcome these obstacles.

GaS is the integration of a multitude of technologies, and the realization of a solution for the automated synthesis of combinational logic circuits. The work falls under the evolvable hardware (EHW) [2] domain which is a very novel field of research. The presented system is fully functional, and has been prototyped onto a computing platform, which embeds itself into a personal computer, with the main communications path being the local bus. The front-end of GaS consists of a command-line interface, whence the user inputs both the truth table representation of the Boolean function to be evolved, as well as a few initializing parameters. The system then proceeds to evolve a solution for the user's problem.

Experimental results are presented in this thesis. In simple terms, the speedup factors are very promising, and in future versions of the system, the improvements that could be realized will usher in a new computing paradigm: **evolvable computing**.

Dedication

This work is dedicated to those minds that always want more, to the ones whose fires are never extinguished. To you, I say, burn on!

Acknowledgements

I would like to thank my thesis supervisor Dr. Voicu Groza for his ongoing support, and great assistance with this thesis, as well as Dr. Emil Petriu for the thought-stimulating conversations. Their bias towards the classical methodology helped me garner the tools necessary to defend the emerging topic. Special thanks go to my family for all their support during this period. It is always comforting to know that I have people backing me in whatever *flying saucer* endeavors I may partake in.

Xilinx. Incorporated and the Canadian Microelectronics Corporation (CMC) are thanked for their donations of hardware and software, respectively. As well, in particular, heartfelt appreciations go out to Delon Levi and Steven Guccione of Xilinx for their insight into GeneticFPGA, Steve Casselman and John Schewel of VCC for their continued support of the HOT II card, Tim Price and Brandon Blodget of Xilinx for their prompt and explanatory responses concerning JBits, Dr. Adrian Thompson of the University of Sussex for his great humility and mind and Michel Racine and Keith White of the University of Ottawa for their unblemished administrative support.

Contents

Chapter

0.1	List of Acronyms	2
1	Overview and Synopsis	5
1.1	"Towards Evolvable Hardware"	6
1.2	Motivations	10
1.3	Objectives and Contributions	12
1.4	Outline of this Thesis	15
2	Terminology and Chronology	16
2.1	Genetic Algorithms	17
2.1.1	GA: from Roots to Leaves	18
2.1.2	Theory and Application	21
2.2	Field-Programmable Gate Arrays	24
2.2.1	FPGA Interworkings and Applications	27
2.2.2	Victory in Quality, not Quantity	28
2.2.3	The Virtex FPGA: a Younger and Better XC6200	30
2.3	Evolvable Hardware	37
2.3.1	EHW Intricacies	39
2.3.2	What's the Big Deal?	39
2.4	Previous Work in EHW	41

3	State of the Art	45
3.1	Evolvable Computing	45
3.2	Reconfigurable Computing	46
3.3	DNA Computing	46
3.4	Quantum Computing	47
3.5	Pervasive Computing	48
3.5.1	Context-Aware Computing	49
3.6	What if...	50
4	The Defined Problem	52
4.1	Functional Decomposition	52
4.2	Technology Mapping	54
4.3	Our Current Solutions	55
4.3.1	Theoretical Synthesis	55
4.3.2	"Educational" Synthesis	56
4.3.3	"Commercial" Synthesis	57
5	The Proposed Solution	58
5.1	System Architecture	58
5.2	The Guts of GaS	59
5.2.1	An In-Depth Look	64
5.2.2	HIGA: Hardware Implementation of a Genetic Algorithm	65
5.2.3	Memory	74
5.2.4	ECLBs: Evolvable Configuration Logic Blocks	75
5.2.5	Processor Interface	77
5.3	A Look at the System Operation	79
5.4	Novelties and Applications	82
5.4.1	A Brief Example	82

5.4.2	LUT Value and Routing Evolution: Apprehensive Combination .	86
5.4.3	Selection Algorithm: Knights in Battle	87
5.4.4	Chromosome Encoding	88
5.4.5	Evolutionary Stages: Till next time...	88
5.4.6	Hardware cache: hcache	91
6	Investigations, dissections and explorations	96
6.1	Functional	96
6.2	Tascal	97
6.2.1	Boolean Function Evolution	97
6.2.2	Input Parameter Variation	97
6.3	Structural	99
6.3.1	LUT and Routing Resources	100
6.4	Behavioral	100
7	Prototyping GaS	103
7.1	Hardware Environment	103
7.1.1	A Look at the Requirements and Candidates	104
7.1.2	"The Chosen One"	105
7.1.3	Brief Synthesis Details	105
7.2	Software Environment	107
7.2.1	A Look at the Requirements and Candidates	107
7.2.2	"The Chosen One"	108
7.3	H/W to S/W: "Would you marry me?"	109
8	(E)Valuations	110
8.1	Comparison with a Hardware-Based GA	110
8.2	GaSage (GaS usage)	112

9	Limitations, Obstacles and Aspirations	113
9.1	The Battles Won	113
9.1.1	Integration of Technologies	113
9.1.2	Realization of System	115
9.1.3	Proof of Concept	115
9.2	The War Ahead	116
9.2.1	Increased Scalability	116
9.2.2	Parallelization and Pipelining	117
9.2.3	Evolving Synthesizable Processors and Computers	118
10	Closing Comments	120
10.1	Significance of Research	120
10.2	The Previous, Current and Next Steps	121
	Bibliography	123
	Appendix	
A	Hardware Design - ASM Diagrams	128
A.1	Random Number Generator	128
A.2	Selection Module	131
A.3	Mutover Module	136
A.4	Fitness Module	142
A.5	Memory Access Module	147
A.6	Initialization Module	152
B	Hardware Design - VHDL Code	157
B.1	Basic Building Blocks	157

B.1.1	24-bit Enabled Address Decoder	157
B.1.2	16-bit Adder	157
B.1.3	16-bit ALU	157
B.1.4	16-bit Full Adder-Subtractor	157
B.1.5	ALU Top Entity	157
B.1.6	1-bit Comparator	157
B.1.7	2-bit Comparator	158
B.1.8	4-bit Comparator	158
B.1.9	8-bit Comparator	158
B.1.10	10-bit Comparator	158
B.1.11	16-bit Comparator	158
B.1.12	2-bit Down Counter	158
B.1.13	5-bit Down Counter	158
B.1.14	5-bit Up Counter	158
B.1.15	10-bit Down Counter	158
B.1.16	D Flip-Flop with Asynchronous Reset	159
B.1.17	D Flip-Flop with Chip Enable	159
B.1.18	D Flip-Flop with Synchronous Reset	159
B.1.19	5-32 Decoder	159
B.1.20	5-32 Decoder for XMM (1)	159
B.1.21	5-32 Decoder for XMM (2)	159
B.1.22	4-bit Full Adder-Subtractor	159
B.1.23	1-bit Full Adder-Subtractor	159
B.1.24	2-1 Multiplexer	159
B.1.25	4-1 Multiplexer	160
B.1.26	8-1 Multiplexer	160
B.1.27	Variable-Length Data Register	160

B.1.28 Variable-Length Data Register with Common Chip Enable . . .	160
B.1.29 Variable-Length Data Register with Independent Chip Enables .	160
B.1.30 SR Latch	160
B.1.31 Synchronized SR Latch	160
B.2 Pseudorandom Number Generator	160
B.2.1 Control Path	160
B.2.2 Data Path	160
B.2.3 Top Entity	161
B.3 Selection Module	161
B.3.1 Control Path	161
B.3.2 Data Path	161
B.3.3 Top Entity	161
B.4 Mutover Module	161
B.4.1 Control Path	161
B.4.2 Data Path	161
B.4.3 Top Entity	161
B.5 Fitness Module	161
B.5.1 Control Path	161
B.5.2 Data Path	162
B.5.3 Top Entity	162
B.6 Memory Access Module	162
B.6.1 Control Path	162
B.6.2 Data Path	162
B.6.3 Top Entity	162
B.7 Initialization Module	162
B.7.1 Control Path	162
B.7.2 Data Path	162

B.7.3 Top Entity	162
B.8 Genetic Pipeline Module	163
B.8.1 Top Entity	163
B.9 GaS Module	163
B.9.1 Top Entity	163
B.10 Chip Module	163
B.10.1 Top Entity	163
C Software Design - C++/Java Code	164
C.1 Device Driver Code	164
C.1.1 jniSpecialNativeLoader.h	164
C.1.2 jniSpecialNativeLoader.cpp	164
C.1.3 gasProject.cpp	164
C.2 JBits Code	164
C.2.1 SimpleLoader.java	164
C.2.2 SpecialNativeLoader.java	165
C.2.3 SpecialLoader.java	165
C.2.4 EvaluateChromosome.java	165
C.2.5 EvaluateChromosomeInterface.java	165
D Reconfigurable Computing: An Overview	166
D.1 The splash effect	166
D.2 How does it work?	173
D.2.1 Reconfigurable Architecture (RA)	173
D.2.2 Run-time environment (RTE)	174
D.2.3 Run-time reconfigurability (RTR)	175
D.2.4 Requirements of a RC system	176
D.2.5 Advantages of a RC system	177

D.2.6 Disadvantages of a RC system	178
D.3 What are the alternatives?	179

Tables

Table

2.1	Typical GA parameters and factors	19
2.2	Virtex configuration breakdown	37
2.3	Evolvable hardware advantages	39
2.4	Evolvable hardware issues	40
5.1	Actual GA parameters and factors	82
6.1	2-input XOR function evolution	101
6.2	3-input AND function evolution	101
6.3	4-input Boolean function evolution	102
6.4	5-input Boolean function evolution	102
7.1	The hardware environment candidates	105
7.2	The software environment candidates	108
8.1	Timing comparisons based on 2 MHz HGA and 33 MHz GaS systems . .	111
8.2	Speedup factors for GaS compared to HGA	111

Figures

Figure

2.1	Bottom-up approach to GAs	18
2.2	Genetic algorithm flow	21
2.3	A genetic algorithm in operation	22
2.4	Basic FPGA structure	25
2.5	Basic FPGA architectures	26
2.6	The XC6200 interconnect	31
2.7	The XC6200 basic cell	32
2.8	The Virtex FPGA (XCV1000E)	33
2.9	Virtex FPGA block architecture	33
2.10	Virtex FPGA slice	34
2.11	Virtex FPGA CLB	35
2.12	Virtex frames architecture	36
3.1	The future of processing systems	50
3.2	The future of computing systems	51
4.1	Hardware design flow	54
5.1	System architecture	59
5.2	Optimizer module architecture	60
5.3	Overall architecture	62

5.4	The GaS programming model	78
5.5	The optimizer operation	80
5.6	The system operation	81
5.7	A sample chromosome	83
5.8	A 4-input LUT structure	84
5.9	A circuit based on the sample chromosome	86
5.10	Evolutionary stage 1	89
5.11	Evolutionary stage 2	90
5.12	Evolutionary stage 3	92
5.13	Evolutionary stage 4	92
5.14	Evolutionary stage 5	93
5.15	The chromosome structure	94
5.16	The chromosome encoding scheme	95
7.1	The HOT II and daughterboard cards	106
A.1	RNG ASM chart	129
A.2	RNG detailed ASM chart	129
A.3	RNG ASM control path	130
A.4	RNG ASM data path	130
A.5	SM ASM chart	132
A.6	SM detailed ASM chart	133
A.7	SM ASM control path	134
A.8	SM ASM data path	135
A.9	XMM ASM chart	137
A.10	XMM detailed ASM chart	138
A.11	XMM ASM control path (Part 1)	139
A.12	XMM ASM control path (Part 2)	140

A.13 XMM ASM data path	141
A.14 FM ASM chart	143
A.15 FM detailed ASM chart	144
A.16 FM ASM control path	145
A.17 FM ASM data path	146
A.18 MAM ASM chart	148
A.19 MAM detailed ASM chart	149
A.20 MAM ASM control path	150
A.21 MAM ASM data path	151
A.22 IM ASM chart	153
A.23 IM detailed ASM chart	154
A.24 IM ASM control path	155
A.25 IM ASM data path	156
D.1 The "loudspeaker bottleneck"	168
D.2 The modified loudspeaker	172

0.1 List of Acronyms

AHDL	Altera Hardware Description Language
ANN	Artificial Neural Network
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
ASM	Algorithmic State Machine
CA	Cellular Automata
CAD	Computer-Aided Design
CLB	Configuration Logic Block
CMOS	Complementary Metal-Oxide Semiconductor
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
DNA	Deoxyribonucleic Acid
DSP	Digital Signal Processing
EA	Evolutionary Algorithm
ECLB	Evolvable Configuration Logic Block
EHW	Evolvable Hardware
EPROM	Erasable PROM
EEPROM	Electrically-Erasable PROM
FM	Fitness Module
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
GA	Genetic Algorithm
GaS	Genetic algorithm Synthesis
GPS	Global Positioning System
HCI	Human-Computer Interface
HGA	Hardware Genetic Algorithm
HIGA	Hardware-Implemented Genetic Algorithm
IC	Integrated Circuit

IM	Initialization Module
IOB	Input/Output Block
IP	Intellectual Property
JDK	Java Development Kit
JNI	Java Native Interface
LC	Logic Cell
LFSR	Linear Feedback Shift Register
LUT	Look-Up Table
MAM	Memory Access Module
MPGA	Mask-Programmable Gate Array
PAR	Placement and Routing
PCI	Peripheral Component Interconnect
PDA	Personal Digital Assistant
PLD	Programmable Logic Device
PM	Programmer's Model
POS	Product-of-Sums
PROM	Programmable Read-Only Memory
QoS	Quality of Service
RAM	Random Access Memory
RC	Reconfigurable Computing
RNG	Random Number Generator
RTL	Register-Transfer Logic
SM	Selection Module
SMD	Single Molecule Detection
SOP	Sum-of-Products
SPR	Special Purpose Register
TAS	Total Analysis System
TSP	Travelling Salesman Problem
VC	Virtual Component
VGA	Variable Genetic Algorithm

VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale Integration
XHWIF	Xilinx Hardware Interface
XMM	Crossover and Mutation Module

Chapter 1

Overview and Synopsis

**"There is one thing stronger than all the armies of
the world, and that is an idea whose time has come"**

(Victor Hugo)

"ComputerSon: welcome to your adolescent years! As all of us have done before you at this age, it is now your privilege and duty to rebel against us, to aspire for a better future, to use your past as a tool and not a memory, to be self-sufficient, to start low and aim high, to make your own mistakes and learn from them, and to become our **proud envy**. Yes, it is now time for you to challenge conventional design methods, to always look for a better resolution, to analyze your past in order to mould your future, to become independent, to begin at mediocrity and to achieve near-perfection (if not perfection!), to learn from previous experiences and apply them in future scenarios, and to **evolve** into the ComputerSon that you are. Look at you now! It is said that no one is dearer than a son, except for a son's son. Well, you have made us so proud. Back when we were young, we started out with nothing but gears and punch cards, and your parents did the best they could with what they had: transistors and integrated circuits which helped them become more efficient in their work and hence, more productive. You are now on the verge of revolutionary breakthroughs: from expert system intelligence, to superconductor technologies, to three-dimensional building blocks; these advancements will aid you in shaping a future that we could only dream of. Wireless telephony,

reconfigurable computing (RC), intelligent agents and self-rebuilding machines are just a few of the HALs [3] of your time.

You are embarking on a long and tedious journey, filled with obstacles that you never thought existed. When the chips are down, and all exits seem blocked, keep in mind that we have been there, and that every door has a keyhole, and every keyhole has a key. May the stream of hope seeping through your keyhole be of a clarifying nature, and not a blinding one. And may our good friend time be as good to you as it has been to us.”

Your loving grandparents, Z3 and Mark I

1.1 “Towards Evolvable Hardware”

And in the end, there were two.

Evolvable hardware: these two words bring uncertainty into the world of computing. Their mere widespread use implies the start of the field’s mainstream acceptance of such an evolutionary change. Our conventional design methods have seen outstanding developments and progress that it is near unethical to stray from those proven methods. Evolvable hardware (EHW), though, has found a niche amongst the vast and reliable computing design techniques. The field has matured into a well-researched, but yet unproven technology. Based on the merging of reconfigurable devices and **genetic algorithms (GAs)**, the ideology has gained popularity with scientists and engineers alike, because of its inherent powerful potential. Imagine if you will, a machine capable of reconfiguring itself, after sensing a change in its environment. The reader will be kept abreast of the breadth and depth of this reconfiguration process, but only later in this thesis.

Evolvable hardware unites engineering practice with biological theory. EHW bases its scheme on one of the oldest *practical theories* known to man, namely evo-

lution. Evolution functions using its most notable and popular operative: **natural selection**. The latter implies that individuals, living in a specific environment, having any advantage, however slight, over others, have the best chance of surviving and procreating their kind. Charles Darwin, the father of *Darwinian ideology*, once noted that:

"Man can act only on external and visible characters: nature cares nothing for appearances, except in so far as they may be useful to any being. She can act on every internal organ, on every shade of constitutional difference, on the whole machinery of life. Man selects only for his own good; Nature only for that of the being which she tends. Every selected character is fully exercised by her; and the being is placed under well-suited conditions of life" [4]

Applying natural selection to practical engineering problems was a brilliant idea. However, it is extremely difficult to introduce such a combination, not only because of the proven deterministic methods that presently exist for the same engineering problems, but because of the proposed technology's *black magic-like* operation. Natural selection has been studied by such prominent scientists such as Charles Darwin and Alfred Russell Wallace, but has received much criticism since its inception around 1859. The thought of nature correcting itself seems far-sighted for some, and near-sighted for others. It is not the fact that natural forces are shaping organisms that is disturbing to some, but it is more the detachment of human interaction from human progress. It is as if we are not in direct control of our future, instead indirectly affecting natural forces and aiding them in their decision making process. To the untrained eye, the theory of natural selection being applied to engineering practice, forges a river of chaos and confusion amongst the ocean of old yet proven technologies. However, to the trained eye, the theory has staying power in four major contexts [5]:

Expert-less problems EHW has the expertise to tackle problems for which human experts are lacking, or non-existent. We can rely on evolutionary concepts to

build strong models and foundations in research areas or topics that are yet to have experts attributed to them;

Algorithmic antidotes EHW offers a systematic approach to the "problem of how to solve problems". We need to develop various processes that guide us whenever we are resolving a particular problem. Evolution is the perfect teacher, having succeeded itself, for numerous generations now, in producing progressive societies;

Dynamics EHW provides a directive for adaptation to changing environments. We are living in a world of constant information bombardment. Our abilities to handle many types of inputs, and be able to adapt to each one differently, is what we are trying to achieve through our human-computer symbiosis; and

Parallel processors EHW is based on a highly parallel process: evolution. It is well understood that resolving numerous problems simultaneously only enhances the methodology, arming it with another human trait: the ability to handle many events at the same time.

The details of EHW and its makeup are being sparingly introduced at this point, as the following chapters will delve into the particulars of the field. It is important to realize that the field is only in its first stages of existence, and much work and research have to be dedicated to it, before we can judge its reliability, scalability, robustness and usefulness. One thing is certain: we are on the verge of revolutionizing the way we think of computing. Between scientists attempting to discover new theories and hypotheses, and engineers attempting to apply them to practical problems, we have formed an effective and reliable relationship, which has allowed us to prosper in scientific and technological growth. We are now able to explain better the world's happenings, and we are now able to better resolve the world's practical problems. This association

between the scientific and engineering worlds has produced one of our most valued innovations: the computer.

The computer has gone through more than five generations of major changes. Ever since we learned how to count, we discovered the need to count faster. We then needed to calculate, to memorize, to extrapolate, to plot, to organize, to read and write, and to even sing. All these were made possible by our enhancements to the computer. The latter has become such an integral part of our lives, that it has become heresy not to be associated with it. When was the last time you have written a letter out by hand? When was the last time you have calculated something out on paper? When was the last time you have purchased a music CD at the store? We have reached a time where the computer is becoming "our best friend". It is the only source of truth we have in a world that may seem chaotic and untrustworthy at times. We have learned a lot from the development of the computer, be it from faster simulations of naturally slow events or from extrapolations of principles that may require a human years to visualize: the computer has touched and enhanced virtually every aspect of our existence. Why not then, associate the computer with scientific and engineering innovations? Why not then, allow the computer to aid us in understanding the world better, and at the same time, designing better tools to cope with the obstacles that face us?

Evolvable hardware is a methodology by which we can utilize the computer and its many advantages (fast computational speed, vast memory space and great processing power, to name a few) in order to not only resolve practical problems, but to *attempt to figure out new ways of doing things*. The computer, that we invented and developed, can operate faster and remember more than we are able to, and thus can, for example, solve a problem, which evolution itself took years to solve, in mere minutes. This inherent power not only allows us to understand and apply concepts faster and better than ever before, but also provides us with the potential of explaining concepts in ways never thought of before.

1.2 Motivations

Having introduced EHW and its many advantages, one question remains unasked: "Why should we deviate from the norm?". The question will remain unanswered for now, as more details are needed before any attempts to answer the question are made. The validity of the question, though, is emphasized by the obstacles faced by any emerging technology. Conventional design methods are so called because they are proven over time. Other methodologies have to break through the "conventionalistic" barrier, and can only do so with time. Much research effort and resources have to be invested before any real results can be validated. Even after major findings, issues such as scalability, reliability and robustness come into play, and the potential technology is doubted again. It is only with a proven track record that the new method can be integrated into the design and user communities.

What is it that entices researchers to enter such a volatile field? Why do people venture into a dark area when there is so much light in other areas? What could possibly come out of trying a new way, only to find that the old one works better? These three questions are very important in analyzing the mindset of researchers involved in unconventional topics. Presented below is a listing of both the short-term and long-term motivations of the author, which should clarify the answers to the aforementioned questions.

Short-term motivations

- Build an autonomous system based on EHW, capable of producing efficient and reliable results;
- Prove the validity of EHW when applied to certain areas of engineering design;
- Understand the interworkings of EHW when faced with real-time system constraints.

Long-term motivations

- Construct a small functional block of an evolvable processor, capable of gaining intelligence through experience and time;
- Model artificial intelligence as applied to engineering thinking and design;
- Pave the route for **evolvable computing**, whereas evolvable hardware concepts are applied to reconfigurable computing methodologies, to allow for intelligent processors capable of independently modifying their computational architectures based on the context of the problem at hand.

The motivations for this work lie in the author's deep belief that the emerging technologies are doing so (emerging) because of the availability of tremendous computational abilities and powerful processing systems. Numerous advances in the fields of chip fabrication, module design, block testing and others, have allowed others to abstract away the many factors involved in each field, and concentrate on merging their successes together to produce computationally intelligent "agents". The latter term has been misused in many different contexts. This will not be another attempt to exactly define the term, but to encompass the many systems, models and designs that are being developed. By agents, the author means anything that ranges from processors to computers to embedded systems. These agents are incrementally being equipped with an intelligence factor not seen in previous generations. Today, cameras are being used in conjunction with embedded systems to construct image recognition applications capable of discerning peoples' faces, for automatic "buzz-in" procedures in apartment complexes. Sensors are being equipped with an electronic data sheet which allows for the communication with the particular sensor in a project-wide environment. Finally, genetic algorithms are being implemented within a car-manufacturing framework, allowing for the better design of fuel distribution mechanisms in cars.

The marriage of these emerging technologies is the key to any sought advancement in the field. The deciding success factor is based on the researchers' deep understanding of the limitations and applications of each technology and their eventual combination with others. This said, it is the author's intention to acquire that knowledge when it comes to evolvable hardware, and successively, reconfigurable computing, and merge the two to form, in the larger scheme of things, *intelligent architectures*.

1.3 Objectives and Contributions

Delving into a topic such as EHW requires not only patience, but also know-how, so as to customize the algorithm to the application at hand. The know-how is gained through experience, and this work has provided the author just that. The learning objectives of the research are thus summarized to be:

- Gaining a deep understanding of genetic algorithms, and their subsequent application to hardware evolution;
- Experiencing an EHW application, and fine-tuning the parameters involved to maximize performance; and
- Judging the power of EHW, and "getting a feel" for their use and limitations.

