Nelson Mandela Metropolitan University

for tomorrow

HARDWARE EVOLUTION OF A DIGITAL CIRCUIT USING A CUSTOM VLSI ARCHITECTURE

SUBMITTED IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF ENGINEERING IN MECHATRONICS

IN THE FACULTY OF ENGINEERING, THE BUILT ENVIRONMENT AND INFORMATION TECHNOLOGY AT THE NELSON MANDELA METROPOLITAN UNIVERSITY

ALLAN EDWARD VAN DEN BERG

UNDER THE SUPERVISION OF DR FAROUK SMITH

PORT ELIZABETH, SOUTH AFRICA
DECEMBER 2013

Nelson Mandela Metropolitan University

for tomorrow

HARDWARE EVOLUTION OF A DIGITAL CIRCUIT USING A CUSTOM VLSI ARCHITECTURE

Submitted in fulfilment of the requirements for the degree of

Master of Engineering in Mechatronics

in the Faculty of Engineering, The Built Environment and Information Technology at the Nelson Mandela Metropolitan University

Allan Edward van den Berg

Under the supervision of Dr Farouk Smith

Department of Mechatronics
Port Elizabeth, South Africa

December 2013

DOCUMENT INFORMATION

Author:	Allan Edward van den Berg
Author's Student Number:	208033000
Supervisor:	Dr Farouk Smith
Institution:	Nelson Mandela Metropolitan University
Faculty:	Engineering, the Built Environment and Information Technology
School:	Engineering
Department:	Mechatronics
Degree:	Master of Engineering in Mechatronics
Degree Code:	75001
NQF Level	9 (Masters Degree)
Pages:	125 (Excluding Appendices)
Word Count:	32696
Date Printed:	10/04/2014 11:05:00
Last Saved:	10/04/2014 11:01:00
Revision Number:	164
File Name:	WorkingMasters

ABSTRACT

Keywords: **Control Circuit, Critical-Path Circuits, Evolutionary Hardware, Evolved Hardware, Field-Programmable Gate Array, Finite-State Machine, Modular Evolution, Portability, Real-World Applications, Two-Bit Multiplier, Virtual Reconfigurable Circuit, Scalability**

This research investigates three solutions to overcoming portability and scalability concerns in the Evolutionary Hardware (EHW) field.

Firstly, the study explores if the V-FPGA—a new, portable Virtual-Reconfigurable-Circuit architecture—is a practical and viable evolution platform.

Secondly, the research looks into two possible ways of making EHW systems more scalable: by optimising the system's genetic algorithm; and by decomposing the solution circuit into smaller, evolvable sub-circuits or modules.

GA optimisation is done is by: omitting a canonical GA's crossover operator (i.e. by using an $1 + \lambda$ algorithm); applying evolution constraints; and optimising the fitness function. The circuit decomposition is done in order to demonstrate modular evolution.

Three two-bit multiplier circuits and two sub-circuits of a simple, but real-world control circuit are evolved.

The results show that the evolved multiplier circuits, when compared to a conventional multiplier, are either equal or more efficient. All the evolved circuits improve two of the four critical paths, and all are unique. Thus, it is experimentally shown that the V-FPGA is a viable hardware-platform on which hardware evolution can be implemented; and how hardware evolution is able to synthesise novel, optimised versions of conventional circuits.

By comparing the $1 + \lambda$ and canonical GAs, the results verify that optimised GAs can find solutions quicker, and with fewer attempts. Part of the optimisation also includes a comprehensive critical-path analysis, where the findings show that the identification of dependent critical paths is vital in enhancing a GA's efficiency.

Finally, by demonstrating the modular evolution of a finite-state machine's control circuit, it is found that although the control circuit as a whole makes use of more than double the available hardware resources on the V-FPGA and is therefore not evolvable, the evolution of each state's sub-circuit is possible. Thus, modular evolution is shown to be a successful tool when dealing with scalability.

ACKNOWLEDGEMENTS

Firstly, I wish to express my gratitude to my supervisor, Dr Farouk Smith, who introduced me to the field of Evolutionary Hardware, assisted me in obtaining financial support, spent many hours proof-reading my documentation and pushed me to submit my findings for review at an academic conference. Your constant concern and patience throughout this process is appreciated; and your knowledge and passion for the FPGA field enabled me to remain enthusiastic throughout my research.

Secondly, the financial assistance of the National Research Foundation (NRF) towards this research is hereby also acknowledged. Opinions expressed and conclusions arrived at are my own and are not necessarily to be attributed to the NRF.

PREVIOUSLY PUBLISHED WORK

A major portion of the work in this dissertation has been previously published (see (van den Berg & Smith, 2013)), and has also been presented at the Annual Research Conference of the South African Institute for Computer Scientists and Information Technologists, held in East London during October 2013. The publication includes the implementation of the V-FPGA and the evolution of the trial two-bit multiplier.

DECLARATION

I, Allan Edward van den Berg, student number 208033000, in accordance with Rule G4.6.3, hereby declare that this dissertation for Master of Engineering in Mechatronics is my own original work, and that it has not previously been submitted for assessment or completion of any postgraduate qualification to another university or for another qualification.

[illegible]

Allan E van den Berg

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF EQUATIONS

DEFINITIONS

$(\mu + \lambda)$-Selection	A selection scheme which selects μ best individuals from a population, and creates λ mutants from these best individuals (Engelbrecht, 2007, p. 139).
Cartesian Genetic Programming	Invented by Miller & Thomson (2000), CGP is a method of arranging the nodes of a circuit using the Cartesian coordinate system.
CP Vector	The collective logic responses of one particular external output when evaluated using all the sets of test vectors.
Critical Path	An output's critical path includes only the relevant circuitry required for that output to function. Thus, it is the direct path linking certain external inputs to a particular external output.
Crossover	Process of combining two individuals' genes in order to create new offspring.
Decomposition	Kalganova (2000, p. 2) defines decomposition as "breaking a large logic circuit into several relatively smaller ones."
Divide and Conquer	The process of dividing a problem into two or more sub-problems until the sub-problems become simple enough to solve. The solved sub-problems can then be recombined to offer a solution for the original problem.
Elitism	The process of ensuring the best individual survives from one generation to the next.
Evolvability	The ability for an EHW system to produce individuals fitter than those found in previous generations (Altenberg, 1994).
Field-Programmable Gate Array	An IC specifically designed to be reconfigured and programmed by a designer after manufacturing.
Finite-State Machine	A simple computational model containing finite number of states. According to the received inputs, the machine transitions from one state to another (Daciuk, 1998). FSM are used as tools to design both computer programs and sequential logic circuits.

Fitness Function	The fitness function is a mathematical formula that is used to assess an individual's ability. An individual is said to be fit if it successful fulfils the requirements of a predefined objective function.
Fitness Landscape	A fitness landscape is a visual metaphor used to describe the fitness of all the possible solutions within the search space. There are peaks and valleys within the landscape, with the highest peaks representing the fittest solutions (Hubert, n.d.). Peaks can be grouped together, or scattered.
Genetic Operator	The programming syntax used to add genetic diversity to population. Examples include selection, mutation, elitism and crossover.
Genotype	The string of integers that represent the necessary logic and routing data needed to implement a particular digital circuit.
Genotype-Phenotype Mapping	The process of encoding a digital circuit, or phenotype, into a genotype (Vassilev & Miller, 2000).
Mutation	The process of randomly changing random genes within a chromosome.
On-Chip Evolution	An evolutionary setup which incorporates the evolutionary algorithm on a separate processor incorporated into the same chip containing the target EHW (Torresen, 2004, p. 6).
Output Element	The logic response of a single external output when evaluated using a single test vector.
Output Vector	The collective logic responses of all external outputs when evaluated using a single test vector.
Phenotype	The genotype's circuit
Portability	The ability of a system to be implemented on different target platforms.
Scalability	Refers to an EHW problem being unsolvable due to the scale of the desired solution. The complexity of a circuit greatly impacts the GA's search space. As a result, complex circuits become exponentially difficult to evolve. Scalability becomes more prominent if: the EHW system uses too few LEs—the desired circuit cannot be evolved due to a lack of resources; or too many LEs—the search space is too large.
Search Space	The set of all the potential solutions.

Soft-Processor	A microprocessor core that is implemented using the programmable logic of semiconductor devices.
Stalling Effect	The fitness of a population does not increase over a substantial number of generations.
System on a Programmable Chip	Mead (2001) defines an SOPC as "a set of functional blocks built on a programmable devices, with at least one computing engine." The programmable device is usually an FPGA, while the computing engine is usually a soft-processor.
Test Vector	A combination of logic high and low signals sent to the V-FPGA's external inputs in order to evaluate a phenotype's response.
Tournament Selection	A selection scheme which selects a group of n random individuals from a population of p individuals, where $n < p$ (Engelbrecht, 2007, p. 137).
Very-Large-Scale Integration	An IC architecture which combines thousands of transistors onto a single chip.

NOMENCLATURE

.vi	LabVIEW VI file-extension
ANN	Artificial Neural Network
API	Application-Program Interface
ASIC	Application-Specific Integrated Circuit
CGP	Cartesian Genetic Programming
$\boldsymbol{Clock}$	Clock line used by the LCM and RCM
CP	Critical Path
CPLD	Complex Programmable Logic Device
DAQ	Data Acquisition
DSM	Dynamic State Machine
$\boldsymbol{D_{Logic}}$	Data line used by the LCM
$\boldsymbol{D_{Routing}}$	Data line used by the RCM
EDA	Electronic Design Automation
EHW	Evolutionary Hardware
EA	Evolutionary Algorithm
FPGA	Field-Programmable Gate Array
FSM	Finite-State Machine
GA	Genetic Algorithm
GDD	Generalised Disjunction Decomposition
HA	Half Adder
HDL	Hardware Description Language
IC	Integrated Circuit
IO	Input or Output
LCM	Logic Configuration Memory
LE	Logic Element
LUT	Look-up Table
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor
NI	National Instruments
NQF	National Qualifications Framework
NRF	National Research Foundation
PC	Personal Computer
PIG	Processing Integrated Grid
PLA	Programmable Logic Array
RAM	Random-Access Memory
RCM	Routing Configuration Memory
RM	Routing Matrix
SOC	System on Chip
SOPC	System on a Programmable Chip
V-FPGA	Virtual Field-Programmable-Gate-Array

VHDL	Very-high-speed Integrated Circuit Hardware Description Language
VI	Virtual Instrument (LabVIEW routine)
VLSI	Very-Large-Scale Integration
VRC	Virtual Reconfigurable Circuit
$\boldsymbol{WA_{Logic}}$	Write-address line used by the LCM
$\boldsymbol{WA_{Routing}}$	Write-address line used by the RCM
$\boldsymbol{WE_{Logic}}$	Write-enable line used by the LCM
$\boldsymbol{WE_{Routing}}$	Write-enable line used by the RCM
XOR	Exclusive OR

Chapter 1

INTRODUCTION

This chapter introduces the field of Evolutionary Hardware and its significance in the context of modern VLSI microelectronics. Evolutionary Hardware makes use of special nature-inspired search algorithms. The chapter discusses these algorithms as well as identifies challenges surrounding the research field, and why it has not yet been widely implemented in industry. Finally, the key objectives for this research are noted, and a paper structure is outlined.

1.1 Background

The development of microelectronics has boomed over the last four decades. Initially, in the sixties, integrated circuits (ICs) only hosted about 10 to 100 transistors (EngineersGarage, 2012). This was known as the Small- to Medium-Scale Integration era. During this era, IC developments were largely funded by the United States government for use in the military and space fields (Schnee, 1978). It was only later, in the seventies, that the manufacturing costs of ICs fell, allowing private firms to start penetrating industrial and consumer markets. By the mid-eighties, the transistor count on ICs well exceeded the 1000 range, thereby marking the Very-Large-Scale Integration (VLSI) age. Common examples of VLSI chips include microprocessors and Field-Programmable Gate Arrays (FPGAs). Today, ICs can typically contain millions of transistors, with programmable-logic device manufacturer Xilinx setting an industry record for 6.8 billion transistors on an FPGA chip (Yannou, 2011).

This rapid growth of technology was first observed by Intel’s co-founder, Gordon Moore, when in 1965 he predicted that the number of transistors on ICs would double approximately every two years (Moore, 1965). However, as the transistors count increases, there has been a desire to make computer hardware more complex and physically smaller.

Consequently, this is causing a bottleneck in the development of circuit designs. Enhancements in hardware design have been achieved by improving hardware-fabrication processes and using better-designed components, rather than optimising conventional circuits to make them more efficient.

Traditional circuit-design methodologies “rely on rules that have been developed over many decades” and require more human expertise for increasingly complex designs, which may be costly (Gordon & Bentley, 2002, p. 1). Complex designs are often tackled using powerful design tools, such as Electronic Design

Automation (EDA), high-level abstraction design-techniques and advanced IP-core libraries. However, the design-productivity gap is still increasing (Cancare, et al., 2011).

One solution to this design problem is Evolutionary Hardware (EHW). EHW is a combination of three disciplines: Computer Science, Electronics Engineering and Biology (Stomeo, et al., 2005, p. 1). Through modelling biological and natural intelligence, engineers and scientists have been able to mimic natural evolution in software for use in hardware design.

EHW has had a growing interest from many institutes across the world. Conferences, such as the Institute of Electrical and Electronics Engineers' International Conference on Evolvable Systems (IEEE ICES), allow researchers to share ideas and solutions in this research area. The IEEE (2013) describes the conference as "the leading conference in the field of evolvable hardware and systems." The conference's domain now covers a large array of research topics, including circuit diagnostics, self-repairing systems, evolutionary hardware design, real-world applications of evolvable hardware and evolutionary robotics. (See Thompson (1995), Higuchi et al. (1999), Hauschildt (n.d.), Barlow & Edwards (2001) and Moreno et al. (1998) as examples)

Despite the increased research and resources in the field, EHW systems remain largely unusable in real-world applications (Keymeulen, et al., 2003). Only a few engineering applications have shown promise (Higuchi, et al., 1999) (Mataric & Cliff, 1996), even though early pioneers claiming that evolution will soon be applied to large-scale machines (de Garis, et al., 1997). Researchers have raised a number of issues that have retarded the growth of EHW applications. These include the difficulties in configuring EHW platforms, scalability (Bedi, 2009), evolution time and problem complexity (Krohling, et al., 2002).

1.2 Evolutionary Hardware

EHW can be categorized into two main areas (Greenwood & Tyrrell, 2006, p. 9), namely Evolvable Hardware and Evolved Hardware:

1. **Evolvable Hardware**, or open-ended evolution, refers to hardware devices that can autonomously adapt to dynamic environments (artificially-intelligent systems), or automatically recover from hardware failures (fault-tolerant systems) (Sekanina & Freidl, 2005). Thus, open-ended evolution continuously evolves solution according to the environmental stimuli, while the best solution found so far is executed (Cancare, et al., 2011). Examples include temperature and radiation tolerant electronics (Keymeulen, et al., 2004) (Stocia, et al., 2004), robot controllers (Nolfi & Floreano, 2000) and image filters (Sekanina, 2002).
2. **Evolved Hardware**, or complete evolution, refers to the automatic synthesis of novel hardware circuits. Once a solution is found, the evolution is stopped. Evolved hardware is often more efficient—uses less hardware components—than conventional circuits (Miller & Job, 1999).
   An example of this efficiency was demonstrated by in 2003, when Koza et al. (2003) examined six different circuit patents, filed between 2000 and 2001, which were issued to universities or

commercial enterprises. Koza et al. (2003) used EHW to automatically synthesize new circuits that duplicated the patented circuits' functionality. It was found that only one of the six evolved circuits infringed on the original circuits' patents. This example clearly highlights the power of EHW, since Koza et al. (2003) only had to identify the desired outcome of the synthesized circuits; and had minimal knowledge on how the circuitry worked.

This research will concentrate on the advantages and downfalls of Evolved Hardware—in particular, how it can be applied when creating control circuitry. But first, in order to understand Evolved Hardware, one needs to examine the two fundamental parts that all EHW systems make use of: an Evolutionary Algorithm (EA) and an evolution platform.

1.2.1 Overview of Evolutionary Algorithms

Natural evolution is a process, taken by organisms or systems, with the aim of optimising the survival rate of the species within a dynamically changing environment (Engelbrecht, 2007). Darwin (1859) first coined the term "natural selection", which became the theoretical foundation of biological evolution. The Darwinian theory of evolution, as summarised by Engelbrecht (2007, p. 127), states:

> *In a world with limited resources and stable* ***populations****, each* ***individual*** *competes with others for survival. Those individuals with the* ***"best" characteristics*** *or traits are more likely to* ***survive and to reproduce****, and those characteristics will be passed on to their* ***offspring****. These desirable characteristics are inherited by the following generations, and over time become dominant among the population.*

The second part of the summary states:

> *During the production of a child organism, random events cause* ***random changes*** *to the child organism's characteristics. If these new characteristics are of benefit to the organism, then the chances of survival for that organism are increased.*

Taking cues from biology, computer scientists have been able to code various search algorithms that mimic evolution, which are collectively known as EAs. EAs and biological evolution share many commonalities, which have been highlighted using bold text in the quotations above, and are discussed further below:

- **A population with individuals**: An initial population is established in which all the individuals represent potential solutions to the computer problem.
- **Selection and survival of the best individuals:** Individuals are first evaluated using a problem-specific fitness function. A fitness function is a function that evaluates how closely an individual is to

achieving the desired objective. Fitness is often represented as a percentage, where a 100%-fit individual would be considered completely fit. Once fitness is established, a selection operator is applied to the population. Many selection operators exist, but fundamentally, they all select one or more of the fittest individuals from the population, usually with a degree of randomness. Then, these selected individuals become the parents that produce offspring.

- **Reproduction to produce offspring:** The selected parents reproduce using a crossover operator. Crossover recombines two parents in such a way as to allow the best characteristics of each parent to be passed to the offspring.
- **Random Changes**: A randomised mutation operator is applied to each offspring, thereby making each offspring unique. This allows diversity to be introduced into the new population. The above crossover and mutation procedures are repeated until a completely new population of offspring is produced—called a generation.

1.2.2 Using Evolutionary Algorithms to Evolve Hardware

When EAs are applied to digital logic, either at gate-level (by using gates such as AND, OR, XOR-gates), or function-level (by using micro-circuits such as adders, multipliers, multiplexers), they can be used to automatically design and synthesise complex circuits. This is essentially how most EHW systems are implemented. The level at which the evolution is performed is called "granularity", with gate-level evolution being fine, and function-level evolution being course (Cancare, et al., 2011, p. 2).

Various forms of EAs have been used to solve EHW problems. Examples of EAs include Cartesian Genetic Programming (CGP) (Miller & Thomson, 2000), Adaptive Genetic Algorithms (Ko, et al., 1997), Parallel Genetic Algorithms (Wang, et al., 2007) and Particle Swarm Optimisation (Eberhart & Kennedy, 1995). One of the more researched EAs used in EHW is the Genetic Algorithm (GA) (used by Gordon (2005), Sekanina & Friedl (2005), Martin & Poli (2002)).

Rustem (2012) claims that from 1990 to 2010, over 80 000 GA[1] journal papers have been published by Springer. In 2010 alone, approximately 12 200 papers were published. The trend seems to be increasing, showing that the GA-research field is still in its infancy.

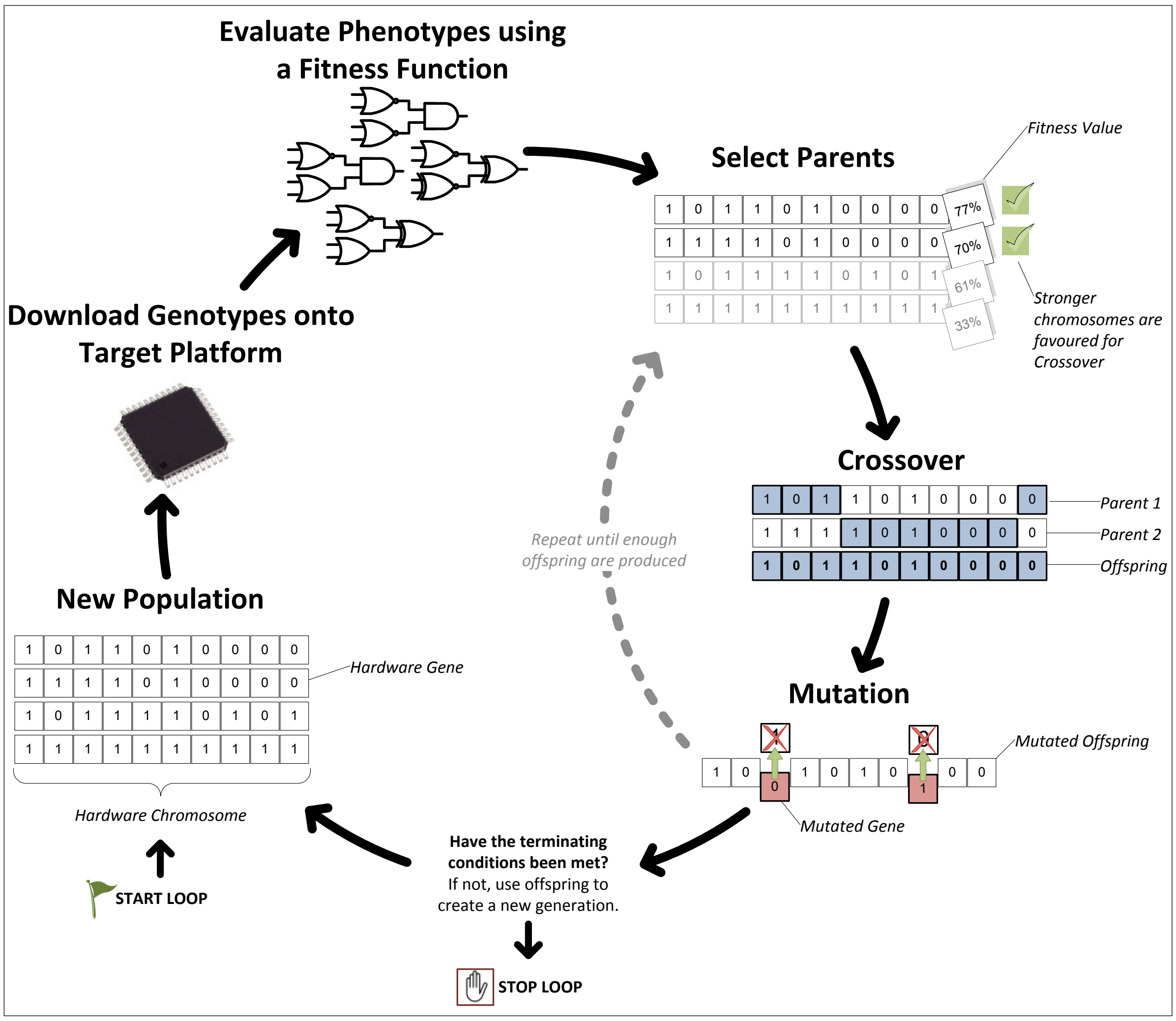
GAs are often referred to as search algorithms because they search, in a structured manner, for a solution that possesses the correct combination of parameters. "The set of all possible combinations of parameter values" is known as the GA's search space (Gordon, 2005, p. 17). Thus, since there are millions of combinations of circuits a GA has to search through, GAs typically have large search spaces from which a solution needs to be found.

[1] To clarify—"80 000 GA journal papers" includes all GAs relating to the computer-science field, and not necessarily GAs relating to only EHW.

Figure 1.1 shows an overview of an EHW system using a GA. The GA is executed using a continuous loop. Every iteration of the loop produces a new generation of individuals.

In EHW, each individual is represented using a hardware chromosome—also called the genotype. A hardware chromosome is a string of integers that represents a certain circuit configuration when decoded. Each single integer in the hardware chromosome is referred to as a hardware gene.

The decoded circuit is known as the phenotype. Phenotypes are configured in terms of functionality (the function of each component) and routing (how the components are connected to one another).



Evaluate Phenotypes using a Fitness Function
Select Parents
Fitness Value
77%
70%
61%
33%
Stronger chromosomes are favoured for Crossover
Download Genotypes onto Target Platform
Crossover
Parent 1
Parent 2
Offspring
Repeat until enough offspring are produced
New Population
Hardware Gene
Hardware Chromosome
Mutation
Mutated Offspring
Mutated Gene
Have the terminating conditions been met?
If not, use offspring to create a new generation.
START LOOP
STOP LOOP

Figure 1.1 Overview of the Hardware Evolution process using a Genetic Algorithm

The Figure 1.1's loop starts by creating an initial population of hardware chromosomes using random values, with each hardware chromosome representing a circuit and a potential solution. The chromosome are then decoded into the phenotypes and downloaded onto the EHW platform. The phenotypes are evaluated using a fitness function in order to establish which circuits have the most desirable characteristics. This is done by either implementing and testing the phenotypes using a simulated software model, or loading the circuits onto reconfigurable hardware. There are many different types of reconfigurable hardware testing platforms, which are discussed in the next section.

Once evaluation is complete, the GA operators are applied to the chromosomes, namely: selection, crossover and mutation. These genetic operators are used to maintain genetic diversity in the population.

The above process is iterated until either a perfectly fit individual is found, or a predefined number of generations are executed.

1.2.3 Hardware Evolution Platforms

To explore the different platforms[2] on which evolved circuits can be evaluated, we need to first consider the rate at which modern circuits and components are advancing. The field of EHW is very reliant on the underlying platform technology it is based on. The idea of implementing evolutionary characteristics into hardware and software systems has been around since the 1950s (Rustem, 2012), but it is only since the early 1990s that the hardware platforms have not constrained EHW systems (Iwata, et al., 1996). In addition, more recently, in the field of microelectronics, the feature size of components has decreased from 2-3µm in 2002 to 0.09µm in 2004 (Greenwood & Tyrrell, 2006), allowing for faster EHW systems to be implemented and thus allowing greater advancements to be made in the EHW field.