The technical aspect of the research focuses more on the attempt to evolve the *minimized* logic solution of a defined Boolean input function, as well as the proof-of-concept involving the comparison to other automated minimization techniques and human methods. This said, the research also encompasses important technical objectives, listed as follows:

- Minimizing the logic representation of simple Boolean functions;
- Evolving the logic circuits of simple structures; and
- Expanding the concept to evolve minimized complex structures.

It is important to note here, that the evolution is done through a hardware implementation of a genetic algorithm, whose details will be discussed in chapter 5, while the minimization is one of resource usage and logic levels. The latter will be demonstrated on a reconfigurable device, further discussed in section 2.2.3.

It is to the delight of every researcher to contribute to their respective field. Contributions, however, are measured in light of their applications and continued education. The former is a straight-forward characteristic, whereby the potential use of the contribution is valued according to simplicity, understanding and power. The latter is a measure of how well the contribution can be picked up in future research ventures, be it by the same researcher or a completely different group of researchers. These two valuations are quite important in determining the overall contributions of the undertaken work. In order to efficiently portray the contributions of a research, a question has to be asked, and answered. This work's question is: "How will GaS benefit circuit synthesis?". GaS is the nomenclature used to designate **Genetic algorithm Synthesis**, and it is the embodiment of the system described in this thesis. GaS, as will be seen later, is directly involved in synthesis-related issues, such as functional decomposition, technology mapping and resource minimization. These topics will be discussed in more detail in later chapters. GaS is the first evolutionary design system that utilizes *both logical and routing resource evolution*, within the device the solution is being developed on. GaS also introduces a fundamental concept to hardware processing, mainly hardware cache. These innovations are verified through the following work, mainly by the realization of a working prototype, and the successful analysis of the complete system. Here is a brief listing of the contributions of this work in the field of circuit synthesis:

Tracking solutions quickly It has been proven that GAs are very efficient in locating a solution in a large search space. A design was realized in which the GA efficiency was used to find a minimized solution to the defined Boolean input

function. Numerous additions were made to the standard GA in order to better guide the process;

Constraining complexity The number of possible *sub-functions* in a logical expression, exponentially increases with an increase in the number of inputs, causing the system complexity to increase as well. A prototype was designed and built to constrain system complexity by progressively eliminating unhelpful sub-functions throughout the evolutionary runs. The prototype proved the theoretical results, and built on the innovations introduced by the overall system;

Maximizing the solution space GaS was realized to yield numerous solutions to the input problem, which can be further analyzed for various other minimization factors (speed, delay, gate structure and so on);

Adapting to parallelism and fault-tolerance GaS was designed to adapt to the different input functions, generating the correct solutions in real-time. The realized system is parallel and fault-tolerant, in that any input change can trigger evolutionary runs, only ending when a valid solution is found to the new scenario; and

Beginning of evolutionary agents concept GaS, along with its prototype, forms an introductory step into the realm of computational agents, able to resolve different problems in parallel, and thus, together forming an intelligent architecture capable of evolving, through adaptation and fault-tolerance, to survive in a hostile environment. This concept was studied and analyzed, and the preliminary hypotheses are presented in this thesis.

1.4 Outline of this Thesis

The remainder of this thesis is organized as follows. In Chapter 2 a basic introduction to the technical details, involved in merging genetic algorithms and reconfigurable architectures, is presented. Also presented is previous work utilizing evolvable hardware to increase application efficiency. Chapter 3 presents the state of the art of six different research areas viewed as integral parts of our technological progress. Chapter 4 defines the problem being resolved and presents the current solutions at our disposal. Chapter 5 describes **GaS**, its makeup, functionality, and introduced novelties. Chapter 6 presents the verification of the design by referring to performance metrics in numerous shades of light (functional, tascal, structural and behavioral). In Chapter 7 the **GaS** prototype is described from design to realization. A few comments about hardware-software co-design are emphasized in this chapter. Chapter 8 presents a comparative study of the system, related to previously developed environments. Chapter 9 develops an argumentative discussion, building on the ideas presented in this chapter. Included in this discussion is the potential future research and the advantages of such endeavours. Chapter 10 presents conclusions and a review of the presented material. The hardware design (Algorithmic State Machine diagrams and VHDL code) is presented in Appendices A and B, while the software design (C++ and Java code) is presented in Appendix C and Appendix D describes in a more formal manner reconfigurable computing.

Chapter 2

Terminology and Chronology

"How can they say my life isn't a success? Have I not for more than sixty years got enough to eat and escaped being eaten?"

(Logan Pearsall Smith)

What is it that is so intriguing about evolution? What can nature teach us that we don't already know? These questions have been incrementally answered by man's deep desire for imitation. Man always attempts to imitate what he cannot control. Through imitation, it is hoped that the desired understanding will be obtained and thus, the mystery unsolved. It is with this understanding that man is controlling what he cannot. Nature has been an uncontrollable entity which entices all who study it. It searches through an immense space of solutions, with help from its two main advocates: time and evolution. The former allows nature to experiment through the different paths, and is the ultimate judge on progress, while the latter guides nature through the infinite paths. Evolution is **parallel and adaptive**, hence it is well qualified as an algorithm used to search through an enormous number of *solutions*. In the 1950s and the 1960s, several researchers began noticing the power and innovation that nature was offering all around them, and started experimenting with rudimentary versions of what are now called **evolutionary algorithms (EAs)**. The first few trials involved evolutionary strategies used to resolve a given problem, through a mapping of genetic-inspired operators. Scientists, such as Box (1957), Rechenberg (1965) and Schwefel

(1975), worked on evolutionary strategies, evolutionary programming and eventually genetic algorithms [6]. The aforementioned methodologies are the main components of EAs, with genetic algorithms being the most popular.

2.1 Genetic Algorithms

A brief introduction to genetic algorithms follows. Not many details are given in this section, as the subsequent sub-sections in this chapter will provide that information. This part is meant to prepare the reader in understanding what genetic algorithms are and what they are useful for.

Developed by John Holland, his colleagues and students at the University of Michigan, GAs are search techniques modeled after the natural selection, including the genetic operators that play an important role in the way in which we survive in our environment [7]. GAs are stochastic algorithms with very simple operators that involve random number generation, and copying and exchanging of string structures. The aforementioned operators act on a population of members, represented by chromosomes. Members are mated by the crossover of their appropriate chromosome, the latter undergoing a mutation operator after crossover. The next generation of members is selected from the current one according to how well (represented by the associated fitness value) the members react to their present environment [6]. GAs have been efficient in their *evolutionary* runs because as new generations are made of older ones, one can easily select the members that are fairing quite adequately in the environment they have been placed in. The crossover operator ensures that the *good* chromosomes have more children in the generations to come, while the *bad* ones have less or no children at all. The mutation operator provides a stabilizing random input of error, to emulate the corresponding phenomenon that occurs in nature [2]. GAs fair well in large search space problems, as **better solutions tend to grow old with time**.

2.1.1 GA: from Roots to Leaves

Let us introduce GAs using a bottom-up approach, as seen in figure 2.1. The reason is that to truly appreciate the *complex simplicity* of a genetic algorithm, the topic has to be introduced by explaining the basic building blocks, the methodology by which they are combined, and the overall functionality of the complete system.

Figure 2.1: Bottom-up approach to GAs

The GA designer commences by selecting a *population* of encoded solutions, or *chromosomes*, composed of genes (bits), each gene being an instance of a particular allele (for example, either a '0' or a '1'). The basic building blocks of a GA are its genetic operators. These are the operations that are performed on the population of chromosomes every iteration through the GA. There are **three major operators** when speaking of a simple GA (SGA).

Selection This operator is used to copy individual chromosomes to the next generation according to their objective function values (fitness values);

Crossover This operator is used to swap out substrings of parent chromosomes to form new children chromosomes

Mutation This operator is used to aid the GA in escaping local optima by randomly flipping a single allele of a chromosome.

The SGA's operation can be summarized in a difference equation as follows:

$$x[t + 1] = s(v(x[t])) \quad (2.1)$$

where $x[t]$ is the population at time t under a representation x

v is the random variation operator, and

s is the selection operator.

Thus, the success of the genetic algorithm is based on the relationship between s and v , as applied to representation x [5]. There are other factors involved in the design of a GA, as shown in table 2.1. These parameters have to be understood and defined before the operation of the GA is explored. Each of these parameters has an effect on the overall success of the evolutionary runs, and any minor modifications to the different values, schemes and criteria can result in a drastic shift in the output. Explored later in this thesis are the actual selected choices for each of these parameters during the design phase of GaS, and the reasons behind each selection. At this point,

Parameter	Description
Population size	<i>Used to designate how many chromosome per generation are available</i>
Chromosome size	<i>Used to designate how many genes (or bits) per chromosome are available</i>
Encoding scheme	<i>Used to designate what type of encoding scheme is used (e.g. binary, gray or Manchester coding)</i>
Number of generations	<i>Used to designate how many generations the GA runs for</i>
Crossover and mutation probabilities	<i>Used to designate the probability of each, crossover and mutation occurring</i>
Fitness function	<i>Used to designate the objective function by which each chromosome is valuated</i>
Selection scheme	<i>Used to designate the criteria of building a new generation from an older one</i>
Stopping scheme	<i>Used to designate the condition of stopping the evolutionary run</i>
Initial random seed	<i>Used to designate the seed injected into the random number generation scheme</i>
Random number generation scheme	<i>Used to designate the algorithm which generates the random numbers</i>
First generation members	<i>Used to designate the methodology by which the initial population is generated</i>

Table 2.1: Typical GA parameters and factors

the only remaining major topic to be discussed before moving to the operation of a

SGA is the **fitness function**. The latter is the objective function applied to every chromosome in the population, in order to receive a valuation, for that member, of how well it performed when placed in a particular environment. The fitness function is used to assign a score to each chromosome, representing how well the chromosome solves the problem at hand. The fitness function embodies the various optimization factors involved in the problem, rewards the chromosomes that perform well in optimization, and punishes the chromosomes that do not perform as well as others. Each GA tailors to a specific application, and hence, each GA sports its own unique fitness function.

The GA is a simple iterative algorithm that encompasses heuristic operations. Figure 2.2 demonstrates the flow of a GA. It can be noticed that the genetic operators, along with the fitness evaluation, form a **genetic pipeline** of operations which each population of chromosomes goes through in order to generate the next population's members. After the initial generation of the first population of solutions (as simple as the outputs of a random number generator, and as complex as the outputs of a neuro-fuzzy system), the different chromosomes (or solutions) are evaluated and assigned a fitness score. The selection operator is then applied to the population in order to select members based on a selection criteria (to be discussed later). This step dutifully weeds out the weaker chromosomes, and enriches the next generation with the stronger members (or combinations of the stronger members). The crossover operator is then applied to the chosen members, which according to a pre-defined probability, combines the parent members to form child members. The latter are subjected to a mutation operator, again according to a pre-defined probability, which helps the GA avoid local "steep hills", and forces the algorithm to better cover the search space.

Figure 2.3 demonstrates the typical flow of a SGA, where one-bit crossover and mutation operations occur. Only two members are chosen through the selection operator, and are placed into the next generation. As can be seen in the figure, the population number and the chromosome length both remain constant, but what is not represented

Figure 2.2: Genetic algorithm flow

in the figure, is the probabilistic operations by which selection, crossover and mutation occur. Not all chromosomes undergo the three major genetic operators during every generational run of the GA, and that is a point to be kept in the mind of the reader while the GA operations are further explored.

The GA proceeds through its generational runs, until a certain stopping condition is met (e.g. the GA has run for a certain number of generations, or a solution has been reached or the chromosomes have reached a state of consistency). The solution is then extracted from the population of chromosomes and is installed into the system as the working resolution to the task at hand.

2.1.2 Theory and Application

Genetic algorithms are used in various applications. Slight modifications are done to the SGA in each of these areas, but the main idea remains constant. The following lists a few of the application areas where GAs have been used advantageously [6] [8].

NP-complete situations Problems such as the boolean satisfiability problem and the Traveling Salesman Problem (TSP);

Figure 2.3: A genetic algorithm in operation

Machine learning Problems, such as the prediction of weather, in machine learning applications;

Operation research Problems where linear or non-linear optimization are major factors;

Finance Problems that deal with economical factors, such as prediction of stock market trends;

VLSI layout Problems that deal with optimizing the layout of a VLSI chip, be it for area, power or speed;

Social systems Problems that try to imitate, and thus study, the evolutionary strategies and aspects of social systems.

Genetic algorithms are based on *Schemata Theory*. A schema is a grouping of strings, formed from a base alphabet, and represented by set notation. For example, the binary alphabet is represented by the following set notation 0,1. A ternary alphabet,

using the binary alphabet, can be represented by utilizing the *don't care* symbol *, to form 0,1,*. The schema 00*10, for example, can be considered as an encoding for the following two strings: 00010 and 00110. There are two important features of a schema: its length and its order. Consider a schema S , whose set notation is $1000^*01^*1^*$. Its length, δ , is the distance between the last and the first specific positions, and thus for this case would be $9 - 1 = 8$. Its order, o is the number of fixed positions present, and thus for this case would be 7. Using these notations, the *Schema Theorem*, or the *Fundamental Theorem of Genetic Algorithms*, has been developed as follows:

Short, low-order, above-average schemata receive exponentially increasing trials in subsequent generations [7]

It can be easily seen that the above theorem encompasses the powerful features of a GA, by noting that chromosomes that are of short length and low-order, but have relatively high fitness values, tend to remain present in some shape throughout the evolutionary runs. The short length and low-order properties aid in the ever-presence of the particular chromosome in that it is tougher to break up a short, low-order member than it is to break up a long, high-order member. For example, a schema such as 1001*10101011 is more easily dispersed into the subsequent generations than a schema such as 10*1. The high fitness values also aid in the ever-presence of the particular chromosome through a direct influence on the selection algorithm being used (more on the selection criteria later).

Building upon the theories of a SGA, there exists variations upon the basic building blocks, as documented below [6] [8].

Inversion This genetic operator allows for the survival of beneficial schemas by direct inversion of part of the chromosome;

Crossover "Hot Spots" This operator allows for a better crossover operation by evolving gene positions in each chromosome where crossover is allowed to occur;

Multi-point crossover This operator allows for a wider selection of schemata by having multiple points of crossover within each chromosome;

Elitism This addition to the SGA allows for maintaining consistency in subsequent generations by retaining some number of the best individuals of the previous generation;

Rank Selection This selection operator allows for lower sequential processing times during the selection operation, as only ranks are compared between chromosomes, instead of absolute fitness values;

Tournament Selection This operator allows for a lower number of runs through a generation, as two individuals are chosen at random, and placed in direct combat against each other. The actual selection process is based on a random number distribution function.

Other operators and strategies exist and have been well documented in the literature: *dominance and diploidy* [7], *duplication and deletion* [1], *crowding* [9] and *niche and customization* [1]. These topics are more advanced and are beyond the scope of this thesis, thus will not be covered any further. Having reached a certain introductory level of understanding of what genetic algorithms are, how they are used and what they are useful for, it is now time to introduce the other component of EHW, namely **field-programmable gate arrays (FPGAs)**.

2.2 Field-Programmable Gate Arrays

The highest capacity general-purpose logic chips available today are the traditional gate arrays sometimes referred to as **Mask-Programmable Gate Arrays (MPGAs)**. MPGAs consist of an array of pre-fabricated transistors that can be customized to the user's logic circuit by connecting the transistors with custom wires.

Customization is performed during chip fabrication by specifying the metal interconnect, and this means that in order for a user to employ a MPGA, a large setup cost is involved and the manufacturing time is long. Although MPGAs are not PLDs, they are mentioned here because they motivated the design of the user-programmable equivalent: FPGAs. Like MPGAs, FPGAs comprise an array of uncommitted circuit elements, called logic blocks, and interconnect resources, but the FPGA configuration is performed through programming by the end user [10]. An illustration of a typical FPGA architecture appears in figure 2.4, courtesy of [11].

Figure 2.4: Basic FPGA structure

FPGAs can be categorized by two major factors: their architecture and their process technology. The former is further sub-divided into four: symmetrical array, row-based, hierarchical PLD and a sea-of-gates (refer to figure 2.5, courtesy of [12]). The main differences between the architectures are the size of the logic blocks, the number of available flip-flops and the logic implementation method. The other category that differentiates today's FPGAs is their process technology. There are currently four technologies in use: anti-fuse, *erasable programmable read-only memory (EPROM)* transistors, *electrically-erasable PROM (EEPROM)* transistors, or static **Random Access Memory (RAM)** cells.

Figure 2.5: Basic FPGA architectures

As previously mentioned, there are four major categories of FPGA process technologies, but they are typically grouped into three, because of the relative similarities between the EPROM and the EEPROM transistor technologies. The following is a brief description of each process technology [13]:

Anti-Fuse Used as a process technology by passing current through a link in order to make a connection. The anti-fuse is in a high-impedance state, and can be programmed to be in a low-impedance (fused) state. FPGAs based on this technology are one-time programmable (OTP), and require large programming transistors;

EPROM/EEPROM transistors Used as programmable switches by placing the transistor between two wires in a way that facilitates implementation of wired AND functions. The EPROM/EEPROM transistors can be reprogrammed externally, and cannot be reconfigured in-circuit;

Static RAM Using pass-transistors, transmission gate or multiplexers that are SRAM-cell controlled, the configurable links are made throughout the FPGA. This type of technology makes its FPGAs in-system programmable (ISP), and re-

programmable, because SRAM-based devices are volatile and thus need to be configured upon power-up. The disadvantage is the large chip, and thus package, size that is needed to implement the SRAM technology.

2.2.1 FPGA Interworkings and Applications

FPGAs provide the benefits of custom **complementary metal-oxide semiconductor (CMOS)** design, while avoiding the initial costs, time delays and inherent risks of a conventional masked gate array. FPGAs are customized by loading configuration data into the internal memory cells. The FPGA can either actively read its configuration data out of external serial or byte-parallel PROM (master mode), or the configuration data can be written into the FPGA (slave mode). The FPGA can be programmed an unlimited number of times.

The FPGA has three major configurable elements (refer to figure 2.4 for an illustration): *configurable logic blocks (CLBs)*, *input/output blocks (IOBs)*, and *interconnects*. The CLBs provide the functional elements for constructing the user's logic. The IOBs provide the interface between the package pins and the internal signal lines. The programmable interconnect resources provide routing paths to connect the inputs and outputs of the CLBs and IOBs onto the appropriate networks. Customized configuration is established by programming internal static memory cells that determine the logic functions and internal connections implemented in the FPGA [11] [10]. The CLBs implement most of the logic in a FPGA. The flexibility and symmetry of the CLB architecture facilitates the placement and routing of a given application.

FPGAs are currently used in many different fields and applications. One of the major applications is FPGA processors. The latter change the view on algorithmic problem solving and have the advantage of being extremely powerful for many applications. Widely-utilized computer architectures have a fixed *central processing unit (CPU)* operating on a memory. Programs determine the sequence of single instructions

executed by the CPU. This is clearly a disadvantage for algorithms, which can be executed in parallel. In contrast, FPGA computers have no given processor structure but offer certain amounts of gates, registers, RAM and routing resources. These can be used for performing logical and arithmetical operations, for variable storage and to transfer data between different parts of the system. Programs do not determine the sequence of execution but the logical structure of the reconfigurable machine. Therefore, algorithms are not only executable in parallel but are executed using a minimum amount of hardware. In a word, FPGA processors combine the high computing power of special purpose hardware with the flexibility of a software solution.

Another major application for FPGAs are **digital signal processing (DSP)** applications. FPGAs provide an easy way to develop DSP boards because each logic block can be programmed by the user. A FPGA also has many of these logic blocks, thus permitting their use in the process of building complex digital circuits. Another promising area for FPGA applications, is the utilization of FPGAs as custom computing machines. This involves using the programmable parts to "execute" software, rather than compiling the software for execution on a regular CPU [14] [15]. Other interesting applications of FPGAs are prototyping of designs later to be implemented in gate arrays, emulation of entire large hardware systems, processing of radar signal, modeling of physical systems, processing of audio/video signals, and for communication purposes.

2.2.2 Victory in Quality, not Quantity

With the advent of **very large scale integration (VLSI)** in the 1980s, engineers began to realize the advantages of designing an *integrated circuit (IC)* that was customized or tailored to a particular system or application rather than using standard ICs. As VLSI technologies became popular, a designer could build a system from a smaller number of components by combining many standard ICs into a few custom ICs. Building a microelectronic system with fewer ICs allowed for a reduction in cost

and an improvement in reliability. The resulting methodologies became widely known as **application-specific integrated circuit (ASIC)** design. The process has grown rapidly and today's predictions are that we will have microprocessors running at twenty gigahertz, containing in excess of a billion transistors, and operating at less than one volt, by the year 2007 [16]. ASICs have grown to become the main implementation medium for hardware designs, but recently, reconfigurable devices, such as FPGAs, have approached the speeds and logic resources of ASICs, and have allowed for a revolutionary type of computing: reconfigurable computing.

RC has several advantages. First, it is possible to achieve greater functionality with a simpler hardware design. Because not all of the logic must be present in the FPGA at all times, the cost of supporting additional features is reduced to the cost of the memory required to store the logic design. The second advantage is lower system cost, which does not manifest itself exactly as you might expect. On a low-volume product, there will be some production cost savings, which result from the elimination of the expense of ASIC design and fabrication. However, for higher-volume products, the production cost of fixed hardware may actually be lower. We have to think in terms of lifetime system costs to see the savings. Here, technical obsolescence drives up the cost of systems based on fixed-hardware designs. Systems based on RC are upgradable in the field. Such changes extend the useful life of the system, thus reducing lifetime costs. The final advantage of RC is reduced time-to-market. The fact that you're no longer using an ASIC is a big help in this respect. There are no chip design and prototyping cycles, which eliminates a large amount of development effort. In addition, the logic design remains flexible right up until (and even after) the product ships. This allows an incremental design flow, a luxury not typically available to hardware designers. You can even ship a product that meets the minimum requirements and add features after deployment.

Traditional FPGAs have the aforementioned characteristic of having to read their

configuration out of a serial EEPROM, one bit at a time, and only when a chip reset signal is asserted. This is disadvantageous in terms of RC, since the FPGA must be flexible for run-time reconfiguration. The following features make the FPGA better suited for RC applications [17]:

Real-Time Reprogrammability Instead of resetting the FPGA, stop the clock from going to some or all of the chip, change the logic within that region, and then restart the clock. This way, there isn't as much wasted time, or configuration overhead;

Partial Reprogrammability Leave most of the internal logic in place and change just one part. Any gate or set of gates may be changed without affecting the state of the others;

Externally-Visible Internal State Save the internal state of the FPGA at any time for future use. This allows the internal state of the FPGA to be read and written just like memory and makes it possible to "swap" hardware designs in much the same way that pages of virtual memory are swapped into and out of physical memory.

For a more formal and detailed introduction to RC, as well as a presentation of the state of the art of the field, the reader is referred to appendix D.

2.2.3 The Virtex FPGA: a Younger and Better XC6200

Evolvable hardware became feasible with the advent of the FPGA technology, but only became useful with the introduction of symmetrical and partially-reconfigurable devices, such as Xilinx' XC6200 family of FPGAs. Xilinx boasts some of the best reconfigurable devices available on the market. The FPGAs mostly use static RAM as a process technology, and contain both reconfigurable logic and routing resources, thus closely exemplify other architectures from different vendors. In this work, two Xilinx

FPGAs are utilized, one from the XC4000 family and another from the Virtex family. The XC4000-family FPGA is used for the implementation of the GA, while the Virtex-family FPGA is used for the actual fitness evaluation of a chromosome (more on this later). Now that the FPGA families have been briefly introduced, let us take a closer look at the XC6200 family of FPGAs (information courtesy of [18]).