There are four major digital hardware platforms on which EHW has been tested (Lambert, et al., 2009):

1. **Application-Specific Integrated Circuits (ASICs):** These application-specific ICs are designed to execute EHW within very specific parameters. The chips are custom made according to the designer's EHW needs. ASICs are considered to be the least flexible hardware platform; and because of their uniqueness, are expensive to manufacture.
2. **Programmable Logic Arrays (PLAs):** A PLA chip provides limited flexibility in that it can only implement circuits consisting of AND and/or OR-gates. They are considered cheap devices.
3. **Complex Programmable Logic Devices (CPLDs):** Similarly to PLAs, CPLDs also have limited flexibility—they can implement circuits using AND, OR and/or XOR-gates.
4. **Field-Programmable Gate Arrays (FPGAs):** These devices are very flexible (allows for the reconfiguration of a circuit's routing and functionality), readily available and competitively priced when compared to ASICs.

Due to point four above, FPGA's have become the preferable choice for many EHW researchers (see (Moreno, et al., 1998) (Lambert, et al., 2009) (Sekanina & Freidl, 2005) (Smith, 2010) as examples), and thus this research will make use of the FPGA platform to evolve a control circuit. However, before exploring FPGAs further, it is first necessary to discuss the design of a control circuit using a finite-state machine.

[2] "Platform" in this case refers to the target hardware (i.e. board, IC, or chip) on which the evolved circuits (phenotypes) are implemented.

1.3 Evolving a Finite-State Machine

In the previous sections, the EHW field was broadly introduced. We will now describe, in the section below, the opportunity EHW presents when synthesising the control circuitry of a finite-state machine (FSM).

State machines are one of the oldest and most used ways of modelling the behaviour of systems (Wright, 2005). They are actively used in systems of: hardware design; biology; genetic; processing and retrieval of text information; and verification (Kryvyi, 2011). All state machines, regardless of their application, make use of the following fundamental principles:

- The whole system is described using a set of states
- The machine can only be in one state at a time
- To change to another state, a triggering event has to be initiated

A FSM, which is simply a state machine with a predefined finite number of states, can be used to help aid the design of a sequential-logic control system. FSMs consist of a set of states, an initial state, a set of finite inputs/outputs and a finite number of triggering events (Black, 2008).

Consider the following example of a FSM, being implemented in hardware as a sequential circuit, as explained by Floyd (2009, p. 436). A general FSM's circuitry, as shown in Figure 1.2, consists of a combinational-logic and memory section. The memory, which is often implemented using flip-flops, stores the state variable, i.e. current state's number. At any given time, the memory is in a state called the *current state*, and transitions to the next state on a clock pulse according to the conditions on the excitation lines. The combinational logic controls the system's excitation lines according the system's inputs.

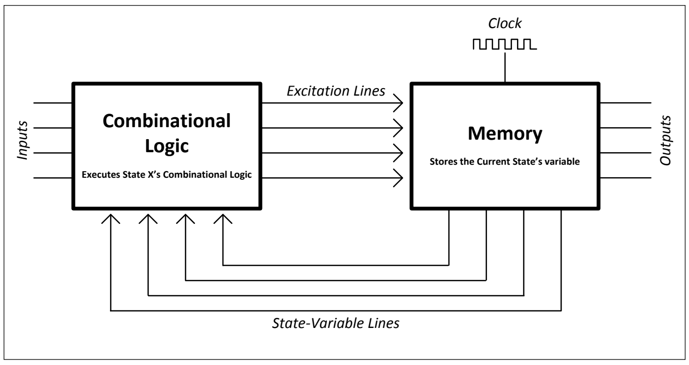


Figure 1.2 A general FSM's circuit implemented as a sequential circuit

Now, consider the combinational logic. In order to advance to the next state, a triggering or input condition needs to be met. If met, the relevant excitation lines are activated, thereby initiating the next state-variable in memory. The state variable is communicated to the combinational logic via the state-variable lines.

Most FSM are cyclic in nature, and execute unconditionally. There are many examples of FSM circuits used as cyclic control systems. Examples include counters, traffic lights, wristwatches, turnstiles and elevators (see (Floyd, 2009) (Harel, 1986) (Wright, 2005) for examples).

In evolved hardware, much emphasis has been placed on evolving simple, functional circuits, as opposed to evolving real-world FSM circuits. The functional circuits include multipliers, adders, filters and oscillators (see (Vassilev, et al., 1999), (Sekanina & Freidl, 2005), (Sekanina, 2002), (Thompson, 1995) for examples). These simple circuits have been popular because they are small and practical enough to evolve, while still allowing the researcher the ability to test and adjust new EHW platforms, GAs and architectures.

Nevertheless, Rustem (2012) did manage to evolve a complex six-state, four-input/four-output FSM. Two key points, taken from Rustem's (2012) research, include:

1. The FSM was evolved using a software simulation, i.e. it was not evolved using a hardware platform
2. The final circuit was tested on a CPLD and not an FPGA

These two points are significant, as even though this research will also aim to evolve a four-input/four-output FSM, it will differ to previous research by not simulating the evolution and not using a CPLD. Instead, the FSM will be evolved and implemented on an FPGA platform.

1.4 Problem Statement

There are three main problems addressed in this study:

1. The difficulty of configuring modern FPGAs to implement EHW
2. The influence of different genetic operators on the efficiency of EHW GAs
3. Using modular evolution to address scalability issues when evolving FSMs

1.4.1 The Virtual Reconfigurable Architecture

Altera (2013) describes FPGAs as "semiconductor devices that can be programmed after manufacturing." This is a particularly vital advantage in that an FPGA allows the designer to reconfigure the hardware architecture according to the designer's EHW needs.

In order to understand this reconfiguration, first consider the simplified internal structure of an FPGA, shown in Figure 1.3, consisting of IO blocks, logic blocks, routing switches and wires. FPGAs use thousands of basic programmable logic blocks which are arranged in a row/column matrix formation and connected via reconfigurable interconnections. By allowing the logic blocks to communicate to one another and to external hardware via the IO blocks, complex combinational logic and sequential logic operations can be performed. This allows the user to implement any hardware circuit on the FPGA by simply reprogramming the chip's configuration.

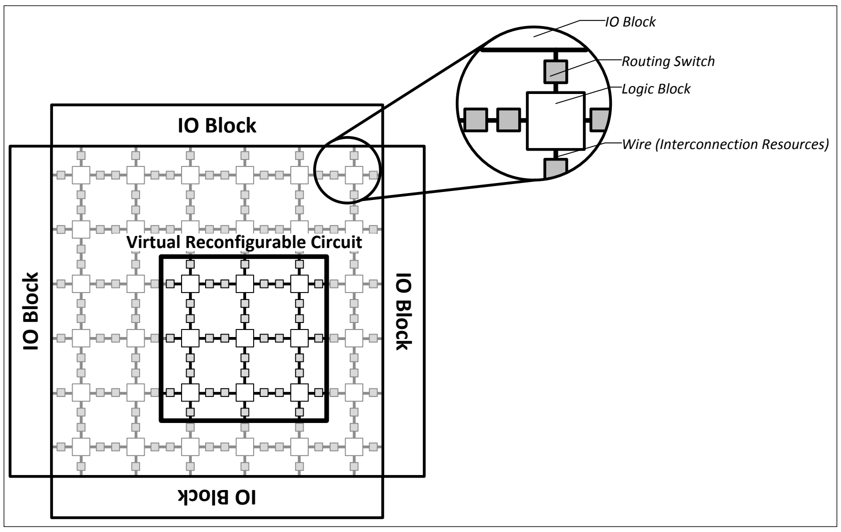


Figure 1.3 Overview of an FPGA with a virtual reconfigurable layer

When programming an FPGA, the designer makes use of the FPGA vendor's software, which loads the circuit's configuration data onto the FPGA using a configuration bitstream. The bitstream is a sequence of bits which describes all the necessary data needed to implement the user-defined circuit, and is stored in the FPGA's configuration memory. In the case of EHW, the hardware genotypes would be decoded into an FPGA bitstream in order to implement the phenotype circuit.

The initial FPGAs used for EHW, such as the Xilinx 6200 used by Thompson (1996) and Gers, et al. (1998), offered many evolutionary-friendly features, including partial reconfigurations, access to single logic blocks, an open architecture and protection against illegal configurations (Haddow & Tufte, 2001). However, the Xilinx 6200 FPGA is now obsolete and most modern FPGAs have become less EHW-friendly.

FPGA manufactures have prohibited the modification of the configuration memory or bitstreams, claiming that it is "intellectual property" or "proprietary to the vendor" (Bedi, 2009, p. 2) (Majzoobi, et al., 2012, p. 200). The only FPGA devices capable of performing bitstream configurations are the "Xilinx 4000" and the "Xilinx Virtex series" (Smith, 2010, p. 1) (Bedi, 2009, p. 1). Still, even though these FPGAs can be used, few evolution features, found on the original Xilinx 6200, have been retained.

To overcome this problem, Xilinx introduced JBits. JBits is a set of Java classes that allow the designer access the FPGA's bitstream via an application-program interface (API) (Guccione & Levi, 1999). This, in turn, gives the designer the capability of designing, modifying and dynamically modifying circuits on JBits-enabled FPGA devices.

However, JBits has shortcomings: larger EHW chromosomes can be cumbersome to dynamically decode into bitstreams; JBits can only be used on Xilinx FPGAs; JBits can only be configured using external hardware attached to configuration ports.

In light of the above-mentioned problems, a more promising solution, used in this research, is the virtual-reconfigurable-circuit (VRC) architecture. A VRC, as shown in Figure 1.3, is a virtual reconfigurable layer residing on top of the FPGA hardware layer. The VRC allows the designer access to its virtual configuration memory (i.e. a set of registers) and bitstream (i.e. the chromosome), thus allowing run-time configuration of the FPGA to be performed.

Notice, in Figure 1.3, that the VRC only makes use of a portion of the FPGA's available logic blocks. This is because VRCs maintain complete encapsulation during evolution. Designers thus have the opportunity to utilise the remaining logic blocks for other digital functions, such as implementing a soft-processor.

There is no standard for VRCs—each VRC is unique, and thus VRCs are made to the exact requirements of a given EHW application. Different designers may create different virtual architectures that perform similarly, but utilise different methodologies.

In 2010, Smith presented a VRC architecture called the Virtual-FPGA (V-FPGA). It was shown, through software simulation, that the V-FPGA was successfully configured to implement a clock-divider circuit. However, the V-FPGA also presented a drawback. Smith (2010, p. 2) noted that any reconfigurable architecture, which is built on top of an FPGA architecture, creates inevitable "inefficient resource utilisation". This disadvantage, however, is negligible for prototyping EHW applications. Most VRC architectures have shown promise in the EHW field (Sekanina & Freidl, 2005).

1.4.2 Optimising Genetic Operators for use on the Virtual-FPGA

Variations in GAs—such as changing the population size, crossover, selection and mutation—may a have vast impact on the performance of the algorithm. From the literature, GA operators are usually optimised for each EHW system according to the makeup of the evolutionary platform and architecture, the complexity of the circuit being evolved, the fitness function and the representation of the hardware chromosome. Thus, every GA is unique to the specific application.

Vassilev et al. (1999) tested the effectiveness of the crossover and mutation operators in a GA configured to evolve a two-bit multiplier. They did this by examining the GA's fitness landscape[3]. The setup made use of a VRC which could reconfigure 16 LEs according to their function and routing. Each cell could function as either a two-input logic gate or a 2-1 multiplexer. Restrictions were also applied to the circuits' outputs: Outputs could only be connected to a select number of cells.

Vassilev et al. concluded that uniform crossover was not a favourable operator for evolving the functionality and internal connectivity of the evolved multiplier. Furthermore, Vassilev et al. states, "the mutation landscapes appear to be relatively smooth, and therefore, it is feasible for an evolutionary search." Although

[3] Fitness Landscape: A collective term for the fitness values of all the possible solutions within the search space.

many EHW researchers have, and still do, obtain success using crossover (see (University of Heidelberg, 2011) (Miller & Thomson, 1998)), Vassilev et al. suggests that by omitting crossover and using mutation only, the GA may be optimised.

The above findings were successfully demonstrated by both Vassilev & Miller (2000) and Selanina & Freidl (2005). The latter evolved both an adder and multiplier circuit by omitting crossover, constraining the LEs' functionality and using effective fitness functions.

It is important to note that all the sources, used in this section, have implemented different EHW architectures, platforms, cell restrictions and GAs. Thus, because each EHW system is unique, it would be advantageous to investigate the outcome of optimising the EHW GA—as done by Selanina & Freidl (2005)—to evolve a two-bit multiplier on the V-FPGA.

Evolving a multiplier circuit is a good starting point when experimenting with a new EHW architecture and setup. Multipliers have been well researched (Miller & Thomson (1998), Vassilev et al. (1999) and Selanina & Freidl (2005)), are mathematically useful and are relatively uncomplicated.

1.4.3 Scalability

Since EHW makes use of search algorithms that need to explore large search spaces, the intricacy and sheer scale of finding a solution becomes more apparent as the solution circuit becomes more complex. Many researchers have recognised that scalability is a hindrance in the successful implementation of EHW in real-world applications (Gordon & Bentley, 2005) (Vassilev & Miller, 2000).

Scalability, in this context, refers to the difficulty of finding a satisfactory solution for large, complex problems—those found in real-world applications—due to the GA's search space being too large, or the solution being too complex to be implemented on the EHW system. To put scalability in context, remember that EHW systems have to evolve circuitry by placing logic gates—often thousands of logic gates in larger systems—in very specific configurations. In addition, as the complexity of the circuit increases, so the genotype length and the time required to calculate the fitness of each phenotype also increases. This results in there being billions of potential solutions which are cumbersome and time-consuming to explore, even with evolutionary techniques.

One suggested solution to scalability is function-level evolution, which uses micro-circuits as the building blocks for larger complex circuits (Higuchi, et al., 1997) (Antola, et al., 2007). However, function-level evolution merely moves the problem: A designer is still needed to successfully identify the suitable function-level circuits. Also, function-level evolution is inheritably inefficient due to the evolution not being performed at the lowest gate-level. For instance, a three-bit multiplier requires only 23 LEs using gate-level evolution (Vassilev & Miller, 2000); compared to 14 LEs and 7 three-input binary multipliers (Miller, et al., 2000).

A more reasonable approach, which follows on from function-level evolution, is modular evolution. (Some literature refers to *modular evolution* as *incremental evolution*.) Modular evolution simplifies an EA's search space by decomposing the desired circuit into smaller modules, and then re-assembling the smaller modules once evolved. This process of circuit decomposition-and-assembly is inspired by the divide-and-conquer principle, which has been observed in biological systems (Torresen, 1998) (Wang, et al., 2007).

Since each module, or sub-circuit, can be independently evolved using gate-level components, the evolution allows for fine-granularity (Gordon, 2005). Gordon (2005, p. 16) goes on to further state that modularisation in evolutionary designs have "proved useful in many respects", however have still not caught up with the modular evolution used in "biological organisms". For example, mammals are modular, comprising of sub-components such as organs and tissue. Each sub-component has been specifically evolved in order for the mammal to thrive in its natural environment.

Modular evolution is not a new concept. Torresen (1998) showed, through experimentation, that the number of generations required, when using smaller sub-systems, "can be substantially reduced when compared to direct evolution." Similar results are concluded by Vassilev & Miller (2000): "To evolve digital circuits using modules is faster, since the building blocks of the circuit are sub-circuits rather than two-input gates."

This research will demonstrate that by modularising EHW problems, EHW systems are able to synthesise larger, complex circuits that are typically not directly evolvable due to scalability. In particular, unlike the previous research, the modular evolution will be applied to a FSM's circuitry.

1.5 Statement of Objectives

1.5.1 Major Objectives

With the above background considered, there are three major objectives addressed in this research:

1. This research will form a continuation of the research done on the V-FPGA architecture designed by Smith (2010). Smith (2010) first introduced the V-FPGA in his paper entitled, "A Virtual VLSI Architecture for Computer Hardware Evolution". The paper's objective was to "provide and method to facilitate hardware evolution" and "not to demonstrate hardware evolution." It thus follows that this study will aim to show, by successfully implementing and demonstrating evolved hardware on an FPGA, that practically the V-FPGA is a viable evolution platform.
2. Following Vassilev et al. (1999) and Selanina & Freidl's (2005) research, this study will explore the effectiveness of optimising the EHW GA for use on the V-FPGA by omitting the crossover operator, applying evolution constraints and optimising the fitness function. The GA will be used to evolve smaller circuits, such as a two-bit multiplier, in order to provide a viable means of studying major EHW issues, such as: scalability, evolution time, circuit efficiency and the V-FPGA configuration.

3. Finally, once the V-FPGA and GA are optimised, modular evolution will be demonstrated by evolving the circuitry required for a simple, but real-world FSM. This will aim to show that larger control circuits may be scalable if the evolution process is constrained.

1.5.2 Minor Objectives

In order to achieve the major objectives, the following minor milestones have to be attained:

- The V-FPGA architecture has to be verified through simulation.
- A genotype representation of the phenotype has to be devised, i.e. a standard by which all the phenotypes' data can be represented and modified in software.
- A software procedure for decoding the genotypes into phenotypes has to be programmed.
- An effective fitness function, which correctly evaluates the outcome of each phenotype, must be developed.
- The size of the GA's search space needs to be investigated.
- A critical-path analysis of each circuit needs to be completed in order to understand the manner in which the circuits are evolved. With this knowledge, better fitness functions and operators can be developed to further enhance the EHW system's efficiency.

1.6 Methodology

The research methodology is outlined as follows:

1. Create a V-FPGA VHDL file and simulate a two-bit multiplier using software.
2. Create a larger V-FPGA, and download the VHDL file onto an FPGA chip.
3. Create a LabVIEW GA which can evolve a two-bit multiplier.
4. Optimise the GA, through experimentation, by adjusting different operators.
5. Use the optimised GA to evolve the combinational circuits used in a FSM.

To create a V-FPGA VHDL-based file, the following parameters have to be defined: the number of logic elements (LEs); the number external inputs and the number of external outputs. Initially, the V-FPGA will be configured to use four external inputs, eight LEs and four external outputs. This configuration will be adequate to implement a two-bit multiplier on the V-FPGA. Through software simulation, the conventional two-bit multiplier will then be tested, thereby validating if V-FPGA architecture works correctly.

Once the simulation is successful, a larger V-FPGA will be configured using four external inputs, twenty LEs and four external outputs. This V-FPGA configuration will be used, in conjunction with the GA, to evolve the control and multiplier circuits. To do this, V-FPGA will be downloaded onto an FPGA chip.

The GA will be programmed using LabVIEW software on a PC. Since the GA and V-FPGA will not be executed on the same processor, this research will demonstrate off-chip evolution. Off-chip evolution occurs when the evolutionary algorithm is performed on a separate processor not incorporated into the chip containing the target EHW (Torresen, 2004).

A two-bit multiplier will first be evolved using different GAs operators, constraints and fitness functions. For the successfully implemented GA, the number of generations, fittest parent per generation, phenotype and evolution time will be recorded. Appropriate statistical and design comparisons will be made between the evolved phenotype and the conventional multiplier circuit.

Finally, an example eight-variable FSM case study will be also be evolved. This case study will demonstrate the modular evolution of a FSM's state circuits. Since the V-FPGA and GA will be constrained to only evolve forward-feed circuits, only the combinational section of the FSM will be evolved. Similar to the evolved two-bit multiplier, the evolution time, number of generations and final phenotype will also be recorded.

1.7 Paper Structure

The dissertation will continue with a literature review in Chapter 2. The chapter will begin with a brief history of the EHW field; and will conclude by identifying several difficulties hindering the implementation of real-world applications.

Chapter 3 will discuss the V-FPGA architecture. The chapter will be presented in twofold: first, how to configure a circuit's LEs within the V-FPGA; second, how to configure a circuit's routing structure using a specialised Routing Matrix. The chapter will close with the simulation of a two-bit multiplier.

Chapter 4 will illustrate the evolution of a two-bit multiplier using the V-FPGA architecture. The chapter will start by describing the experimental hardware setup. Part of the setup will include configuring and compiling the V-FPGA on an FPGA chip. Once configured, the effectiveness of two different GAs will be investigated. Finally, the chapter will end with an account of three evolved multiplier phenotypes as well as a discussion on the arising evolution difficulties, the possible causes and solutions.

Chapter 5 will be to demonstrate a means of overcoming scalability by using modular evolution. The chapter will starts by introducing a case study and a real-world FSM. Then, the FSM will be modelled using a state and block diagram. Finally, each state's combinational sub-circuit in the FSM will be is independently evolved. The chapter will end with a discussion.

The final chapter will conclude the research by reviewing the study's contributions, and presenting considerations for future work.

Chapter 2

LITERATURE REVIEW

This chapter surveys the benefits and shortcomings of previous EWH systems. The chapter starts with a brief history of the EHW field, which was primarily driven by the development of reconfigurable FPGAs. Then, from the literature, several difficulties of implementing real-world applications are identified. In particular, scalability is addressed in detail, with a number of solutions being proposed.

2.1 A History Driven by Target Platforms

2.1.1 The Early Years

EHW can first be traced back to 1993, when de Garis (1993) published a paper on "Darwin Machines". It was proposed that Darwin Machines would use EHW to evolve artificial nervous systems, which would be used to control artificial creatures and/or robots. de Garis believed that "software configurable hardware", in particular FPGAs, would allow the Darwin Machines to be realised "within a year or two".

However, this was not to be. In 1995 Xilinx introduced the XC6200 FPGA device-family—the first reconfigurable FPGAs (Lazzaro, 2010). It was only after the introduction of the XC6200 that the first research on implemented EHW was published. The research, conducted by Thompson (1996) over a three week period, showed how an EHW system managed to evolve a frequency discriminator able to distinguish between 1kHz and 10kHz square waves. The XC6216 FPGA allowed Thompson to directly reconfigure the FPGA's logic blocks by manipulating the configuration bitstream.

Thompson's work can be considered an important milestone in EHW, since it was the first publication to demonstrate the advantages of EHW through bitstream manipulation.

However, in addition to the evolution, Thompson's experimentation revealed that the evolved discriminator was susceptible to external conditions, such as temperature, and was device-dependent, i.e. the evolved discriminator malfunctioned when implemented on different FPGAs. These factors made the evolution volatile.

To address this volatility, Thompson turned his attention to the EHW system's "robustness". Robustness, as defined by Thompson (1998, p. 1), means "to be able to maintain satisfactory operation when certain variations in the circuit's environment or implementation occur." To increase robustness, Thompson created an "Evolvatron" tool to simultaneously test the evolved circuits in different conditions and on different devices. By testing the circuits in all conditions, a more precise fitness function could be used to accurately award fitness values (Thompson, 1999). The Evolvatron proved to be a success.

Besides Thompson, other notable research done before 2000 was conducted by Torresen (1999), Higuchi, et al. (1999) and Macias (1999).

Torresen and Thompson's research have similarities: Both used the Xilinx 6200 FPGA and both initially lacked robustness. Torresen managed to evolve a character recognition system which performed as expected, however lacked "noise robustness". His paper also highlighted some reasons as to why EHW had not yet been widely applied—mostly due to scalability.

Higuchi, et al.'s research, in contrast to Torresen and Thompson, investigated an ASIC, which was named the "GRD chip", to dynamically reconfigure gates within the chip. The GRD chip, which made use of a RISC processor, evolved the hardware using an Artificial Neural Network (ANN) algorithm. The aim of the study was to introduce an EHW chip capable of solving a variety of real-world problems, such as robot navigation and the control of a prosthetic hand.

Similarly, Macias also created an EHW ASIC, called the "Processing Integrated Grid", or simply PIG, consisting of a reconfigurable-element grid which allowed the implementation any large digital circuit. What made the PIG architecture unique is that it was capable of parallel reconfiguration. Hence, different elements within the circuit could be evolved simultaneously.

2.1.2 The Virtual-Reconfigurable-Circuit Era

Besides specialised ASIC chips, most FPGA-based EHW research before 2000 was performed on a Xilinx XC6200 device. Cancare, et al. (2011) and Hollingworth, et al. (2000) highlight some of the XC6200 features:

- **Safe Reconfiguration:** The FPGA could not be physically damaged through implementing illegal bitstream configurations that created short circuits.
- **Partial Reconfiguration:** Selected portions of the FPGA could be reconfigured independently.
- **Fast Reconfiguration:** The FPGA used a parallel interface, allowing faster configurations compared to previous devices.
- **Static Random-Access Memory (RAM)**: The FPGA made use of static RAM that could be swiftly accessed and rewritten through standard interfaces.
- **Known Data Format**: The well-known bitstream format allowed the designer to alter individual parts of the configuration.

In 1998, Xilinx withdrew the XC6200 device-family from further development, thereby indirectly halting any EHW research done using direct bitstream-manipulations (Lazzaro, 2010). The reasoning behind the discontinuation, as Cancare, et al (2011, p. 3) explains, is that “the open bitstream format was allowing the reverse engineering of proprietary-hardware intellectual-property cores.”

Later in 1998, Xilinx released the Virtex FPGA family to replace the XC6200. It retained most of the XC6200 features, but had drawbacks: Virtex FPGAs did not allow the designer direct access to the FPGA’s bitstream or static RAM. In addition, the Virtex series allowed multi-directional routing, thus allowing unsafe and potentially damaging routing configurations (Hollingworth, et al., 2000).

Subsequently, the discontinuation was a major turning point. To combat the Virtex shortfalls, many designers attempted direct bitstream-manipulation by means of well defined APIs such as JBits (as discussed in Chapter 1) (Hollingworth, et al., 2000). However, although successful, JBits was limited and clumsy, and there was a need to find a more universal and standardised EHW approach—one which did not rely purely on Virtex FPGAs. As such, researchers proposed VRCs.

VRC-based solutions, such as those designed by Slorach & Sharman (2000), Sekanina & Azeddien (2000), Sekanina & Freidl (2005) and Smith (2010), generally always consisted of:

- **Logic Elements (LEs):** Programmable elements, sometimes called cells
- **Programmable Interconnection Network:** Connected the LEs and external IOs
- **Configuration Memory:** Stored the VRC’s virtual bitstream so that the desired circuit could be implemented

To create a VRC, the above components would first be modelled at a higher level of abstraction, using either C++ or C. Thereafter, an executable file would be generated to produce the required Hardware-Description-Language (HDL) code needed to deploy the VRC on a target FPGA.

VRC architectures were able to make EHW systems portable by providing a standardised, device-independent means of configuring circuits on FPGAs through virtual bitstreams. Thus, VRCs provided a major advantage over specialised direct-bitstream-FPGA and ASIC systems.

2.1.3 Future Platforms

The latest research done by Dobai & Sekanina (2013) investigated two new potential target platforms: Xilinx’s latest EHW offering—a stand-alone programmable system-on-chip (SOC) called Zynq-7000; and a hybrid-VRC architecture.

The Zynq-7000 integrates programmable logic with a dual-core ARM processor. The main difference between the Zynq and previous FPGAs is that FPGAs are typically built around the programmable logic, with on-chip processors being an extension, i.e. part of the programmable logic is configured into a soft-processor. On the contrary, the Zynq-7000 is an FPGA platform built around its hard-processor (Dobai & Sekanina, 2013). Thus, the processor and programmable logic are independent, and can be accessed and configured independently.

For example, the configuration bitstream can be downloaded from memory onto the programmable logic without utilising the processor.

Hybrid VRCs make use of both direct-bitstream and VRC architectures, and can thus only be used on specific Xilinx FPGAs. Essentially, a hybrid VRC uses the direct-bitstream approach to program the LEs' functionality, and the VRC architecture to route the LEs. In this way, hybrids are able to take advantage of VRCs' faster reconfigurations, while lowering the hardware resources needed to implement the VRC.

Finally, Dobai & Sekanina (2013, p. 7) concluded that "the hybrid approach represent computationally equal power to the pure VRCs." However, VRCs still present device-independent advantages. In addition, although the Zynq platform is promising, further work is needed to "exploit all the features and advantages of this platform."

2.2 Factors Delaying Real-World Applications

There has been extended research, with some success, into VRCs and target platforms to improve the portability of EHW systems. Thus, systems are becoming less device and application specific. However, there are additional factors which are delaying the field's progress.

Many researchers agree that real-world applications of EHW systems, such as the control circuitry on robots, have been, and still are, a major challenge (Higuchi, et al., 1999) (Cancare, et al., 2011) (Wang, et al., 2007) (Sekanina, 2003). From the literature, the following key areas have been identified as troublesome:

1. **Performance:** Conventional circuit-design methods currently outperform EHW. For example, only basic adder and multiplier circuits have been evolved.
2. **Cost:** Designing and manufacturing an EHW ASIC, or using the latest FPGAs, is costly.
3. **Time:** For intrinsic evolution, testing each chromosome becomes increasingly time-consuming as the number of external IOs increase.
4. **Scalability:** Digital circuits generally do not scale proportionally. For example, a two-bit multiplier can be evolved within 5000 generations, whereas a 3-bit multiplier could take as much as three million generations to evolve (Miller, et al., 1999).
5. **Chromosome Length:** Most EHW chromosomes use one-to-one mapping, where one gene configures either a single LE or a routing node. Thus, to evolve larger circuits, the length of the chromosome can be cumbersome and computationally taxing.
6. **Variety of Research:** Because the research field is varied (with each system using different target platforms, EAs and being designed for different applications) the research is scattered with each researcher's work tending to be isolated. Also, most literature only describes the EHW system and/or relevant results. Little technical detail such as coding and exact architecture implementation is included.

It is vital for designers to consider all the above factors holistically when designing EHW systems, as they all interlink. The factors are now discussed further in the subsections below.

2.2.1 Performance, Cost and Time

On both the Xilinx (2013) and Altera (2013) websites, the FPGA manufacturers state that FPGAs provide faster time-to-market and no non-recurring engineering costs—the costs of developing, designing and testing an IC—when compared to ASICs. Xilinx (2013) further states that FPGA devices do not require “complex and time-consuming floor-planning, place-and-route, timing-analysis and mask stages.” Thus, FPGAs provide a clear advantage over ASICs when considering costs and flexibility—which are two important points considered by EHW researchers.

However, as discussed in Section 2.1.2 , FPGAs can be limiting, and as FPGA vendor’s release newer models, procuring the latest technology, such as the Zynq platform, can also be costly.

Generally, VRCs do help to minimise FPGA costs, since they have no special hardware requirements. But, even so, designers should be aware that although cheaper, VRCs do compromise on performance. This is because VRCs, by their very nature, are secondary configurable layers, which make use of FPGA logic blocks configured as multiplexers and registers to implement logic. Direct-bitstream evolution, on the other hand, does not require additional hardware resources. Thus, the evolution can be performed at higher frequencies with the evaluation of individuals being faster (Dobai & Sekanina, 2013).

To address evaluation times in VRCs, researchers have proposed multi-VRC solutions (Wang, et al., 2007) (Cancare, et al., 2011). Multi-VRCs, as the term suggests, make use of several independent VRCs implemented on a single commercial FPGA chip. This allows the EA to simultaneously download and evaluate different individuals on different VRCs, thereby dramatically reducing evolution time and improving performance.

2.2.2 Scalability and Chromosome Length

Scalability and chromosome length are directly related. As the complexity, i.e. number external IOs, of the solution circuit increases, so does the size of the search space and length of the chromosome string. Long chromosome strings are an inevitable side-effect of complex systems. Thus, by simplifying or scaling-down complex circuits, chromosomes string can be reduced.

In Chapter 1, scalability and function-level evolution was briefly introduced, and modular evolution using circuit decomposition was proposed as a solution. These techniques are all based on reducing the EA’s search space. However, a different approach to scalability is to increase the EA’s computing power. To do this, researchers have proposed parallel evolution (Wang, et al., 2007) (Cancare, et al., 2011). Parallel-evolution schemes make use of two or more independent EAs, i.e. the EAs run in parallel, thereby allowing multiple circuits to be evolved simultaneously.

2.2.2.1 Overcoming Scalability using a Multifaceted Approach

In 2007, Wang, et al. (p. 33) evolved both three-bit adders and multipliers in less than three seconds, which was reportedly "untouchable by any other reported evolvable system." The research showed that three-bit adders/multipliers were scalable if a mutifaceted approch was taken. Wang, et al. (p. 25) suggested overcoming scalability using three techniques: optimising the EA; limiting the chromosome length; and "decreasing the computational complexity of the problem."

Firstly, Wang, et al. made use of a GA optimised, as suggested in Chapter 1, by omitting the crossover operator. Also, a multi-VRC platform was used, thereby allowing the GA to test the candidate circuits faster. Although not done by Wang, et al., other optimisations could also include finding more effective mutation and crossover operators (if used), and improving the fitness function (see (Martin & Poli, 2002) and (Vassilev, et al., 1999)).

Secondly, the chromosome length was limited by decomposing the solution circuit into modules, as done in modular evolution. There are different ways of decomposing circuits, each with varying levels of success and complexity. Examples include Shannon decomposition (Kalganova, 2000), disjunction decomposition (Stomeo, et al., 2006) and output decomposition. Wang, et al. made use of output decomposition, which decomposes a circuit according the number of external outputs. Figure 2.1 shows the decomposition of a two-bit multiplier into two modules. Each module has the same external inputs, but only half of the available external outputs.

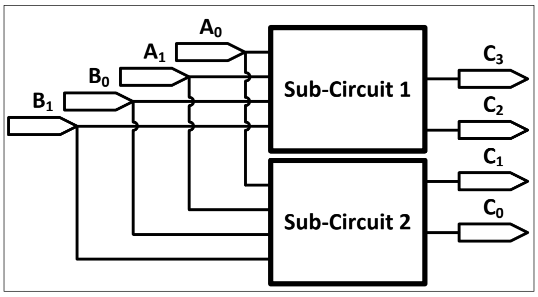


Figure 2.1 Output decomposition of a four-output circuit.

Thirdly, to decrease the computational complexity, parallel evolution was used. It is important to understand that unlike modular evolution, parallel evolution does not decrease the complexity of the solution circuit. It only improves the computational capacity of the evolution. Wang, et al. used a two-core system, with each core running independent GAs and VRCs, and evolving a single sub-circuit. Theoretically, there was no limit to the number of implemented cores and VRCs.

The results showed that modular evolution decreased the number of generations required from over 18-million generations for standard evolution, to approximately 133 thousand. In addition, parallel evolution was able to improve the evolution time from approximately 77 seconds for standard evolution, to a mere 2.6 seconds.

Finally, Wang, et al. (2007) acknowledge that more complex circuits would still need to be tested, adding that "future work will be devoted to applying this scheme to other more complex real-world applications."

2.2.3 Variety of Research

The diversity of EHW research may be considered both beneficial and problematic: beneficial because diversity promotes progress and unique solutions; problematic because there is little standardisation within the field.

To analyse the diverse EHW research, EHW systems can be classified, as suggested by Torresen (2004), into the following categories: evolutionary algorithm, evolution level, target platform/architecture, degree of evolution and scope. The evolutionary algorithm, evolution level and target platform have been discussed in Chapter 1. Thus, the remaining categories are defined below:

- **Fitness Computation:** Refers to the manner in which the fitness of a circuit is computed. *Extrinsic evolution* only downloads the elite chromosome to the target platform. Thus, much of the evolution is simulated. *Intrinsic evolution* implements and tests each chromosome in hardware.
- **Degree of Evolution:** Refers to the whether or not "the evolutionary algorithm is performed on a separate processor incorporated into the chip containing the target EHW" (Torresen, 2004, p. 6). *Off-chip evolution* does not make use of an incorporated processor, while *on-chip evolution* does. *Complete evolution* does not use a processor, but rather uses specialised hardware.
- **Scope:** *Static evolution* only puts the evolved circuit to use once evolution is complete. *Static evolution* is typically used in Evolved Hardware. *Dynamic evolution* is undertaken while the evolved circuit is used. Thus, *dynamic evolution* is used in Evolvable Hardware.

Table 2.1 shows an overview of eleven EHW research papers conducted from 1996 to 2013.

Table 2.1 Overview of FPGA and ASIC-based EHW applications, found in literature

Case Study	Evolutionary Algorithm	Evolution Level	Target Platform and Architecture	Fitness Computation	Degree of Evolution	Scope
Robot control system (Thompson, et al., 1996)	GA	Gate	Direct-Bitstream on FPGA	Extrinsic	On Chip	Dynamic
Function-level EHW (Higuchi, et al., 1997)	ANN	Function	Direct-Bitstream on FPGA and PLD	Extrinsic	On chip	Dynamic
Proposed ASIC for a CATV modem and prosthetic EMG-controlled hand (Murakawa, et al., 1999)	ANN	Gate	ASIC	Intrinsic	On chip	Dynamic
Crossover optimisation (Martin & Poli, 2002)	GA	Gate	VRC on FPGA	Extrinsic	Off chip	Static
Human gene recognition	GA	Gate	VRC on FPGA	Extrinsic	Off Chip	Static

(Yasunaga, et al., 2003)						
Digital circuit synthesis (Sekanina & Freidl, 2005)	GA	Gate	VRC on FPGA	Intrinsic	Complete	Static
Sequential and combinational logic synthesis (Popa, et al., 2006)	GA	Gate	Direct-bitstream on FPGA and CPLD	Extrinsic	Off chip	Static
Multiplier and adder synthesis (Wang, et al., 2007)	GA	Function	VRC on FPGA	Intrinsic	Complete	Static
Proposed EHW system for use on an Inverse Pendulum Problem (Cancare, et al., 2010)	GA	Gate or Function	Direct-Bitstream on FPGA	Extrinsic	On chip	Dynamic or Static
Analogue circuit synthesis (University of Heidelberg, 2011)	GA	Gate	ASIC	Extrinsic	Off chip	Static
Image filtering (Dobai & Sekanina, 2013)	GA	Function	Zynq-7000, VRC Hybrid on FPGA	Intrinsic	Complete	Dynamic or Static

From the table, the variations in the research are apparent. Most research deals with simple digital/analogue circuit synthesis, image filtering and system refinements. Again, there is little evidence of real-world applications being implemented, and thus most research is still focussed on refining systems, and testing these refinements by evolving simple circuits such as adders and multipliers.

2.3 Chapter Summary

The EHW field was primarily target-platform driven, with the Xilinx 6200 FPGAs initially being the platform of choice. Later, post 2000, the Xilinx Virtex FPGAs limited direct-bitstream evolution, leading to the popularisation of VRCs, with many designers favouring VRCs due to cost and portability advantages.

Research is now concentrated on scalability—EHW needs to become more prominent in real-world applications. From the literature, there are many solutions to scalability, but the most promising solutions will need a multifaceted approach.

Chapter 3

THE VIRTUAL-FPGA

This chapter discusses the V-FPGA architecture designed by Smith (2010). The V-FPGA is a virtual reconfigurable electronic circuit dedicated to the implementation of EHW by means of partially reconfiguring an FPGA chip. The chapter is presented in twofold: first, how to configure a circuit's logic elements within the V-FPGA; second, how to configure a circuit's routing structure using a specialised Routing Matrix. The chapter ends with the simulation of a two-bit multiplier using QSim software.

3.1 Introduction

Like the VRC shown in Figure 1.3, the V-FPGA is a second reconfigurable layer residing on top of an FPGA, which takes the form of a two-dimensional array of logic blocks. It only makes use of a partial section of the FPGA.

For clarity, the terminology used to describe a V-FPGA circuit will first be addressed. Each circuit created on the V-FPGA consists of external inputs, logic elements (LEs) and external outputs. The LEs can be configured to carry out any digital-logic function. Depending on how the LEs are routed, i.e. connected to one another, the LEs can form either a combinational or sequential digital circuit. Thus, when creating digital logic using the V-FPGA architecture, the designer needs to define/configure:

1. The function of each LE
2. The way in which the LEs, external inputs and external outputs are routed

The circuit's functionality and routing configuration is stored on the V-FPGA's Logic and Routing Configuration Memories. These memories are implemented using registers.

Because the V-FPGA's configuration memory, style of reconfiguration, granularity and size can be designed exactly according to the requirements of a given application, designers can create an optimised application-specific reconfigurable device. Furthermore, the V-FPGA is described in VHDL and is thus independent of a target platform.

The V-FPGA's VHDL code is obtained by running a C program. The C program outputs a VHDL file which can then be downloaded to any FPGA. In theory, the C code can generate a V-FPGA of any desired size.

The development of the V-FPGA architecture is described further in the sections below.

3.2 The V-FPGA's Logic Elements

The V-FPGA architecture is fine-grained with each LE consisting of two inputs. Each LE models a two-bit lookup table (LUT), which is implemented using a four-to-one multiplexer, and can encode any two-bit Boolean function. Figure 3.1 illustrates how a multiplexer is used to implement an AND-gate. The LE's multiplexer is configured as follows:

- The multiplexer's four input-bits determine the type of gate. For the V-FPGA architecture, there are four one-bit registers—found in the Logic Configuration Memory (LCM)—which are connected to the four multiplexer inputs in order to supply the multiplexer input values. This concept is illustrated in Figure 3.1. Depending on the function to be implemented, the contents of the four one-bit registers is set to 0 or 1. It is set to 1 for all the 1-minterms of a two-variable Boolean function, and to 0 for all 0-minterms.
- The multiplexer's two select-lines form the inputs of the LUT—illustrated using line $LE0\_I\_0$ and $LE0\_I\_1$. Depending on the values of these two inputs, the value from one of the four multiplexer inputs is passed to the LE's output. For example, if both select-line were *high* (binary "11"), then the fourth multiplexer input would be selected, resulting in LE 0's output being *high*. This is the expected value, since a *high* output is produced if both inputs to an AND-gate are *high*.

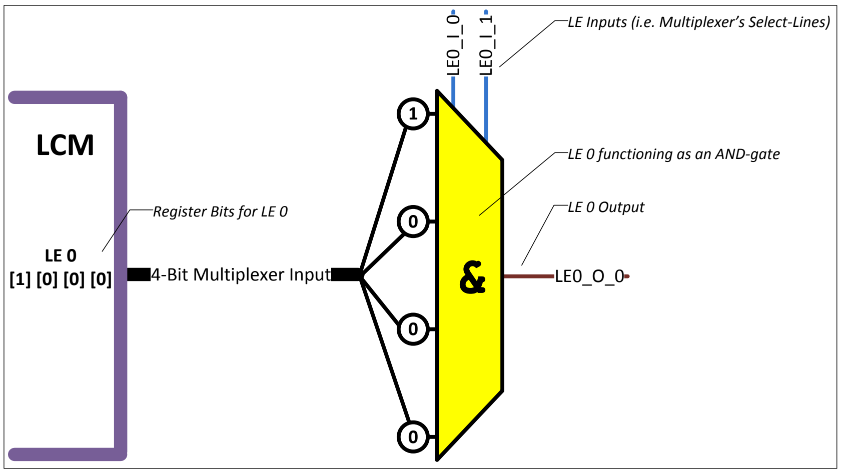
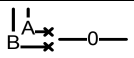
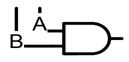
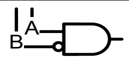
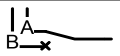
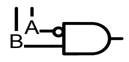
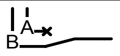
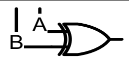
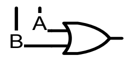
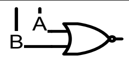
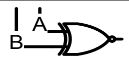
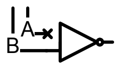
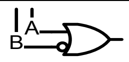
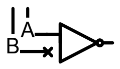
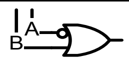
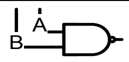
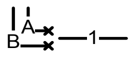


Figure 3.1 AND-gate implementation using a multiplexer

Table 3.1 illustrates all the different two-bit logic gates that can be implemented on the V-FPGA. The binary number next to each gate represents the four-bit multiplexer input needed to implement that particular logic gate. The gates are categorised as fundamental and non-fundamental. The fundamental gates include all the basic functions, namely: AND, NOT, XOR, OR and the wire gates. The gates with bubbled inputs or outputs, except for the NOT-gate, are considered non-fundamental.

Table 3.1 V-FPGA logic-gates

Decimal Value	D_Logic Binary Value				Gate Name	Simplified Boolean Expression	Circuit Diagram	Fundamental Gate?
	$\bar{A}\bar{B}$	$\bar{A}B$	$A\bar{B}$	AB				
0	0	0	0	0	Always Off	0		
1	0	0	0	1	AND	$A.B$		☑
2	0	0	1	0	Negated-Input AND	$A.\bar{B}$		
3	0	0	1	1	A	A		☑
4	0	1	0	0	Negated-Input AND	$\bar{A}.B$		
5	0	1	0	1	B	B		☑
6	0	1	1	0	XOR	$A \oplus B$		☑
7	0	1	1	1	OR	$A + B$		☑
8	1	0	0	0	NOR	$\overline{A + B}$		
9	1	0	0	1	XNOR	$\overline{A \oplus B}$		
10	1	0	1	0	NOT B	$\bar{B}$		☑
11	1	0	1	1	Negated-Input OR	$A + \bar{B}$		
12	1	1	0	0	NOT A	$\bar{A}$		☑
13	1	1	0	1	Negated-Input OR	$\bar{A} + B$		
14	1	1	1	0	NAND	$\overline{A.B}$		
15	1	1	1	1	Always On	1		

Now, consider an array of n LEs, as specified by the designer. Each LE is represented by one LE multiplexer—as shown in Figure 3.2. Hence, for n LEs there are n LE multiplexers.

$$Multiplexers_{LE} = n$$

Equation 3.1 Maximum number of LE multiplexers

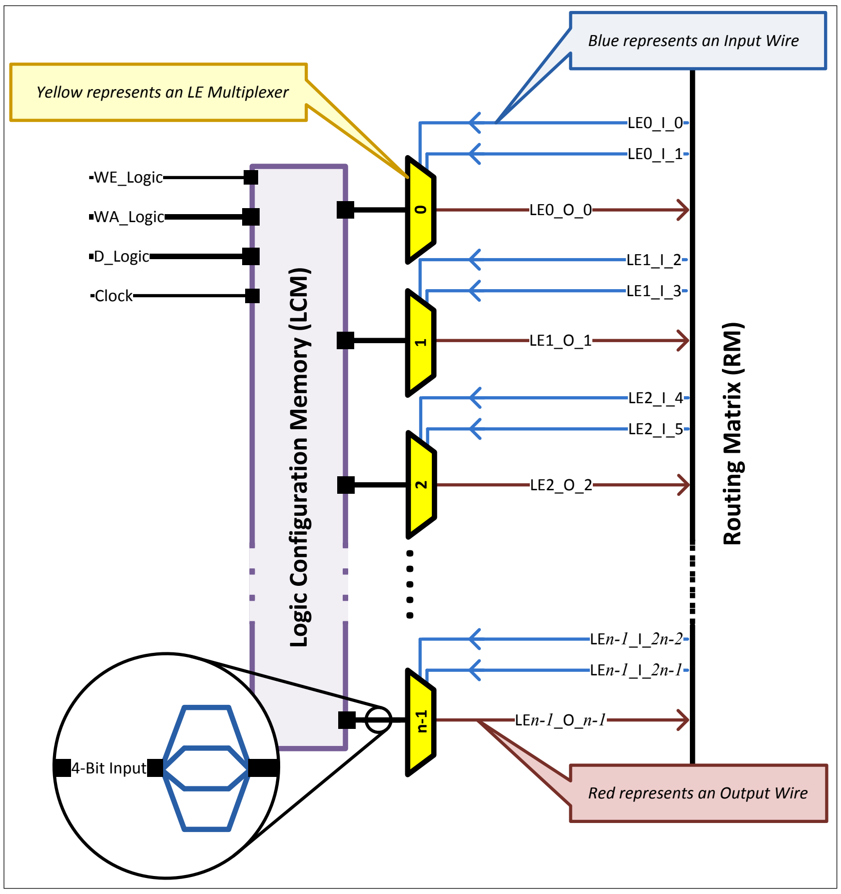


Figure 3.2 Logic Configuration Memory and Routing Matrix connected to the LE multiplexers

The n LEs are addressed from 0 to $n-1$. As previously explained, the four-bit inputs are connected to the LCM, while all n LEs' outputs ($LE0\_O\_0$ to $LEn-1\_O\_n-1$) and select-lines ($LE0\_I\_0$ to $LEn-1\_I\_2n-1$) are connected to the Routing Matrix (RM).

The RM allows the various LEs to connect to one another in order to route a circuit. But, before discussing the RM further (Section 3.3.1), first consider the LCM. Since the LCM stores the information that determines the functionality of each LE, it is important for a designer to be able to load configuration data onto the LCM.

3.2.1 Programming the Logic Configuration Memory

In order to access the LCM, a write-enable line (WE_{Logic}), write-address line (WA_{Logic}), a data line (D_{Logic}) as well as clock line ($Clock$) is provided.

When programming an LE via the LCM, the designer needs to consider:

1. The address of the LE
2. The desired logic function of the LE, i.e. the gate type

Generally, data can be loaded into the LCM by:

1. Setting WE_{Logic} to *high*
2. Specifying the LE address with WA_{Logic}
3. Setting the logic function of the LE with D_{Logic}
4. Clocking the data with a positive-edge of a $Clock$ pulse

WA_{Logic} addresses the LEs from 0 to $n-1$. Thus, $n-1$ represents the maximum value that can be transmitted over the WA_{Logic} line.

D_Logic is the four-bit line used to implement the sixteen different logic functions (as described in Table 3.1). Thus, "1111" (decimal 15) is the maximum value that can be transmitted over D_{Logic}.

Consider the following example: Suppose we wish to use LE 1 in Figure 3.2 to implement a two-input OR-gate. To do this, the following procedure would be followed:

1. Set WE_{Logic} equal to logic "1" (set enable to *true*)
2. Set WA_{Logic} equal to "1" (select LE 1)
3. Set D_{Logic} equal to "0111" (configure the OR-gate)
4. Produce a positive-edge $Clock$ pulse

The above example is illustrated in Figure 3.3. It shows, using a timing diagram, the configuration data being loaded onto the LCM. The LCM, in turn, sends "0111" to multiplexer 1 via its four-bit input, thereby creating the desired OR-gate.