Each device belonging to the XC6200 family (XC6209, XC6216, XC6236 or XC6264) contains a sea of gates, whereby each cell can be programmed to implement a combinational or a registered function. As well, each device can be viewed as a hierarchical cellular array, whereby a group of neighboring cells form a, let's say, 4x4 block of cells. Four of these blocks can form a 16x16 block of cells, and so on. These blocks have routing resources associated with them, aiding in the partitioning of the design, as well as forming a symmetrical view to the entire device (see figure 2.6).

Figure 2.6: The XC6200 interconnect

The basic XC6200 cell is composed of inputs from the four neighboring cells (N, S, E and W), from inputs four cells away (N4, S4, E4 and W4), as well as a functional unit that generates the function output F (figure 2.7). The functional unit contains five multiplexers and one D-type flip-flop, allowing for both purely combinational Boolean function outputs or registered outputs. The clock and asynchronous clear signals are routed to the functional unit's D-type register.

The XC6200 device configuration also allows for the following features:

interested in the features that the family provides. However, there are some powerful advantages to the devices, that have since been ported to the newest family of Xilinx FPGAs: the Virtex family. The latter boasts many devices ranging from 50,000 gates to over 8 million gates, and thus is more industrially viable. Take a look at figure 2.8 for a real-life Virtex FPGA.

Figure 2.8: The Virtex FPGA (XCV1000E)

Each Virtex device consists of CLB and IOB arrays, arranged in a columnar format (figure 2.9). The IOBs are used to interface the package pins to the internal CLBs, but are not further discussed because they do not greatly impact the undertaken experiment. The CLBs, however, are quite integral, and are thus discussed in more detail (information courtesy of [18]).

Figure 2.9: Virtex FPGA block architecture

Each CLB is made up of four *logic cells (LCs)*, organized in two similar slices (figure 2.10 [19]), with each LC containing a 4-input function generator, carry logic and a storage element. In addition to the four LCs, each CLB contains logic elements

allowing for the construction of outputs of five or six inputs. The function generators are actually 4-input **look-up tables (LUTs)**, each capable of operating as a function generator, or as a 16 x 1-bit synchronous RAM. Since there are two LUTs within each slice of a CLB, they can be grouped to form a 16 x 2-bit or 32 x 1-bit synchronous RAM, or a 16 x 1-bit dual-port synchronous RAM. Along with the LUTs, the storage elements are made up of D-type registers, with additional logic used to aid in arithmetic operations, such as a dedicated carry logic for high-speed adder functionalities (refer to figure 2.11).

Figure 2.10: Virtex FPGA slice

Along with the programmable logic resources, each Virtex device contains ample routing resources including:

- Local routing for intra-cell interconnections
 - (1) Connecting LUTs, flip-flops and the routing matrix

Figure 2.11: Virtex FPGA CLB

- (2) Connecting CLB feedback paths into LUT inputs
- (3) Connecting directly the CLB outputs (without use of the general purpose matrix)
- General purpose routing for inter-cell interconnections
 - (1) Provides 24 single-length lines for adjacent CLBs
 - (2) Provides 72 buffered Hex lines for routing to another CLB six blocks away
 - (3) Provides 12 Longlines that span the height and width of the device
- I/O routing for CLB-to-IOB interconnections
 - (1) Uses the VersaRing as additional IOB routing resources
 - (2) Pin-swapping and pin-locking are facilitated through the additional resources

The logical and routing resources of a Virtex device are programmable so as to change the makeup and the interconnections of the functions being implemented. This occurs through the programming of the device. The reader is referred to [20] for detailed

outline of how to configure a Virtex device, but presented below are brief outlines of the reference.

The configuration bitstream is made up of a collection of configuration bits that represent configuration commands and configuration data. The bits are grouped into *vertical frames* that extend from the top to the bottom of the device (as seen in figure 2.12). The frame is the *atomic unit of configuration*, and is grouped into a set of columns. Hence, the Virtex device can be reconfigured down to the column level, while, as aforementioned, the XC6200 device could be reconfigured down to the cell level. This feature, however, allows the Virtex device to be a solid candidate for EHW applications. The columns themselves are sub-divided according to the block being configured.

Figure 2.12: Virtex frames architecture

There is one center column which includes the configuration information for the four global clocks. The IOB columns include the configuration information for the IOBs on the left and the right of the device. The majority of the remaining columns are CLB ones, and they include the CLBs along the column, as well as the IOBs above and below those CLBs. Refer to table 2.2 to see how the columns are grouped in a Virtex device.

The columnar hierarchy of the Virtex FPGA aids EHW applications in that the configuration bitstream is consistent for each column. Boolean functions can be implemented in one CLB column, and moved to another, without structural changes, other than the location of the programming bits within the configuration bitstream

Columns	Number of frames	Number of frames per device
Center	8	1
CLB	48	Same as number of CLB columns
IOB	54	2
Block Select RAM Interconnect	27	Same as number of Block Select RAM columns
Block Select RAM Content	64	Same as number of Block Select RAM columns

Table 2.2: Virtex configuration breakdown

itself.

2.3 Evolvable Hardware

Now that both GAs and FPGAs have been formally introduced, it is time to turn our attention to the result formed by their combination. Evolvable hardware is a term coined by Hugo De Garis in 1992 [2]. *Evolware*, as it is often referred to, consists of reconfigurable hardware that can undergo a number of evolutionary cycles, producing a suitable solution to the application at hand [21]. The goal of evolvable hardware is to implement a flexible and fault-tolerant architecture, capable of *morphing* its structure so as to match a dynamic environment. The basic concept behind EHW is viewing the *configuration bitstream* of a reconfigurable device, such as a FPGA, as the *chromosome* of a GA. Hence, the random distribution of chromosomes, and their eventual selection, crossover and mutation makes up the actual circuits, evaluated on the FPGA, and assigned a fitness value according to how well they perform. The chromosomes that fare well will have more children in the upcoming generations, while the ones that don't fare as well will have less or no children at all. *Evolutionary engineering* is mentioned in [2] and represents the complexity independence of GAs, and EAs in general. The internal system complexity does not matter to an EA, as long as the fitness levels are increasing.

Compared to conventional design methodologies, which requires the placement of constraints on any abstraction level of a problem, EHW allows for interactions between

abstraction levels, and works by evaluating actual physical changes [22]. As well, using conventional techniques, the specifications of the system have to be known beforehand, while with evolvable hardware the system can reconfigure itself without the actual hardware specifications. EHW, as aforementioned, basically uses parallelism and adaptation to implement genetic learning and reinforcement learning, as applied to the synthesis of analog and digital systems.

In [23], the authors define four classes of EHW as follows.

Software circuit design Techniques applied through the use of software, with the final solution possibly being loaded on a real circuit;

Hybrid circuit design Operations are still mostly carried out in software, but a real circuit is used during the evolutionary runs;

Hardware circuit design All operations are carried out *online*, including selection, crossover, mutation and fitness evaluation, but there is still a predefined goal in mind;

Open-ended circuit design The *truest* evolutionary process involving both **undirected and unconstrained** evolution.

Another common measure of differentiating EHW systems is whether the inherent evolution is *intrinsic* or *extrinsic*. Intrinsic hardware evolution involves the direct generation of an analog or digital circuit on a hardware device, and falls under the "hardware circuit design" class. Extrinsic hardware evolution, on the other hand, involves running the EA on an external device, with the resulting solution being downloaded and evaluated on a real hardware device. This type of evolution falls under the "software circuit design" class.

2.3.1 EHW Intricacies

Evolvable hardware offers a powerful combination of EAs and FPGAs which re-wires a potential solution in a very short time step. This, and many more advantages, are listed in table 2.3 [24].

Advantage	Description
Execution speed	Extremely quick reconfiguration and evaluation speeds if done in hardware
Low cost	No fabrication is needed and risk is minimized, as fault-tolerance allows for design faults
Fault-tolerance	Instead of using redundant systems, EHW provides a way to resolve environmental faults through the inherent adaptation of the system
Resource minimization	EHW provides a balance between exploration of uncharted regions and exploitation of the good parts of the tested regions
Novel techniques	EHW breaks away from conventional design techniques and truly tests out the environmental relationships. EHW also transforms the system designer into a functional designer, instead of an architectural designer

Table 2.3: Evolvable hardware advantages

There are some unresolved issues when evolvable hardware is concerned, and the major ones are listed in table 2.4. Finally, there are numerous applications that EHW is suitable for, and the following is a brief listing of the major ones where EHW has been found successful: pattern recognition, medical applications, data compression, artificial life, circuit synthesis, space exploration and instrumentation and measurement [21].

2.3.2 What's the Big Deal?

So, why is evolvable hardware grabbing the attention of a myriad of world-wide researchers? It is with this question that we embark on the next topic of discussion. *What's the big deal?* The fact of the matter is that without the advent of FPGA technology, EHW seemed improbable, more of a dream than reality. Researchers were waiting for reconfigurable devices so that they can apply the GA concept on a programmable logic device of some kind. With the invention, and subsequent rising popularity, of the FPGA, the EHW community grew rapidly and evolvable projects sprang up mainly in

Issue	Description
Robustness	This includes the testing of the elite circuit under different operating conditions such as temperature, packaging, circuit position and output load
Fault-tolerance	This includes the confidence that a system designer has concerning the circuit facing both environmental and design faults
Scalability	This includes the concept of how well evolvable systems scale from the basic building blocks
System analysis	This includes the measure of how well the evolved system can be compared to more conventional techniques

Table 2.4: Evolvable hardware issues

Europe and Japan. The excitement over this new field of research became fuelled by the ongoing breakthroughs being made in the reconfigurable device market: the faster the chips are able to be reprogrammed, the better evolution can be tested on these devices. The introduction of devices, such as the XC6200 family or the Virtex family devices, allowed for a great boom in the field, as they were based on practically *undamageable* technologies.

The evolvable hardware field received its first major breakthrough through Adrian Thompson's tone discriminator experiment in 1996. The evolved circuit was the result of an intrinsic hardware evolution experiment, and ideally matched the functionality of a standard bi-tone discriminator. The evolved circuit was tested for robustness and fault-tolerance, and it was found that, with a few more generational runs, an ideal solution eventually emerged. The results were quite outstanding, and it was analyzed that the algorithm had found a solution that contained **output-affecting, yet unconnected** cells. These latter entities could not be removed from the circuit without affecting the output, but had no connected path by which to influence that output. This finding prompted Thompson to note that "possible mechanisms (to explain the phenomenon) include interaction through the power-supply wiring or electromagnetic coupling" [22]. This resulting hypothesis might be the reason why EHW is becoming

more attractive as a design technology: its *capability of embedding a solution into the problem environment*.

2.4 Previous Work in EHW

Evolvable hardware research stemmed from projects aimed at implementing genetic algorithms using reconfigurable hardware devices such as FPGAs. Rintala [25] used a programmable logic device (PLD) and the Altera Hardware Description Language (AHDL) as a testbed for a GA. The goal of the implementation was to be able to test different parameters of the GA with minimal reconfiguration: rewiring the hardware, and testing it on the PLD. Wirbel [26] designed a number of GAs in a text compression chip.

Scott [8], on the other hand, developed a GA on a FPGA, using the Xilinx XC4000 family as the testbed. The design goals were the optimization of an input function, as well as the use of the inherent parallelism and pipelining capabilities of a genetic algorithm when mapped out to hardware. One of the successes of the project is the ability to modify the hardware-based fitness function. This allows for the implementation of different fitness functions on the same GA hardware. Speedup factors close to 19 were reached when comparing the Hardware GA (HGA) and the SGA, the latter implemented in software. The work also presented an architecture that is easily scalable to provide parallel runs of the HGA, using different external fitness function evaluation boards.

Other research work has been done in the field of reconfigurable hardware, but not necessarily related to GA-related research. These projects are beyond the scope of this thesis, but the reader is referred to appendix D for a detailed description of past and present reconfigurable computing research.

Evolvable hardware research has rapidly grown since its inception in 1992. Higuchi et. al. [27] used EHW to develop a gate-level pattern recognition system, as well as a function-level variable GA (VGA), whose chromosome length is dynamic, used in an

artificial neural network (ANN) application. The results were impressive as the GA recognized the patterns 100% of the time. Arslan et. al. [28] used a GA to synthesize a range of circuits using multiple-constrained fitness functions. It also possesses the ability to deal with multi-input/multi-output circuits. The reader is referred to [29] and [2] for detailed descriptions of previous EHW-related projects and experiments.

EHW has been previously used to tackle circuit minimization issues, such as resource usage or multi-level optimization. Louis and Rawlins [30] used a GA to design a structure that performs a pre-defined function subject to certain constraints. The GA used was a designer GA (DGA) and used the elitism and masked crossover operators to implement the genetic evolution of a function. The chromosome represented an array of gates (one of AND, OR, NOT, XOR and WIRE, where a WIRE is the representation of the absence of a gate), that could not back-propagate their outputs back into a previous level's inputs. The DGA was used, and compared to a classical GA (CGA), as a 2-bit adder and a 4-bit parity generator were designed. It was shown that the only price to pay for using a DGA over a CGA, is at most a constant factor per generation, emanating from a sort of the generation and from the mask propagation. The major contribution of the research is the incorporation of domain knowledge into the GA's operators, but the latter hindered the GA's general search capabilities. A similar research project was undertaken in 2001, when Coello et. al. [31] designed a GA-based evolvable system in order to generate the circuit solution of a combinatory problem, containing a minimal number of gates. The authors chose to attempt to minimize the number of gates, rather than the gate delay, and used a similar chromosome representation to that of [30]. They also utilized a 2-point crossover operator, a conventional mutation operator and the elitism selection operator. The major results of the work include the superior speed of convergence and total number of performed evaluations that the n-cardinality encoding (NGA) portrayed, as compared to the traditional binary GA (BGA). As well, the NGA and BGA circuit solutions noticeably contained more XOR gates (even nested ones)

than the human design methodologies (Quine-McCluskey procedure, Karnaugh maps). Finally, the NGA seemed to reuse components within a more complex circuit, so as to reduce the total number of gates, proving that the GA was tailoring to its specific fitness function. An interesting note is that minimizing the total number of gates, was actually accomplished through the maximization of the WIRE element, and hence the minimization of the total number of gates necessary for the implementation of the circuit. The results are very promising, but major drawbacks of the representation are the inability to handle more than two circuit inputs, as well as the extremely long resultant Boolean expressions. As well, there are issues concerning how well the GA converges to a certain solution, resolved only through a careful analysis and customization of the chromosome representation and matrix dimensions. This does not help in making the tool more generalized for the design of any sort of combinational circuit.

In [32], the authors attempt to evolve circuit solutions for combination logic system onto the Xilinx 6000 family FPGAs. The authors evolved a 1-bit adder, a 2-bit adder and a 2-bit unsigned multiplier. In the 1-bit adder design, it was found that the carry was independent from the sum. The 2-bit adder circuit used the 1-bit adder circuit to evolve the 2-bit one. Finally, the 2-bit unsigned multiplier circuit was evolved using less gates and gate delays, as compared to other human-designed circuits of the same functionality. The goal of the research was to attempt to design a software implementation of an alternative to classic automatic place and route (APR) tools. The evolvable approach was found to be more organic, with the circuit embedding itself on the FPGA, thus ridding itself of the need for routing.

Other logic circuit work has been done by Koza [33], who used genetic programming techniques to assist him in the design of combinatorial circuits. The main goal of Koza's research was the successful generation of the circuits, and not their minimization, hence do not completely apply to the task at hand. A similar comment can be matched to Iba et. al. [34] and their work in gate-level evolvable hardware. The focus

of the authors' research is more on the evolutionary learning, rather than the actual circuit optimization.

Chapter 3

State of the Art

**"The only people for me are the mad ones, the ones who
are mad to live, mad to talk, mad to be saved, desirous
of everything at the same time, the ones who never yawn
or say a commonplace thing, but burn, burn, burn, like
fabulous yellow roman candles exploding like spiders
across the stars and in the middle you see the blue
centerlight pop and everybody goes "Awww!" "**

(Jack Kerouac)

Given the extensive length of the previous chapter, as well as the wide variety of topics that were introduced, this chapter serves as a summary of the state of the art for the technologies that the author believes to be directly affecting the flow of our computing ideologies. Topics ranging from computers that embed themselves into *the very essence of our lives* (context-aware), or devices that utilize *the very essence of our lives* (DNA) are discussed in this chapter. The idea is to provide the reader with a brief, and possibly unrealistic, outlook of the future that awaits us.

3.1 Evolvable Computing

This topic has been extensively discussed in the previous chapter (see section 2.3), and thus will not be further expanded upon in this section. However, it is worth noting the following points about evolvable computing, the actual processing methodology of EHW.

- allows for the embedding of a particular solution into a general problem environment;
- presents a logical procedure to the problem of how to solve problems; and
- attempts to figure out new ways of doing things.

3.2 Reconfigurable Computing

This topic has also been already introduced and discussed in the previous chapter (see subsection 2.2.2), and again, the reader is referred to appendix D for a clear discussion of the state of the art of the reconfigurable computing field of research.

3.3 DNA Computing

The major drawback that computing professionals are faced with today is the fact that the computer is mostly a sequential machine, capable of crunching instructions at an impressively fast rate. The inherent instruction execution bottleneck is perceived by many researchers as the true hindrance to today's computing prowess. That said, many scientists and engineers are embarking on the path of attempting to find out what, if not silicon, will the computer of the future consist of. One such road is the DNA computing approach, where computers are made up of deoxyribonucleic acid, or simply DNA.

Adleman [35], in 1994, came up with the idea of using DNA strands to solve problems, traditionally resolved by silicon-based computers, such as the "Hamiltonian path problem", better known as the TSP. Instead of using the traditional binary (1's and 0's) representation to encode the solutions, Adleman turned to the DNA sequences that make up the cells within our own bodies. By allowing the various DNA strands and their complements to bond, itself an extremely fast reaction, computations that would be executed sequentially on an electronics computer, take about 1000 times less time, and consume 1000 times less energy. Not only are there time and energy savings, but

the computations are **all executed in parallel**. Such parallel computing power has one disadvantage, in that simple operations, such that of multiplying two 50-digit integers, are hard on a DNA-based computer, but its underlying methodology of resolving complex problems is quite unparalleled (no pun intended), as it is based on an entity that makes up a very complex machinery: the human being.

3.4 Quantum Computing

Quantum computing is another technology that rivals that of the conventional sequential computer. This time, instead of using a new substance (DNA vs. silicon), we analyze a new way of using the available silicon. Feynman [36] introduced the world to quatumelectrodynamics (QED), and opened up the field of research that stemmed from the basic ideas of quantum physics. In quantum computing, instead of a bit representing one state at a time (either a 1 or a 0), the *qubit* can be in both states at the same time. The reader is referred to [37] for a very clear experiment explaining how a qubit can simultaneously be in both states. As the number of qubits is increased, so is the number of states a qubit can simultaneously be in. The increase is an exponential increase, with L qubits having 2^L simultaneous states, and thus are **capable of storing 2^L numbers**. This is quite an advantage over the classical computer, in that the latter needs to perform the same computation 2^L times or have 2^L different processors performing the computation, in order to match the power of a L -bit quantum computer.

Much research work is still needed in order to produce quantum gates and quantum circuits, as well as applying the three most fundamental theories of quantum physics to quantum computation[38]:

Quantization matter is made up of distinct quanta;

Interference a quantum event depends on all its possible histories;

Entanglement two separate events, that have interacted in the past, may still contain

inaccessible information common between them.

The potential of quantum computers is immense and the parallelism offered by these machines can be used in the fields of cryptography, algorithmic searching and factorization; basically applications that currently require supercomputers, and massively parallel processing machines. The current state of microprocessors has seen us cramming more and more bits into silicon, down to a level where transistors are becoming a few atoms wide. Down at this atomic scale, matter obeys the laws of quantum mechanics, and hence a shift in the way we perceive algorithms, as well as a novel method of computation are both needed. Quantum computation intends to fill both needs.

3.5 Pervasive Computing

Pervasive computing is the embodiment of numerous mobile devices that are casually accessible, allowing for easier computing at more frequent locations. The idea refers to the "computing everywhere" approach, or in more technical terms, *ubiquitous computing*. With more and more devices being connected to the world wide Internet, it is now common to have our personal gadgets communicating amongst themselves, in order to provide their owners (us) with information about our daily lives. Digital compasses, global positioning system (GPS) receivers, personal digital assistants (PDAs), mobile phones, calculators, watches, printers, cameras and the like can all now communicate with each other and share information. This evolution of communication is attributed to the great progress that was achieved in the acquisition, storage and transmission of data, and is not a new methodology of computing or information processing. **Pervasive computing refers to the quick, efficient and effortless management of the information available to us.**

There are quite a few obstacles facing the pervasive computing way of life, such as the outline of a standard interface between components, the design of and adaptation to

intelligent devices capable of accessing relevant information securely and efficiently, and the creation of novel protocols and environments capable of masking away the details of device communication. That mentioned, there exists one major hurdle that needs to be cleared before considering the just mentioned obstacles, and that is the introduction of new services and applications to be run on the present devices. The shortage of such endeavours, and the advancement in process technologies, is producing smaller and faster chips, but not enough things to do with them. Until the true potential of today's devices is achieved, pervasive computing will remain as it is in this writing: on paper.

3.5.1 Context-Aware Computing

A major field of computing within pervasive computing is context-aware computing. The latter offers functionality driven by a user's context, be it physical, social, emotional and even mental. Hence, context-aware computing grasps context knowledge, such as where we are and what we're doing, in order to allow for device interaction with the environment. A simple example is a mobile phone, capable of recognizing whether its user is in a quiet room or an outdoor location, and producing a vibrating ring tone in the former, and a loud ring tone in the latter. The main goals of this type of computing is the bridging of the physical and the virtual worlds, the creation of an environment where connectivity is unobtrusive and always available, and the advancement of human-computer interfacing (HCI).

In context-aware computing, it is essential for the device in question to sense both implicit (e.g. time, bandwidth and processing power) and explicit (e.g. orientation, position and temperature) information, in order to make an informed decision about the environment. This, again, is not a new way of computing, but the **natural progression of computing as intelligent sensors are making their way on networked devices.**

The switch to a totally context-aware computing approach is met with a few problems such as the definition of standard data formats and protocols, the establishment of a secure and private network of communication and the formulation of a reliable quality of service (QoS) applications. The resolution of these topics will pave the way to a surreal future, where computers will not only be everywhere, but will change their functionality according to our needs.

3.6 What if...

The prefix of *genii* itself: *What if... ?*: it is the question that begets innovation and excellence. It is the beginning of every revolutionary movement. The future of computing lies in the envelope of the mentioned technologies: EVOLVABLE COMPUTING, RECONFIGURABLE COMPUTING, DNA COMPUTING, QUANTUM COMPUTING and PERVASIVE COMPUTING (including CONTEXT-AWARE COMPUTING). As can be seen in figure 3.1 (courtesy of [39]), the evolution of information processing is moving towards evolvable construction systems, which span the electronic, mechanical and biological fields.

Figure 3.1: The future of processing systems

Quantum and pervasive computing are not shown on the figure, but are part of a larger pool of resources, that make up the future of computing systems (figure 3.2). As can be seen in the diagram, it is the eventual combination of the major computing trends, namely intelligent architectures, computing everywhere concepts and parallel algorithms, which will eventually aid in building the computer of the future. It is this same computer which will allow us to send structures into space, that will evolve, reconfigure, learn, process, store and transmit, in the ultimate goal of surviving in a completely unknown, and possibly hostile, environment.

Figure 3.2: The future of computing systems

Chapter 4

The Defined Problem

**"The advantage of modern means of communication is they
enable you to worry about things in all of the world"**

(Laurence Peter)

Now that we know where we came from (chapter 2) and where we're heading (chapter 3), let us take a look at what we're doing. This chapter introduces the specific problem being investigated. Discussed are the topics of functional decomposition and technology mapping, and presented are the current solutions to both areas.

4.1 Functional Decomposition

A major factor of logic synthesis is how well a function can be sub-divided into less complex sub-functions. The actual methodology by which this division takes place is labelled as *functional decomposition*. Very complex circuits can be simplified through their division into smaller circuits, or sub-circuits, where one or more of these sub-circuits implement recurrent functionalities of the bigger circuit. For example [40], consider the following Boolean function f

$$f = \overline{x_1}x_2x_3 + x_1\overline{x_2}x_3 + x_1x_2x_4 + \overline{x_1}\overline{x_2}x_4 \quad (4.1)$$

, which through factoring of x_3 and x_4 becomes

$$f = (\overline{x_1}x_2 + x_1\overline{x_2})x_3 + (x_1x_2 + \overline{x_1}\overline{x_2})x_4 \quad (4.2)$$

, and thus can be reduced through functional decomposition, and a sub-circuit of $g = x_1x_2 + \overline{x_1x_2}$

$$f = \overline{g}x_3 + gx_4 \quad (4.3)$$

The previous example demonstrates the power of the functional decomposition technique for logic synthesis, as a large and complex circuit was broken down into the combination of smaller and simpler sub-circuits. This is the reason why this technique is very popular amongst modern *computer-aided design (CAD)* tools.

A major problem still, however, exists with functional decomposition. As the number of inputs increases within an input function, the number of possible sub-functions increases along with it. The search space quickly becomes extremely vast, and traditional or deterministic algorithms degrade in performance in these scenarios. As well, the design goal of having optimal solutions becomes secondary to the goal of finding an acceptable solution in the first place, within a reasonable period of time and exhaustion of effort and resources. Instead of relying on the traditional methodologies, heuristic algorithms are utilized. By definition, *heuristic* means

"Relating to or using a problem-solving technique in which the most appropriate solution of several found by alternative methods is selected at successive stages of a program for use in the next step of the program [41]"

Thus, GAs are heuristic by nature (no pun intended), and introduce a novel method by which functional decomposition can be used for large input functions. The vast search space of possible sub-functions becomes the environment where the GA operates, and attempts to weed out the harmful sub-functions and *enhance* the useful ones. In this case, instead of providing the exactly minimal expression, the heuristic algorithm (GA) can provide an approximately minimal expression, within a shorter window of execution time.

4.2 Technology Mapping

Functional decomposition aids in one of the steps of the hardware design flow: *technology mapping*. The latter is part of the optimization process that is performed upon a circuit before the final design is created (refer to figure 4.1, remade from [11]). Technology mapping consists of the mapping of the logical blocks (or sub-functions) onto the components of the particular technology library being mapped to [42]. Thus, if the device is made up of NOR gates, then technology mapping is going from the logical expressions that were input and minimized (through factoring or functional decomposition), and mapping them onto NOR gates. The current most popular technology library in FPGAs involves LUT-based architectures, and thus the technology mapping step involves the mapping of the minimized logical sub-functions onto LUTs.

Figure 4.1: Hardware design flow

The combination of functional decomposition and technology mapping allows a designer to fit a desired logical circuit into the blocks that are offered by the target

technology. These blocks become the main storage units for the sub-functions that are found to be the most worthwhile to implement in the realization of the larger circuits. It is interesting to note [40] that technology mapping is not to be confused with *technology-independent logic synthesis*, with the latter involving techniques such as factoring, functional decomposition and minimization through cubical representation, to minimize a circuit without any particular target mapping. Technology mapping, on the other hand, involves techniques that are used to ensure that the circuits produced by the CAD synthesis tools are actually realizable on the target technology, considering speed, area and architecture constraints.

4.3 Our Current Solutions

As mentioned in [43], the vast number of prime implicants present in large input functions makes it unrealistic to recognize all of them using a deterministic approach. A heuristic approach, as previously discussed, is necessary to attempt to retrieve a close-to-exact, if not exact, solution. The issue does not rest there, as the problems increase substantially when speaking of multioutput or multilevel logic minimization problems [44]. Multioutput logic minimization refers to the minimizing of a circuit which has multiple outputs and common inputs, while multilevel logic minimization refers to the minimizing of a circuit into multiple levels of logic, instead of just two in the AND-OR circuit minimization technique, for example. Both scenarios introduce advantages, such as fewer logic gates, less area and less delay, but have inherent constraints brought along with them, such as complexity and fan-in/fan-out problems.

4.3.1 Theoretical Synthesis

Developed in order to simplify logic functions through graphical representations, Karnaugh maps, or K-maps, represent, graphically, a function's truth table. K-maps aid a designer in finding the sum-of-products (SOP) or product-of-sums (POS) minimized

forms of expression for a particular logic function of two, three or four variables. As soon as the number of variables increases above five, K-maps lose their efficiency and become too cumbersome to utilize for the minimization of logic functions.

Another more systematic approach is the Quine-McCluskey approach to logic minimization. In separate documents, W.V. Quine [45] described an algorithm that could easily be translated onto a computer, and E.J. McCluskey [46] improved upon that algorithm, which became known as the Quine-McCluskey algorithm [47]. The method utilizes an iterative process in order to find the minimal set cover of prime implicants. The Q-M algorithm utilizes a tabular representation, which is more flexible within a computer implementation, based on implicants, prime implicants, and essential prime implicants. Extensive discussions of the details of the Q-M algorithm are found in many digital design textbooks, such as [44], [43] and of course, [47]. It is left up to the reader to peruse through the interworkings of the Q-M algorithm, but it is worthwhile to note that for functions with more than 6 input variables, the Q-M method is better performed through a computer program, as it becomes very tedious on paper. However, as the number of input variables increases, the Q-M method requires more execution speed, time and memory. The Q-M algorithm is not a search algorithm, and thus when the search space for all the prime implicants that will form the minimal set becomes extremely vast, the method loses its efficiency and becomes useless.

4.3.2 "Educational" Synthesis

Extrapolating upon the advantages of both the K-map and the Q-M minimization techniques, research at educational institutions has been receiving community wide acceptance. A heuristic method in particular, named Espresso [48], has received a lot of attention for its capability to find a near-minimal solution, by relaxing the minimization goals so that it is not absolutely imperative to find a minimum-cost realization of the circuit. The algorithm has been fine-tuned for two-level logic minimization and

has evolved into Espresso-II, an algorithm currently fully or partly being utilized as the basis for most educational and commercial synthesis endeavors. One of these educational projects produced, also at the University of California at Berkeley, two software products for the minimization of multilevel circuits: MIS [49] and SIS [50]. The former allows the user to apply various optimization schemes, such as factoring and functional decomposition, to different parts of the circuit, while the latter concentrates on the synthesis of sequential circuits, through the use of the Espresso algorithm, as well as many other heuristic approaches.

4.3.3 "Commercial" Synthesis

The main commercial CAD tools are provided by Cadence Design Systems [51], Mentor Graphics [52] and Synopsys [53]. Each of these, as well as other CAD tool vendors, include logic synthesis software within their products, and utilize proprietary logic optimization strategies based on heuristic methodologies. As aforementioned, the Espresso algorithm is utilized in many CAD programs for circuit synthesis. The algorithm has two main derivatives, Espresso-MV and Espresso-EXACT [54], which are multiple-valued-based optimization schemes, that have great commercial potential, but has not crossed over from the academic side. As can be seen, most of the current commercial synthesis is based on the academic research done in the logic synthesis domain. That said, there is a lot more ground to cover in this field, which will lead to better placement and routing (PAR) tools.

Chapter 5

The Proposed Solution

**"The person who says it cannot be done
should not interrupt the person doing it"**

(Chinese proverb)

The field is logic synthesis, and the problem is the attempt to find a minimal, or close to minimal, solution to realize the circuit of a logic function using technology mapping. This chapter will discuss in detail the proposed solution, GaS, and its architecture and operation. A look into the novelties and applications that stem from this work is also presented at the end of the chapter.

5.1 System Architecture

The overall system is composed of two PLDs, appropriately called *the optimizer* and *the host* (refer to figure 5.1). The optimizer is where the input function logic is "evolved", whereas the host is where the input function solution is actually instantiated and tested. The host will be included in the design as a sub-block of the optimizer (please see section 5.2.4). The optimizer's main objectives are as follows:

- interface with the CPU and the host;
- start off the **Hardware-Implemented Genetic Algorithm (HIGA)** on its evolutionary run;
- store the solutions to each input truth table;

- evaluate the fitness of each candidate solution.

The host is used in order to instantiate the functional units that each logical function needs for execution. Hence, the optimizer performs the evolutionary work, writes the best solution to memory and instantiates the solution in the host, the latter being an originally empty PLD. As an implementation note, all PLDs in this research are actually FPGAs [55]. The optimizer/host model can be referenced to a client/server model, with the optimizer acting as a server, ready to accept requests from the client, or host in this case. The fact of the matter is that the model can be extended to include multiple hosts, serviced by the same optimizer, thus allowing for the evolution of multiple functions, in parallel, on one optimizer.

Figure 5.1: System architecture

5.2 The Guts of GaS

The optimizer is made up of the following (refer to figure 5.2): a HIGA subsystem, a memory subsystem, an **Evolvable Configuration Logic Blocks (ECLBs)**

subsystem and a CPU interface subsystem.

Figure 5.2: Optimizer module architecture

HIGA This subsystem provides the optimizer with the "nucleus", or brain, of the operation. According to the input function (represented by a truth table) presented to the optimizer, the HIGA begins to run in order to determine, through intrinsic hardware evolution, which "chromosome" (or solution) is best suited for the operation. The HIGA is implemented in the *very high speed integrated circuit hardware description language (VHDL)* and contains the following modules: Random Number Generator (RNG), Initialization, Memory Access, Selection, Crossover, Mutation, and Fitness modules.

Memory This subsystem is what is referred to as the "**hardware cache**" (hcache) of the system. It resides on the optimizer as memory cells that, upon optimizer programming, could be loaded from a ROM device which contains the previously stored circuits of logic functions. This subsystem stores the "solutions" to all the previously-encountered truth tables, and thus acts as a common storage facility for designs that could be utilized in the future. This is initially external memory, as opposed to what is shown in figure 5.2.

ECLBs These logic blocks are empty on boot up, but are used to evaluate the chromosomes in order to provide the HIGA subsystem with feedback on how well the candidate chromosome performed in the current environment. This is initially an empty logic device, as opposed to what is shown in figure 5.2. This subsystem is viewed as the current replacement for the host. It has the same functionalities as the latter, but also acts as an area for the actual evolution of each circuit.

CPU Interface This subsystem provides the interface to the CPU. It consists of data, control and status registers, as well as a myriad of **special purpose registers (SPRs)**. The interface integrates the programmer's model of the optimizer and the host communication functions, in order to easily and successfully allow for the hardware/software co-execution and control.

As seen in figure 5.3, the HIGA receives its inputs from on-board shared memory. Currently, the user manually enters the input truth table, as well as the input GA parameters (maximum population, crossover and mutation probabilities, etc...). The HIGA proceeds to iteratively evaluate chromosomes in the ECLBs subsystem. This real-time, on-board evaluation constitutes the "intrinsicity" of this project. The output of the HIGA is a best-fit function-specific solution, which is stored in the memory/hcache subsystem. The stored solution can then either be kept in the hcache or moved to external memory, for future reference.

As previously mentioned, the HIGA subsystem is composed of the following modules:

Figure 5.3: Overall architecture

Random Number Generator (RNG) This module periodically provides the input instantiation, selection, crossover and mutation modules with pseudo-random numbers.

Selection (SM) This module accepts a member and a random number as inputs. It uses a hybrid algorithm (roulette wheel selection [6] and tournament selection [7]) to decide whether the member is to be selected for the next generation or not.

Crossover/Mutation (XMM) These modules accept members as input, and use the input pseudo-random numbers to decide whether crossover/mutation are to be performed upon the individuals or not. Typical single-point crossover and single-bit mutation are used.

Fitness (FM) This module is used to evaluate the fitness of each individual. Initially, it proceeds in generating the first population of chromosomes and receives its fitness values after the instantiation and testing of the chromosomes in the ECLBs subsystem. Afterwards, this module is responsible for receiving two chromosomes at a time, and evaluating their fitness values onto the ECLBs subsystem, and reporting these values back to the device driver running on the host.

Initialization (IM) This module initializes all other modules with their respective values. The selection module receives its selection probability, the mutover module receives its crossover (two of them) and mutation probabilities, as well as the low end fitness (LEF) and high end fitness (HEF) values to be explained later in section 5.2.2.3. The random number generator also receives its seed value, while the fitness module receives the number of bits per chromosome to be used in the generation of the initial population.

Memory Access (MAM) This module is similar to a typical control module, handling the transfer of individuals between the modules and memory. It stores the new individuals in memory and passes them on to the selection module. This module is also responsible for routing each module's request to the appropriate memory block.

To summarize the operation of the HIGA, the truth table and GA parameter inputs are stored in shared memory. The initial generation of chromosomes is generated pseudo-randomly, and stored in shared memory. Each member undergoes the three major genetic operators, processed in the selection, crossover and mutation modules, and is either wholly or partially picked to move on to the next generation. Once picked, the new member is evaluated in order to receive a fitness value. The memory access module coordinates all the transfers between the other modules, thus ensuring a successful sequential operation [55].

5.2.1 An In-Depth Look

GaS currently runs with a simple command-line interface. The user enters the desired truth table, as well as the initializing GA parameters, and starts off the system by launching the application. GaS proceeds to load the parameters into their appropriate modules, generate and evaluate the first population of individuals and successively iterate through the **genetic pipeline**, consisting of the SM, XMM and FM (refer to figure 5.3). Once evolution is complete, and the stopping condition has been met, the application exits. Throughout the evolutionary runs, the application writes out to a file each generation's information, including the individuals, their associated fitness values, the generation's optimal, maximum, minimum, average and sum fitness values. The latter (optimal and average fitness values) are used to determine the stopping condition. Every ten generations, GaS saves its most recent generation's information, in order to

allow for a simple restart if an unforeseen fault should occur (e.g. network connection goes down, or accidental reboot of host). As well, GaS can be interrupted at any time, and restarted from where it left off. This allows for easy access into the internals of the system, for debugging purposes.

There are a few other implementations of genetic algorithms in hardware, utilizing different schemes for memory and processing, [8] [32] to cite a couple. The main advantage that GaS presents in these aspects is genetic pipeline and the shared memory architecture. The genetic pipeline allows, as in [8], for forming parallel genetic processing modules, each capable of locating the solution of a specific Boolean function. This adds to the parallelism feature of GaS, and will be discussed in more detail in section 9.2.2. The shared memory architecture allows the online HIGA to interact with the controlling device driver software, as well as the host. The chromosomes in each generation can easily be saved and viewed in file format, and at the same time applied through the genetic pipeline in the HIGA. As will be seen later, the host (or the ECLBs subsystem in this case) is currently a set of library calls used to emulate the functionality of a Xilinx Virtex device, and thus the shared memory provides another advantage, in that the emulator will be replaced by a physical device in future spin-offs of the application, thus will need to interact with the optimizer through a communications medium. The host and optimizer can interact through a set of signals, overseen by a controller, but the peer-to-peer interaction that the shared memory offers, is simpler to implement and more efficient for scalability purposes.

5.2.2 HIGA: Hardware Implementation of a Genetic Algorithm

The first subsystem to be discussed is the most important one: the HIGA. It has already been briefly introduced in the previous section, but will be elaborated upon in this section. The reason for the elaboration is to document the design procedures undertaken to realize each of the module's functionalities, as well as the overall module

interaction, in order to accomplish the hardware implementation of a genetic algorithm. The HIGA provides the nucleus of GaS, and hence must be studied in detail. The overall design methodology used to realize this subsystem involves the combination of the **algorithmic state machine** concept [44] , combined with an **register-transfer logic (RTL)**-level hardware implementation method. This combination is very powerful and efficient, as the ASM method outputs the control path and data path of each module, in very specific schematics involving combinational and sequential logic. The latter paths are then used to realize the system in VHDL, at the RTL level. Compared to previous implementations of GAs in hardware, which were accomplished by using the behavioral style of VHDL, this work is very easily mapped out to hardware devices, as each module has a known logic area. There are no hidden surprises in terms of how well the synthesis algorithm of the specific CAD tool being utilized maps out to the target technology. As well, for scalability aspects, GaS can be easily multiplied, as each module's control and data paths are well-defined.

The ASM design methodology also allows for a clear description of each module's control and data paths. This clarity provides for an easy explanation and handover to a third-party member, not involved in the design phase of the work. The ASM method also systematically aids in the hardware design of any project. Its simplistic nature, including the definition of the ASM chart, the data path, the detailed ASM chart, and the control path, permits the designer to directly integrate numerous modules to each other, and to the controlling device driver software. Rather than using the *finite state machine (FSM)* approach, which hides away many implementation details of the system, the ASM process exposes the interworkings of the system for all to see and comprehend.

VHDL was chosen as the HDL for this thesis, because of its wide use in the community of hardware designers, its well-documented and supported tools, as well as its simplicity when it comes to structurally building a system. The following is a brief description of each tool utilized in the design phase.

Cadence NC-VHDL Simulator - v2000a Used to simulate the initial design for functionality purposes only.

Synopsys VHDL Analyzer, Simulator and Debugger - v2000.6 Used to compile, simulate and debug the VHDL implementation of GaS. This tool was used for all phases of simulation, functional and timing (refer to figure 4.1).

Synopsys Design Analyzer - v2000.5 This tool was used to synthesize the VHDL design into gate-level components for the XC4000 family of devices. It was replaced later in the design by the Xilinx proprietary CAD tools (Alliance and Foundation).

Xilinx Alliance - v3.1 Initially used to synthesize the Synopsys Design Analyzer outputs into a bitstream that can be loaded into XC4000 FPGAs. Later, it was found that it was better to synthesize and build the bitstream from the same tool: Foundation.

Xilinx Foundation - v2.1i Used to synthesize the VHDL design, and to build the bitstream which programs the XC4000 FPGA. This CAD tool allows for very finite control of the realized circuit, in terms of speed, area and memory constraints. It is very well tailored for the Xilinx FPGA devices, and hence is the optimal tool to be used when working with those devices.

Now, let us take a look at each of the modules that make up the HIGA.

5.2.2.1 Pseudorandom Number Generator

The RNG module is based on the theory of cellular automata (CA), proven to be better RNGs than linear feedback shift registers (LFSRs) [56]. The CA used consists of 16 alternating cells, which exhibit their behavior according to the rules 90 and 150, described in [57]:

Rule 90 $s_i^+ = s_{i-1} \oplus s_{i+1}$

Rule 150 $s_i^+ = s_{i-1} \oplus s_i \oplus s_{i+1}$

In the latter rules, s_i refers to the current cell, while s_i^+ refers to the next state of that same cell, and $\oplus$ is the exclusive-OR operator. The 16-cell CA updating itself using the 150-150-90-150 ... 90-150 proceeds to output all possible 2^{16} bit patterns (except the all 0s pattern) with very balanced probability distribution. It also provides more randomness than the LFSR based RNG. This work's RNG is, as in [8], also based on the CA scheme, but utilized RTL coding technique to implement the functionality. The module accepts a random seed value, provided by the IM upon initialization, and proceeds to output a different 16-bit pseudorandom bit pattern every clock cycle. The pseudorandom strings are sent to the following modules:

SM Sent to aid in deciding whether selection is to occur or not on particular members;

XMM Sent to aid in deciding whether crossover and/or mutation are to occur on particular members, and if true, at which point crossover and/or mutation are to occur;

FM Sent to aid in generating the first population of members.

The RNG module's ASM chart and detailed chart are shown in figures A.1 and A.2, respectively. The control path is shown in figure A.3. The data path, on the other hand, is shown on the following page, in figure A.4.

The VHDL code for the RTL realization of the pseudorandom number generator is shown in section B.2.

5.2.2.2 Selection Module

The SM uses a *grooved* roulette wheel selection algorithm to choose two members out of the current generation of chromosomes. Both chromosomes are then pitted

against each other in a tournament style selection algorithm. The hybrid selection algorithm combines the stochastic sampling with replacement methodology with the tournament selection process. When the two members are selected from the population, based on the grooved roulette wheel, their fitness values and corresponding addresses are stored in registers. The chromosomes' fitness values are aggregated, and when their running sum reaches a pre-defined value, equal to the generational fitness sum logically AND'ed with a 16-bit random string, the chromosome that caused the overflow is selected to move on to the next step of the selection algorithm. The logical AND operator is used instead of a direct multiplication, because it guarantees that the resulting value is not greater than the current generational fitness sum, and that a chromosome from the current generation will always be picked in this phase of the algorithm.

The chromosomes are then pitted against each other in the following fashion: Compare a random number r to a preset parameter k , and if r is smaller than k , then select the member with the higher fitness value, otherwise select the member with the lower fitness value. The winning chromosome is passed on, by fitness and address, to the mutover module which handles the crossover and mutation genetic operators. The selection algorithm is performed iteratively in the genetic pipeline, until the next generation is complete, whence it resets back to its initial state. The *elitism operator* is also utilized in order to retain the best chromosome from the previous generation. The member possessing the best fitness value is copied over to the next generation, by address and fitness value, from the current generation. When evolution is complete, the SM is idled and awaits its original start signal, indicating the start of a new evolutionary run.

The SM's ASM chart and detailed chart are shown in figures A.5 and A.6, respectively. The control path is shown in figure A.7. The data path, on the other hand, is shown on the following page, in figure A.8.

The VHDL code for the RTL realization of the selection module is shown in

section B.3.

5.2.2.3 Mutover Module

The XMM groups both the crossover and the mutation genetic operators into one operator, labelled the *mutover operator*. Instead of a crossover module fetching a word from memory, deciding whether to crossover or not, then passing it to the mutation module, to mutate each bit or not, and then finally storing the word into memory, the current realization of the XMM fetches the word, performs the mutover operator and stores the word back into memory.

Upon receiving a winning chromosome from the SM, the XMM proceeds to test out whether crossover is to be performed upon the member. Two values, namely LEF and HEF, are utilized in the decision making process. The LEF value represents the low end fitness value used to distinguish those chromosomes that did not fare so well in their current environment. The HEF value, on the other hand, represents the high end fitness value used to distinguish those chromosomes that fared well in their current environment. The idea is to emulate the class system used in some societies of today, where two members belonging to very different classes are less probable to crossover. The LEF and HEF values distinguish these two classes, with the result being that two chromosomes are very likely to crossover, if they belong to the same class. If, for example, one fitness value is greater than the HEF value, and the other fitness value is less than the LEF value, then that combination of chromosomes is less probably to crossover, then, say, two chromosomes whose fitness values are between the LEF and HEF values, or both greater than the HEF, or less than the LEF values. This class system introduces a natural bias for the crossover operator, and utilizes a random number to decide at which point crossover is to occur (if at all). Thus, the operator implements what is called **1-bit cast crossover**. Implied with this operator, is the need for two crossover probabilities: one to select whether crossover is to occur, and the

other to select where crossover is to occur, if it is occurring.

If crossover occurs or not, the chromosomes are cycled through their words in memory, and 1-bit mutation is applied to every bit of each member. The mutation operator inverts a single bit, if and only if, the current random bit string is smaller than the pre-defined mutation probability. The actual values of the probabilities are defined when the user enters the initializing GA parameters. Please see section 5.3 for a more detailed discussion of the GA parameters.

The XMM's ASM chart and detailed chart are shown in figures A.9 and A.10, respectively. The control path is shown in figures A.11 and A.12. The data path, on the other hand, is shown on the following page, in figure A.13.

The VHDL code for the RTL realization of the mutover module is shown in section B.4.

5.2.2.4 Fitness Module

The FM has two main purposes. The first is, upon receiving the main "Start" signal, the module proceeds to generate and evaluate the first generation of members. The generation process depends on which evolutionary stage the actual evolution is (more on this in section 5.4.5), as different stages will require different chromosome lengths. Never the matter, the main procedure is to use the 16-bit random strings output by the RNG module, and to construct the appropriately long chromosome (for example, if it is the first evolutionary stage, then the chromosomes are 32 bits long, and only require two clock cycles for each chromosome generation). Once each chromosome is generated, it is passed on to the MAM, which proceeds to evaluate the member on the host, and write out its fitness value in shared memory.