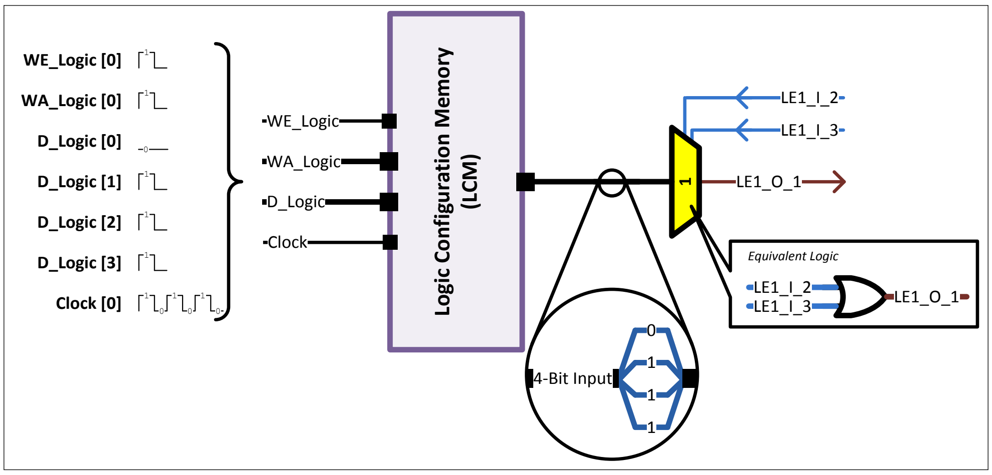


Figure 3.3 An example implementation of an OR-gate using the LCM and multiplexer 1

3.3 The V-FPGA's Routing Architecture

To route a circuit within the V-FPGA, the RM and Routing Configuration Memory (RCM) is used.

3.3.1 The Routing Matrix

The V-FPGA contains an RM (or switch box) for inter-LE communication. Similarly to the LE-multiplexer array, the RM is also implemented using multiplexers. This feature, although expensive in term of space and delays, avoids short circuits that could occur, either when partially reconfiguring the V-FPGA, or during an unconstrained evolution process (Smith, 2010).

The purpose of the RM is to connect an LE's output or an external input to the input of any other LE or external output. In order to connect an LE output or external input to any other element within the V-FPGA, all LE outputs and external inputs are simultaneously connected to the corresponding inputs of every multiplexer within the RM. This concept is illustrated in Figure 3.4, callout A.

The output of each RM multiplexer—see callout B as an example—is in turn connected to either an LE input or an external output. By making use of this matrix routing architecture, each element can be connected to multiple elements within the RM.

The number of multiplexers required in the RM depends on the number of LEs and external outputs. For a V-FPGA with n LEs, the RM will require $2n$ multiplexers to represent the LEs' inputs (since each LE has two inputs). In addition, for O external outputs, the RM will need require O multiplexers. Thus, the total number of RM multiplexers required is defined by:

$$Multipexers_{RM} = 2n + O$$

Equation 3.2 Maximum number of RM multiplexers

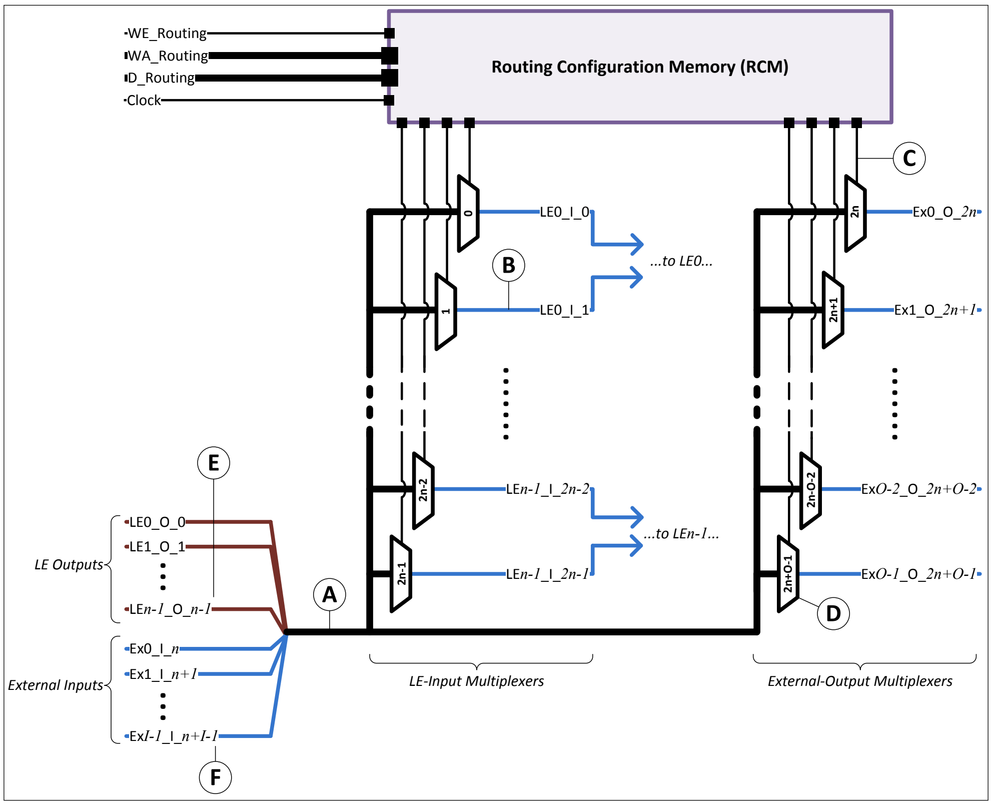


Figure 3.4 The Routing Matrix's multiplexers linked to the Routing Configuration Memory

3.3.2 Programming the Routing Configuration Memory

The RCM controls the routing resources within the RM. The RCM consists of registers connected to the select-lines of the RM multiplexers so as to route a particular input to an output, as shown in Figure 3.4, callout C.

In order to load routing data into the RCM, a write-enable line ($WE_{Routing}$), write-address line ($WA_{Routing}$) as well as a data line ($D_{Routing}$) is provided. The same clock line ($Clock$), used to clock the LCM, is used.

When routing a circuit on the V-FPGA, the $WA_{Routing}$ line defines where the circuit connection is going to. $WA_{Routing}$ addresses either an LE-input or external-output multiplexer in the RM. As described previously, there are $2n + O$ multiplexers, however, since the multiplexers are addressed from 0 (zero), the last multiplexer has address $2n + O - 1$ (see callout D). Thus, $2n + O - 1$ represents the maximum value that can be transmitted over the $WA_{Routing}$ line.

The $D_{Routing}$ line defines where a connection is coming from. $D_{Routing}$ may address either an LE's output or an external input. In Figure 3.4, the LEs' output addresses are numbered from 0 (zero). Thus, if there are n LEs, the last LE's output will have address $n - 1$ (callout E). The external inputs are numbered after the last LE's output. Therefore, the first external input's address will be n, the next external input $n + 1$ and so forth. For a circuit with I external inputs, the last external input will have address $n + I - 1$ (callout F). $n + I - 1$ represents the maximum value that can be transmitted over the $D_{Routing}$ line.

For further clarity, let us consider an example: A connection coming from LE 3 (address $LE3\_O$) and going to external output 1 (address $Ex1\_O$) needs to be setup.

First, it is necessary to know how many LEs, external inputs and external outputs there are in the V-FPGA. Hence, the internal architecture of the V-FPGA must be known in order to generate configuration data. For this example, let's assume there are five LEs, three external inputs and three external outputs available. In order to generate the routing configuration data, the following procedure would be followed:

1. Set $WE_{Routing}$ equal to logic "1" (set enable to *true*)
2. Set $D_{Routing}$ equal to logic "011" (decimal 3). This will select the fourth input of an RM multiplexer, addressed $LE3\_O\_3$.
3. Set $WA_{Routing}$ equal to logic "1011" (decimal 11). This will address the eleventh RM multiplexer ($2 \times 5\ LEs + 2\ external\ outputs - 1$) whose output is linked to external output 1 (address $Ex1\_O\_11$).
4. Produce a positive-edge $Clock$ pulse

Figure 3.5 shows the final routing configuration of the above example. This configuration will route the fourth input of RM multiplexer 11 (i.e. the output of LE 3) to its output. In this case, the multiplexer output is connected to external output 1.

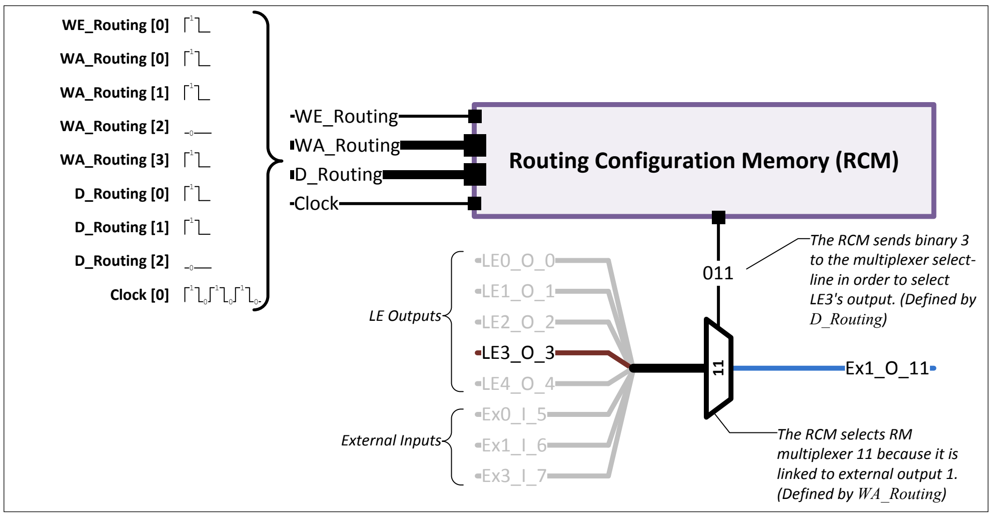


Figure 3.5 Example implementation of a RM multiplexer

3.4 V-FPGA Simulation

This section describes the simulation of a two-bit multiplier circuit using the V-FPGA architecture. The purpose of this simulation is to assess the practical manner in which the logic and routing lines are used to program the LCM and RCM. Also, the following questions are addressed: Can the LCM and RCM be programmed simultaneously? Is it necessary to have the clock running once the V-FPGA is configured? Does the V-FPGA's output correspond to the desired circuit's output for a particular input combination? The answers to these questions will also help validate that the V-FPGA architecture is operating correctly, before commencing with evolution.

Figure 3.6 shows a conventional two-bit multiplier, found in most introductory digital electronics textbooks. The multiplier is implemented using eight LEs, four external inputs and four external outputs. The external inputs are grouped as two two-bit inputs, A and B, and one four-bit output, C. A_0, B_0 and C_0 represent the IO bits' least-significant bit. The desired truth table of the multiplier is shown in Table 3.2.

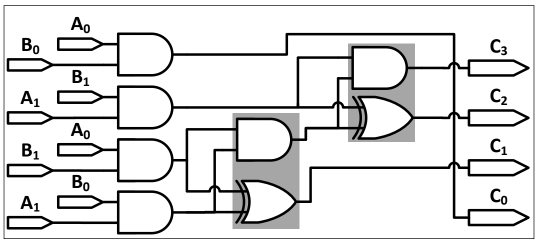


Figure 3.6 A conventional two-bit multiplier

Table 3.2 Two-bit multiplier truth table

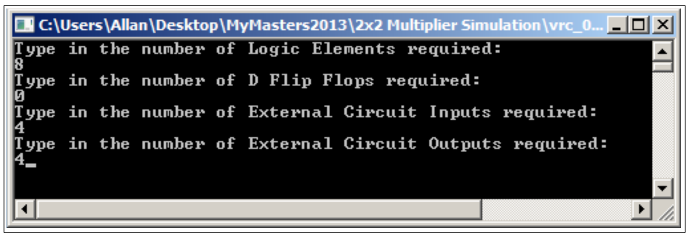
Decimal Value of A	A_1	A_0	Decimal Value of B	B_1	B_0	Decimal Value of C	C_3	C_2	C_1	C_0
0	0	0	0	0	0	0	0	0	0	0

0	0	0	1	0	1	0	0	0	0	0
0	0	0	2	1	0	0	0	0	0	0
0	0	0	3	1	1	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0
1	0	1	1	0	1	1	0	0	0	1
1	0	1	2	1	0	2	0	0	1	0
1	0	1	3	1	1	3	0	0	1	1
2	1	0	0	0	0	0	0	0	0	0
2	1	0	1	0	1	2	0	0	1	0
2	1	0	2	1	0	4	0	1	0	0
2	1	0	3	1	1	5	0	1	1	0
3	1	1	0	0	0	0	0	0	0	0
3	1	1	1	0	1	3	0	0	1	1
3	1	1	2	1	0	6	0	1	1	0
3	1	1	3	1	1	9	1	0	0	1

To simulate the multiplier using the V-FPGA, the entire V-FPGA had to first be coded using C language. Once coded, the compiled C-code then prompted the user to enter four variables, shown in Figure 3.7. Based on the circuit in Figure 3.6, the following variable values were used:

• Number of LEs (n):	8
• Number of D flip-flops:	0
• Number of external inputs (I):	4
• Number of external outputs (O):	4

With these variables, the C-code then generated a unique VHDL-based V-FPGA file, which was the hardware description of the V-FPGA.


C:\Users\Allan\Desktop\MyMasters2013\2x2 Multiplier Simulation\vrc_0...

```
Type in the number of Logic Elements required:
8
Type in the number of D Flip Flops required:
0
Type in the number of External Circuit Inputs required:
4
Type in the number of External Circuit Outputs required:
4_
```

Figure 3.7 C-code variable-prompt which configures the V-FPGA's size

The configured eight-LE V-FPGA used 28 multiplexers, of which eight were LE multiplexers (from Equation 3.1) and twenty were RM multiplexers (from Equation 3.2). Sixteen of the RM multiplexers were needed for the eight LEs, as each LE has two inputs. The remaining four RM multiplexers were used by the four external outputs.

Once the VHDL file was generated, it was compiled using Altera's Quartus II software package. Thereafter, the V-FPGA was ready to be simulated using a Quartus II software add-on, called QSim.

To program the multiplier's configuration bits onto the V-FPGA's LCM and RCM, a QSim timing diagram was used. The logic data was programmed using the D_{logic} and WA_{logic} lines, while the routing data used the $D_{Routing}$ and $WA_{Routing}$ lines. Each line was manually altered in the timing diagram in order to implement the desired configuration bits. However, although done manually in simulation, ultimately during evolution the GA would need to automatically produce these configuration bits.

3.4.1 Simulation Results

The resulting QSim timing analysis is shown in Figure 3.8. All the logic bits were downloaded onto the LCM between callouts A to B. Similarly, for the routing lines, the routing bits were downloaded between callouts A to C. Thus, the logic and routing data could be programmed simultaneously, thereby answering the first question—can the LCM and RCM be programmed simultaneously? Stated differently, the LCM and RCM are independent entities.

The clock was only active between callouts A to C, i.e. when downloading data onto memory. Thus, once the circuit was configured on the V-FPGA, the clock pulse was no longer needed. Hence, the second question—is it necessary to have the clock running once the V-FPGA is configured—has been answered.

The third question—does the V-FPGA's output correspond to the desired circuit's output for a particular input combination—can be addressed by making a comparison between the simulation output (between callouts C to D) and the truth table from Table 3.2. All the external-input combinations were simulated on lines "Ex_In_A0A1" and "Ex_In_B0B1". The corresponding output lines, "External_Output", showed that the multiplier correctly computed the binary products.

In closing, since a designer can swiftly implement any digital logic circuit on the V-FPGA (and indirectly on an FPGA) by simply manipulating the V-FPGA's configuration bits, the above findings confirm that the V-FPGA architecture does provide a viable platform to facilitate and perform hardware evolution.

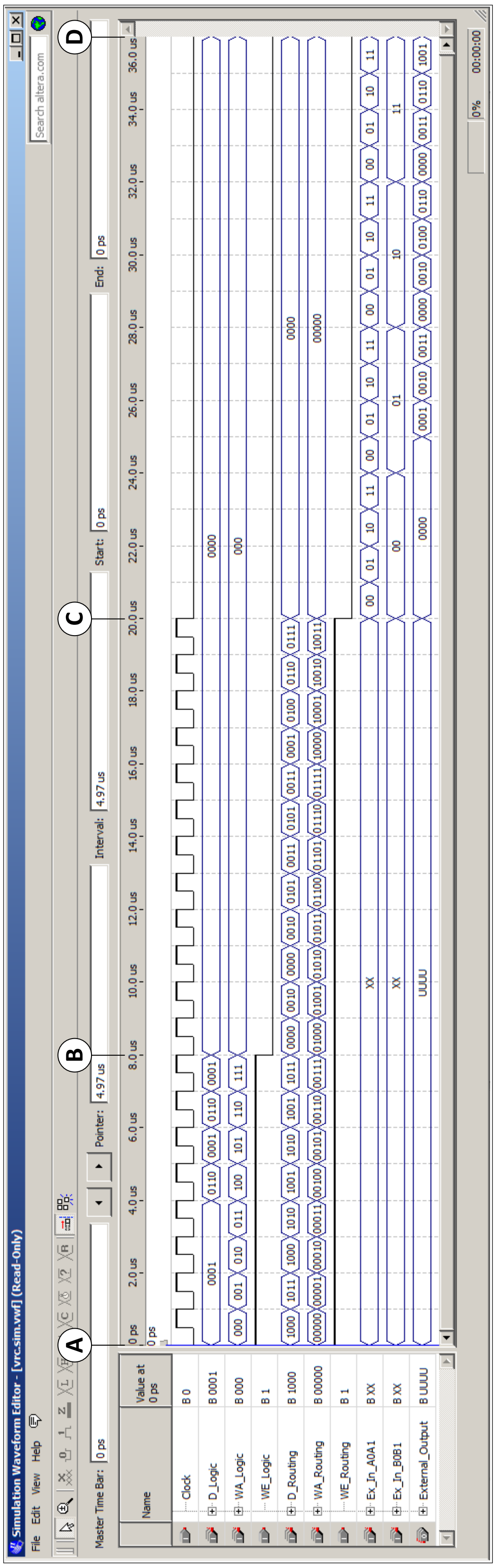


Figure 3.8 Simulation results

Chapter 4

EVOLVING A TWO-BIT MULTIPLIER CIRCUIT

This chapter demonstrates the evolution of a two-bit multiplier using the V-FPGA architecture. The chapter starts by describing the experimental hardware setup. Part of the setup includes configuring and compiling the V-FPGA on an FPGA chip, and connecting interfacing hardware to a PC. Once the setup is complete, the effectiveness of two different GAs, namely the canonical and $1+\lambda$ GA, is investigated. Finally, the chapter closes with an account of three evolved phenotypes, as well as a discussion on the arising evolution difficulties, the possible causes and solutions.

4.1 Introduction

A combinational-multiplier circuit is an electronic circuit that computes the product of two unsigned binary numbers. Multipliers are often useful for computing mathematical instruction-sets in PCs' arithmetic-logic units. Most multiplier circuits use the scheme of first computing the inputs' partial products, and then summing the partial products to form the final product. The summing is done using adders, as shown in Figure 4.1.

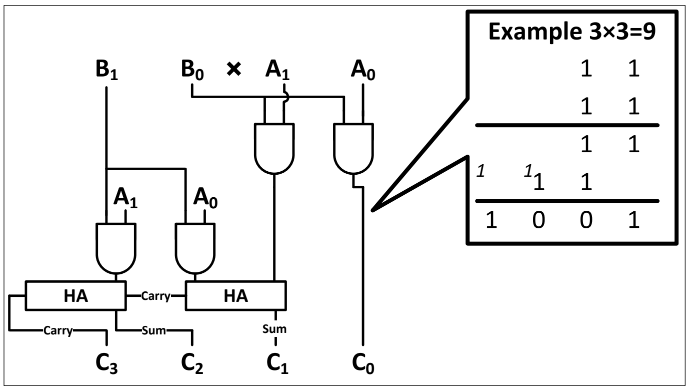


Figure 4.1 A two-bit multiplier using half-adders (HA) to sum the partial products

The complexity of a multiplier's circuit increases exponentially with an increase in the number of output bits. For example, a conventional two-bit multiplier may use approximately eight LEs configured to implement two half-adders, while a four-bit multiplier may use up to 64 LEs configured to use four half- and eight full-adders (Katz, 1993). Considering the above complexity, the simple two-bit multiplier circuit is a favourable initial circuit to evolve since it small enough to demonstrate EHW while still being a practical sub-circuit for many digital ICs.

The rest of this chapter describes the implementation of the software and hardware needed to evolve a two-bit multiplier. This includes setting up the hardware components, compiling and downloading the V-FPGA, and programming the GAs.

There are two EHW GAs demonstrated in this chapter. The first GA variant, referred to as a canonical GA, uses the tournament-selection, uniform-crossover and mutation genetic operators. The outline of this GA was first described by Holland (1975), who played a vital role in popularising GAs. It is interesting to note that initial studies on canonical GAs did not value mutation, and thus mutation did not often feature (Engelbrecht, 2007). This is in direct contrast to the second GA variant, referred to as a $1 + \lambda$ GA.

The $1 + \lambda$ GA relies only on the mutation operator. It makes use of $(\mu + \lambda)$-selection, where μ represents the number of parents and λ the number of offspring. For example, a $1 + \lambda$ GA uses a single parent that is mutated λ times until a new generation is computed. This GA variant does not make use of tournament selection or crossover.

As mentioned in the Chapter 1, the literature claims that $1 + \lambda$ GAs—GAs without crossover—are more efficient for EHW problems. This chapter will investigate this claim by evolving a two-bit multiplier using both GAs. However, before any GA can be executed or discussed, the experimental hardware and V-FPGA needs to be examined.

4.2 Hardware and V-FPGA Setup

4.2.1 Hardware Components

Figure 4.2 shows the EHW experimental setup[4], which used three hardware components:

1. A PC running the GAs using LabVIEW software
2. A National Instruments (NI) data-acquisition card (DAQ card)
3. An Altera DE2 development-and-education board running the V-FPGA

[4] For a more detailed experimental-setup schematic, the reader is referred to the A3-sized page entitled "APPENDIX A: Electrical Schematic of Complete EHW System" found in the Appendix.

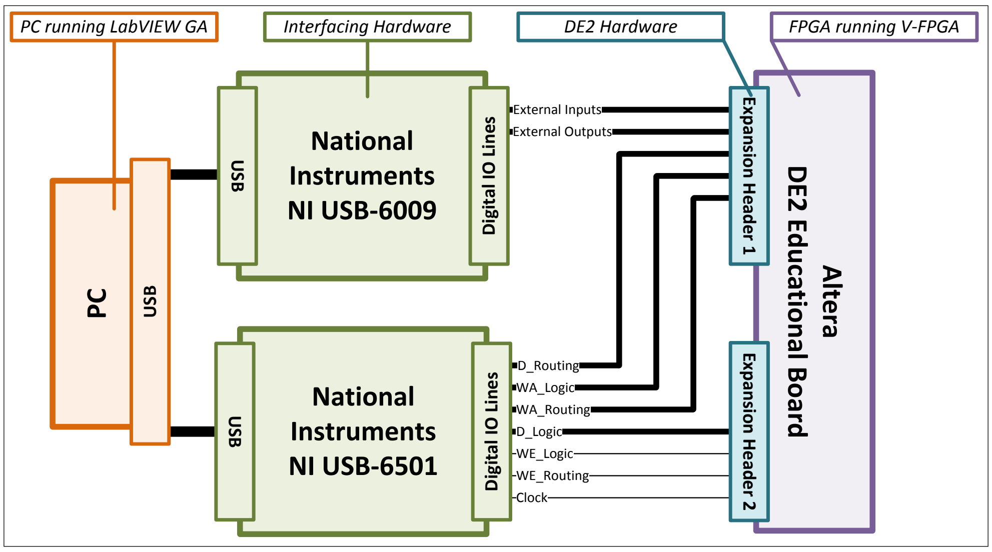


Figure 4.2 Schematic of the hardware components used in the experimental setup

4.2.1.1 PC running the LabVIEW Genetic Algorithm

The LabVIEW software, in which the GAs were programmed, was developed by NI specifically to ease engineering prototyping. LabVIEW makes use of a graphical programming interface which helps with debugging and allows for uncomplicated data acquisition.

LabVIEW programs and subroutines are called virtual instruments (VIs), and are saved using the *.vi* file extension. Each VI program makes use of a front panel and a back-panel diagram. The front panel uses an array of controls and indicators; and is used purely to display or indicate results. The back panel, presented in a block-diagram format, is where the user programs the source code of the VI.

An important aspect of VIs is that they can operate independently, or run as sub-routines on other VIs. Hence, LabVIEW can implement VIs in a hierarchal structure.

4.2.1.2 National Instruments' Data Acquisition Cards

The NI DAQ cards have "plug-and-play USB connectivity" and are "simple enough for home or academic applications, but robust and versatile enough for laboratory or industrial applications." (National Instruments, 2012) The cards have been specifically designed to integrate with LabVIEW software.

In the experimental setup, shown in Figure 4.3, the DAQ cards were used as the interfacing hardware, allowing the PC to access the FPGA's pins via the DAQ cards' digital IO lines.

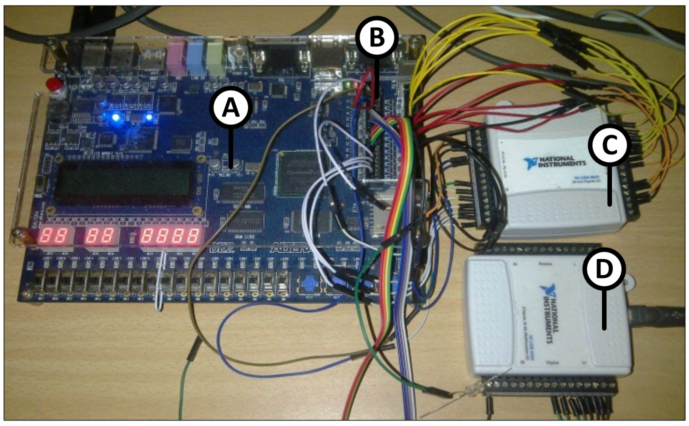


Figure 4.3 Actual hardware setup

For the two-bit multiplier evolved in this study, a total of 31 FPGA pins were used. (The derivation of these pins is shown later in Table 4.1.) Since each pin required one DAQ digital-IO line, a total of 31 IO lines were needed. This presented a problem, as the available DAQ cards had only 24 IO lines. To solve this, two DAQ cards were used:

1. The NI USB-6501, in Figure 4.3 (callout C) and Figure 4.4, is a 24-line bidirectional digital-IO card. Of the 24 available lines, 23 lines were used as digital outputs.
2. The NI USB-6009, in Figure 4.3 (callout D), is an 8-input, 14-bit multifunction IO card. It has 12 bidirectional digital-IO channels. Four lines were configured as digital outputs, connected to the FPGA's external-input pins; and four lines were configured as digital inputs, connected to the FPGA's external-output pins.

Figure 4.4 NI USB-6501 DAQ card (National Instruments, 2012)

4.2.1.3 Altera DE2 Development-and-Education Board

The Altera DE2 FPGA board, shown in Figure 4.3 (callout A), was launched in 2006, and is described by Altera (2008) as a "board specifically designed for education." The DE2 makes use of, amongst other components, an Altera Cyclone II 2C35 FPGA chip, a USB host/slave controller and two 40-pin expansion headers. Figure 4.5 shows the layout of the board with labelled key components.

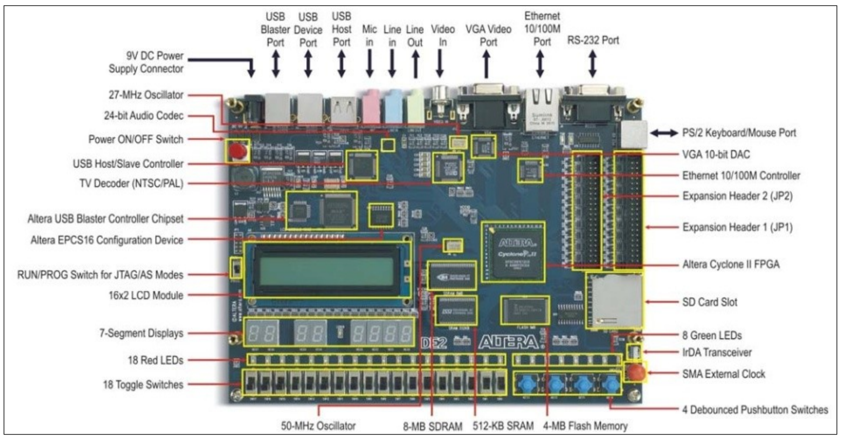


Figure 4.5 The DE2 Board (Altera, 2006)

The 2C35 FPGA chip makes use of 33 216 logic elements and 475 user IO-pins. Of the 475 pins, 72 are connected to the two 40-pin expansion headers. The expansion headers also provides two V_{CC} 5-volt pins, two V_{CC} 3.3-volt pins and four GND ground pins (Altera, 2006). (Hence, the expansion headers have a total of 80 available pins.) These header pins, when connected to the DAQ cards' IO ports, provided a channel for LabVIEW to communicate to the V-FPGA.

In order to use the DE2 board, the user needs to be familiar with Altera's Quartus II software. This software, as used in Section 3.4 (V-FPGA Simulation, page 30), allows the user to compile and execute the hardware description code on the DE2 board. The board is programmed via the on-board USB blaster.

4.2.2 Creating a 20-LE VHDL-Based V-FPGA File

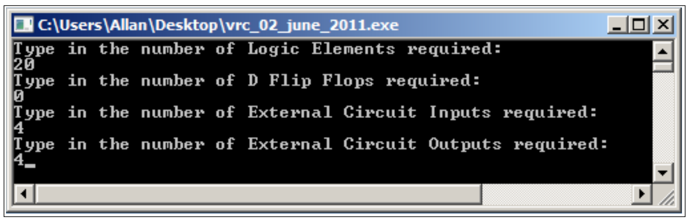
Theoretically, when generating a V-FPGA, the C-code can generate a V-FPGA of any desired size, i.e. any number of LEs and IOs (Smith, 2010). However, the size can also significantly influence the GAs effectiveness in finding a solution. If too few LEs are selected, the desired phenotype will never fully evolve since the solution circuit requires more LEs than the number of available LEs, i.e. the search space is too small and does not include a solution. On the contrary, creating a V-FPGA with too many LEs increases the search space, thereby decreasing the probability of the GA finding a solution. The above two problems are directly related to scalability, where the probability of finding a solution greatly deteriorates if the selected search space is too small or large.

Looking at past research on evolved multipliers, Vassilev et al. (1999) made use of a 16-LE VRC, while Miller & Thomson (1998) used a 21-LE VRC. Using these figures as guidelines, the V-FPGA was configured to use 20 LEs.

As done in Section 3.4 (V-FPGA Simulation, page 30) and shown in Figure 4.6, a new VHDL-based V-FPGA file was configured using the following parameters:

- Number of LEs (n): 20

- Number of D flip-flops: 0
- Number of external inputs (I): 4
- Number of external outputs (O): 4



```
C:\Users\Allan\Desktop\vrc_02_june_2011.exe
Type in the number of Logic Elements required:
20
Type in the number of D Flip Flops required:
0
Type in the number of External Circuit Inputs required:
4
Type in the number of External Circuit Outputs required:
4_
```

Figure 4.6 C-code variable-prompt used to configure a 20-LE V-FPGA

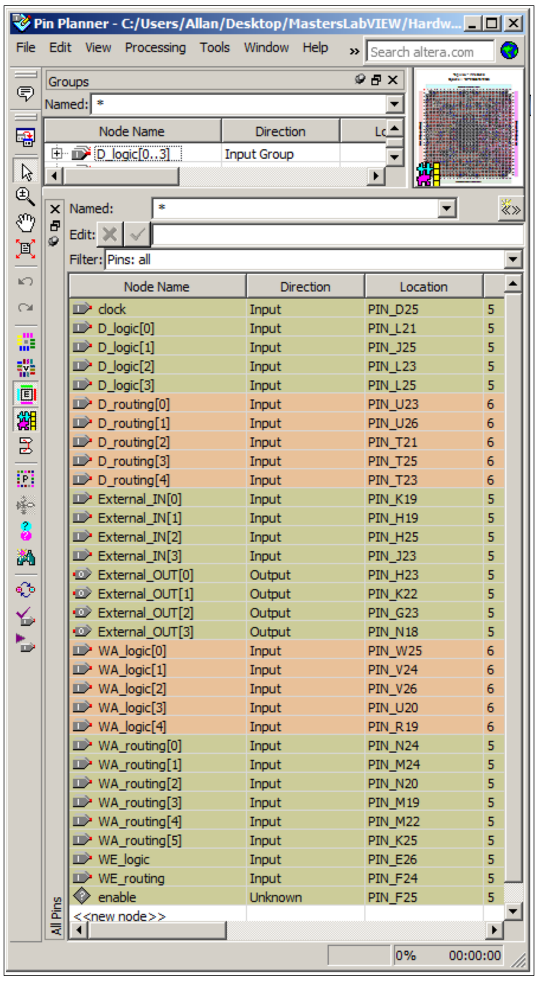
4.2.2.1 Compiling the V-FPGA

An important aspect of the hardware setup was assigning every FPGA pin used by the V-FPGA to a particular header pin on DE2's expansion headers. Table 4.1 shows the pin requirements of the 20-LE V-FPGA. A total of 31 pins were assigned.

Table 4.1 Number of pins used

	Maximum Decimal Value	Reference	Corresponding Binary Value	Number of Pins
D_{Logic}	15	Table 3.1	1111	4
WA_{Logic}	$n-1$ $20-1 = 19$	Section 3.2.1	10011	5
WE_{Logic}	1		1	1
$D_{Routing}$	$n+I-1$ $20+4-1 = 23$	Section3.3.2	10111	5
$WA_{Routing}$	$2n+O-1$ $40+4-1 = 43$	Section 3.3.2	101011	6
$WE_{Routing}$	1		1	1
External Input	15		1111	4
External Output	15		1111	4
Clock	1		1	1
				31

The pins were assigned using the Quartus II Pin Planner software, as shown in Figure 4.7. Pins belonging to the same logic- or routing-configuration lines were generally grouped next each other on the expansion headers to allow for easier troubleshooting.

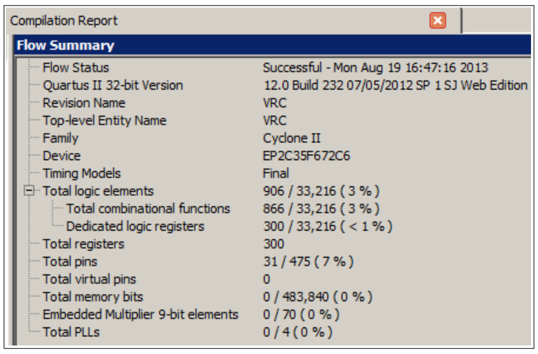

Pin Planner - C:/Users/Allan/Desktop/MastersLabVIEW/Hardw...
File Edit View Processing Tools Window Help
Search altera.com
Groups
Named: *
Named: *
Edit:
Filter: Pins: all
All Pins
0%
00:00:00

Node Name	Direction	Lc
D_logic[0..3]	Input Group	

Node Name	Direction	Location	
clock	Input	PIN_D25	5
D_logic[0]	Input	PIN_L21	5
D_logic[1]	Input	PIN_J25	5
D_logic[2]	Input	PIN_L23	5
D_logic[3]	Input	PIN_L25	5
D_routing[0]	Input	PIN_U23	6
D_routing[1]	Input	PIN_U26	6
D_routing[2]	Input	PIN_T21	6
D_routing[3]	Input	PIN_T25	6
D_routing[4]	Input	PIN_T23	6
External_IN[0]	Input	PIN_K19	5
External_IN[1]	Input	PIN_H19	5
External_IN[2]	Input	PIN_H25	5
External_IN[3]	Input	PIN_J23	5
External_OUT[0]	Output	PIN_H23	5
External_OUT[1]	Output	PIN_K22	5
External_OUT[2]	Output	PIN_G23	5
External_OUT[3]	Output	PIN_N18	5
WA_logic[0]	Input	PIN_W25	6
WA_logic[1]	Input	PIN_V24	6
WA_logic[2]	Input	PIN_V26	6
WA_logic[3]	Input	PIN_U20	6
WA_logic[4]	Input	PIN_R19	6
WA_routing[0]	Input	PIN_N24	5
WA_routing[1]	Input	PIN_M24	5
WA_routing[2]	Input	PIN_N20	5
WA_routing[3]	Input	PIN_M19	5
WA_routing[4]	Input	PIN_M22	5
WA_routing[5]	Input	PIN_K25	5
WE_logic	Input	PIN_E26	5
WE_routing	Input	PIN_F24	5
enable	Unknown	PIN_F25	5
<<new node>>			

Figure 4.7 Quartus II Pin Planner screen dump

The pin assignment was followed by the V-FPGA compilation, which was also done using Quartus II. An interesting observation, noted during the compilation, was that the V-FPGA used a conservative amount of FPGA resources. Figure 4.8 shows, of the 33 216 available LEs on the FPGA, only 906 were used by the V-FPGA—or approximately 3%. Furthermore, approximately 7% of the available pins were used.



Flow Status	Successful - Mon Aug 19 16:47:16 2013
Quartus II 32-bit Version	12.0 Build 232 07/05/2012 SP 1 SJ Web Edition
Revision Name	VRC
Top-level Entity Name	VRC
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	906 / 33,216 (3 %)
Total combinational functions	866 / 33,216 (3 %)
Dedicated logic registers	300 / 33,216 (< 1 %)
Total registers	300
Total pins	31 / 475 (7 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Figure 4.8 Compilation report

4.3 Implementing the LabVIEW Genetic Algorithms

So far, the hardware and V-FPGA setup has been examined. We now turn our attention to the system's software requirements.

4.3.1 The Canonical Genetic Algorithm

The LabVIEW algorithm[5] consisted of a main routine, called *MainEHW*, which called eight smaller subroutines. Example subroutine files included: *D&T.vi*, *NoDoubleInputs.vi*, *RemoveOutputDups.vi*, *TourSelect.vi*, *Mutation.vi*, *Crossover.vi*, *FindParents.vi* and *CreateCircuits.vi*. In addition, the *D&T.vi*—or the download-and-test VI—called the *FitnessTest.vi* and the *Download.vi* files. Figure 4.9 shows the hierarchical structure of the GA's VIs.

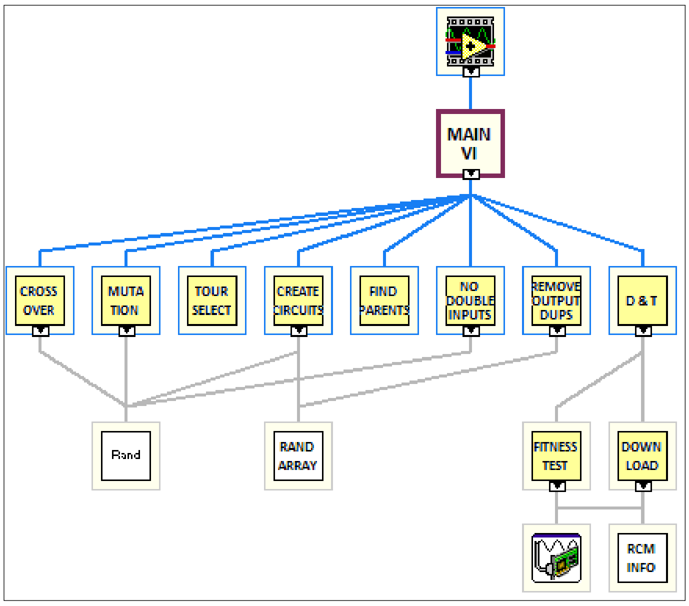


Figure 4.9 Hierarchical structure of the LabVIEW VIs

Using the above VIs, Figure 4.10 shows a flowchart overview of the canonical LabVIEW GA.

[5] Refer to Appendix D for the LabVIEW code of the selected VIs.

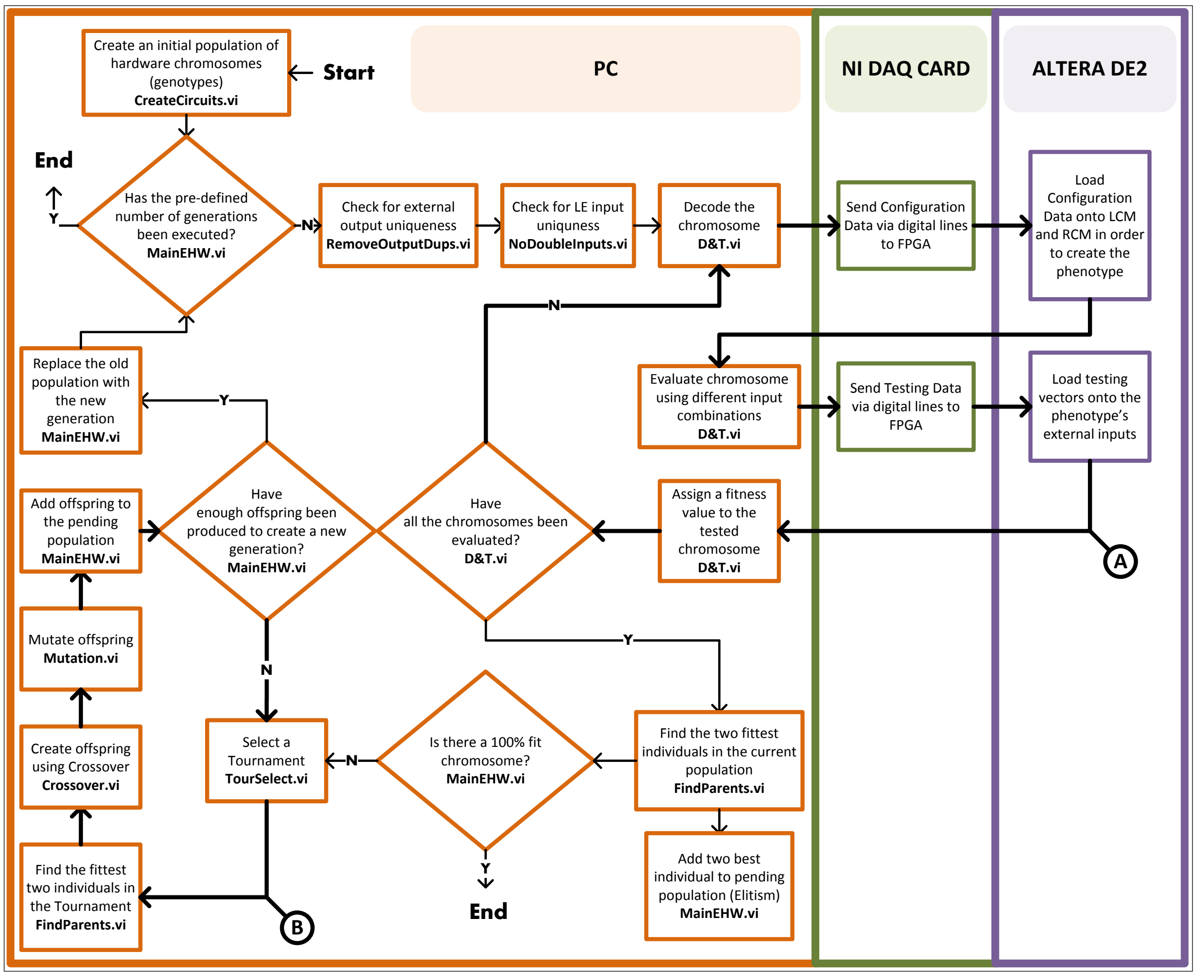

Create an initial population of hardware chromosomes (genotypes) **CreateCircuits.vi**
Start
PC
NI DAQ CARD
ALTERA DE2
End
Y
Has the pre-defined number of generations been executed? **MainEHW.vi**
N
Check for external output uniqueness **RemoveOutputDups.vi**
Check for LE input uniquness **NoDoubleInputs.vi**
Decode the chromosome **D&T.vi**
Send Configuration Data via digital lines to FPGA
Load Configuration Data onto LCM and RCM in order to create the phenotype
N
Replace the old population with the new generation **MainEHW.vi**
Y
Evaluate chromosome using different input combinations **D&T.vi**
Send Testing Data via digital lines to FPGA
Load testing vectors onto the phenotype's external inputs
Add offspring to the pending population **MainEHW.vi**
Have enough offspring been produced to create a new generation? **MainEHW.vi**
Have all the chromosomes been evaluated? **D&T.vi**
Assign a fitness value to the tested chromosome **D&T.vi**
A
Mutate offspring **Mutation.vi**
N
Y
Create offspring using Crossover **Crossover.vi**
Select a Tournament **TourSelect.vi**
N
Is there a 100% fit chromosome? **MainEHW.vi**
Find the two fittest individuals in the current population **FindParents.vi**
Find the fittest two individuals in the Tournament **FindParents.vi**
Y
B
End
Add two best individual to pending population (Elitism) **MainEHW.vi**

Figure 4.10 Flowchart of the LabVIEW GA with the applicable VIs highlighted using bold text

The algorithm is summarised as follows:

1. Create a random six-individual population of chromosomes using the *CreateCircuits.vi* VI.
2. Has the predefined number of generations been met? If yes, stop the algorithm.
3. Check the validity of each chromosome. (These constraints are discussed later in Section 4.3.5)
4. Execute loop A in Figure 4.10 using the download-and-test (*D&T.vi*) VI:
    a. Decode each chromosome and load each phenotype onto the V-FPGA
    b. Test the loaded phenotype by applying all possible input-combinations on the phenotype's external inputs
    c. Using the fitness function, assign a fitness value to each phenotype
5. Find the best parent using the *FindParents.vi* VI and directly copy this parent into the next generation (elitism).
6. Is there a 100%-fit individual? If yes, stop the algorithm.
7. Execute loop B in Figure 4.10:

a. Create a new generation using the GA's genetic operators, namely: tournament selection, crossover and mutation. Use the *TourSelect.vi*, *Crossover.vi*, *Mutation.vi* and *FindParents.vi* VIs.

b. When enough offspring have been created for a new generation, return to Step 2.

4.3.2 The $1 + \lambda$ Genetic Algorithm

In order to optimise the canonical GA, three different techniques, implemented successfully by Sekanina & Freidl (2005), were investigated: the $(\mu + \lambda)$-selection method; the limiting of the LEs' functionality and the optimisation of the fitness function. The LEs' functionality (Section 4.3.5) and the optimisation of the fitness function (Section 4.3.8) are discussed later.

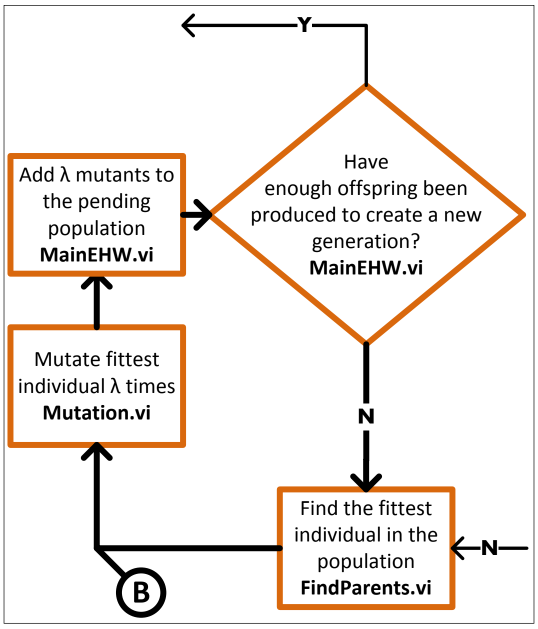

Y
Add λ mutants to the pending population MainEHW.vi
Have enough offspring been produced to create a new generation? MainEHW.vi
Mutate fittest individual λ times Mutation.vi
N
Find the fittest individual in the population FindParents.vi
N
B

Figure 4.11 $1 + \lambda$ mutation loop

To implement the $(\mu + \lambda)$-selection method, only loop B in Figure 4.10 was replaced with the loop shown in Figure 4.11. The $1 + \lambda$ GA makes use of one parent that is mutated five times. The $1 + \lambda$ loop's algorithm is summarised as follows:

1. Execute mutation loop B in Figure 4.11:
   a. Find the fittest individual using the *FindParents.vi* VI.
   b. Create λ mutants, where $\lambda = 5$. Use the *Mutation.vi* VI.
   c. When enough mutants have been created for a new generation, return to Step 2 on page 43.

4.3.3 Genetic Operators

This section explains the tournament selection (*TourSelect.vi*), crossover (*Crossover.vi*), mutation (*Mutation.vi*) and elitism (*MainEHW.vi*) genetic operators[6], shown in Loop B of Figure 4.10. The same mutation VI is also used in the $1 + \lambda$ loop shown in Figure 4.11.

4.3.3.1 Tournament-Selection Virtual Instrument

Tournament selection, as used by Miller & Thomson (2000), selects a tournament (or group) of n individuals randomly from the population, where $n < p$ and p is the total number of individuals in the population (Engelbrecht, 2007). The performance of all n individuals is compared, and the best two individuals from the tournament—called the parents—are selected for crossover.

The advantage of tournament selection, provided n is not too large, is that it prevents the best individuals in a population from dominating the evolutionary process, thus allowing weaker individuals to add diversity to the population (Engelbrecht, 2007).

In the case of the canonical GA, the tournament was set to 50% of the total population. (Thus $p = 6$ and $n = 3$.)

4.3.3.2 Crossover Virtual Instrument

$1 + \lambda$ GAs make use of asexual reproduction, where offspring are generated from one parent (Engelbrecht, 2007). On the contrary, in sexual crossover, as used in the canonical GA's *Crossover.vi* VI, two parents are used to produce one or two offspring (Engelbrecht, 2007). These parents reproduce by swapping random genes, or by combining the genes. The probability of the genes being swapped/combined is controlled by the crossover rate.

A dynamic crossover, which depends on the weaker parent's fitness, was used. For example, if parent 1 had a fitness of 46% and parent 2 49%, parent 1 will cause a crossover rate of $100 - 46 \; = \; 54\%$. This means that 54% of the genes will be swapped between the two parents.

The dynamic crossover allowed for a larger exchange of genetic data when the parents are weak, thereby allowing the GA to explore a larger search space (global search). However, as the parents' fitness values improved, the crossover rate is reduced in order to refine the GA search (local search).

4.3.3.3 Mutation Virtual Instrument

During mutation, the *Mutation.vi* VI simply replaced random genes with a random integer value that satisfied the configuration criteria discussed later in Section 4.3.5 .

[6] See Appendix D for the genetic operators' LabVIEW code.

The probability of mutation is determined by the mutation rate. For the canonical GA, a dynamic mutation rate was used. The dynamic mutation rate was determined by the weaker parent's fitness. However, the mutation rate was restricted to the range of 0 to 50%. For example, a weak parent, with the fitness of 46%, will yield a mutation rate of 50%, and not $100 - 46 = 54\%$.

Since the $1 + \lambda$ GA only used one parent, a constant mutation rate of 20% was chosen after several trial runs.

4.3.3.4 Elitism Virtual Instrument

Elitism is an important operator that ensures the best individuals from each generation survives. The best individuals are copied to the new generation without being modified by crossover or mutation.

Elitism should be used with care. The more individuals that are copied, the less diversity the new generations will have.

Due to the nature of the $1 + \lambda$ GA, only one elite individual was retained. This elite individual was used to produce the λ mutants. For the canonical GA, two elite individuals were retained.