The second main task of the FM is to evaluate, after the first generation, two chromosomes sequentially to receive fitness values for each. Upon the completion of one generation of chromosomes, the FM signals the controlling software in order to obtain

a fitness sum through the MAM. Once that is complete, the FM returns back to its initial state, awaiting either the next set of two chromosomes of the next generation, or if evolution is complete, the beginning of another evolutionary run.

The FM's ASM chart and detailed chart are shown in figures A.14 and A.15, respectively. The control path is shown in figure A.16. The data path, on the other hand, is shown on the following page, in figure A.17.

The VHDL code for the RTL realization of the fitness module is shown in section B.5.

5.2.2.5 Memory Access Module

The MAM is used to control the memory accesses of each of the other HIGA modules, as well as the interaction between them and their controlling software, the latter residing on an external processor. The MAM provides a *system bus* (refer to figure 5.3) for all the modules to hook onto. The bus provides the IM with all the values needed to initialize the rest of the modules, the SM with the fitness value of the next chromosome needed to perform the hybrid selection algorithm, the XMM with a path to memory either to read a word from memory (to perform crossover and mutation on the current chromosome), or to write a word to memory (to store the chromosome after mutover operator is performed, if at all). Finally, the MAM's system bus provides the FM the means by which the fitness sum is read within the HIGA.

The MAM also interacts with the controlling software in order to control the operation of the HIGA, and most importantly the genetic pipeline. The module accepts requests and provides feedback of the state of the HIGA, in order to aid in deciding what to do next. It is imperative to discuss the fact that in shared memory, there are two generations that are stored simultaneously. The MAM controls the read/write operation to shared memory from the HIGA side, and thus reads from the current generation and writes to the next generation. If M_1 represents the current population

in memory, and M_2 represents the next population in memory, then at the completion of every generation, M_1 is swapped with M_2 and thus the current generation becomes the next generation, and the next generation becomes the current generation. The pointers are updated both in software and in the MAM, with the latter recognizing the switch, and correcting its operation so that it is reading and writing to the correct memory blocks. The shared memory also has memory blocks for the truth table values, the GA parameters, as well as a test area for future use.

The MAM's ASM chart and detailed chart are shown in figures A.18 and A.19, respectively. The control path is shown in figure A.20. The data path, on the other hand, is shown on the following page, in figure A.21.

The VHDL code for the RTL realization of the selection module is shown in section B.6.

5.2.2.6 Initialization Module

The IM is used to send the initializing values to each module. The RNG receives its seed value, the SM receives its selection probability used in the tournament phase of the algorithm, the XMM receives the mutation and both crossover probabilities for the mutover operator, as well as the LEF and HEF values used only in the crossover operator, and the FM receives the number of bits per chromosome for the current evolutionary stage. The initializing parameters are actually stored in the **programmer's model (PM)**, such that the latter can be accessed through the system bus, and signaled back to the controlling software, in order to determine the current state of the HIGA.

The IM uses the GA parameters section of memory to store the values needed for initialization, and signals each module to fetch its corresponding parameter, and to store that parameter in a SPR. The IM completes when the parameters are all stored, and idles back to its initial state, awaiting another evolutionary run.

The IM's ASM chart and detailed chart are shown in figures A.22 and A.23, respectively. The control path is shown in figure A.24. The data path, on the other hand, is shown on the following page, in figure A.25.

The VHDL code for the RTL realization of the initialization module is shown in section B.7.

5.2.3 Memory

As aforementioned, the memory subsystem (or hcache) is used to store the evolved solutions for future use. In the current realization of GaS, the memory subsystem is actually residing off-chip, in an external memory block. This is due to the lack of a system-on-chip device containing both a ROM and a customizable logic core. The availability of such devices has just been expanded to the research community, and a future realization of GaS will incorporate the hcache within the optimizer's block architecture.

Once the HIGA has completed an evolutionary run, it stores the resulting chromosome in the hcache, and proceeds to the next run through. Every solution is stored as a chromosome, along with the initializing parameters and finalizing results (such as evolutionary stage number, number of LUTs and so on). This information is used when that specific logic function is later needed on the host, and can be extracted from hcache. If the host needs a function that has not been evolved yet, and thus is not present in hcache, then the optimizer will begin running, and generate such a solution, for storage in hcache. The latter is termed so because it resembles the operation of software cache, in that a hit or miss occurs, depending on the availability of the circuit solution in memory. A solution can be compared to a cache page, and it can be referenced from the external processor controlling the optimizer's operation.

The idea behind hcache is to store all the solutions, and if that's not possible, the most recently used solutions, in a memory block. The ROM architecture of hcache

allows us to retain all the data, even after power is shut off. Thus, upon power-up, the hcache contains all the previously evolved solutions, ready for placement on the host, if the need arises. Another major advantage to having hcache, is the ease with which larger building blocks can be built, if they contain simpler functions, already realized in hcache.

5.2.4 ECLBs: Evolvable Configuration Logic Blocks

The ECLBs subsystem is used to evaluate how well the chromosomes fared during the current evolution. Ideally, as shown in figure 5.1, this block is called the host, and consists of an independent PLD, empty on power-up, and ready to accept chromosomes for fitness assignment. The ECLBs subsystem is a replacement to the host, since due to financial limitations, a second PLD could not be easily procured. However, the pressing need for a physical device as the host was extinguished through the use of a very innovative **application programming interface (API)**, developed at Xilinx Incorporated, called JBits. The latter is a set of Java classes which provide a fast and simple interface into the Xilinx Virtex FPGA bitstream. This interface operates on either bitstreams generated by Xilinx design tools, or on bitstreams read back from actual hardware. The provided capabilities, using the API, are the design and dynamic modification of circuits to be realized on Xilinx Virtex FPGA devices.

The advantage of using JBits is the simplicity with which a Virtex FPGA bitstream is viewed. The logical resources, mainly the CLBs, are viewed as a two-dimensional array of bits, while the routing resources adjacent to the selected CLBs are also made available through the interface. JBits is implemented in Java, thus bitstreams can be generated very quickly, and parameterizable cores can be easily generated and verified. The use of JBits has allowed us to emulate the functionality of a Virtex device, and thus realize the operation of the ECLBs subsystem. If the need arises for a physical Virtex device, it is very easy to port the Java implementation to the real Virtex device,

through another Xilinx interface, the Xilinx Hardware Interface (XHWIF), which allows for the use of prototyping boards instead of the JBits Virtex simulator.

Thus, the ECLBs subsystem is implemented in the Java language, and interfaces to the controlling software, the latter written in the C++ language, through the **Java Native Interface (JNI)**. The JNI allows for the following main functionalities [58]:

Invoke Java methods The JNI allows for the invocation of Java methods from native (e.g. C++) code;

Create a JVM The JNI allows for the creation of a Java virtual machine (JVM) from native code;

Function table access The JNI allows for all JVM accesses, from the native side, to occur through a function table residing between the native side and the JVM.

JNI also allows for communication between the JVM and the native side, thus scenarios such as calling a C++ class method from a Java method are possible, but are not covered in this thesis, as that functionality is not useful. The main idea is that the communication path between the Virtex JBits simulator and the HIGA is done through the JNI. As each chromosome reaches the fitness evaluation stage, it is transmitted to the ECLBs subsystem and decoded in the JBits application running on the host. The Virtex bitstream is built from the outputs of the decoding process, and is instantiated onto the Virtex JBits simulator. The device that is being utilized is the XCV50, which contains a maximum of 57,906 system gates and 180 I/O pins. The reason this device was selected is because it is the smallest (in terms of gates) Virtex device, and suffices for the experiments described herein. After the instantiation of the bistream onto the device, the circuit is tested by running through all the different available input combinations for the particular boolean function being evolved. The outputs are recorded and sent back to the native side for further processing. The native side, realizing the optimizer's

controlling software and residing on an external processor, will be discussed in section 5.3.

5.2.5 Processor Interface

The processor interface is seen in figure 5.1 as the CPU interface. It consists of a set of status, control and data registers which realize the programmer's model of the optimizer, in order to allow for a simple handshaking protocol between the hardware and software components. Refer to figure 5.4 for a detailed breakdown of each SPR. There are five status and control registers, mainly one for each of the selection, mutover, fitness, RNG and initialization/memory access modules. All status and control registers are 8-bits wide. There are also two main data registers, one containing the fitness module's current fitness value, and the other containing the current generational fitness sum. Both data registers are 32-bits wide. Additionally, there are four peripheral component interconnect (PCI) bus registers, used to communicate between the HIGA and the external processor running the controlling software. These registers are all 32-bits wide, and are the actual components that make up the processor interface.

The implementation details of each register are spared here, but the reader is referred to Appendix B for the VHDL code of all modules, and the actual realization of each status and control bit, as well as the data and PCI registers. As well, the reader can reference the bits in figure 5.4 to the ASM charts in Appendix A to understand how each bit is realized in hardware.

Figure 5.4: The GaS programming model

5.3 A Look at the System Operation

The host having been introduced and discussed in section 5.2.4, it is now time to take a look at the optimizer. The latter is composed of the HIGA and its controlling software. The software is a device driver, that permits us to control the operation of the HIGA, and thus will be referred to as so for the rest of this writing. The device driver is, as aforementioned, implemented in the C++ language. It controls the execution of the HIGA through the processor interface. It also allows for the communication path between the Virtex simulator and the HIGA, through the JNI. A JVM is hence instantiated from the device driver, in order to assemble the valid bitstream encoded in the chromosome.

The optimizer operation is shown in figure 5.5. As can be seen, the initial step is to generate the first population of members. The FM takes care of this step, by finding the fitness values for the first chromosomes. The SM subsequently takes over and selects 2 members out of the current population according to the already presented hybrid selection algorithm. The XMM then proceeds to crossover and mutate the selected chromosomes, and send the resulting members to the FM. The latter proceeds to instantiate the chromosomes into the ECLBs subsystem and evaluate each chromosome, and stores the member, along with its fitness value, into the next generation. This process is iterated over until 24 members are selected by the SM, as the last member is the elite member (there are 25 chromosomes per generation, as later seen in table 5.1). Once that is complete, the generational pointers are switched, resulting in the current generation becoming the next, and the next generation becoming the current, and the fitness sum is calculated and stored in data register 1.

As can be seen in figure 5.6, the system has three main components interacting with each other: the HIGA, the device driver and the host. There also are four triggering events: global reset (GR), evaluate, assign and update. Upon receiving the first trigger,

Figure 5.5: The optimizer operation

both the HIGA and the device driver enter their respective initialization states. The HIGA proceeds to enter the genetic pipeline operation, and when trigger 2 arrives, it indicates to the device driver, that a chromosome is ready for evaluation. That event causes the device driver to signal the host in order to decode, instantiate and test a circuit on the Virtex simulator. The results are passed back from the host to the device driver to the HIGA, which makes up the third trigger, indicating the assignment of a fitness value to a chromosome. If a generation has just completed with the latest chromosome evaluation, then the fourth trigger (update) is activated, which results in the calculation of the fitness sum and the generational switch in the device driver, otherwise the genetic pipeline starts up again. As well, if evolution is complete, hence a solution has been located, then the whole process terminates, and all components enter their finalizing state.

Figure 5.6: The system operation

Below is table 5.1 which contains the actual GA values that optimized the operation of GaS. As well, information such as the stopping condition and the encoding scheme are presented.

Parameter	Description
Population size	25 chromosomes per generation
Chromosome size	Variable (depends on evolutionary stage) See section 5.4.5
Encoding scheme	Binary coding
Number of generations	10000 per evolutionary stage
Crossover and mutation probabilities	Crossover - 0.85, Mutation - 0.01
Selection probability	Tournament - 0.75
Fitness function	Direct Boolean truth table mapping
Selection scheme	Elitism + roulette wheel + tournament
Crossover scheme	1-bit random cast crossover
Mutation scheme	1-bit random standard mutation
Stopping scheme	A perfect member must be present in current generation AND fitness average must be over 90% of ideal fitness value
Initial random seed	Variable every evolutionary stage
Random number generation scheme	Cellular automata (16-cell)
First generation members	Generated randomly

Table 5.1: Actual GA parameters and factors

5.4 Novelties and Applications

In this section, a brief discussion of the project's novelties and applications is presented. The major application for this endeavor is the capability of evolving functional and combinational circuits onto today's FPGA devices. The combination of technology mapping with evolvable hardware permits for this application, and it is foreseen that complex circuits will be designed in a similar fashion.

5.4.1 A Brief Example

Before we present the novelties and applications, let us first take a look at an example that will introduce the terminology utilized in this section. A chromosome is nothing more than a series of binary digits, that encode a certain trait. A sample chromosome is shown in figure 5.7. The chromosome is made up of 68 bits, because it is a stage 2 chromosome. Each stage's chromosomes have different lengths to account

for the increasing complexity of the Boolean function. A more detailed description will be presented in section 5.4.5.

The sample chromosome shown in figure 5.7 is made up of the following fields:

A, B, C and D (4 bits) Each of these fields is used as a variable pointer to indicate which input variable connects to the A, B, C and D inputs of the first look-up table, LUT0 in this case;

E, F, G and H (1 bit) Each of these fields is used as a control bit to designate whether the E, F, G and H inputs of the second look-up table should receive their signal from the previous LUT output or the original input variables. If the control bit is a logical '0', then the appropriate input receives its signal from the input variable pointed to by the next variable pointer. However, if the control bit is a logical '1', then the appropriate input receives its signal from the previous LUT output;

E, F, G and H (4 bits) Each of these fields is used as a variable pointer to indicate which input variable connects to the E, F, G and H inputs of the second look-up table, LUT1 in this case;

LUT0 and LUT1 (16 bits) Each of these fields is used to hold the *truth table values* of each sub-function being evaluated by the chromosome.

Figure 5.7: A sample chromosome

Referring to figure 5.7, we can see that the chromosome is made up of groups of bits, that each represent a certain resource on the FPGA. To better analyze the

chromosomes, they are viewed in hexadecimal format instead of in binary. Thus, the sample chromosome presented above would be translated to *0x247861c32ba36de2a* in hexadecimal (0x is used here to denote a hexadecimal number). As well, the sample chromosome is a stage 2 chromosome, and thus has that specific configuration. Other chromosomes will have similar configurations, but will be of various lengths, as they have to cover a different number of LUTs and interconnections. Finally, by viewing the sample chromosome, we can notice that it possesses potential sub-functions in the LUT0 and LUT1 fields. The chromosome is attempting to solve an **arbitrary six variable Boolean function**, by placing it into two LUTs, rather than the traditional three LUTs.

Refer to figure 5.8 to view the basic structure of a 4-input LUT. As can be seen, there are four inputs (*A, B, C* and *D*) and one output (*f*). Within the LUT are 16 *memory locations* that store the bit that is to be accessed when a particular input combination is presented at the inputs. For example, if *ABCD = 0010*, then the bit (a logical 1 in this case) stored at the memory location tagged by *0010* in figure 5.8 is accessed and placed on the *f* output line.

Figure 5.8: A 4-input LUT structure

Before moving to the following description, refer to figure 5.9, which represents the circuit embedded within the aforementioned sample chromosome, currently at the

second stage of evolution.

A input (LUT 0) The A connection of the first LUT is variable x_2 , since *0010* is 2 in binary;

B input (LUT 0) The B connection of the first LUT is variable x_4 , since *0100* is 2 in binary;

C input (LUT 0) The C connection of the first LUT is defaulted to ground, since the variable being pointed to (7) is not available in this configuration;

D input (LUT 0) The D connection of the first LUT is defaulted to ground, since the variable being pointed to (8) is not available in this configuration;

E input (LUT 1) The E connection of the second LUT is variable x_1 , since the first E bit is a 0, meaning that there is no connection between the output of the first LUT $f1$ and E. As well, *0001* points to variable x_1 in binary;

F input (LUT 1) The F connection of the second LUT is the output of the first LUT, since the first F bit is a 1, meaning that there is a connection between $f1$ and F, which effectively overrides the variable being pointed to (12);

G input (LUT 1) The G connection of the second LUT is the output of the first LUT, since the first G bit is a 1, meaning that there is a connection between $f1$ and G, which effectively overrides the variable being pointed to (3);

H input (LUT 1) The H connection of the second LUT is variable x_4 , since the first H bit is a 0, meaning that there is no connection between the output of the first LUT $f1$ and H. As well, *0001* points to variable x_1 in binary;

LUT 0 values The first LUT contains the 16 values that are shown in the *LUT 0* component of the sample chromosome;

LUT 1 values The second LUT contains the 16 values that are shown in the *LUT 1* component of the sample chromosome.

Figure 5.9: A circuit based on the sample chromosome

By viewing the sub-function outputs, $f1$ and $f2$, we are able to verify whether this chromosome has a high or low fitness value, by sending the truth table combinations to all six variables, and recording the sub-function outputs. Since in this particular example, there are six variables, we will be required to store 64 (2^6) 1-bit outputs per sub-function. Hence, each sub-function, in this case, outputs a 64-bit output, representing the actual values when this chromosome is fitness evaluated. If one or both of the sub-functions closely match the input truth table output values, then the chromosome is assigned a high fitness value to indicate that it contains a possibly useful sub-function; otherwise the chromosome is assigned a low fitness value to indicate that it does not possess any useful sub-functions for the purpose of functional decomposition.

5.4.2 LUT Value and Routing Evolution: Apprehensive Combination

Levi and Guccione [59] from Xilinx, realized a Java-based tool (GeneticFPGA), based on the HereBoy [60] algorithm, which evolves the LUT values, to attempt to find a circuit solution. The design, however, utilizes static routing, which does not allow for the routing resources within the Virtex device to be modified by the algorithm. GaS,

on the other hand, evolves **both the LUT values and the inter-CLB**, as well as **intra-CLB routing** in order to find a correct solution to the input function. GaS allows the GA to modify the LUT values, as well as the routing resources adjacent to the the LUT. This provides for a more powerful evolutionary environment, which has total control of the Virtex device. This is one of the first designs to take advantage of both resources, and although, GaS has many more solutions to search through, it does have a higher inclination to find a better (minimal) solution.

GaS guarantees the instantiation of synchronous circuits onto the Virtex device, by ensuring the absence of any asynchronous feedback loops and by ensuring that the clock is not gated at any place in the circuit. The first condition is met by guaranteeing that the LUT inputs only come from flip-flop outputs, and the second condition is met by routing one global clock to all the synchronous devices in the FPGA. These two conditions account for only instantiating synchronous circuits, thus preventing any coupling between the output data and the circuit producing that data. As well, using synchronous circuits allows for simple understanding of the circuit's behavior by examination of the output data. Finally, the combination of both LUT value and routing evolution is made partly possible by the JBits API, due to the partial reconfiguration and the resource contention avoidance functionalities. The API allows for partial reconfiguration of the Virtex device by saving the old bitstream in memory and only modifying, on the device itself, the sections that have been changed in the new bitstream. As well, the API has built-in functionality that makes sure that resource contention is avoided, and thus prevents any *randomly-generated circuits*, such as the ones in this work, to damage the device, such as two sources driving the same resource.

5.4.3 Selection Algorithm: Knights in Battle

The chosen selection algorithm has been previously discussed in section 5.2.2.2. The caveat is the combination of the roulette wheel selection and the tournament selec-

tion, which provides the GA with more flexibility, to determine which chromosome gets selected to move on the next generation. The first pass through involves the regular roulette-wheel scheme, which generates two members. These two members then face off each other, similarly to knights doing battle, in order to determine which member goes on to the next generation.

5.4.4 Chromosome Encoding

As aforementioned, the evolution takes advantage of both the LUT and routing resources present on the FPGA. Thus, the chromosomes are made up of both LUT and route bits (refer to figure 5.16). The LUT bits are used to set the appropriate LUT, while the route bits are used to decide whether the input to a LUT comes from an input variable or the previous sub-function output. Note that if a LUT input has been assigned to a previous sub-function, then no input variable can be routed to it.

The chromosomes determine how the circuit is mapped and routed, while evolving with each generation. The LUT bits are mapped out to the actual LUTs in the FPGA, and the route bits select certain lines within the FPGA Virtex slice to decide where the LUT input is coming from, and where the slice output is going to. These decisions are done by the decoding process, which takes place in the simulator, and forms the valid bitstream from the chromosome bits.

5.4.5 Evolutionary Stages: Till next time...

There are five stages of evolution. Each stage uses the same number of chromosomes per generation, but differently sized chromosomes, thus this work is also classified under the variable GA (VGA) heading. The reason for the various lengths is the increasing complexity inherited with every stage. Only feed-forward paths are allowed, and the LUT inputs are, as mentioned, guaranteed to be registered outputs. Refer to figures 5.10, 5.11, 5.12, 5.13, 5.14 in the following description:

Stage 1 Uses 1 CLB and generates 1 potential sub-function. Each chromosome in this stage is 32 bits long.

Stage 2 Uses 2 CLBs and generates 2 potential sub-functions. Each chromosome in this stage is 68 bits long.

Stage 3 Uses 4 CLBs and generates 4 potential sub-functions. Each chromosome in this stage is 128 bits long.

Stage 4 Uses 8 CLBs and generates 8 potential sub-functions. Each chromosome in this stage is 352 bits long.

Stage 5 Uses 16 CLBs and generates 16 potential sub-functions. Each chromosome in this stage is 864 bits long.

Figure 5.10: Evolutionary stage 1

It is also interesting to note the number of bytes per chromosome per stage as follows:

Stage 1 8 bytes, equivalent to 2 word (32-bit) transfers

Figure 5.11: Evolutionary stage 2

Stage 2 12 bytes, equivalent to 3 word (32-bit) transfers

Stage 3 20 bytes, equivalent to 5 word (32-bit) transfers

Stage 4 48 bytes, equivalent to 12 word (32-bit) transfers

Stage 5 116 bytes, equivalent to 29 word (32-bit) transfers

Thus, the worst-case required memory space to store the chromosomes is 1 Mb, if the evolution is in the fifth stage, and there are 16 potential sub-function outputs, and 64k truth table outputs ($16 * 65536 = 1048576$). As well, the worst-case number of word transfers is 58, again in the fifth stage, with two chromosomes being evaluated ($2 * 29 = 58$).

The chromosome structure of every stage is presented in figure 5.15. It is interesting to note that depending on the chromosome length, pad values are added to respect the 32-bit boundary. As well, the current and next generations are both initialized to 1's, and are then modified by the writing of the chromosome structures.

5.4.6 Hardware cache: hcache

The hcache has already been discussed in detail, but is outlined again to present its novelty. With the introduction of parallelism into GaS, hcache will become a common storage facility that any host can use to decide whether to evolve the specific truth table or not. The hcache allows for the integration of different structures within the overall system, through a 'plug-and-play' feature. It also aids in SoC designs, by supplying a firm representation of a virtual component (VC). The latter is accomplished by the representation of a digital circuit in every chromosome. As the circuits become more and more complex, the encoded evolved solutions will hold the structure of a VC, which can be used to build intellectual property (IP) cores for use in a SoC design environment.

Figure 5.12: Evolutionary stage 3

Figure 5.13: Evolutionary stage 4

Figure 5.14: Evolutionary stage 5

Figure 5.15: The chromosome structure

Figure 5.16: The chromosome encoding scheme

Chapter 6

Investigations, dissections and explorations

"When I examine myself and my methods of thought, I come close to the conclusion that the gift of fantasy has meant more to me than my talent for absorbing positive knowledge"

(Albert Einstein)

This chapter will present the results that were obtained after going through multiple evolutionary runs. The main goals are the minimization of logical resources (number of LUTs) and processing time. There are numerous methods to analyze the results, and a few are presented herein.

6.1 Functional

The first analysis method consists of a functionality test. Does the system properly function? In a short answer, yes. The system correctly evolved a myriad of simple Boolean functions. The first of such evolutions, is the 2-input XOR function. The latter was chosen as it is the simplest and proves whether the system works or not. The evolution was allowed to take place, and the results are shown in table 6.1. As can be seen, the obtained elite chromosome in the final population was 0x1d06043e. That member also appears 11 other times in a population of 25 members. Its fitness value is 4, which is the optimal fitness value, as this is a 2-input XOR function, and thus will have 4 different outputs. Since the member's fitness value is 4, then it correctly matches all the outputs. It is also interesting to note that the fitness sum went up from a first

population fitness sum of 45 to a final population sum of 77, meaning the mean fitness was raised from 1.8 to 3.08. One final note about this evolutionary run is that it took 173 generations to complete, with each generation taking a mean of 120 seconds of total computational time, for a total of 5.8 hours of evolutionary time.