4.3.4 Genotype-Phenotype Mapping

First, the terminology used in this chapter needs clarification. In this chapter, *genotype* and *chromosome* are used synonymously, while the term *phenotype* is used to describe the circuit on the FPGA represented by the genotype. When converting a circuit's genotype into the corresponding phenotype by decoding the genotype's genes, the process is known *genotype-phenotype mapping*.

Now, although there are many variations of GAs, all GAs make use of chromosomes—in this case, a chromosome describing the desired hardware circuit. The hardware chromosome carries all the necessary $WA_{Routing}$, $D_{Routing}$, WA_{Logic} and D_{Logic} data in order for any desired genotype to be mapped into a phenotype. For illustrative purposes, an example chromosome is represented using an integer column-vector in Table 4.2 (shaded grey).

Table 4.2 An example hardware-chromosome (shaded grey)

	Gene Number	**Chromosome**	$WA_{Routing}$	$D_{Routing}$	WA_{Logic}	D_{Logic}
Routing Data used to program the RCM	0	**1**	0	1		
	1	**6**	1	6		
	2	**2**	2	2		
	…	**…**	…	…		
	$2n + O - 2$	**9**	$2n + O - 2$	9		
	$2n + O - 1$	**2**	$2n + O - 1$	2		

Logic Data used to program the LCM	$2n + O$	**1**	0	1
	$2n + O + 1$	**8**	1	8
	$2n + O + 2$	**5**	2	5
	…	**…**	…	…
	$3n + O - 2$	**9**	$n - 2$	9
	$3n + O - 1$	**7**	$n - 1$	7

Each gene in the column-vector chromosome carries two pieces of data—the index of the element and the value of the element.

Consider the first section of the chromosome, which represents the phenotype's routing data. The index of the element represents the $WA_{Routing}$ data, and the value represents the $D_{Routing}$ data. The second section of chromosome represents the logic data, which defines the LEs. The index of the element represents the WA_{Logic} data, and the value represents the D_{Logic} data.

Now, consider an example. Gene 0 in Table 4.2 (highlighted yellow), when downloaded onto the RCM, will cause binary zero to be transmitted over the $WA_{Routing}$ line and binary one over the $D_{Routing}$ line. Similarly, gene $3n + O - 2$ (highlighted green) will transmit the binary number represented by $n - 2$ over the WA_{Logic} line and binary nine over the D_{Logic} line to the LCM.

Since $3n + O - 1$ represents the last gene, the multiplier chromosomes consist of a total of $3n + O = 3(20) + 4 = 64$ genes, of which $2n + O = 2(20) + 4 = 44$ are routing, and $n = 20$ are logic. Thus, the genotype is represented in LabVIEW software using a 64-integer array.

A vital advantage of the devised chromosome representation in Table 4.2 is that it does not allow illegal configurations. Illegal configurations occur when two LE outputs or external inputs are connected to a single LE input or an external output, as shown in Figure 4.12. These configurations can create shorts, thereby physically damaging the FPGA chip.

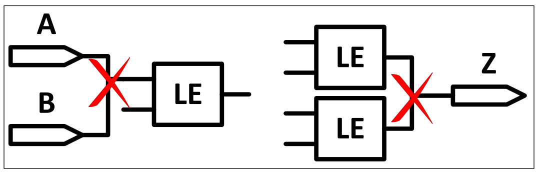


Figure 4.12 Illegal configurations

The chromosome innately prevents illegal configurations by making each gene's $WA_{Routing}$ data unique. Remember, the $WA_{Routing}$ data represents where a routing connection is going to, while $D_{Routing}$ shows where a connection is coming from. Thus, each LE input or external output is unique and can only have one connection, whereas each LE output or external input can be connected to many different LE inputs or external outputs.

4.3.5 Evolution Constraints

To encourage efficient and fast evolution, certain constraints were imposed on the evolved phenotypes. These constraints decreased the GA's search space, thereby refining the search and minimising scalability.

4.3.5.1 The Search Space

To put in perspective the size of the GA's search space, consider the following hardware-chromosome permutations[7], which are based on the 64-gene chromosomes described in Section 4.3.4 :

- For the routing of the external inputs and LEs, there are forty LE inputs (two inputs per LE) which can be connected to one of any of the other twenty LEs' outputs, or any of the four external inputs. Thus, there are $24^{40} \approx 1.62 \times 10^{55}$ possible LE permutations.
- For the routing of the external outputs, there are four external outputs which can be connected to the output of any of the twenty LEs, or any of the four external inputs. Thus, there are $24^{4} \approx 331 \times 10^{3}$ possible external-output permutations.
- For the logic section of a chromosome, there are twenty LEs, each of which can be configured into one of 16 different logic functions. Thus, there are $16^{20} \approx 1.21 \times 10^{24}$ possible logic permutations.

The above three permutations clearly highlight the vast scale of possible solutions (or search space) that the GA has to explore. This is what makes GA algorithms unique—their ability to find solutions strategically and systematically in an almost infinitely large search space.

However, GAs can be aided in finding solutions faster if these large search spaces are minimised. There are two strategies for reducing a GA's search space—reduce the V-FPGA's size or impose evolution constraints.

Notice, in the above analysis, that the number of external IOs and LEs has a direct influence on the number of permutations and size of the search space. This reconfirms that the V-FPGA's size is critical—it is important to choose an adequate, but not large, V-FPGA size. If too large, the GA runs the risk of not finding a solution in a reasonable amount of time.

Evolution constraints can reduce the search space by lowering the possible routing and functionality permutations. For example, by limiting the LEs' functionality to seven gates and not 16, the logic permutation can be reduced to $7^{20} = 8 \times 10^{16}$ possibilities. This simple constraint reduces the number of possible logic permutations by over $\frac{16^{20}-7^{20}}{16^{20}} \approx 99\%$.

4.3.5.2 Reducing the Search Space

[7] See "Appendix E: Search-Space Permutations" for an in-depth explanation.

In light of the above discussion, each circuit phenotype was routinely checked for the following configurations, which were not permitted during evolution:

1. **Each LE input had to be unique, i.e. an LE could not have the same two inputs**
   Having identical inputs simply changes the LE's gate-type. However, since every two-bit Boolean function can be represented by an LE's multiplexer, changing the LE's gate-type by having identical inputs is unnecessary. For example, a NAND-gate using two identical inputs acts as a NOT-gate. If a NOT-gate is required for that particular LE, the LE should be directly configured as a NOT-gate and not as an identical-input NAND-gate. Hence, the identical-input NAND-gate is unnecessary.
   The *NoDoubleInputs.vi* VI in Figure 4.10 was used in the GA to validate each LE input.
2. **Each LE's function was limited**
   The LEs' functionality was limited to the seven fundamental gates, shown in Table 3.1 on page 25. All the non-fundamental gates from Table 3.1 can be created using a combination of fundamental gates. Thus, the main advantage of limiting the LEs' functionality is to reduce the GA's search space.
3. **An external input could not be directly connected to an external output**
   If connected directly, all the LEs would be bypassed.
4. **Each external output had to be unique**
   None of the external outputs could be connected to the same LE. This would create duplicate external outputs.
   Figure 4.10's *RemoveOutputDups.vi* VI was used to validate each external output's uniqueness.
5. **No feedback loops were allowed, i.e. only feed-forward circuits were allowed**
   Feedback loops can create memory elements within circuits. This causes instability, thereby creating unreliable fitness values. For example, a feedback-loop circuit, producing a fitness of $x\%$ during one evaluation, may produce a completely different fitness value when evaluated again. This unstable circuit will cause inevitable genetic problems, since there is a high probability of the feedback loops being passed to the offspring.

   An LE array of five rows by four columns, shown in Figure 4.13, which draws similarities from Cartesian Genetic Programming (CGP), was used to prevent feedback loops (Miller & Thomson, 2000). The CGP array of LEs works as follows: An LE's output can only be connected to an external output or the input of another LE which is in a following column. If an LE's output is connected to a preceding column's LE input, there is a risk of creating a feedback loop.

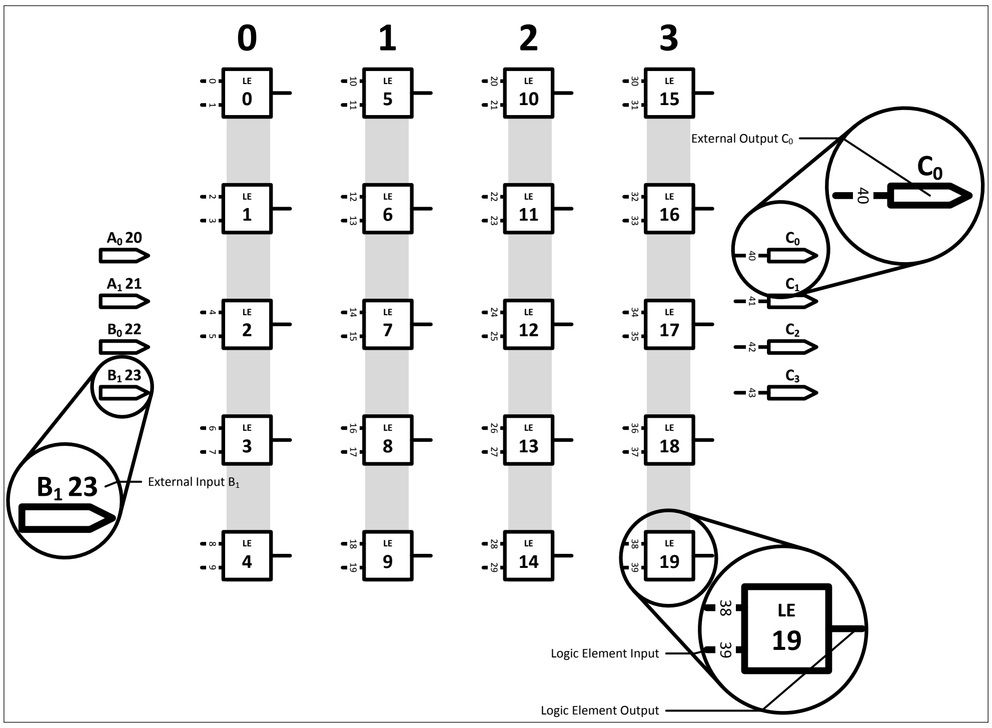


Figure 4.13 CGP layout of LEs used to permit only feed-forward circuits

Figure 4.14 shows an example two-by-three LE array with three feedback loops (bold lines) which are not permitted in the CGP configuration.

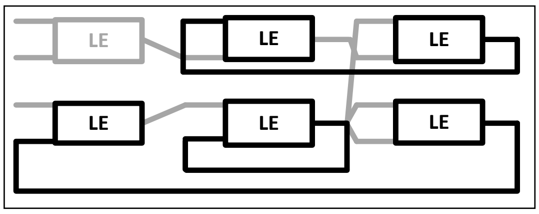


Figure 4.14 A CGP LE array with prohibited feedback loops

6. **External inputs could only be connected to column-zero LEs**

   In order to further reduce the GA's search space, the external inputs could only be connected to the inputs of LE 0 to 4 in column zero in Figure 4.13.

4.3.6 Downloading a Genotype

To download a chromosome onto the V-FPGA, the LabVIEW algorithm first decoded the chromosome into the logic and routing genotypes, as shown in Figure 4.15. Then, each gene was sequentially downloaded, via the relevant routing or logic lines, onto the V-FPGA's RCM or LCM.

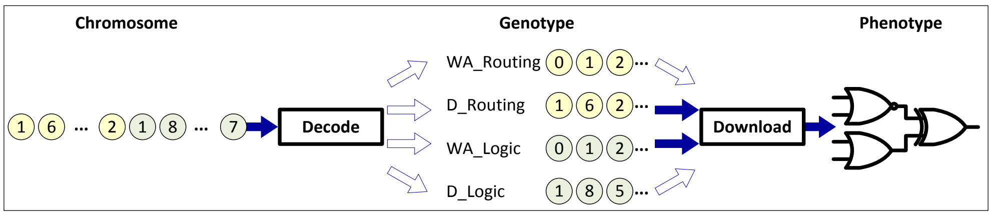


Figure 4.15 Decoding and downloading procedure

The above decode-and-download procedure was performed using the *D&T.vi* VI, shown in Figure 4.9. This VI made use of software for- and while-loops to activate the appropriate DAQ output lines, which, in turn, activated the appropriate FPGA pins.

The V-FPGA architecture only allowed the configuration memories to be altered on a rising-edge of the $Clock$ line. Hence, the execution time of the downloading procedure was governed by the $Clock$ frequency. The *D&T.vi* VI required one $Clock$ pulse per downloaded gene; or a total of 46 $Clock$ pulses for every gene per chromosome download.

4.3.7 Evaluating a Phenotype

To evaluate each downloaded phenotype, a sub-VI of the *D&T.vi* VI, called the *FitnessTest.vi* VI (shown in Figure 4.9), was used. During each generation, every phenotype was evaluated. Testing was done by comparing a phenotype's output to a truth table modelled on the solution circuit.

Consider the truth table shown in Figure 4.16. The inputs on the left form binary row-vectors which count from 0 to $2^I - 1 = 2^4 - 1 = 15$, where I represents the number of external inputs. These binary row-vectors are called test vectors. There are a total of $2^I = 2^4 = 16$ test vectors for a four-external-input V-FPGA.

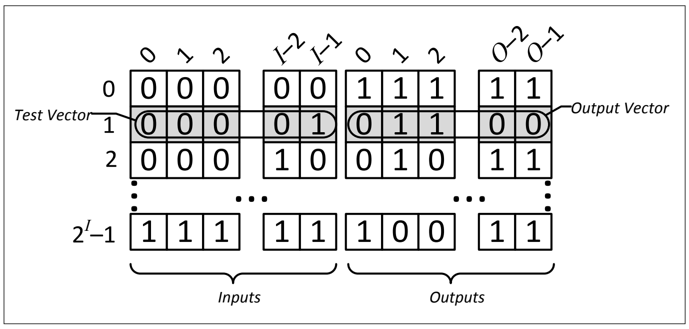


Figure 4.16 General truth table

Testing a phenotype involved LabVIEW sequentially loading each test vector onto the FPGA's external-input pins via the appropriate DAQ output lines. Once loaded, the LabVIEW GA read the DAQ input lines, which were connected to the FPGA's external-output pins.

Figure 4.17 shows an example test vector $[0, 0, 0, \dots, 1, 0]$ being loaded onto the V-FPGA's external inputs. The resultant output vector, which in this arbitrary example is $[0, 1, \dots, 1, 1]$, can then be compared to the corresponding output vector from the solution circuit's truth table. According to how well the output vectors compare, the phenotype is assigned a fitness value. A 100%-fit phenotype will therefore have the exact same output vectors as the solution truth-table.

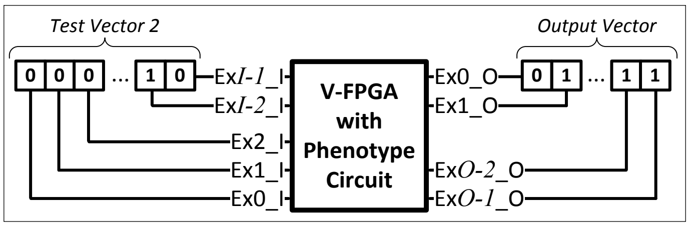


Figure 4.17 Test vector with corresponding output vector

4.3.8 The Fitness Function

To explain how a phenotype's fitness is calculated, first consider the conventional multiplier-circuit in Figure 4.18.

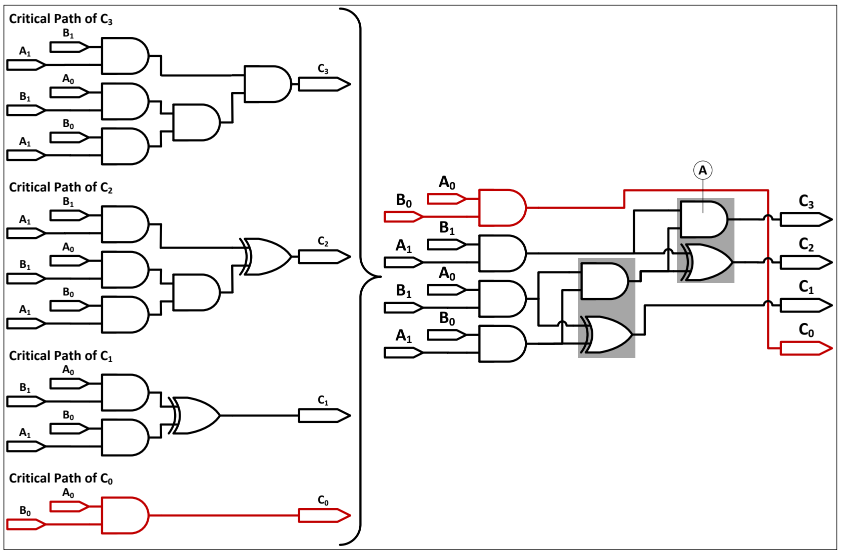


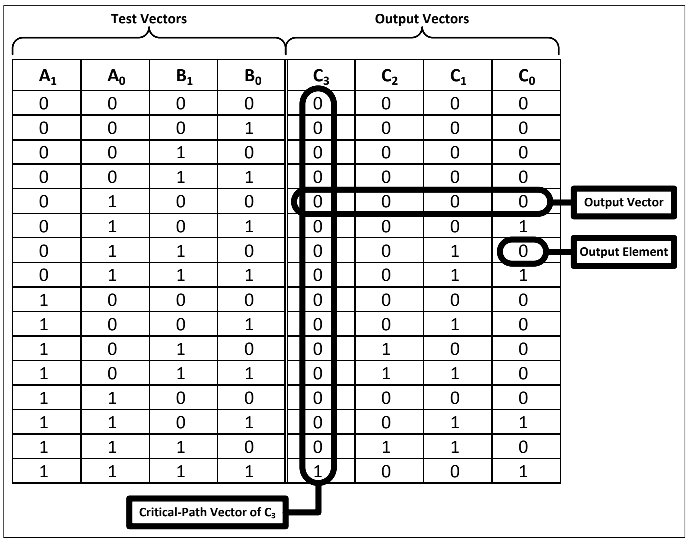
Figure 4.18 Conventional multiplier circuit (right) comprising of four smaller CP circuits (left)

Each external output has its own critical path (CP). A CP is the direct path linking the external inputs to a particular external output. Hence, the multiplier circuit can be thought of as four, smaller CP circuits that have been coupled.

CPs may be classified as either independent or dependent. An independent CP does not have any LEs in common with other CPs. For example, C_0's CP is independent since none of its LEs are used by any other CP (highlighted red in Figure 4.18). Dependent CPs, such as those of C_1, C_2 and C_3, have LEs in common.

Now, consider the multiplier's truth table, shown in Figure 4.19. There are three sets of data within the truth table that can be used to derive a phenotype's fitness, namely:

1. The 16 output vectors of the corresponding test vectors
2. The 64 individual output elements of the output vectors
3. The 4 CP vectors of each external output



A_1	A_0	B_1	B_0	C_3	C_2	C_1	C_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	0	1	0
0	1	1	1	0	0	1	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	1	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	1	1	0
1	1	1	1	1	0	0	1

Test Vectors
Output Vectors
Output Vector
Output Element
Critical-Path Vector of C_3

Figure 4.19 Analysis of a 2-bit multiplier's truth table

First, consider the output vectors. In initial evolution trial-runs, the phenotypes were awarded fitness values according to Equation 4.1, which expresses the percentage of correct output vectors. For example, if a phenotype had 12 correct output vectors, it would score an output-vector fitness of $F_{OV} = \frac{12}{16} = 75\%$.

$$F_{OV} = \frac{\text{Correct Output Vectors}}{\text{Output Vectors}}\%$$

Equation 4.1 Fitness function of the output vectors

However, it was soon discovered that this fitness scheme was flawed. To prove this, consider the AND-gate marked as callout A in Figure 4.18. This gate only affects the output of C_3. If the gate's function was arbitrarily changed, most of C_3's outputs would be incorrect. This would, in turn, lead to most output vectors also being incorrect, thereby yielding a low fitness value.

Nevertheless, this low fitness value would actually be underrated; since even though the phenotype's output vectors are mostly incorrect, the phenotype's LEs and routing is generally correct (only the changed gate is incorrect). For this reason, assigning fitness values using the output vectors was considered inaccurate and misleading.

A more useful and precise fitness value was derived from the output elements and CP vectors.

There are $O(2^I) = 4(2^4) = 64$ output elements for O external outputs and I external inputs. Equation 4.2 expresses the number of correct elements as a percentage. Thus, if an example phenotype had 24 of the 64 elements correct, a fitness of $F_{Elements} = \frac{24}{64} = 37.5\%$ would be assigned.

$$F_{Elements} = \frac{\text{Correct Output Elements}}{\text{Output Elements}} \%$$

Equation 4.2 Fitness function of the output elements

The advantage of using output elements over output vectors is that every correct element in the output vectors contributes towards the final fitness value. For example, if all the C_3 output elements in Figure 4.19 were incorrect, the truth table would yield zero correct output vectors but 16 incorrect output elements. Thus, the phenotype would attain a fitness value of $F_{Elements} = \frac{48}{64} = 75\%$ for an element evaluation, but $F_{OV} = 0\%$ for the output-vector evaluation. The output-element evaluation would provide a more accurate fitness value since the phenotype is partially correct.

$F_{Elements}$ does not, however, encourage the correct evolution of CPs. It merely gives an overall indication of a phenotype's correctness.

To understand why the correct evolution of the CPs is important, recall that CPs are often dependent. Because dependent paths rely on other CP LEs, correctly evolving one CP inevitably partially solves other CPs. For example, if C_1's CP in Figure 4.18 was to be successfully evolved, two of the five LEs in paths C_2 and C_3 would, by default, also be correct. This would in turn make the GA more efficient.

To encourage CP evolution, the CP vectors in Figure 4.19 need to be evaluated. To do this, the CP fitness, or F_{CP}, is used to expresses the percentage of correct CP vectors, as shown in Equation 4.3. For example, if a phenotype has three correct CP vectors, it would score a CP-vector fitness of $F_{CP} = \frac{3}{4} = 75\%$.

$$F_{CP} = \frac{\text{Correct Critical Path Vectors}}{\text{Critical Paths Vectors}} \%$$

Equation 4.3 Fitness function of the CPs

However, F_{CP} is very rigid, and only awards fully evolved CP vectors—partially evolved CPs are not awarded. For partially evolved CPs, Equation 4.4 is used.

$$F_{CP,Partial} = \frac{\sum_{i=0}^{3} 0.5(\text{\% of correct } true \text{ bits for } C_i\text{'s CP Vector}) + 0.5(\text{\% of correct } false \text{ bits for } C_i\text{'s CP Vector})}{\text{Critical Paths Vectors}} \%$$

Equation 4.4 Fitness function of the partial CPs

To explain Equation 4.4, first consider each CP vector in Figure 4.19:

- C_0's CP vector requires 4 $true$ bits and 12 $false$ bits
- C_1's CP vector requires 6 $true$ bits and 10 $false$ bits
- C_2's CP vector requires 3 $true$ bits and 13 $false$ bits
- C_3's CP vector requires 1 $true$ bit and 15 $false$ bits

For each CP vector, the percentage of correct $true$ and $false$ bits is calculated, and weighted in a $0.5{:}\,0.5$ ratio in Equation 4.4. For example, if C_0's CP vector yielded 3 correct $true$ and 10 correct $false$ bits, a fitness of $0.5\left(\frac{3}{4}\right) + 0.5\left(\frac{10}{12}\right) = 79.2\%$ would be assigned. Once all four CP-vectors have been assessed, the mean of the four CP-vector fitness values can then be expressed as $F_{CP,Partial}$.

An important aspect of Equation 4.4 is the $0.5{:}\,0.5$ ratio, in which the percentage of correct $true$ and $false$ bits are weighted. This ratio is essential. If not used, simply finding the percentage of correct bits will yield inaccuracies. To prove this, consider the circuit in Figure 4.20, which will always register a logic low regardless of the input signals. If, for example, this circuit was evolved as C_3's CP, the CP vector would register 16 $false$ bits. Thus, the CP-vector fitness evaluation would identify 15 of these $false$ bits as correct and only one as incorrect.

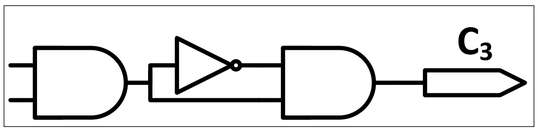


Figure 4.20 Logic-low circuit

However, these 15 correct bits are deceptive. From Figure 4.19 note that C_3's CP is one of the multiplier's most complex CPs and makes use of five LEs. When comparing the desired C_3 CP circuit in Figure 4.19 to the evolved circuit in Figure 4.20, there is substantially difference. Thus, assigning a CP-vector fitness value of $\frac{15}{16}$=93.8% will result in an overrated fitness score, since the evolved CP does not resemble the desired CP.

One way to curb this inaccuracy is to place an equal amount of emphasis on both the $true$ and $false$ bits. This is what the $0.5{:}\,0.5$ ratio does. If this ratio were to be applied to the above example, a fitness of $0.5\left(\frac{0}{1}\right) + 0.5\left(\frac{15}{15}\right) = 50\%$ would be achieved. This lowered fitness value is more truthful, as it more fittingly describes the poorly evolved CP circuit.

Finally, all three fitness values, namely $F_{Elements}$, F_{CP} and $F_{CP,Partial}$ could now be combined in order to describe the phenotype's overall fitness, as shown in Equation 4.5. Both $F_{Elements}$ and $F_{CP,Partial}$ were given an equal weighting of 30% of the overall fitness. However, to ensure that fully evolved CPs are preserved during the evolution process, a slightly higher 40% weighting was given to the F_{CP} fitness.

$$Fitness_{Overall} = 0.3F_{Elements} + 0.4F_{CP} + 0.3F_{CP,Partial}\%$$

Equation 4.5 Overall fitness function

In summary: $F_{Elements}$ provides an overview of a phenotype's correctness; F_{CP} ensures that correct CPs are sustained and $F_{CP,Partial}$ encourages the correct evolution of partially evolved CPs.