6.2 Tascal

This section describes the second analysis methodology. It concentrates on the *tascal* aspect of the system. The main point in this case is the evolution of tasks with varying parameters. These tasks are more complex functions, which are tougher on the synthesis tools to map onto LUTs.

6.2.1 Boolean Function Evolution

The subsequently evolved function was the 3-input AND function. Any other 3-input function could have been chosen in this case, but the AND function was selected for its popular use. The results are shown in table 6.2. As can be seen, the obtained elite chromosome in the final population was 0xb9f573ae. That member also appears 3 other times in a population of 25 members. Its fitness value is 8, which is the optimal fitness value, as this is a 3-input AND function, and thus will have 8 different outputs. Since the member's fitness value is 8, then it correctly matches all the outputs. Again, the fitness sum went up from a first population fitness sum of 77 to a final population sum of 153, meaning the mean fitness was raised from 3.08 to 6.12. This evolutionary run took 2070 generations to complete, with each generation taking a mean of 100 seconds of total computational time, for a total of 57.5 hours of evolutionary time.

6.2.2 Input Parameter Variation

Let us now vary the input parameters, in order to see whether it has much of a difference on the outcome. The main parameters that were modified are the GA

initializing parameters, including the selection, crossover and mutation probabilities. After careful analysis and numerous evolutionary runs, the optimal parameters were found to be the following

Selection probability The optimal range for this probability was found to be 0.65-0.85 (for the tournament-style selection)

Crossover probability The optimal range for this probability was found to be 0.6-0.7

Mutation probability The optimal range for this probability was found to 0.1-0.15

The next evolutionary run takes us to 4-input Boolean functions, using parameters in the optimal ranges. The results are presented in table 6.3. As can be seen, the obtained elite chromosome in the final population was 0xd76f31fe. That member also appears 3 other times in a population of 25 members. Its fitness value is 16, which is the optimal fitness value, as this is a 4-input Boolean function, and thus will have 16 different outputs. Since the member's fitness value is 16, then it correctly matches all the outputs. For the fourth time, the fitness sum went up from a first population fitness sum of 182 to a final population sum of 296, meaning the mean fitness was raised from 7.28 to 11.84. It is very interesting to note that this evolutionary run only took 438 generations to complete, with each generation taking a mean of 110 seconds of total computational time, for a total of 13.38 hours of evolutionary time. The much-improved total computational time is attributed to the optimized GA parameters and the flexibility of the entire system.

In order to further prove the success of the system, a 5-input Boolean function was evolved. The reason for this trial is that any current synthesis solution (academic or commercial) will map out the 5-input function truth table into 2 separate 4-input LUTs, since there are 32 possible outputs, and 16 possible locations in each 4-input LUT. Thus, the 5-input variable truth table will require two LUTs to map out its functionality. Let us take a look at how GaS performed (table 6.4). Note that the actual Boolean function

is irrelevant in this case, but just in case the curiosity of the reader gets the better of them, the realized Boolean function is

$$F = [(A \cdot B) + (C \cdot D)] \oplus E \quad (6.1)$$

As can be seen from the previous table, the obtained elite chromosome in the final population was 0xe7046ee4. That member also appears 5 other times in a population of 25 members. Its fitness value is 32, which is the optimal fitness value, as this is a 5-input Boolean function, and thus will have 32 different outputs. Since the member's fitness value is 32, then it correctly matches all the outputs. For the fifth time, the fitness sum went up from a first population fitness sum of 414 to a final population sum of 696, meaning the mean fitness was raised from 16.56 to 27.84. It is also greatly interesting to note that this evolutionary run only took 1252 generations to complete, with each generation taking a mean of 100 seconds of total computational time, for a total of 34.78 hours of evolutionary time. The much-improved total computational time is attributed to the optimized GA parameters and the flexibility of the entire system. As well, the final and most important analysis aspect is the fact that GaS was able to find multiple solutions for the mapping of a 5-input variable Boolean function onto a 4-input LUT. The most optimal solution is the elite, but in case there are other constraints on the system, there are many other ideal solutions that have optimal fitness values.

6.3 Structural

This section describes the third analysis method. This particular method pays attention to the structure of the circuit solutions that were generated in the evolutionary runs. In this case, we will take a look at the 2-input XOR as it is the simplest one to analyze, while the remaining ones are too obtrusive for explanation.

6.3.1 LUT and Routing Resources

If we take a look at the final chromosome of the 2-input XOR function, mainly 0x1d06043e, we notice that the chromosome contains eight hexadecimal digits, and thus 32 bits in all. Referring back to figure 5.8, the first hexadecimal digit represents the input variable that connects to the A input, and since it is a 0x1, then it is the first input variable, x_1 . The second hexadecimal digit represents the input variable that connects to the B input, and since it is a 0xd, then a ground is connected to B, as there are only 2 variables. The third and fourth hexadecimal digits are decoded in a similar fashion. All that remains now is the LUT resources. The last four hexadecimal digits, namely 0x043e, represent the values for the 4-input LUT in CLB1 and slice 1. Another note to keep in mind, is that as the evolutionary stages grow, the number of bits per chromosome grows. This means there is more decoding to be performed, but the same rules explained in section 5.4.2 apply.

6.4 Behavioral

The final analysis method is the behavioral one. In this case, the main issues involve what occurred during the evolutionary runs, and whether it matched the underlying theory or not. As previously presented, GAs operate on the underlying principle of the Schemata Theory. It is easily noticeable that practice meets theory in this case, as in all the evolutionary runs, low-order and high-valued schemas appeared in the final population within the remaining, or surviving, members. For example, looking at the 5-input function evolution, the schema $\{^*704^*ee4\}$ is prevalent in most of the members of the final population. That schema proved too powerful to be subdivided by the crossover and mutation operators, and was repeatedly picked by the selection algorithm to move on to the next generation. The same analysis can be matched to the first four evolutionary runs.

Chromosome Number	Initial Population Chromosome	Fitness Value	Final Population Chromosome	Fitness Value
1)	68746408	3	1d06043e	4
2)	f011f434	1	1d06043e	4
3)	68746408	3	dd16b43e	1
4)	7e59272d	3	0106043e	2
5)	e0432240	3	1d16b43e	1
6)	f571fe5d	1	1d06043e	4
7)	3bcc732e	3	1d06043e	4
8)	70e16802	2	1d06043e	4
9)	74cb72ce	3	dd06042e	3
10)	c65d7e72	3	1d06043e	4
11)	435d6585	3	1d06045e	4
12)	e0a2d557	1	1d06043e	4
13)	e203a64d	1	1d06043e	4
14)	a41d3197	3	5d04843e	1
15)	4f7ae596	1	1d06043e	4
16)	96cd0934	1	1d06043e	4
17)	c34c999b	1	dd02443e	3
18)	aa540900	3	1d06043e	4
19)	7924e60b	1	1d06043e	4
20)	8b7c976c	1	dd06043e	3
21)	6ae6cdf2	1	1106b47e	1
22)	e2931d7c	3	0d06043e	3
23)	dc06743f	3	1106b47e	1
24)	1d5eba34	1	dd06043e	3
25)	11b179af	3	dd06743e	3

Table 6.1: 2-input XOR function evolution

Chromosome Number	Initial Population Chromosome	Fitness Value	Final Population Chromosome	Fitness Value
1)	afa76761	7	b9f573ae	8
2)	afa76761	7	b15773ae	5
3)	7ce97d5d	7	b9f573ae	8
4)	cc39df10	1	01f573ae	3
5)	05560b65	7	81f573ae	5
6)	8e66c060	1	31d573ae	5
7)	994b2e46	7	01f573ae	3
8)	7d60c9be	1	b1f573ec	5
9)	e88ee9e8	1	b9f573ae	8
10)	95d1add7	1	19d533ee	7
11)	7cd81fd5	7	01f573ae	3
12)	26340aac	7	b1f573ae	5
13)	50717bb1	3	b17573ae	5
14)	92f41b05	7	b1f573ae	5
15)	28ac61fb	5	b9f573ae	8
16)	c1bfc841	3	b1f573ae	5
17)	4023f654	3	31f573ae	5
18)	d0b53ec8	5	31f573ee	5
19)	04721703	7	08f573ae	5
20)	4b2eed71	1	315773ae	5
21)	eae0d936	3	b0f573ae	5
22)	164784cc	3	81f572ae	5
23)	f143e0d4	1	b1f573ae	5
24)	372b045b	7	01f553ae	5
25)	93febe11	1	b1f57ba6	5

Table 6.2: 3-input AND function evolution

Chromosome Number	Initial Population Chromosome	Fitness Value	Final Population Chromosome	Fitness Value
1)	a3e926af	12	d76f31fe	16
2)	4a54ae5d	4	d76f21fe	14
3)	714af701	4	d76f312c	14
4)	a3e926af	12	7977f5fe	15
5)	8bcb93ce	4	d72f31fe	16
6)	e692ee68	4	d72f31fe	16
7)	7d5bd985	4	d52f31fe	14
8)	808f45bc	12	576f31fe	14
9)	083a46b6	8	d76f31fa	14
10)	bb0c7cdf	8	d76f31fe	16
11)	5f032e17	8	d76f31fe	16
12)	3cb8aa0b	4	d76f31fe	16
13)	8ce30216	12	d76f21fe	14
14)	21e90f9f	12	d74b81fe	14
15)	fc2b628a	12	574f31fe	14
16)	575d0414	12	d748f72c	14
17)	8b0a3182	12	774b01be	15
18)	6824dae9	4	470f31fe	14
19)	726341e1	12	d74b01fe	14
20)	34e6b6d3	4	574b01fe	14
21)	7942192a	12	574b31fe	14
22)	6d0b7c87	8	d74fb1fe	14
23)	ae777982	12	d74f31fe	16
24)	6e2366ab	12	d74f31fe	16
25)	01f5741d	6	d74f31de	14

Table 6.3: 4-input Boolean function evolution

Chromosome Number	Initial Population Chromosome	Fitness Value	Final Population Chromosome	Fitness Value
1)	c7040ae2	22	e7046ee4	32
2)	9c765bc4	16	e7046ee4	32
3)	4cc8a79a	16	e7044ef6	32
4)	7b7f8e9e	16	e7044ee4	32
5)	a1e85542	16	c7046ef6	32
6)	832b6097	16	c7444ee4	32
7)	e6f7a9c4	16	f7044ed4	22
8)	1f6b599f	16	e7044ee4	32
9)	ca44c5fa	16	e7046ee4	32
10)	a3e96be7	16	e7046ee4	32
11)	9acba5a1	16	e7046ee4	32
12)	b9c679de	16	a7444ee6	16
13)	99de9a8d	16	e7044ee4	32
14)	4164dcc0	16	e741cea4	16
15)	efc44c6f	16	e6064ef4	18
16)	674b8600	16	07044ee4	20
17)	46166ba5	16	07144ee4	18
18)	6334d514	16	07205ce4	20
19)	ea549f5a	16	c7044ce4	32
20)	d21eb578	16	c7044ef4	32
21)	4c3d3077	16	c7044ce4	32
22)	b29calb0	16	c3044ee4	22
23)	f8f0353c	16	e7046ee4	32
24)	c7040ae2	22	e7044ee4	32
25)	a107d2a5	10	c7046ef4	32

Table 6.4: 5-input Boolean function evolution

Chapter 7

Prototyping GaS

**"I have offended God and mankind because my
work didn't reach the quality it should have"**

(Leonardo da Vinci)

The GaS system presented in figure 5.3 was prototyped and tested. In this chapter, the details of the prototyping environment will be presented in order to give the reader a better practical feel for the system.

7.1 Hardware Environment

The hardware component of GaS consists of the HIGA and the hcache. Actually, the host (or ECLBs subsystem) is also considered part of the hardware component, but the JBits API is being used as a software replacement in this work. The HIGA is the major piece of the component, since the hcache is simply external memory. Thus, the major discussion of the realization of the hardware environment will be based on the requirements of the HIGA subsystem. The major decision that was to be taken was whether the hardware environment should be a stand-alone one or a coprocessor-type one. The stand-alone environment has many advantages including ease of portability, and inherent flexibility. On the other hand, the coprocessor-type environment is enhanced by the presence of the processor and all its associated peripherals. The latter was chosen because of its availability and enhanced capabilities.

7.1.1 A Look at the Requirements and Candidates

While selecting a test bed hardware environment, it was very important to monitor the gate level count of the HIGA subsystem, as that will be one of the underlying factors in the decision process. The integral point is that the selected device/board must have enough logic to support the expanding design of the HIGA subsystem and any future spin-offs. Let us now take a brief look at the requirements of the hardware environment:

Gate count for optimizer The optimizer must have a reconfigurable device present that is capable of supporting the HIGA subsystem, with a few gates to spare for future work;

PCI bus availability The chosen board must have access to the PCI bus in order to communicate with the host, which is being simulated on the external processor through the JBits API;

Shared memory The chosen board must have enough (at least 2 Mb) of memory to store the current and next generations of chromosomes. As well, the memory must be shared between the optimizer board and the external processor;

hcache memory Either the shared memory or the external processor's memory will have to be used as hcache;

Prototyping area The chosen board must have available space for prototyping, as well as access to the FPGA pins.

The requirements were very stringent, and as the HIGA design was nearing completion, it was very obvious where the gate level count would fall. Prototyping boards were researched and listed in table are the possible candidates.

Table 7.1: The hardware environment candidates

Company	Product	Description
Avnet	Xilinx prototyping board	<i>XCV300BGA432 with a 64-bit PCI bus interface 16 MB of SDRAM, 2 USB and 1 bi-directional RS 232</i>
Insight Electronics	Xilinx prototyping board	<i>XCV100-4PQ240 with a PCI bus form factor 1 RS 232 and a 32x6 integrated LCD display</i>
Associated Pro	APS-X240 board	<i>XCV50QFP240 with an ISA carrier board 1 RS 232 and modules can be stacked together</i>
Virtual Computer Corporation	HOT II board	<i>XC4062XLA with a PCI bus interface 4 MB of SRAM and prototyping area</i>
Alpha Data	ADM-XRC board	<i>XCV400 with PCI bus interface 8 MB of SDRAM and prototyping area</i>
XESS	XSV-50 board	<i>XCV50 on a stand-alone board</i>

7.1.2 "The Chosen One"

The selected platform was the VCC HOT II board (see figure 7.1 courtesy of [61]). The selection fulfills all the state requirements, in that the FPGA (XC4062XLA) is large enough to fit the HIGA subsystem, as well having PCI bus support, enough memory for both generations and a decent prototyping area. All these characteristics are met by the HOT II board, with the addition of the HOT daughterboard card, in order to provide for the prototyping area, as well as access to the FPGA pins for debugging purposes through the use of a logic analyzer. Each of the other candidates had its pros and cons, but the ultimate decision fell on the overall flexibility provided by the board and the support provided by the vendor. Both of these were the main advantages of the HOT II board, which also contains 2 Mbytes of configuration flash memory, 512 Kbytes of configuration SRAM cache, a programmable clock (360 KHz to 100 MHz), PCI target and initiator interfaces, support for the major commercial operating systems, and a multitude of design examples.

7.1.3 Brief Synthesis Details

The completed HIGA synthesized in 1631 flip-flops (out of a total 4608, for a utilization percentage of 35), and 1487 CLBs (out of a total of 2304, for a utilization

Figure 7.1: The HOT II and daughterboard cards

percentage of 64).

The following is a verbatim outlook from the Xilinx synthesis tool (Foundation):

Design statistics:

Minimum period: 29.998ns (Maximum frequency: 33.336MHz)

Maximum path delay from/to any node: 24.807ns

Number of External IOBs	172 out of 193	89%
Flops:	42	
Latches:	0	
Number of Global Buffer IOBs	2 out of 8	25%
Flops:	0	
Latches:	0	
Number of CLBs	1487 out of 2304	64%
Total CLB Flops:	1632 out of 4608	35%
4 input LUTs:	2019 out of 4608	43%

3 input LUTs:	762 out of 2304	33%
Number of BUFGLSs	2 out of 8	25%
Number of READBACKs	1 out of 1	100%
Number of READCLKs	1 out of 1	100%
Number of STARTUPs	1 out of 1	100%
Number of TBUFs	358 out of 4800	7%

7.2 Software Environment

The software component of GaS consists of the device driver and the host, being simulated through the JBits API. The former was developed in the C++ language, while the latter was developed in the Java language. In this case, the major decision fell onto the methodology by which the device driver and the host will be communicating. The system requires both entities to be in direct synchronization with the hardware, hence it is not helpful for them to be residing on separate platforms. Thus, the decision was made to have both the device driver and the host on the same platform, in direct coupling with the external processor. This saves the author from having to carry overhead control data between the two entities, and from the design of a handshaking protocol.

7.2.1 A Look at the Requirements and Candidates

The main requirements in this case, consist of direct control of the HIGA subsystem by the device driver, and the seamless communication between the device driver and the host. In the case of the former requirement, it was easily met by the supporting APIs provided by VCC, in order to access the HOT II board, and eventually the XC4062 XLA device residing on the board. The APIs are implemented in C++, and thus the device driver was implemented in the same language for conformity. As for the second requirement, there were two main candidates:

Candidate	Advantages	Disadvantages
CORBA	<ul style="list-style-type: none"> -Capable of integrating OS-independent, language-independent and platform-independent modules together -Allows for dynamic modification of code without shutdown of server 	<ul style="list-style-type: none"> -Extremely large memory and processor requirements -Not generic and not easily integrated
JNI	<ul style="list-style-type: none"> -Capable of having C/C++ method invocations from Java, and vice-versa -Extremely portable and flexible -Easily integrated into existing modules 	<ul style="list-style-type: none"> -Poor garbage collection schemes -Instantiations of 2 JVMs is not feasible

Table 7.2: The software environment candidates

7.2.2 "The Chosen One"

As aforementioned, the JNI was used to provide a communication path between the device driver and the host. The reason is that the JNI provides a flexible and easily integrated solution to the issue at hand. The fact that the JNI solution only requires the designer to include certain dynamic library calls, and to modify the syntax for the invocation of Java methods from the native side, far outweighs the advantages offered by the Common Object Request Broker Architecture (CORBA) solution. As well, the latter requires a large amount of processing and memory overhead that are not available in this work.

The JNI solution is quite sufficient, but does suffer from the two drawbacks mentioned in table 7.2. The automated garbage collection schemes offered by Java are very useful when it comes to dealing with Java applets or applications, but when the native side is involved, much research and work is still ahead of the designers of such tools. The other major drawback is the difficulty of instantiating more than one JVM from the native side. This greatly hindered the parallelization schemes of GaS, as it did not allow for multiple chromosomes to be evaluated simultaneously. These limitations are mentioned in a later chapter, but are worth being introduced in this section.

7.3 H/W to S/W: "Would you marry me?"

As the last section of this chapter, it is interesting to note the different hardware to software interactions that occur in GaS.

ASM → VHDL → H/W The design methodology took the author from the ASM design charts, to the VHDL realization of these ASM charts in conjunction with the device driver controlling software. Finally, the VHDL code was optimized and synthesized into hardware that was instantiated on the XC4062XLA device;

JBits → Java → S/W The JBits API, along with the Java programming language, were used to simulate the functionality of an actual Virtex FPGA. The resulting software took care of the decoding of each chromosome and its instantiation on the Virtex FPGA;

H/W → DD → S/W The main processing flow consists of the actual hardware running the HIGA, communicating with the device driver through the programmer's model, which in turn communicates with the host through the JNI.

Chapter 8

(E)Valuations

**"Perfection does not exist. To understand this is the triumph of human intelligence;
to expect to possess it is the most dangerous kind of madness**

(Alfred de Musset)

Having realized and verified the system, it is now time to briefly look at the valuations and evaluations of its results. There are two major drawbacks to GaS; the first being the poor utilization of the automated garbage collection schemes when it comes to JVM-native code interaction, and the second is the JBits simulator's long computational times. Both of these drawbacks are associated with the Java language, and thus both will be eliminated in future versions of GaS, whence an actual Virtex device will take the place of the simulator, and true online evolution will occur. The system will become a hardware one, only requiring an extra processor in order to execute the device driver code. The latter can be accomplished through the use of SoC-type chips, which eventually, will be able to store all of GaS' components onto one single device.

8.1 Comparison with a Hardware-Based GA

GaS was compared to the standard Hardware Genetic Algorithm (HGA) realized in [8]. The reason behind the comparison is that the HGA is closely related to GaS, although differing in functionality and structure. The similarity is that both systems attempt to minimize, through the use of a hardware-based GA, a certain fitness function.

The HGA attempts to minimize an actual deterministic function such as $f(x) = x + 5$. The resulting design churned at 2 MHz on a BORG prototyping board, consisting of five Xilinx FPGAs (three 4003s and two 4002s). As shown in the previous chapter, GaS operates at slightly above 33 MHz, and thus respects the timing of the PCI bus, which is also at 33 MHz. The superior speed is accounted by the detailed and synchronized design using RTL-level style realization instead of a high-level behavioral style, as well as the state of the art FPGA (XC4062XLA) that the GA was implemented on. As well, since the design fit onto one FPGA, all communication overhead was eliminated. Now, on the other hand, the JBits Virtex simulator caused the major delays, when computing the total processing time for each run. This is alleviated, for the sake of comparison, by assuming that in a real Virtex device, it takes, in the worst-case scenario, 18 clock cycles (1 for setup, 1 for instantiation and at most 16 for readback of sub-function outputs) to place and test a chromosome, the JBits time component is replaced by 0.545 ns. Taking that into consideration, let us take a look at the comparisons in tables 8.1 and 8.2.

Fitness Function	Number of Generations	HGA Clock Cycles	HGA Time (ms)	Number of Inputs	Number of Generations	GaS Clock Cycles	GaS Time (ms)
x	20	10622	5.311	2	173	81483	2.47
$x + 5$	20	10945	5.473	3	2070	974970	29.54
$2x$ (add)	20	10659	5.330	4	438	206298	6.25
$2x$ (mult)	20	10659	5.330	5	1252	589692	17.87

Table 8.1: Timing comparisons based on 2 MHz HGA and 33 MHz GaS systems

Fitness Function	Number of Inputs	GaS Speedup (Cycles/Generation)	GaS Speedup (Time/Generation)
x	2	$531.1/471 = 1.128$	$0.266/0.014 = 18.63$
$x + 5$	3	$547.25/471 = 1.162$	$0.274/0.014 = 19.57$
$2x$ (add)	4	$532.95/471 = 1.132$	$0.267/0.014 = 19.04$
$2x$ (mult)	5	$532.95/471 = 1.132$	$0.267/0.014 = 19.04$

Table 8.2: Speedup factors for GaS compared to HGA

8.2 GaSage (GaS usage)

GaS has been proven to be efficient for meeting its intended goals. The big advantage, in terms of speedup factors, have also been shown. The overall cost of such a system is comprised of the optimizer and host costs. In future versions, both would be ideally combined onto one chip, and thus the cost of the entire system is reduced to the cost of one SoC-like device. The economic benefits of such a device are great, in that the HIGA, the ECLBs, the hcache and the processor interface subsystems would all reside on the same chip. Not only are there economic benefits, but throughput benefits, as many handshaking protocols and control overhead would be abolished through the combination of all the subsystems onto one hardware device.

GaS has also proven to be successful in mapping out Boolean functions onto a constrained area of a reprogrammable device. This reduces the designer base needed to design future combinational circuits. The process has now (for simple functions) become automated, and as the complexity of the circuits grows, the need for designers to implement the unenviable tasks of mapping logic onto hardware devices would diminish, leaving these same designers with the freedom to tackle other CAD issues, such as timing simulation and verification.

When it comes to combinational circuits, GaS is a very good tool for the automated design and synthesis of minimized circuit realizations, onto FPGA devices. Combinational circuits are now mainly being used in address decoding applications and combinational multipliers and dividers. The need for more complex combinational circuits is already present, since the process technology is always decreasing the transistor length, and thus decreasing the inherent gate delay, while the advantages of utilizing combinational logic are becoming too overwhelming. Coupled with the evident need of a quick and flexible automated solution for the combinational circuit synthesis problem, GaS is an attractive test bed for such a solution.

Chapter 9

Limitations, Obstacles and Aspirations

**"The only way to discover the limits of the possible
is to go beyond them into the impossible"**

(Arthur C. Clarke)

The previous eight chapters introduced in great detail the interworkings of GaS. The following chapter will focus more on the accomplishments and achievements of the undertaken work, and the obstacles that await the researchers of the EHW field.

9.1 The Battles Won

EHW is such a novel field that it has taken a long time to entice researchers to enter it, but it has come in the right time. The convergence of technologies has provided a stable household for the EHW field to grow and flourish. This work, along with many others, will help to define the field better, so as to realize what the limitations of EHW are, as well as tap into its yet unrealized potential. The following section will summarize the cleared hurdles during the period it has taken for GaS to enter its prototyping phase.

9.1.1 Integration of Technologies

The major obstacles faced in the design and implementation of GaS were not in theory, but in practice. The issue here is that the system was built from the bottom up, and did not utilize many pre-built blocks, other than the device driver and JBits

APIs. This brought many challenges to the foreground, especially when the number of technologies being integrated grew above two.

Firstly, the VHDL realization of the hardware, consisting of compilation and verification (both functional and timing), synthesis and instantiation of the HIGA subsystem onto the FPGA device was systematic. The ASM methodology proved very valuable in not only the design of the hardware, but the overall co-design of the controlling software and underlying hardware.

Secondly, the C++ realization of the device driver was structured. The controlling software used a simple handshaking protocol, whence the programmer's model indicated to the software what state the hardware was in. The software wrote into the control registers whenever it needed to signal the hardware, and the hardware wrote into the status registers whenever it needed to signal the software. This busy-waiting methodology allowed the designer to slow down the operation of the system and ignore the many modules and ASMs that are simultaneously executing on the hardware, and concentrate on the module at hand. The PCI registers immensely aided in the communication between the device driver and the hardware, as four of these 32-bit registers were available, allowing for 128 bits to be transmitted at the same time back and forth, be it control, status or simply data information.

Lastly, the Java realization of the host was very problematic. The previously mentioned drawbacks of having a JVM instantiation from the native side were very prevalent during the design of the host. The Java Development Kit (JDK) from Sun Microsystems [62] was utilized to run the JBits API and the Java application itself. Sun itself has acknowledged the lack of a viable solution for having more than one JVM instantiated from the native side. As well, the drawback of having an automated garbage collector was quite hindering. Usually, the garbage collection schemes are very useful, in that they allow the designer to concentrate on the functionality itself, rather than concentrating on allocating and freeing up memory, whenever the latter becomes

scarce or avoiding memory leaks. However, in this work, the garbage collection schemes interrupt the system in order to collect what seems like free-able objects. The objects, though, are actually being referenced through the JNI, and thus are being utilized, which then prevents their garbage collection. In addition, the memory which does get collected from the Java application is returned to the JVM, but the latter does not return that recently freed memory to the OS properly. This causes a memory leak that eventually causes the application to stagnate. The resolution to these issues was actually a blessing in disguise. It was decided to interrupt the operation of GaS every 10 generations, in order to allow for the memory to be transmitted back to the OS. As well, and this is the extra caveat, this prompted the designer to save the context of the running application out to file. Thus, GaS can be interrupted by any source, and still be able to recover when it is started up again.

9.1.2 Realization of System

The realization of the system proves that the integration of the technologies is not only feasible, but also successful. This actually aids the research field in that it offers a working version of an EHW system capable of finding solutions to simple Boolean functions. The realized system can now be fine-tuned in order to better sell the idea of EHW entities capable of searching through large spaces in order to find a constrained solution. Other improvements can be made in order to make the system run faster, and consume less resources, but the main concept is a working one as is.

9.1.3 Proof of Concept

Speaking of concepts, the major block that was overcome in this work is the proof of concept. The idea that through EHW systems, digital circuits can be realized so as to abide by some pre-defined constraints, such as minimum number of utilized LUT and routing resources, seems somehow far-fetched. However, through the underlying work, it

was proven that the idea is realizable and with the combination of technology mapping, functional decomposition and evolvable hardware, it is possible to realize these circuits onto a contemporary FPGA.

Through this proof, it is now easier to look ahead at the realization of more complex systems, as will be mentioned in the next section. As well, the idea that these complex systems can learn from their simpler counterparts is quite exciting. Work has commenced in this area such as in [31], but more research is required before we can affirm that these intelligent systems are learning from their previous evolutionary runs, in order to construct the more complex blocks.

9.2 The War Ahead

There is much work left to be done for us to begin to understand the details of intelligent architectures. It is, however, promising to see the field taking shape. As introduced in the first few chapters, the future of computing systems not only lies in one field, but in the conglomeration of multiple fields (refer to figure 3.2). There is a major war ahead of us, but to each battle an army, and we have armies of researchers ready for each battle.

9.2.1 Increased Scalability

The first major topic of future work involves the increased scalability offered by EHW systems. Looking at GaS in particular, it is imperative to extend the work to more complex circuitries. This will allow the author to use the acquired knowledge to implement and analyze the operation of GaS when faced with building bigger structures, either from scratch, or utilizing the building blocks already evolved and residing in hcache (or off-chip memory).

To have success in this continued endeavor, the major drawbacks of the Java application implementation will be eliminated through the use of a physical hardware

device. The XHWIF can then be put to use in order to directly communicate with the physical Virtex device, and establish a path between the device driver and the host. As well, internal improvements within the HIGA design would make GaS run faster. These improvements can include:

- Parallel evaluation of multiple chromosomes would account for a much higher over throughput;
- Simultaneous read and write operations to the respective generations would avoid the present bottleneck;
- Better handshaking protocols would allow for a reduction in delays associated with module interactions;
- Distributed module communication would allow each module to operate without having to worry about the others. This requires extra synchronization, but reduces control overhead;
- Parallel genetic pipelines would permit for the evolution of multiple solutions simultaneously.

9.2.2 Parallelization and Pipelining

As mentioned in the previous paragraph, the parallelization and pipelining of GaS is extremely integral for future versions. The genetic pipeline is already implemented successfully. As can be seen from the ASM charts, each module goes through a number of steps, in order to execute its functionality. The entire process can be sped up if each module was not so dependent on the previous' output. This is where internal parallelism can be seen, where multiple SMs, XMMs and FMs can be instantiated within the same system. As for external parallelism, GaS would ideally contain multiple genetic pipelines,

each capable of processing chromosomes and outputting minimal solutions to different Boolean functions.

The advantages of such structures are numerous, including the avoidance of various bottlenecks within the genetic pipeline, the decreased contention for memory accesses, the reduction of inter-module overhead and the increased system throughput (more solutions per iteration). As well, with increasing parallelism various simpler structures can be evolved and combined, through an increased inter-pipeline communication bandwidth, to form more complex structures.

9.2.3 Evolving Synthesizable Processors and Computers

As mentioned throughout this thesis, repeated below are the long-term objectives for the undertaken work

- Construct a small functional block of an evolvable processor, capable of gaining intelligence through experience and time;
- Model artificial intelligence as applied to engineering thinking and design;
- Pave the route for **evolvable computing**, whereas evolvable hardware concepts are applied to reconfigurable computing methodologies, to allow for intelligent processors capable of independently modifying their computational architectures based on the context of the problem at hand.

Bluntly stated, the author's main future goal is to delve into the reconfigurable computing domain, in order to attempt to link both the RC field with the EHW field. This combination is a very powerful one, capable of realizing the intelligent architectures that were discussed earlier on.

The realization of GaS allows for the first objective to be near completion. It is now the author's intent to tackle the development of a library of evolved solutions, and

provide those circuit representations to GaS to analyze how well the latter is at gaining intelligence through experience and time.

The last long-term objective, and the most glamorous one, is the evolution of completely synthesizable processors and computers. The field is far away from this event, but the ongoing work will open up many closed doors. The ongoing exploration will uncover many truths about the technology itself: its power, its limitations and its real applications. The road towards complete evolvable computing is ahead of us, but with many obstacles in the way. As a short catch phrase for evolvable computing, the author presents the following : "We have to get what we need from what we know" (Anonymous).

Chapter 10

Closing Comments

"All men dream, but not equally. Those who dream by night in the dusty recesses of their minds wake in the day to find that it was vanity; but the dreamers of the day are dangerous men, for they may act their dream with open eyes, to make it possible"

(T. E. Lawrence [Lawrence of Arabia])

And in the beginning, there were two.

Evolvable hardware: these two words bring life into the world of computing. Isn't it ironic that evolution and society began thousands of years ago, and yet they are just beginning again. It seems that every time a new concept or ideology emerges from the ashes of darkness for all to see, there is a sense of beginning. Although EHW has not been around for long, it seems that it has been with us for a long time. The 'deroulements des évènements', or the successive occurrences of events, caused the idea to take shape. First it was basic integrated circuit design, then it was microprocessor design, and finally it was FPGA design that culminated with the birth of the field of EHW.

10.1 Significance of Research

The undertaken work produced GaS, a system with many innovative ideas (LUT and routing evolution, evolutionary stages and hcache, for example) that allow for the

fast and directed evolution of solutions. GaS is the underlying foundation to the evolution of more complex circuitries and drives the shift of design from a structural view to a functional one. The latter is a fundamental result of applying GAs to configurable computing, as the designer becomes one of functions, instead of structures, alleviating the intricacies associated with the internal protocols and operation of systems, and permitting the designer to think of new functionalities and dream up new possibilities.

The main objectives of this research were the minimizing of simple Boolean functions, the evolution of simple logical structures, and the expansion of the concept to permit evolution of minimized complex structures. All three were met with great success, and the fire to continue with the attempt to attain the long-term objectives has been fuelled by the underlying work. The research field has now gained another hardware evolutionary system, capable of finding the minimized logical solutions to simple Boolean functions. This system is very well optimized, as it is designed using the ASM technique, and realized in the RTL-level methodology. The exact area and timing of each module aids in any future rework, where portability to state of the art FPGAs becomes very simple knowing the dimensions of the complete system and its modules. Parameterizable cores are also easily constructed, as the interface of each module is well defined, and realized.

10.2 The Previous, Current and Next Steps

As in all conclusions, it is necessary to retrace our steps to see where we started from, and where we're heading. The field of EHW is a novel one, having been started in the early 1990's. Through its conception and early endeavors, the field has grown to become internationally recognized by many groups. Due to breakthrough experiments such as Thompson's evolvable discriminator circuit, the field has flourished and is gaining wide acceptance in the area of VLSI design and CAD tool implementation. Through the past few years, the field has expanded to include the evolution of combinational,

simple and synchronous circuits. The current work has added to that list of successes, but also has laid the foundation for a larger pool of successes, as larger and more complex circuits, as well as sequential and asynchronous circuits, can now be tackled by evolution.

The next step is to combine all of the computing trends, including the undertaken evolvable computing trend, in order to achieve the ultimate four stages of information systems: **acquire it, store it, process it and distribute it**. Once those capabilities are associated with computing systems and their interaction with information, and the computing trends converge onto *ComputerSon* as shown in figure 3.2, then it would be safe to assume that society evolved another entity, apart from humans, on its own.

Bibliography

- [1] J. Holland, **Adaptation in Natural and Artificial Systems**, The University of Michigan Press, Ann Arbor, Michigan, 1975.
- [2] H. De Garis, "Evolvable hardware: Principles and practice," Last viewed on September 20, 2000, <http://www.hip.atr.co.jp/degaris/papers/CACM-E-Hard.html>.
- [3] A. C. Clarke, **2001: A Space Odyssey**, PWS Publishing Company, 20 Park Plaza, Boston, Massachusetts 02116, 2nd edition, 1992.
- [4] C. Darwin, **The Origin of Species**, PWS Publishing Company, 20 Park Plaza, Boston, Massachusetts 02116, 2nd edition, 1992.
- [5] Neittaanmaki K. Miettinen, Makela and Periaux, **Evolutionary Algorithms in Engineering and Computer Science**, John Wiley & Sons Ltd., 1999.
- [6] M. Mitchell, **Introduction to Genetic Algorithms**, MIT Press, Cambridge, Massachusetts, 1996.
- [7] D. E. Goldberg, **Genetic Algorithms in Search, Optimization, and Machine Learning**, Addison-Wesley Publishing Company, Incorporated, Reading, Massachusetts, 1989.
- [8] S. D. Scott, "Hga: A hardware-based genetic algorithm," M.S. thesis, University of Nebraska, August 1994.
- [9] K.A. De Jong, **An Analysis of the Behavior of a Class of Genetic Adaptive Systems**, Ph.D. thesis, University of Michigan, 1975.
- [10] S. Trimberger, **Field-Programmable Gate Array Technology**, Academic Publishers, Kluwer, 1994.
- [11] S. Knapp, "Faq about programmable logic," Tech. Rep., OptiMagic, 1997, <http://www.optimagic.com/faq.html>.
- [12] N. Weste and K. Eshraghian, **Principles of CMOS VLSI Design: A Systems Perspective**, Addison-Wesley, Massachusetts, 1993.
- [13] E. Hamdy, "Dielectric-based antifuse for logic and memory ics," **IEEE International Electro-Devices Meeting Technical Digest**, pp. 786–789, 1998.

- [14] C. A. Mead and L. A. Conway, **Introduction to VLSI Systems**, Addison-Wesley, Reading, MA, 1980.
- [15] T. E. Price, **Introduction to VLSI Technology**, Prentice-Hall, Englewood Cliffs, NJ, 1994.
- [16] Intel Corporation, "Intel researchers build world's fastest silicon transistors," Internet.
- [17] University of Cincinnati, "Reconfigurable computing," Internet, <http://www.ece.uc.edu/dinesh/RC/RC-links.html>.
- [18] Xilinx Incorporated, **The Programmable Logic Data Book**, San Jose, California, 2000.
- [19] Xilinx Corporation, "Jbits documentation," Package, JBits 2.8.
- [20] Xilinx Corporation, "Virtex series configuration architecture user guide," Internet, <http://www.xilinx.com/xapp/xapp151.pdf>.
- [21] T. Gordon, "Introduction to evolvable hardware," Last viewed on September 20, 2000, <http://www.cs.ucl.ac.uk/staff/T.Gordon/ehwnote.html>.
- [22] A. Thompson, "Silicon evolution," **Proceedings of Genetic Programming**, pp. 444–452, 1996, MIT Press.
- [23] M. Tomassini and M. Sipper, "An introduction to evolvable hardware," Last viewed on January 25, 2001, <http://evonet.dcs.napier.ac.uk/evoweb/evonews/news3/ehard.htm>.
- [24] H. Iba T. Higuchi and B. Manderick, **Evolvable Hardware**, chapter 12, pp. 399–421.
- [25] T. Rintala, "Hardware implementation of ga," Last viewed on September 20, 2000, <http://www.uwasa.fi/cs/publications/2NWGA/node60.html>.
- [26] L. Wirbel, "Compression chip is first to use genetic algorithms," **Electronic Engineering Times**, p. page 17, December 1984.
- [27] I. Kajitani T. Higuchi, M. Iwata and M. Murakawa, "Hardware evolution at gate and function levels," **Proceedings of the International Conference on Biologically Inspired Autonomous Systems: Computation, Cognition and Action**, March 1996.
- [28] D.H. Horrocks T. Arslan and E. Ozdemir, "Structural cell-based vlsi circuit design using a genetic algorithm," **IEE Electronic Letters**, 1996.
- [29] D. Keymulen H. Sakanashi, M. Iwata and M. Murakawa, "Evolvable hardware chips and their applications," **Proc. of the 1999 IEEE Systems, Man, and Cybernetics Conference (SMC'99)**, pp. 559–564, 1999.

- [30] S. J. Louis and G. J. Rawlins, "Using genetic algorithms to design structures," Tech. Rep. 326, Computer Science Department, Indiana University, Bloomington, Indiana, Feb. 1991.
- [31] A. D. Christiansen C. A. Coello Coello and A. H. Aguirre, "Towards automated evolutionary design of combinational circuits," **Computers and Electrical Engineering**, vol. 27, no. 1, pp. 1–28, January 2001.
- [32] J.F. Miller T.C. Fogarty and P. Thomson, "Evolving digital logic circuits on xilinx 6000 family fpgas," **2nd Online Workshop on Soft Computing**, 1997.
- [33] J. R. Koza, **Genetic Programming. On the Programming of Computers by Means of Natural Selection**, The MIT Press, 1992.
- [34] M. Iwata H. Iba and T. Higuchi, "Gate-level evolvable hardware: Empirical study and application," **Evolutionary Algorithms in Engineering Applications**, pp. 259–276, 1997, Springer Verlag.
- [35] L. Adleman, "Molecular computation of solutions to combinatorial problems," **Science**, vol. 266, pp. 1021–1024, 1994.
- [36] R. P. Feynman, **Quantum Electrodynamics**, Benjamin, 1961.
- [37] A. Sanpera A. Barenco, A.Ekert and C.Machiavello, "A short introduction to quantum computation," Internet, November 1996, <http://www.qubit.org/intros/comp/comp.html>.
- [38] J. Gruska, **Quantum Computing**, McGraw Hill, 1999.
- [39] J. S. McCaskill and P. Wagler, "From reconfigurability to evolution in construction systems: Spanning the electronic, microfluidic and biomolecular domains," **FPL 2000 - Springer-Verlag Berlin Heidelberg**, pp. 286–299, 2000.
- [40] S. Brown and Z. Vranesic, **Fundamentals of Digital Logic with VHDL Design**, McGraw Hill, 2000.
- [41] **The American Heritage Dictionary of the English Language**, Houghton Mifflin Company, 4th edition, 2000.
- [42] M. Mano and C. Kime, **Logic and Computer Design Fundamentals**, Prentice Hall, 2000.
- [43] J. Wakerley, **Digital Design Principles and Practices**, Prentice Hall, 1994.
- [44] S. Lee, **Design of Computers and Other Complex Digital Devices**, Prentice Hall, 2000.
- [45] W. V. Quine, "A way to simplify truth functions," **American Mathematics Monthly**, vol. 62, no. 9, pp. 627–631, 1955.
- [46] E. J. McCluskey, "Minimization of boolean functions," **Bell Sys. Tech. Journal**, vol. 35, no. 5, pp. 1417–1444, 1956.

- [47] E. J. McCluskey, **Logic Design Principles**, Prentice Hall, 1986.
- [48] G.D. Hachtel R.K. Brayton, C. McMullen and A. Sangiovanni-Vincentelli, **Logic Minimization Algorithms for VLSI Synthesis**, Kluwer Academic Publishers, 1984.
- [49] A. L. Sangiovanni-Vincentelli R. K. Brayton, R. Rudell and A. R. Wang, "Mis: A multiple-level logic optimization system," **IEEE Tans. on Computer-Aided Desing**, vol. 6, no. 6, pp. 1062–1081, 1987.
- [50] E.M. Sentovic K.J. Singh L. Lavagno C. Moon R. Murgai A. Saldanha H. Savoj P.R. Stephan R.K. Brayton and A. Sangiovanni-Vincentelli, "Sis: A sytem for sequential circuit synthesis," Tech. Rep., Department of Electrical Engineering and Computer Science, University of California in Berkeley, 1992, Technical Report UCB/ERL M92/41.
- [51] Cadence Design Systems, , " Internet, <http://www.cadence.com>.
- [52] Mentor Graphics, , " Internet, <http://www.mentor.com>.
- [53] Synopsys, , " Internet, <http://www.synopsys.com>.
- [54] R.L. Rudell and A. Sangiovanni-Vincentelli, "Multiple-valued minimization for pla optimization," **IEEE Tans. CAD**, vol. 6, no. 5, pp. 727–750, 1987.
- [55] R. Abielmona and V. Groza, "Circuit synthesis evolution using a hardware-based genetic algorithm," **CCECE 2001 - Canadian Conference on Electrical and Computer Engineering**, pp. 963–967, June 2001.
- [56] M. Serra T. Slater J. C. Muzio and D. M. Miller, "The analysis of one-dimensional linear cellualr automata and their aliasing properties," **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, vol. 9, no. 7, pp. 767–778, July 1990.
- [57] S. Wolfram, "University and complexity in cellular automata," **Physica**, vol. 10D, pp. 1–35, 1984.
- [58] R. Gordon, **Essential JNI: Java Native Interface**, Prentice Hall, Upplle Saddle River, New Jersey, 1998.
- [59] D. Levi and S. A. Guccione, "Geneticfpga: Evolving stable circuits on mainstream fpga devices," **Proceedings of the The First NASA/DOD Workshop on Evolvable Hardware**, 1998.
- [60] D. Levi, "Hereboy: A fast evolutionary algorithm," **Proceedings of the The Second NASA/DoD Workshop on Evolvable Hardware (EH'00)**, 2000.
- [61] Virtual Computer Corporation, "Vcc," Internet, <http://www.vcc.com>.
- [62] Sun Microsystems, "Sun," Internet, <http://www.sun.com>.
- [63] R. Schaller, "Moore's law: past, present and future," **IEEE Spectrum**, pp. 53–59, June 1997.

- [64] J. C. Lopez P. Morino and M. Jacome, "A hardware operating system for dynamic reconfiguring of fpgas," **Field-Programmable Logic Applications: From FPGAs to Computing Paradigm. 8th International Workshop**, pp. 431-435, 1998.
- [65] N. Tredennick, "Get ready for reconfigurable computing," **Computer Design**, April 1998.
- [66] T. Makimoto, "The rising wave of field programmability," .
- [67] M. Barr, "A reconfigurable computing primer," **Multimedia Systems Design**, September 1998.
- [68] R. Hartensteiin, "A decade of reconfigurable computing: A visionary retrospective," .
- [69] J. Cardoso and M. Vestias, "Architectures and compilers to support reconfigurable computing," <http://www.acm.org/crossroads/xrds5-3/rcconcept.html>, last viewed on 08/25/2001.
- [70] R. Cant R. Swan, A. Wyatt and C. Langensiepen, "Re-configurable computing," <http://www.acm.org/crossroads/xrds5-3/ntu.html>, last viewed on 08/27/2001.
- [71] O. Diessel and G. Wiley, "Opportunities for operating systems research in reconfigurable computing," .
- [72] K. Compton and S. Hauck, "An introduction to reconfigurable computing," **IEEE Computer**, April 2000.
- [73] A. DeHon and J. Wawrzynek, "Reconfigurable computing: What, why and implications for design automation," .
- [74] M. J. Alexander and M. O'Toole, "Implications of reconfigurable vlsi in a globally networked world: Delivering hardware over the net," Tech. Rep., WSU Technical Report, 1997.
- [75] S. Goldstein and H. Schmit, "Reconfigurable computing course notes," Tech. Rep., Carnegie Mellon University, 1998.
- [76] N. Tredennick, "Dynamic logic vs. computing," **Dynamic Silicon**, 2001.
- [77] http://3gnewsroom.com/3g_news/news_0937.shtml, **Reconfigurable Parallel Architecture IP Cores for Use in 3G+ Base Stations**, last viewed on 11/16/2001, Upper Saddle River, New Jersey, July, 2001.