4.4 Results

The results will be addressed in three parts: the clock frequency, the outcome of the canonical evolution and the outcome of the $1 + \lambda$ evolution.

Since this chapter's objective is to compare the canonical and $1 + \lambda$ GAs, the following control variables were kept constant:

- Both GAs made use of a six-individual population
- Both GAs made use of the same fitness function (Equation 4.5)
- Both GAs made use of the same GA constraints (Section 4.3.5.2)

4.4.1 Clock Frequency

The first major difficulty experienced, when using the DAQ and DE2 setup, was the rate at which the LabVIEW application-programming interface (API) could be synchronised with the V-FPGA's clock. The *D&T.vi* VI was the GA's most time-consuming subroutine.

Recall that the downloading routine is governed by the $Clock$ line. Thus, the clock frequency has a major impact on the downloading accuracy and the execution time of the GA. It was imperative, to the success of the GA, that each chromosome was evaluated accurately. If inaccuracies occurred, false fitness values could have caused the GA to reject strong individuals.

To test the download accuracy, a 100%-fit individual was first downloaded, and then tested on the 20-LE V-FPGA. The download was deemed completely accurate if the evaluation revealed a fitness of 100%.

The results[8] in Figure 4.21 show the download accuracy at different clock frequencies. Each data point represents the average accuracy of 20 downloads taken at a particular frequency. Readings were taken at 10 different frequencies, which varied according to the clock's period from 10ms to 80ms. From the graph, it is evident that clock frequencies above 20Hz were to be avoided.

[8] Refer to Appendix C for the raw data.

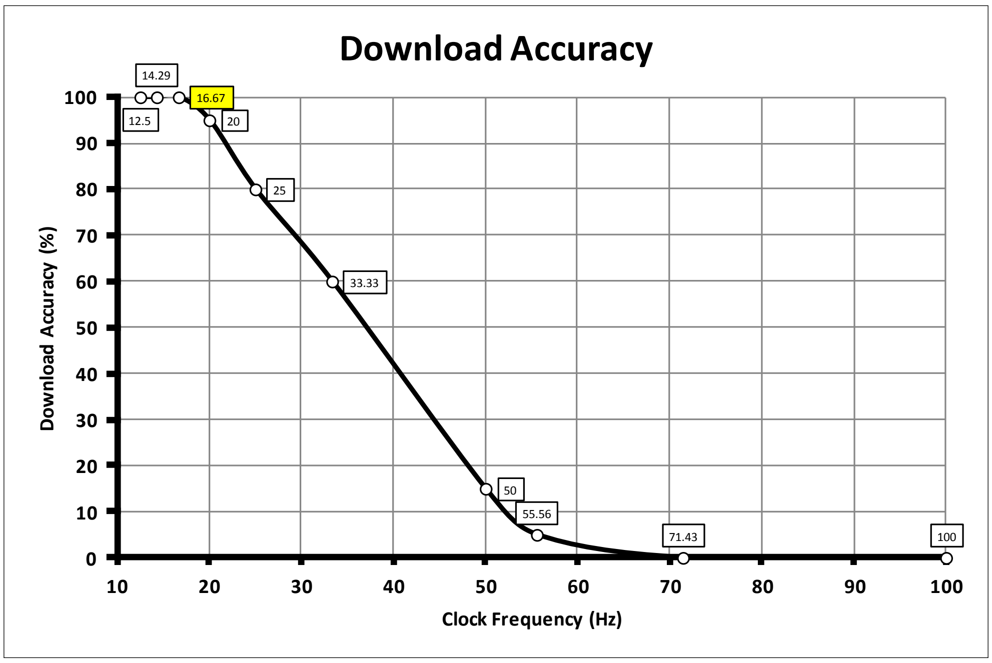


Figure 4.21 Download accuracy vs. clock frequency

Now, consider Figure 4.22. Each data point shows the average download time of 20 downloads at varying frequencies. As expected, since the frequency is inversely proportional to the period, the download-time graph shown is exponential. The graph settles from as it approaches 100Hz, with only incremental improvements made on the download time from 70Hz and faster. Thus, any frequency above 70Hz would be ideal.

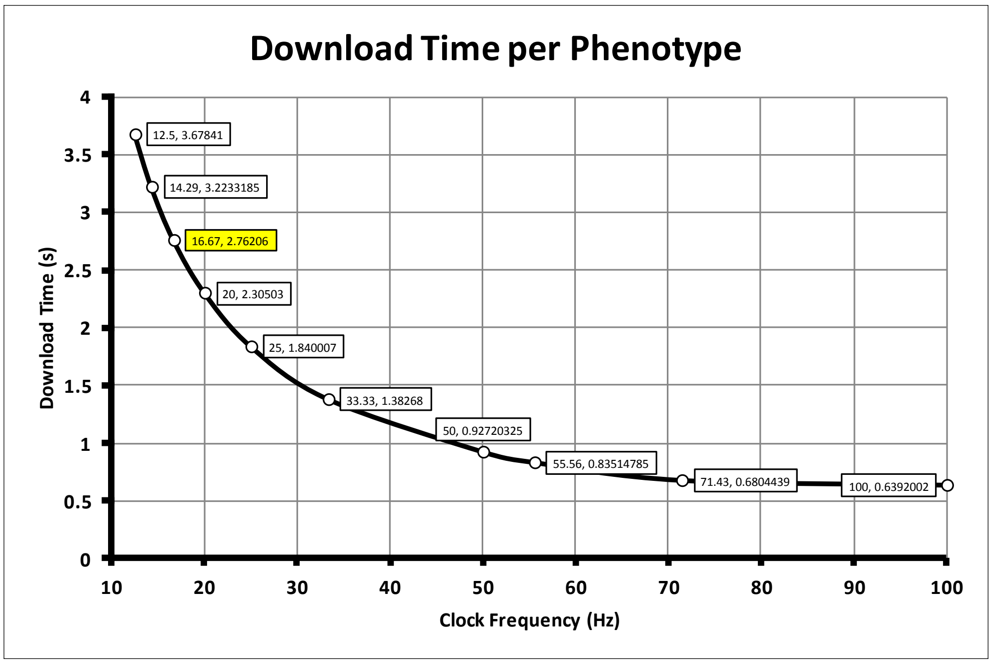


Figure 4.22 Download time vs. clock frequency

However, taking both the accuracy and download time into account, the 16.67Hz clock was selected as the optimum due to it being the fastest accurate clock frequency. The download time of an individual using the 16.67Hz clock was calculated as 2.762 ± 0.006 s (95%)[9].

An additional observation worth noting is that the $Clock$ line did not affect the testing VI. Thus, once a phenotype was downloaded onto the V-FPGA, it was tested as quickly as the LabVIEW program permitted. The time taken for the testing VI to test one individual was calculated as 0.123 ± 0.005 s (95%)[10].

By comparing the downloading and testing times, downloading took over 22 times longer to execute than testing. This is expected, as the testing VI was not clocked and only needed to load 16 test vectors, compared to the 46 clock pulses needed by the download VI (as mentioned in Section 4.3.6).

4.4.2 Canonical Evolution

The canonical-evolution results are presented in two subsections below. The first section discusses the results of an initial trial run; the second section discusses the final-canonical GA results. The trial GA has been included purely to show the reader the various variables which were considered while deriving the final-canonical and $1 + \lambda$ GAs. The final-canonical and $1 + \lambda$ GAs make use of all the operators and constraints previously discussed in this chapter.

4.4.2.1 Trial-Canonical Results

Since hardware evolution, using the V-FPGA, had never been performed before, all aspects of the implemented GA had to be experimentally optimised. To do this, many trial runs were executed and analysed. Each trial made use of different population sizes, elitism schemes, $Clock$ frequencies and fitness functions. These trials were very time-consuming, with one trial typically taking well over 40 hours to complete. In addition, the trials yielded poor fitness values.

However, after much experimentation, one trial run did evolve a 100%-fit phenotype. This successful-trial GA made use of fifty individuals; and all LE functionality was allowed, i.e. all 16 LE functions were permitted. (This is in direct contrast to the final-canonical GA's parameters, which used a six-individual population and only fundamental LEs.) Furthermore, another notable difference was the trial's fitness function shown in Equation 4.6. The function does not consider the CP vectors.

$$Fitness_{Trial} = 0.75F_{Elements} + 0.25F_{OV}\%$$

Equation 4.6 Trial fitness function

Nevertheless, even though the trial GA did not use optimised parameters, it was still successful.

[9] The result is displayed using the standard deviation of the mean at a 95% confidence interval. Thus, 95% of the measured values should lie within the given range.
[10] See footnote 9.

Figure 4.23 shows the progress of the fittest individual using the successful-trial GA. The first parent had an initial fitness of 51.5%, which steadily increased during the first 50 generations. Notice that there are large jumps in the graph. This is expected, since changing the routing or function of one LE can dramatically improve, or deteriorate, a phenotype's fitness.

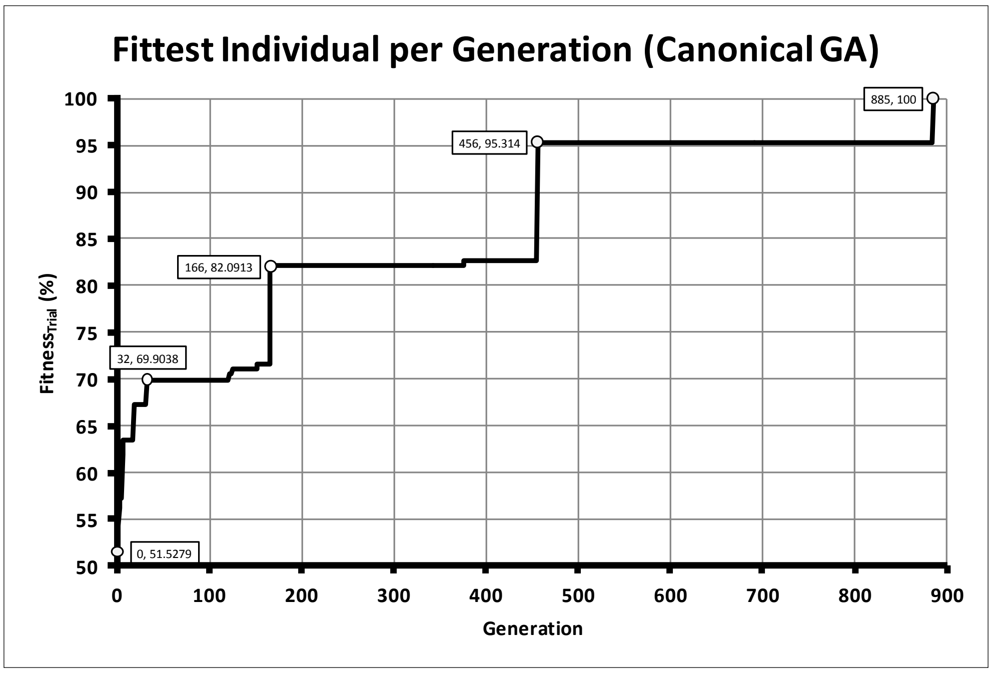


Figure 4.23 The results of the successful trial

The following milestones are also noted in Figure 4.23's graph:

- At fitness values 69.9%, 82.1%, 95.3% and 100%, the outputs C_0, C_3, C_1 and C_2 respectively are correctly evolved. C_0's CP passes two LEs, C_3 and C_1's CPs pass three LEs, while C_2's CP passes six LEs.
- It took a total of 885 generations to evolve the phenotype. This amounted to an evolution time of approximately 49 hours and 14 minutes.

Figure 4.24 shows the evolved phenotype. The bold circuitry shows the CPs. Notice that of the twenty available LEs, only ten were used in the evolved circuit.

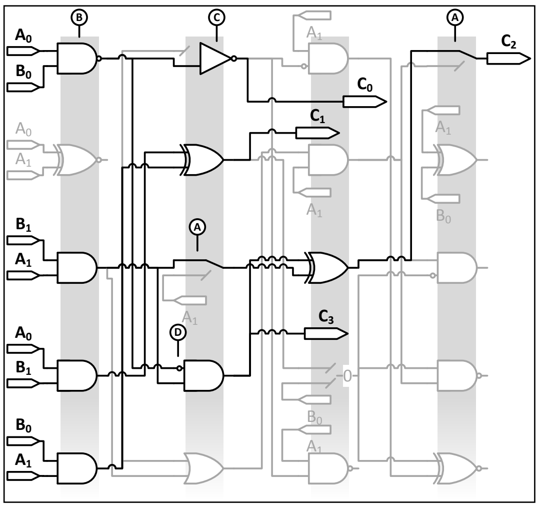


Figure 4.24 Evolved trial phenotype

Figure 4.24 can be further simplified into the circuit shown in Figure 4.25[11] by removing:

- The wire LEs, i.e. LEs which only pass data through them. There are two examples of wire LEs in Figure 4.24, which have been demarcated as callouts A.
- Redundant NOT-gates. The NAND-gate (callout B) is connected to a NOT-gate (callout C) and an inverted input to an AND-gate (callout D). Thus, the two bubbles and NOT-gate are, in reality, redundant and can be removed.

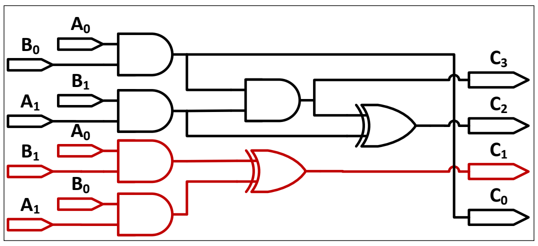


Figure 4.25 Simplified trial phenotype (A red outline indicates an independent CP)

4.4.2.2 Final-Canonical Results

After analysing the successful trial runs, the importance of rewarding correctly evolved CPs was realised. This led to the derivation of the $Fitness_{Overall}$ fitness function (Equation 4.5).

The final-canonical GA was executed eleven times before a 100%-fit phenotype was evolved. Each run was limited to 3000 generations, due to time constraints.

[11] To prove that the simplified evolved phenotypes do indeed produce the correct outputs, please refer to the document titled "APPENDIX B: Truth Tables of the Evolved Phenotypes". All phenotypes included in this dissertation have been proven correct in this appendix.

Figure 4.26 shows the progress of the fittest parent using the final-canonical GA. Within the first 200 generations, the GA had evolved two of the four CPs. Like with the trial-canonical GA, notice that there are large jumps in the graph. However, unlike in the trial, these jumps are largely due to the F_{CP} function in the $Fitness_{Overall}$ fitness evaluation. When a correct CP is evolved, the F_{CP} variable increases the overall fitness by 10%, resulting in noticeable jumps.

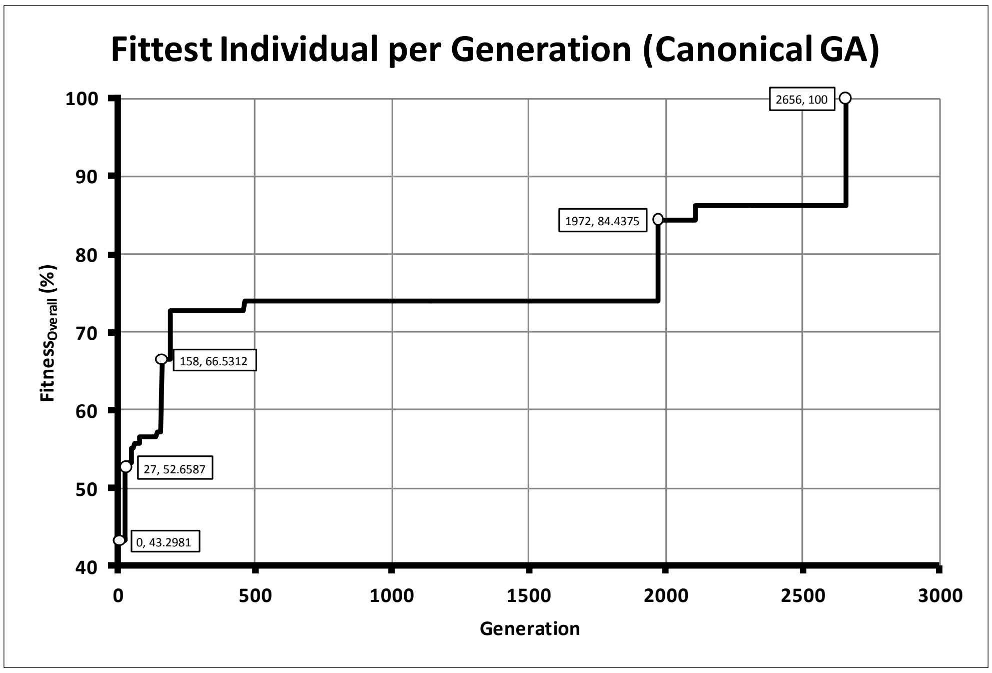


Figure 4.26 Results of the final-canonical GA

The following milestones are also noted in Figure 4.26:

- At fitness values 52.7%, 66.5%, 84.5% and 100%, the outputs C_0, C_3, C_2 and C_1 respectively are correctly evolved. C_0's CP passes one LE, C_3 and C_1's CPs pass three LEs, while C_2's CP passes five LEs.
- It took a total of 2656 generations to evolve the phenotype. This amounted to an evolution time of approximately 15 hours and 49 minutes.

The final-canonical phenotype is shown in Figure 4.27, with the bold circuitry showing the CPs. Out of the twenty available LEs, nine were used.

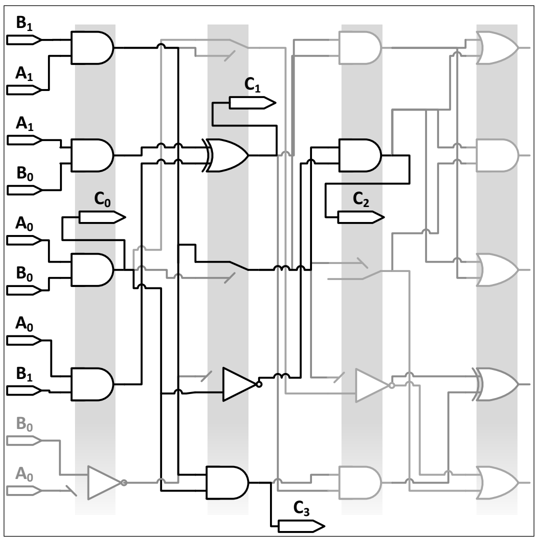


Figure 4.27 Canonical phenotype

Figure 4.27 can be further simplified into the circuit shown in Figure 4.28 by removing the wire LEs. Since the final-canonical GA only made use of fundamental gates, there are no redundant bubbled or NOT-gates.

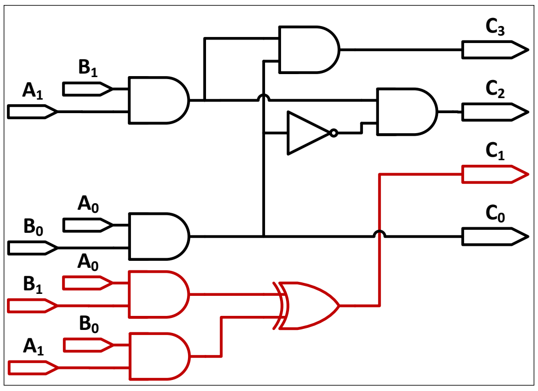


Figure 4.28 Simplified canonical phenotype (A red outline indicates an independent CP)

4.4.3 $1 + \lambda$ Evolution

The $1 + \lambda$ GA was executed eight times before a 100%-fit phenotype was evolved. Like with the canonical GA, each run was limited to 3000 generations.

Figure 4.29 shows the progress of the fittest individual using the $1 + \lambda$ GA. Notice that there are four spikes in the graph (demarcated with ×). These spikes represent downloading errors, where the phenotype has been incorrectly downloaded onto the V-FPGA.

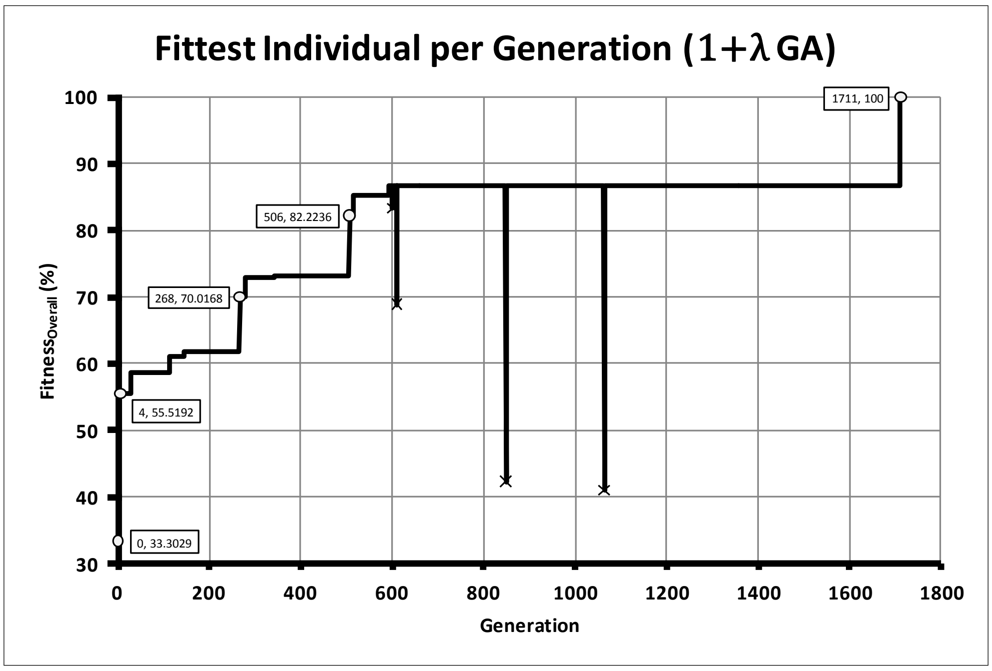


Figure 4.29 Results of the $1 + \lambda$ GA

The following milestones are noted in Figure 4.29's graph:

- At fitness values 55.5%, 70.0%, 82.2% and 100%, the outputs C_0, C_1, C_3 and C_2respectively are correctly evolved. C_0's CP passes one LE; C_1's CP passes three LEs; C_3's CP passes five LEs; while C_2's CP passes nine LEs.
- It took a total of 1711 generations to evolve the phenotype. This amounted to an evolution time of approximately 9 hours and 59 minutes.

The final $1 + \lambda$ phenotype, shown in Figure 4.30, made use of eleven LEs. Of these eleven, three were wire LEs.

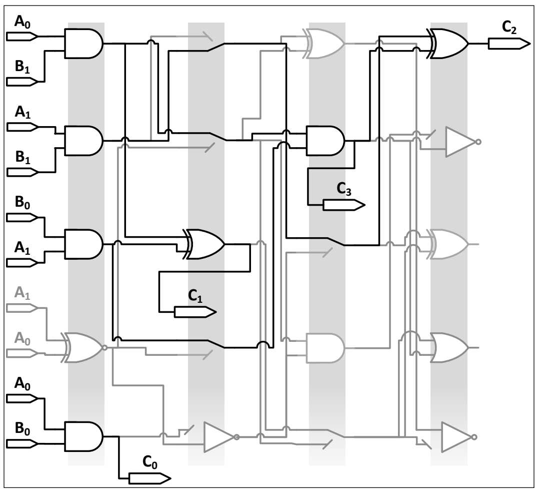


Figure 4.30 $1 + \lambda$ phenotype

Again, the wire LEs have been removed in the simplified circuit, shown in Figure 4.31. C_0's CP is the only independent CP.

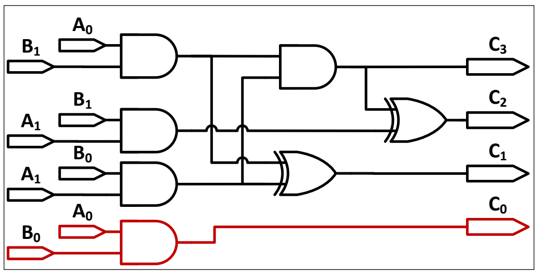


Figure 4.31 Simplified $1 + \lambda$ phenotype (A red outline indicates an independent CP)

4.5 Discussion

4.5.1 The V-FPGA and Hardware Setup

From the compilation report (Figure 4.8), it is apparent that the 20-LE V-FPGA used a minimal amount of hardware resources.

If one had to compare the V-FPGA to direct-bitstream evolution, due to the nature of the V-FPGA using registers and multiplexers to implement logic, the V-FPGA uses more FPGA logic blocks to configure a phenotype than by directly configuring the phenotype on the FPGA. Nonetheless, this trade-off is minimal, especially since the V-FPGA should to be kept small to avoid scalability problems.

In future research, the remainder of the FPGA's logic blocks could be used to implement a soft-processor, thereby creating a system-on-a-programmable-chip (SOPC) solution, as suggested by Smith (2010). The soft-processor would manage the GA and fitness functions, while the V-FPGA would implement and test the phenotypes.

Consider Figure 4.32, which shows an example SOPC. It makes use of Altera's Nios II soft-processor, which runs on most of Altera's FPGA chips, including the DE2's Cyclone II chip. Although Altera's products were used in this research, in principle, any FPGA vendor's offering could be used. For example, Xilinx's MicroBlaze soft-processor could be implemented in a similar manner.