Appendix A

Hardware Design - ASM Diagrams

A.1 Random Number Generator

Figure A.1: RNG ASM chart

Figure A.2: RNG detailed ASM chart

Figure A.3: RNG ASM control path

Figure A.4: RNG ASM data path

A.2 Selection Module

Figure A.5: SM ASM chart

Figure A.6: SM detailed ASM chart

Figure A.7: SM ASM control path

A.3 Mutover Module

Figure A.9: XMM ASM chart

Figure A.12: XMM ASM control path (Part 2)

Figure A.13: XMM ASM data path

A.4 Fitness Module

Figure A.14: FM ASM chart

Figure A.15: FM detailed ASM chart

A.5 Memory Access Module

Figure A.18: MAM ASM chart

Figure A.19: MAM detailed ASM chart

Figure A.20: MAM ASM control path

A.6 Initialization Module

Figure A.22: IM ASM chart

Figure A.24: IM ASM control path

Appendix B

Hardware Design - VHDL Code

B.1 Basic Building Blocks

B.1.1 24-bit Enabled Address Decoder

Refer to CD - /src/vhdl/bblocks/add_decoder.vhd

B.1.2 16-bit Adder

Refer to CD - /src/vhdl/bblocks/adder_16.vhd

B.1.3 16-bit ALU

Refer to CD - /src/vhdl/bblocks/alu_and16.vhd

B.1.4 16-bit Full Adder-Subtractor

Refer to CD - /src/vhdl/bblocks/alu_fas.vhd

B.1.5 ALU Top Entity

Refer to CD - /src/vhdl/bblocks/alu_top.vhd

B.1.6 1-bit Comparator

Refer to CD - /src/vhdl/bblocks/comparator_1.vhd

B.1.7 2-bit Comparator

Refer to CD - /src/vhdl/bblocks/comparator_2.vhd

B.1.8 4-bit Comparator

Refer to CD - /src/vhdl/bblocks/comparator_4.vhd

B.1.9 8-bit Comparator

Refer to CD - /src/vhdl/bblocks/comparator_8.vhd

B.1.10 10-bit Comparator

Refer to CD - /src/vhdl/bblocks/comparator_10.vhd

B.1.11 16-bit Comparator

Refer to CD - /src/vhdl/bblocks/comparator_16.vhd

B.1.12 2-bit Down Counter

Refer to CD - /src/vhdl/bblocks/counter_2.vhd

B.1.13 5-bit Down Counter

Refer to CD - /src/vhdl/bblocks/counter_5.vhd

B.1.14 5-bit Up Counter

Refer to CD - /src/vhdl/bblocks/counter_5_up.vhd

B.1.15 10-bit Down Counter

Refer to CD - /src/vhdl/bblocks/counter_10.vhd

B.1.16 D Flip-Flop with Asynchronous Reset

Refer to CD - /src/vhdl/bblocks/d_flipflopAR.vhd

B.1.17 D Flip-Flop with Chip Enable

Refer to CD - /src/vhdl/bblocks/d_flipflopCE.vhd

B.1.18 D Flip-Flop with Synchronous Reset

Refer to CD - /src/vhdl/bblocks/d_flipflopSR.vhd

B.1.19 5-32 Decoder

Refer to CD - /src/vhdl/bblocks/decoder_5_32.vhd

B.1.20 5-32 Decoder for XMM (1)

Refer to CD - /src/vhdl/bblocks/decoder_5_32_cross1.vhd

B.1.21 5-32 Decoder for XMM (2)

Refer to CD - /src/vhdl/bblocks/decoder_5_32_cross2.vhd

B.1.22 4-bit Full Adder-Subtractor

Refer to CD - /src/vhdl/bblocks/fas_four.vhd

B.1.23 1-bit Full Adder-Subtractor

Refer to CD - /src/vhdl/bblocks/fas_one.vhd

B.1.24 2-1 Multiplexer

Refer to CD - /src/vhdl/bblocks/multiplexer2_1.vhd

B.1.25 4-1 Multiplexer

Refer to CD - /src/vhdl/bblocks/multiplexer4_1.vhd

B.1.26 8-1 Multiplexer

Refer to CD - /src/vhdl/bblocks/multiplexer8_1.vhd

B.1.27 Variable-Length Data Register

Refer to CD - /src/vhdl/bblocks/simple_data_register.vhd

B.1.28 Variable-Length Data Register with Common Chip Enable

Refer to CD - /src/vhdl/bblocks/simple_data_registerCE.vhd

B.1.29 Variable-Length Data Register with Independent Chip Enables

Refer to CD - /src/vhdl/bblocks/simple_data_registerCES.vhd

B.1.30 SR Latch

Refer to CD - /src/vhdl/bblocks/sr_latch.vhd

B.1.31 Synchronized SR Latch

Refer to CD - /src/vhdl/bblocks/sr_sync_latch.vhd

B.2 Pseudorandom Number Generator**B.2.1 Control Path**

Refer to CD - /src/vhdl/rng/rng_controlpath.vhd

B.2.2 Data Path

Refer to CD - /src/vhdl/rng/rng_datapath.vhd

B.2.3 Top Entity

Refer to CD - /src/vhdl/rng/rng_top.vhd

B.3 Selection Module**B.3.1 Control Path**

Refer to CD - /src/vhdl/sm/sm_controlpath.vhd

B.3.2 Data Path

Refer to CD - /src/vhdl/sm/sm_datapath.vhd

B.3.3 Top Entity

Refer to CD - /src/vhdl/sm/sm_top.vhd

B.4 Mutover Module**B.4.1 Control Path**

Refer to CD - /src/vhdl/xmm/xmm_controlpath.vhd

B.4.2 Data Path

Refer to CD - /src/vhdl/xmm/xmm_datapath.vhd

B.4.3 Top Entity

Refer to CD - /src/vhdl/xmm/xmm_top.vhd

B.5 Fitness Module**B.5.1 Control Path**

Refer to CD - /src/vhdl/fm/fm_controlpath.vhd

B.5.2 Data Path

Refer to CD - /src/vhdl/fm/fm_datapath.vhd

B.5.3 Top Entity

Refer to CD - /src/vhdl/fm/fm_top.vhd

B.6 Memory Access Module**B.6.1 Control Path**

Refer to CD - /src/vhdl/mam/mam_controlpath.vhd

B.6.2 Data Path

Refer to CD - /src/vhdl/mam/mam_datapath.vhd

B.6.3 Top Entity

Refer to CD - /src/vhdl/mam/mam_top.vhd

B.7 Initialization Module**B.7.1 Control Path**

Refer to CD - /src/vhdl/im/im_controlpath.vhd

B.7.2 Data Path

Refer to CD - /src/vhdl/im/im_datapath.vhd

B.7.3 Top Entity

Refer to CD - /src/vhdl/im/im_top.vhd

B.8 Genetic Pipeline Module

B.8.1 Top Entity

Refer to CD - `/src/vhdl/genetic_model/genetic_top.vhd`

B.9 GaS Module

B.9.1 Top Entity

Refer to CD - `/src/vhdl/genetic_model/gas_top.vhd`

B.10 Chip Module

B.10.1 Top Entity

Refer to CD - `/src/vhdl/genetic_model/chip_top_UB.vhd`

Appendix C

Software Design - C++/Java Code

C.1 Device Driver Code

C.1.1 jniSpecialNativeLoader.h

This file is the header for the custom JNI loader

Refer to CD - /src/c++/jniSpecialNativeLoader.h

C.1.2 jniSpecialNativeLoader.cpp

This file is the source for the custom JNI loader

Refer to CD - /src/c++/jniSpecialNativeLoader.cpp

C.1.3 gasProject.cpp

This file is the source for the device driver

Refer to CD - /src/c++/gas-project.cpp

C.2 JBits Code

C.2.1 SimpleLoader.java

This file is the source for the Java loader, which extends ClassLoader

Refer to CD - /src/java/SimpleLoader.java

C.2.2 SpecialNativeLoader.java

This file is the source for the JNI loader, which extends SimpleLoader

Refer to CD - /src/java/SpecialNativeLoader.java

C.2.3 SpecialLoader.java

This file is the source for the Java custom loader, which compiles into the JNI Loader dynamic link library

Refer to CD - /src/java/SpecialLoader.java

C.2.4 EvaluateChromosome.java

This file is the source for the JBits application

Refer to CD - /src/java/EvaluateChromosome.java

C.2.5 EvaluateChromosomeInterface.java

This file is the source for the interface to the JBits application

Refer to CD - /src/java/EvaluateChromosomeInterface.java

Appendix D

Reconfigurable Computing: An Overview

D.1 The splash effect

More than 35 years ago, Moore's Law presented the technological research and development communities with a driving force that is incomparable, even in today's time-to-market pressures and their subsequent driving forces. Moore's Law provided us with a "throttle" that is only governed by our own limitations. Its self-fulfilling prophetic nature once prompted a software representative to say, "The length of eternity is 18 months, the length of a product cycle" [63]. This ideology has steered us towards the deep sub-micron levels, where we can now fit millions of gates on a single chip. Today's breakthroughs in process technologies, such as Intel's latest achievement (20-nm transistors), are proof of our constant ingenuity and drive in order to meet, and even exceed, Moore's Law. The predictions are that we will have microprocessors running at twenty gigahertz, containing in excess of a billion transistors, and operating at less than one volt, by the time 2007 rolls around [16].

These gains have not come without their associated losses. The **splash** effect is defined as:

- a *schism* between the traditional ASIC tools and their required capacities,
- a *plethora* of gates and not enough designers to use them,
- a *lack* of a technology to which Moore's Law can be extrapolated to,

- an *augmentation* of the cost of fabrication plants,
 - * US \$14 million in 1966;
 - * US \$ 1500 million in 1995;
 - * US \$ 3 billion in 1998;
 - * Projected to be US \$ 10 billion in 2005 [63]
- a *separation* between the digital producer and the digital consumer, and
- a *hindrance* of the optical lithographical process caused by physical limitations.
 - * Physics could signal the end of Moore's Law.

The splash effect has taken years to materialize, but we have finally reached an era where the typical consumer interacts daily with devices such as a personal computer (PC), a workstation, a laptop, a digital camera, a printer, a scanner, a cellular phone, a pager, a personal digital assistant (PDA), a calculator, a watch and/or a global positioning system (GPS) receiver. The myriad of devices that are microprocessor-based keeps diverging, but the general functionality of each manufactured device keeps decreasing. The splash effect is an outcome of this "decreasing divergence" of technologies. Different markets have been dynamically created and filled, to the point where, the in-the-near-future consumer will require a microprocessor-based device to control all these microprocessor-based devices (e.g. HomeNet). Figure D.1 illustrates what is labeled as the "loudspeaker bottleneck".

Figure D.1: The "loudspeaker bottleneck"

This bottleneck is caused by the decreasing generalization, or specialization, of the outcome of today's best process technologies, and the increasing number of devices and markets that utilize these devices, or dynamic market creation. The bottleneck clearly subdivides the *digital producer (DP)* from the *digital consumer (DC)*, creating a market utilization gap, where the consumer is flooded with more devices than he/she can handle; and the producer is flooded with the constant need to design faster and bigger chips, in order to survive in the market (Moore's Law in its survival-of-the-fittest form). This symbiosis between the DP and the DC is not what we had hoped for: today's consumer is buying more devices more frequently. We need an injection of new techniques in order to combat the loudspeaker bottleneck: enter **reconfigurable computing (RC)**.

RC research began in the late 1980's, but had not gained widespread acceptance as a viable computing paradigm until the past few years. The principal factor, which drove the rapid advancement of the field, was the advent of the field-programmable gate array, or FPGA for short. An FPGA is a 2-D array of compute elements, simultaneously separated and joined, by an interconnect system. The FPGA is a reconfigurable device, whence both the logic cell and interconnect architectures can be modified by a simple stream of bits. The most popular of today's FPGAs are based on SRAM technology, which although occupy a large physical space, exhibit extremely fast reconfiguration times (on the order of microseconds).

RC offers a viable solution to the splash effect faced by our current technologies, as well as an influx of standardization to alleviate the loudspeaker bottleneck. A reconfigurable computer has been defined as a:

workstation host loosely coupled via a general purpose bus to a reconfigurable logic processor [64]

The basic idea is to have a balance between the high level of performance of software and the high level of flexibility of hardware. Software has allowed today's computer

to perform various operations with very little change to the system architecture (e.g. the microprocessor). Hardware has lagged behind either fitting under the application-specific (customizable) circuits or the general-purpose processors. The amount of flexibility that hardware inherently provides is not being utilized in order to realize the computer's true multitasking abilities [65]. By direct comparison, high-level programming languages such as Ada, Java or C++ perform "multitasking" by switching tasks in and out of the processor so rapidly, that it seems that the tasks are being executed in parallel. True multitasking will only be available when, for example, we are able to carry a voice conversation with someone, while editing a document, on a uniprocessor machine.

Reverting back to the splash effect, here is how RC will resolve the various issues:

- a *schism* ...
 - * A new and innovative set of tools and compilers have already been researched and developed to aid and improve the design of reconfigurable architectures.
 - * More research is required to fulfill this need.
- a *plethora* ...
 - * A larger designer pool has to be targeted. The computer did not truly flourish world-wide until software and the Internet allowed the free exchange of ideas and applications amongst its consumers.
 - * The same holds for reconfigurable computers. A larger pool of RC designers and a medium of communication between them, will propel the RC community into the future.
 - * Produces more efficient designs and helps in achieving more functionality with less hardware.

- a *lack* ...

- * Possible technologies being currently investigated are : nano-technology, DNA computing, chaos-based computing, quantum computing and Xputers.

- an *augmentation* ...

- * RC will dramatically lower fabrication costs, as the main computing medium is the FPGA (or another reconfigurable architecture), which is significantly cheaper than an ASIC or a microprocessor.
- * The addition of new features to the system suffers its only cost by the amount of memory that is needed to store the extra logic functionalities.
- * System cost is also reduced due to field-upgradable designs.

- a *separation* ...

- * Chips are currently dissipating more power, as more gates are being squeezed into them, with transistors switching on and off at faster and faster speeds. This is directly contrasted by the increasing mobile market which requires less power to operate (e.g. PDAs, cell phones, pagers)
- * "Standardized in manufacturing, but customized in application" [66]. There is a need for post-fabrication customization in order to help converge the DP-to-DC relationship.

- a *hindrance* ...

- * RC opens up various fields where the hardware is shared in both space and time. More hardware could execute than we have gates to fit it in, greatly reducing processor idle time [67].

Along with the previous characteristics, RC promises to narrow the gap between the DP and the DC (as mentioned in *separation ... above*). Imagine if you will, a device that can be configured (and reconfigured) in order to meet the consumer's demands. The embedded systems market would have witnessed its first true "revolution", and the RC field would have found its first true "killer-application". Instead of equipping the consumer with various devices, each customized for its particular application, arm him/her with a device that can be reconfigured to tailor to the intended application and build the resources needed to perform that application, both on demand and in real-time. This would be taking the image in Figure D.1 and blowing it up to look like Figure D.2, where the overall effort exerted in our process technologies, and by our system designers (the digital producer), directly translates into added variable functionality for the consumer. No customization is required, and no scenarios where controllers of controllers are present.

Figure D.2: The modified loudspeaker

D.2 How does it work?

Let us begin by exploring the detailed interworkings of an RC system:

D.2.1 Reconfigurable Architecture (RA)

- PLD-based design systems, coupled with a microprocessor/microcontroller used to combine the strengths of both hardware and software
- Basic architecture is to have RAM-based structural synthesis, meaning that both the algorithms and the resources that they run on have become dynamic [68], therefore an architecture that provides both the flexibility and fast turnaround times is a must
- Can be classified in four different categories based on coupling strategies [69][70]:
 - * Working as an *external processor* coupled through the *I/O bus*;
 - * Working as an *attached processor* coupled through the *local bus*;
 - * Working as a *coprocessor directly* coupled to the main processor;
 - * Working as a *functional unit* coupled through the *datapath* of the main processor.
- Multiple architectures have been researched and devised in the past and have been showing very impressive results, where coarse-grained architectures are concerned. By coarse-grained, we usually mean that the minimal path width is at least greater than one. These architectures are an alternative to the FPGA-based design systems, which are fine-grained, due to their one bit path widths. Please refer to [68] for a detailed discussion of the types of reconfigurable architectures researched in the last decade

D.2.2 Run-time environment (RTE)

- Needed to manage resources in an abstract manner
 - * Must create a model that designers can relate to without being thoroughly knowledgeable in terms of the actual resources or the tools used to design the resources
 - * Must *virtualize* the resources through models that are independent of the circuits realized on them
- Current management software are part of the **operating system (OS)** residing on the host, and offer nothing more than device driver support in order to achieve full connectivity to the resource, at the expense of designer know-how of the resource
- The more concrete the architecture model is, the **more** a designer knows about the architecture, but on the other hand, the more abstract a complex architecture is to a designer, the **easier** it is for the designer to create applications for the architecture
- The run-time environment is needed to promote RC design to a much wider pool of designers and will aid in the transition from previous design methodologies. More and more designers will become application designers, who are only required to upgrade/refresh their concurrency and real-time design skills. They are **not required to immerse themselves** in the details of the architecture to design their applications
- The following is a simple breakdown of the run-time environment as offered by [71]:
 - * Single-task FPGAs:

- Design capture
- Partitioning
- Placement
- Routing
- Scheduling
- Swapping

*** Multi-task FPGAs:**

- Task sizing
- Task placement
- Location independence
- Swapping and caching
- Input and output
- Inter-task communication
- Allocation and scheduling
- Security

- Each of the aforementioned topics has their own open questions that only research in their respective fields can resolve. The questions are too numerous to be posed in this context, but some will be further explored in a following section

D.2.3 Run-time reconfigurability (RTR)

- Involves the direct manipulation of the available hardware resources at run-time, in order to respond to the surrounding requirements placed upon the system
- RTR allows for
 - * Time sharing of different tasks (hence, temporal partitioning), and thus minimizes the required silicon area

- * Birth of the virtual hardware concept
 - * Cycle-by-cycle context switching
 - * Post-fabrication adaptation to new standards, features, implementations or bug fixes and field updates
 - * Acceleration of the application through hardware realizations of software intensive loops and
 - * True multitasking of applications and algorithms
- There are different techniques for RTR:
 - * Single context, where full reconfiguration is required to modify the structure of a single task;
 - * Multi-context, where the full reconfiguration of one context is required, while other contexts are still operational;
 - * Partial reconfiguration, where only the task that is required to be swapped is, while all other tasks remain operational [72].

D.2.4 Requirements of a RC system

- Along with a reconfigurable architecture and an associated run-time environment, the following is required for the successful realization of an RC system
- Dynamic reprogrammability
 - * No down time must be incurred
 - * The functionality gained must offset the reconfigurability times
- RTR/Partial reconfiguration

- * While other parts of the design remain active and fully functional, the clock is stopped to a part of the design, that part is swapped out and another is swapped in, and the clock is re-started to that part
- Accessible and visible internal state
 - * Memory mapping of device is a requirement
 - * Eases task switching, scheduling and allocation
 - * Permits the swapping of hardware designs
 - * Virtualizes hardware [67]
- Embedded processor to handle OS functionalities
 - * Necessary to handle both hardware and software resources
 - Context switching, priority-schemes, scheduling, allocation of resources, hardware routing and so on

D.2.5 Advantages of a RC system

- Lower system, production and component costs as base architecture is fabricated and re-used over and over again
- Reduced time-to-market as design turnaround times are in the hours, instead of months or years
- Temporal partitioning of spatial resources allows for highly parallel execution of algorithms, and for true multitasking of applications
- Multiple contexts are present and can be swapped in and out of the architecture, thus costing the system in resources only the memory needed to store the context, and in power only that needed to store that context
- Problem customization is possible, for different problems

- Combines strengths of hardware (high parallelism of digital circuitry) and software (high amount of flexibility acquired through the modification of an instruction stream to respond to different problems) [73]
- Minute control over fine-grained operations, such as bit operations. For example, one bit reads for an 8-bit register normally involves at least 8 shift and read instructions, but in RC would only involve one
- Dramatically lowers design, fabrication and verification times
- Significant speed gains are achieved because instructions are not fetched, decoded and executed anymore and a parallel execution of "instruction streams" is allowed for
- Eases field upgrades and ports to newer process technologies[74]
- Increases fault-tolerance, as faults can be isolated and switched out on location
- Current embedded systems market targeting applications in multimedia, encryption, filters and DSP, and not the typical word processing functionalities associated with portable devices. The targeted applications reap tremendous benefits from an RC design
- Offers "upward compatibility" in that ports to larger devices will allow the "compiled code" to run more efficiently [75]

D.2.6 Disadvantages of a RC system

- Impacts overall performance due to the reconfiguration times and other extra functionalities built into the run-time environment (a new scheduling algorithm for example)

- Renders some implementations on FPGAs infeasible due to the reconfiguration constraints (task sizing for example)
- Loss of complete control over the resources, thus takes away from the designer's ingenuity which aids in today's post-fabrication products
- Difficulty of programming for reconfigurable computers is compounded by the lack of supporting tools and compilers, and the lack of designers willing to adopt the emerging methodologies
- Software/configware partition and compilation are either still done manually by expert system designers or through existing inadequate CAD tools

D.3 What are the alternatives?

Having explored the detailed operations and characteristics of an RC system, a venture into the alternative market is due. There are a few alternatives to RC design, but the main ones are the *general-purpose microprocessor*, the *digital signal processor*, the *application-specific integrated circuit (ASIC)* and the emerging *system-on-chip (SoC)* designs.

Let us first explore the microprocessor/DSP alternative. The microprocessor replaced design using TTL blocks because it made problem solving programmable. Instead of designers having to map a fixed problem onto fixed resources, they were mapping variable problems onto fixed resources. This offered a major breakthrough in digital logic design. Designers were now free to explore a multitude of problems, previously unexplored due to the customization and complexity of the TTL component circuits. The boom in the microprocessor industry is being currently hampered, though, by two major bottlenecks: the instruction execution bottleneck and the computational efficiency bottleneck. The instruction execution bottleneck rises from the fact that an instruction has to be fetched from memory, decoded by the control path and then executed on the

data path. Engineers and designers alike have been trying to decrease the number of instructions in an instruction-set architecture (ISA), or execute more instructions per cycle (IPS). Either way, the road seems blocked because there is so much complexity that could be handled in a system, and that is exactly what both designer solutions are increasing. The second bottleneck, and the more pressing one, is the computational efficiency bottleneck. To be "fairly" compared, different devices, such as microprocessors, DSPs, ASICs and FPGAs, abide by the computational efficiency benchmark. This characteristic designates the amount of computation that is performed multiplied by the power used to accomplish that work. In the case of both the microprocessor and the DSP, to increase performance, we need to increase the frequencies that the devices operate at [76]

$$Performance = [frequency * IPS] \quad (D.1)$$

$$/[numberofinstructions]$$

which in turn increases their power dissipation

$$Power = 0.5 * capacitance * [voltage]^2 * frequency \quad (D.2)$$

In the near future, the embedded systems market will dominate next-generation devices when it comes to large computationally efficient systems, and microprocessors and DSPs do not fit into that solution space.

Next come the ASICs, which are very powerful customized chips operating at very high speeds, and consuming much less power than typical microprocessors/DSPs. The big knock against ASICs is that they are not flexible. They are not reconfigurable and do not offer very high programmability. Along with that major stumbling block, ASICs are very expensive to design and produce. ASICs require expertise designers who thoroughly understand digital logic design and are masters of transforming problems into hardware description language (HDL) representations and finally into circuitries.

Finally, ASICs have a very long time-to-market, as small changes in a design might cost the product cycle months at a time. What if a customer is supplied with an ASIC whose supported protocol needs modifications half a year later? Changes in the applications environment and the ability to support dynamic standards rule out ASICs as a viable solution.

A third alternative worth delving into is SoC design. As soon as the process technologies entered the sub-micron levels, vast designer communities began thinking about grouping various cores together, controllable in both size and complexity, and offering the required functionality. The mixing and matching of such cores would produce system-on-chip designs, which can be modified as the need arises. Although in its infancy, SoC design faces many challenges, including but not restricted to: interfacing the different cores together, verifying their individual and combined functionalities, building large databases of IP cores that could be shared in the community and handling all the licensing and legal issues that come about when vendors need to offer their IPs for industrial or academic use in designs [67]. Although efforts have been made to expand the IP cores library such as in [77], the ideal situation of "shopping" for IPs and integrating them together to make a functionally correct design is not that far off, but it is not in the very-near future either. SoC and its related activities have to wait a little longer for realization.