The Nios II uses the Avalon switch fabric to interface with any other embedded peripherals on the FPGA; and is accessed by the host computer via a JTAG debug module. A special JTAG UART interface is used to connect the USB-Blaster circuitry, which provides a USB link, to the host computer.

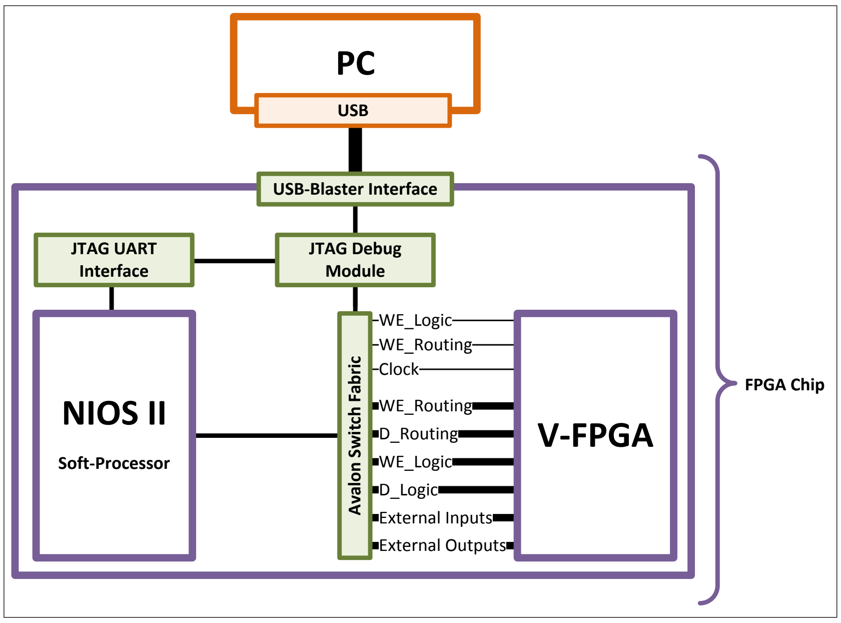


Figure 4.32 Future EHW SOPC

This above solution demonstrates on-chip evolution. On-chip evolution could decrease the GA's execution time by minimising the hardware resources needed to interface the GA with the V-FPGA. Even the slightest improvement in the $Clock$ frequency will see significant improvements in the evolution time. For example, decreasing the download time by a half second per individual, for a six-individual population running for 3000 generations, will save $0.5 \times 6 \times 3000 = 9000s$ or 2.5 hours. This time saving becomes more significant as the number of generations increase.

4.5.2 The Genetic Algorithms

Comparing the final-canonical and $1 + \lambda$ results, the final-canonical GA took 945 generations longer to find a solution, amounting to a further 5 hours 50 min of evolution time. In addition, the $1 + \lambda$ GA found a solution after eight attempts, compared to the final-canonical GA's eleven.

Although the study was based on a small sample of evolution attempts, overall, the results suggest that the $1 + \lambda$ GA was more efficient, and agree with the findings of Vassilev, et al (1999) and Sekanina & Freidl (2005). The subsections below discuss possible reasons why.

4.5.2.1 The Crossover and Mutation Operators

In traditional GAs, used to solve mathematical problems, the chromosomes are represented using floating-point notation. The crossover operator creates new offspring by combining the parents' genes. This is done, for example, by finding the arithmetic mean of each pair of genes. The result is that the offspring's fitness is never worse than a parent's fitness.

The above idea falls under the topic of “evolvability”. Evolvability is defined as the ability for an EHW system to produce individuals fitter than those found in previous generations (Altenberg, 1994). To examine why EHW systems have poor evolvability, we need to consider the GAs’ fitness landscapes.

For traditional GAs, the fitness landscape is considered to be smooth, resulting in the offspring always converging towards a solution. However, this is untrue in EHW systems.

EHW systems have rugged fitness landscapes, where small changes in a gene dramatically influence the fitness. For example, one altered gene can map a NOT-gate in a fit phenotype, thereby inverting all the output signals and completely spoiling the phenotype. Similarly, simple routing changes can also influence the fitness of a phenotype.

Vassilev et al. (1999, p. 1) elaborates: “The difference [between a smooth and rugged landscape] originates in the structure of the genotypes, which are strings defined over two completely different alphabets, and are responsible for the functionality and connectivity of the array of logic cells.” Stated differently, there is no mathematical correlation between the phenotype’s fitness and the genotype’s logic and/or routing genes.

During evolution, two parents may have similar fitness values, but their phenotypes can be completely dissimilar. This raises concerns as to how to implement crossover, if at all. Simply swapping the parents’ genes—randomly combining segments of two different parent phenotypes’ topologies and functionality—will inevitably result in weak offspring.

Thus, to maintain a system’s evolvability, crossover should be used with caution when applied to digital circuitry. Instead, as used in the $1 + \lambda$ GA, an EHW GAs should rely on mutation. By making small adjustments to a phenotype, there is a greater probability of producing fit offspring.

4.5.2.2 The Population Size

In the initial trials, large populations, with fifty or more individuals, were used. These large populations inevitably took longer to execute since there were more individuals to evaluate.

However, it was later found in the final-canonical and $1 + \lambda$ GAs that large populations were unnecessary. To explain why, first consider a fifty-individual population. To create a new generation, the fittest parent in the population is crossed-over and/or mutated fifty times according to the implemented GA. This means that even if the first offspring is fitter than the parents, the GA will continue to crossover/mutate the parents from the original population until fifty new offspring are created. Only once the new generation is formed will this fitter offspring become the new parent.

Now, consider a smaller six-individual population, where the GA crosses-over/mutates the parent six times. Unlike in the large population, if the first offspring is fitter, the GA only has to create five more offspring (and not 49) in order to form a new six-individual population. Again, once complete, this fitter offspring will become the new parent.

Because smaller populations are evaluated in smaller batches, the fittest parents are updated more regularly than in larger populations. This ensures the mutation and crossover operators are more effective and the GA has a greater level of efficiency. Thus, the smaller six-individual population was favoured.

4.5.3 The Evolved Phenotypes

4.5.3.1 Comparing the Simplified Evolved Phenotypes

The conventional and evolved multiplier circuits show similarities in that they all made use of the same external-input combinations to the AND-gates. These AND-gates are crucial since they calculate the partial products of the multiplicand and multiplier. However, unlike in the conventional circuit which adds the partial products, none of the evolved phenotypes made use of the half-adders.

All three simplified phenotypes are unique, showing that there is more than one solution circuit within a GA's search space. In particular, the final-canonical phenotype is interesting because it did not evolve a second XOR-gate (as found in the trial-canonical and $1 + \lambda$ phenotypes). This uniqueness is a by-product of the inherent degree of randomness a GA possess, as seen in the random mutation, crossover and initial population. Also, the uniqueness demonstrates a major advantage of using an EHW system—they can autonomously find unusual, novel and often more efficient solutions to problems.

To shows that the evolved phenotypes are often more efficient, consider Table 4.3. The table summarises the total number of LEs used by each simplified phenotype, as well as the number of LEs used by each CP.

Table 4.3 LE summary of the simplified phenotypes

	C_0	C_1	C_2	C_3	**Total number of LEs**
Conventional Multiplier	1	3	5	5	8
Trial-Canonical Phenotype	1	3	4	3	7
Final-Canonical Phenotype	1	3	4	3	8
$1 + \lambda$ Phenotype	1	3	5	3	7

From the table, the following is observed:

- C_0 and C_1's CP remained unchanged in both the evolved and conventional circuits.
- C_2's CP in the conventional multiplier used five LEs. This was improved upon in both canonical phenotypes by using only four LEs, but remained unchanged in the $1 + \lambda$ phenotype.
- Again, C_3's CP in the conventional multiplier used five LEs. This was improved upon in all evolved phenotypes by using only three LEs.
- Both the conventional circuit and final-canonical phenotype made used of eight LEs, while the trial-canonical and $1 + \lambda$ phenotypes only used seven.

Thus, in summary, of the four evolved CPs, two remained unchanged; one was usually improved upon; while one was always improved. No phenotype was less efficient than the conventional circuit, with two of the three evolved phenotypes improving the circuit's efficiency by one LE. All evolved phenotypes were unique.

4.5.3.2 The Evolved Critical Paths

In all the experiments, using both the canonical and $1 + \lambda$ GAs, the sequence in which the CPs were evolved was also unique. For the three successfully evolved phenotypes, the sequence of the evolved CPs was as follows:

- Trial-canonical GA: $C_0\ C_3\ \boldsymbol{C_1}\ C_2$ where C_1 is independent
- Final-canonical GA: $C_0\ C_3\ C_2\ \boldsymbol{C_1}$ where C_1 is independent
- $1 + \lambda$ GA: $\boldsymbol{C_0}\ C_1\ C_3\ C_2$ where C_0 is independent

The above CP results are summarised in Table 4.4:

1. The first row in the table shows the number of LEs used by each CP.
2. During evolution, due to some CPs being dependent, some LEs are shared and thus only need to be evolved once. This is shown in the second row. Thus, all independent CPs and the first evolved dependent CPs will always have no previously evolved LEs, and will yield a 0 in the second row of the table.
3. Finally, by finding the difference between the number of used LEs and the number of previously evolved LEs, the net number of LEs that was needed to be evolved for the particular CP can be calculated, as shown in the final row.

Table 4.4 Summary of the net number-of-LEs evolved by each CP (Red text indicates an independent CP)

	Trial-Canonical Phenotype				**Final-Canonical Phenotype**				**$1 + \lambda$ Phenotype**			
	C_0	C_3	C_1	C_2	C_0	C_3	C_2	C_1	C_0	C_1	C_3	C_2
Number of evolved LEs used by the CP	2	3	3	6	1	3	5	3	1	3	5	9
Number of LEs, used by the CP, that were previously evolved by other CPs	0	1	0	3	0	1	2	0	0	0	2	5
Net number of LEs that were evolved (Net evolved LEs)	2	2	3	3	1	2	3	3	1	3	3	4

From the above table, the following is observed:

- C_0's CP was always evolved first, regardless of its dependence. This is due to its simplicity, i.e. it only used one or two LEs.
- C_1's CP sequence varied—from third to fourth to finally second place.

In both canonical GAs, C_1 was independent and thus was never partially evolved with the evolution of the other CPs. This explains why the CP took longer to evolve when compared to the $1+\lambda$ phenotype.

For the $1+\lambda$ phenotype, C_1's CP was the first dependent CP to be evolved. This is due to its simplicity when compared to the other dependent CPs in the $1+\lambda$ phenotype, i.e. it used three LEs compared to the five and nine LEs used by C_3 and C_2 respectively.

- C_2's CP was evolved either third or last, mostly due to its complexity. In all three phenotypes, C_2's CP used the most LEs—it used five LEs in the final-canonical phenotype, six LEs in the trial-canonical phenotype and nine LEs in the $1+\lambda$ phenotype. However, due to C_2's CP always being dependent, the net number of evolved LEs was much lower, and thus C_2's CP evolved in a reasonable amount of time. This is particularly evident in the $1+\lambda$ GA, where C_3's CP is a key component to C_2's CP, providing five of the nine required LEs.
- Although C_3's CP was evolved either second or third, it was always the second dependent CP to evolve due to it neither being the simplest, nor the most complex dependent CP.

The net-evolved-LE number reveals an important insight into the manner in which a GA evolves the phenotypes. Notice that the net number, for a particular phenotype, increases for each CP. This shows that GAs tends to evolve CPs with a smaller net numbers first. Thus, the fewer net LEs a CP requires, the more likely a GA will correctly evolve the CP. This is expected, since intuitively there is a higher probability of correctly evolving a simpler CP which uses fewer LEs.

4.5.4 Repeatability

The results show that the GAs' have low repeatability, with all the GAs being executed numerous times before being successful.

Poor repeatability is common in EHW systems. In Chapter 2, it was mentioned that Wang, et al. (2007) was able to scale a three-bit multiplier and adder. But, even so, Wang, et al.'s results were also not repeatable, with a success rate of only 50%.

One of the first symptoms of poor repeatability is the stalling effect. The stalling effect, as coined by Stomeo, et al. (2006), is defined as "non-improvements of fitness values during the evolutionary process." Figure 4.33 shows two sample failed attempts using the $1+\lambda$ GA, with the stalling effect clearly visible from the 1500th generation and onwards.

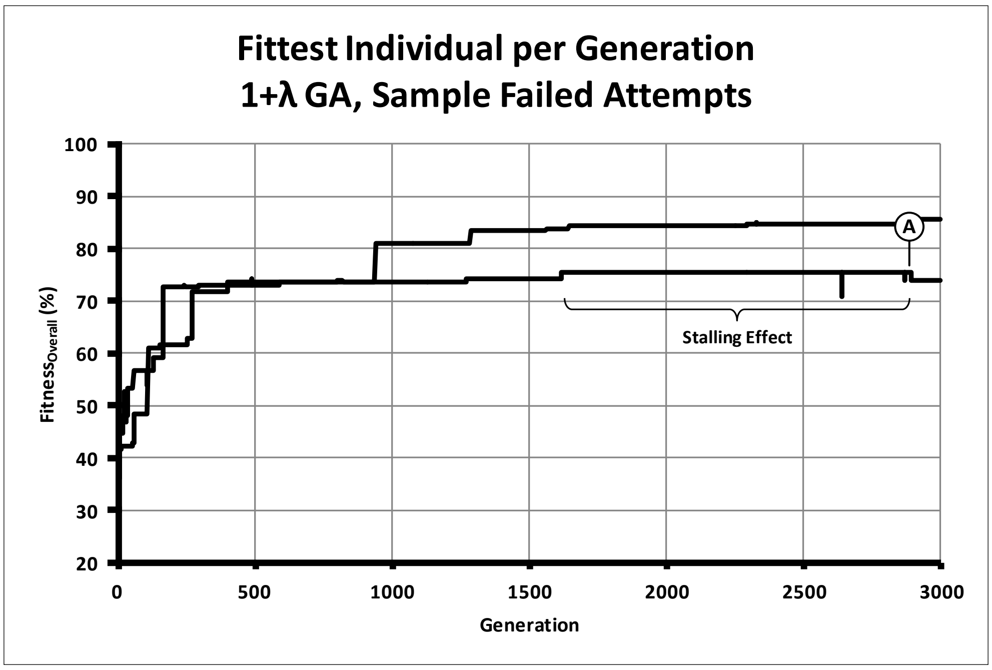


Figure 4.33 Sample failed attempts showing the Stalling Effect.

Three potential reasons for poor repeatability have been identified, with the first two reasons contributing to stalling fitness values:

1. Scalability
2. Erroneously-evolved chromosomes
3. Inaccurate phenotype downloads

4.5.4.1 Scalability

As discussed in Section 4.3.5.1 , even though the two-bit multiplier is considered to be a simple circuit, it has an almost infinite search space. In the experimentation, the search space was reduced using evolution constraints and a smaller V-FPGA. However, although reduced, the space still remained large.

Since scalability is a direct consequence of a large search space, one cannot completely rule out the notion that the unsuccessful evolution attempts were partially due to scalability issues. The unsuccessful evolutions, in turn, resulted in poor repeatability.

Thus, to summarise, the larger or more complex a circuit, or the larger the V-FPGA, the more prominent scalability issues become and consequently, the less repeatable the results.

4.5.4.2 Erroneously-Evolved Chromosomes

During evolution, as more CPs are successfully evolved, so the probability of mutating a correct LE increases; and the number of available LEs decreases. This implies that as individuals become fitter, so the difficulty finding a solution also increases.

For example, say for the 20-LE V-FPGA, one CP is correctly evolved using four LEs, with the remaining sixteen LEs yet to be evolved for the other CPs. One may think that the GA now has a greater chance of success since there are fewer routing or logic configurations, i.e. there are only sixteen potential LEs to be evolved compared to the initial twenty. However, this is not the case, since these four evolved LEs can still be altered by the GA. During evolution, the GA cannot distinguish between correctly and incorrectly evolved LE. Consequently, the GA can modify any LE—even if correct. Thus, in actuality, there is an increased chance of erroneously altering a correct LE, thereby making it less likely to successfully evolve the next CP. The above explanation is reflected in the results. Most failed attempts, such as those shown in Figure 4.33, managed to evolve three of the four CPs, with the complex or independent CP failing to evolve.

To further clarify the above explanation, consider another example. Say a GA has correctly evolved three of the four CPs. During evolution, even if the GA correctly evolves the fourth CP in a particular phenotype, there is high probability that the GA, in the process of evolving this fourth CP, will erroneously alter the other three CPs. Thus, the fitness function will return a low value for this phenotype since one or more of the original three CPs are now incorrect.

Future research could investigate using a GA that can identify and isolate correct genes within a chromosome. By doing this, there will be no chance of erroneously modifying correct LEs, and thus mutation and crossover will only be applied to the genes still requiring further evolution.

4.5.4.3 Inaccurate Phenotype Downloads

Though the 16.67Hz was chosen due to its perceived accuracy, some random downloading errors did still occur, as seen in Figure 4.29. In that particular example, the GA managed to recover from the downloading errors. However, this is not always the case, and in many of the trials, the downloading accuracy negatively impacted on the GA's success. An unsuccessful recovery is illustrated in Figure 4.33, callout A.

For the canonical GA in Figure 4.26, there are no perceived downloading errors because the canonical GA made use of two elite individuals (see Section 4.3.3.4), i.e. the two fittest individuals in a generation were retained. These two individuals usually had the same fitness values, differing from each other only for a few generations when a new elite individual was found. Thus, if the fittest individual failed to download properly, the second individual simply replaced the failed one.

Chapter 5

THE MODULAR EVOLUTION OF A FINITE-STATE MACHINE

The aim of this chapter is to demonstrate a means of overcoming scalability by using modular evolution. Scalability is more prominent when evolving larger circuits, such as those used in real-world FSMs, since larger circuits naturally also have larger search spaces. The chapter starts by introducing a case study: A FSM's control circuit, to be used in a typical mechatronics application, needs to be designed and evolved. To do this, the system's control requirements, components and operation are first defined. Then, based on these parameters, the FSM is modelled using a state and block diagram. The state diagram describes each state's operation and transition requirements, while the block diagram describes the sequential- and combinational-logic sub-circuits used to create the complete control circuit. The combinational-logic portion is then further analysed and described using truth tables, which later forms a core part of the GA's fitness function. Finally, by using the hardware setup and $1 + \lambda$ GA discussed in Chapter 4, each state's combinational sub-circuit is then independently evolved. The chapter closes with a discussion.

5.1 Introduction

A packaging company[12], which manufactures corrugated boxes, makes use of a FSM control circuit that controls the production of glue in two tanks. The first tank—the mixing tank—is used to mix starch and water together, at a specified temperature, in order to produce a batch of glue. The predefined temperature set-point is selected by the tank's operator via a numeric keypad. This set-point determines the glue's viscosity—an important aspect influencing the integrity of the final box. After mixing and heating, the glue is then pumped into a second tank—the holding tank. The holding tank stores the glue, also at a specific temperature, until it is needed by the factory's gluing machinery. When the glue is pumped from the holding

[12] The unique case study examined in this chapter is loosely based on an example control circuit discussed by Floyd (2009, p. 268).

tank, it is pumped in a ten-second-on, five-second-off cycle. The pumped glue is used to produce the board needed for the boxes.

5.2 System Components

The mixing and holding tanks make use of a number of digital sensors and actuators, i.e. the sensors can only output *high* or *low*, while the actuators can only be in an *on* or *off* state.

Consider Figure 5.1 and Figure 5.2, showing all control components used by the mixing and holding tanks. The mixing tank makes use of four sensors and four actuators, while the holding tank uses only two sensors and two actuators. Each sensor and actuator is connected to an external IO on the FSM control circuit.

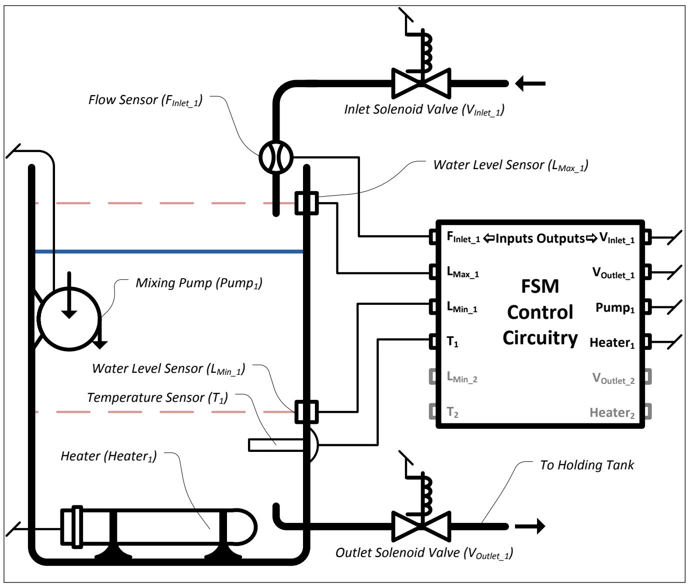

Flow Sensor (F_{Inlet_1})
Inlet Solenoid Valve (V_{Inlet_1})
Water Level Sensor (L_{Max_1})
F_{Inlet_1} ⇦Inputs Outputs⇨ V_{Inlet_1}
L_{Max_1}
V_{Outlet_1}
FSM Control Circuitry
L_{Min_1}
$Pump_1$
T_1
$Heater_1$
L_{Min_2}
V_{Outlet_2}
T_2
$Heater_2$
Mixing Pump ($Pump_1$)
Water Level Sensor (L_{Min_1})
Temperature Sensor (T_1)
Heater ($Heater_1$)
To Holding Tank
Outlet Solenoid Valve (V_{Outlet_1})

Figure 5.1 Mixing-tank components

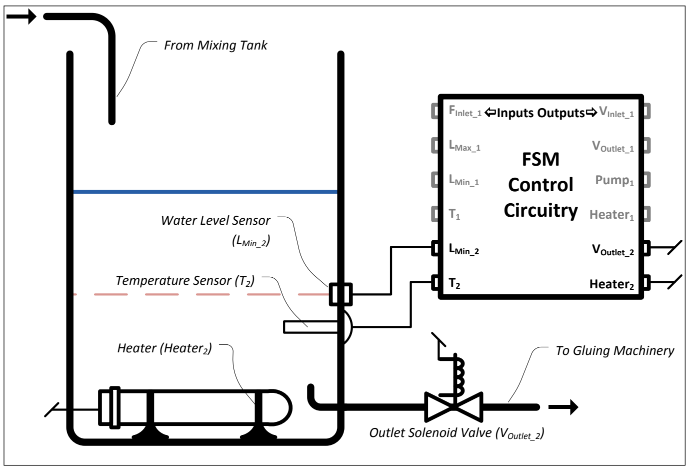


Figure 5.2 Holding-tank components

Table 5.1 and Table 5.2 summarises the tanks' components from the above two figures. Each component's external IO on the control circuit has been given a variable name.

Table 5.1 Overview of the mixing-tank variables

Variable	IO	Sensor/Actuator Name	Description
$\boldsymbol{F_{Inlet\_1}}$	Input	Flow Sensor	Determines whether water is running into the mixing tank. Outputs *high* if water flows into tank.
$\boldsymbol{L_{Max\_1}}$	Input	Maximum Water-Level Sensor	Detects the maximum water level. Outputs *high* if water covers the sensor.
$\boldsymbol{L_{Min\_1}}$	Input	Minimum Water-Level Sensor	Detects the minimum water level. Outputs *high* if water covers the sensor.
$\boldsymbol{T_1}$	Input	Temperature Sensor	Detects the water's temperature. Outputs *high* if temperature reaches the set-point.
$\boldsymbol{V_{Inlet\_1}}$	Output	Inlet Solenoid Valve	Allows water into tank
$\boldsymbol{V_{Outlet\_1}}$	Output	Outlet Solenoid Valve	Allows glue out of tank
$\boldsymbol{Pump_1}$	Output	Mixing Pump	Mixes tank's content
$\boldsymbol{Heater_1}$	Output	Water Heater	Heats tank's content

Table 5.2 Overview of the holding-tank variables

Variable	IO	Sensor/Actuator Name	Description
$\boldsymbol{L_{Min\_2}}$	Input	Minimum Glue-Level Sensor	Detects the minimum glue level. Outputs *high* if glue covers the sensor.
$\boldsymbol{T_2}$	Input	Temperature Sensor	Detects the glue temperature. Outputs *high* if temperature reaches the set-point.
$\boldsymbol{T_{L\_I\_2}}$	Input	On-Delay Long Timer	Determines whether the long countdown has elapsed. Outputs *high* after 10 seconds.
$\boldsymbol{T_{S\_I\_2}}$	Input	On-Delay Short Timer	Determines whether the short countdown has elapsed. Outputs *high* after 5 seconds.
$\boldsymbol{V_{Outlet\_2}}$	Output	Outlet Solenoid Valve	Allows glue out of tank
$\boldsymbol{Heater_2}$	Output	Water Heater	Heats glue
$\boldsymbol{T_{L\_O\_2}}$	Output	Long Timer	Activates or resets the long timer
$\boldsymbol{T_{2\_O\_2}}$	Output	Short Timer	Activates or resets the short timer

In addition to the holding tank's control components, the tank also makes use of two timers. These timers are not represented by external sensors or actuators, but rather by timing circuitry found within the control circuit itself. Thus, the timers do not need, or have, external IOs on the control circuit. Nevertheless, Table 5.2 has included the timers' variables for completeness.

5.3 System Operation and Analysis

From the system components, the tanks' operation can now be analysed. The system will be modelled as a FSM, using two states—the mixing-tank and holding-tank states. Each tank subsystem will be explained independently before examining the system as a whole.

5.3.1 State One: The Mixing Tank

Consider the mixing tank's flowchart, shown in Figure 5.3.

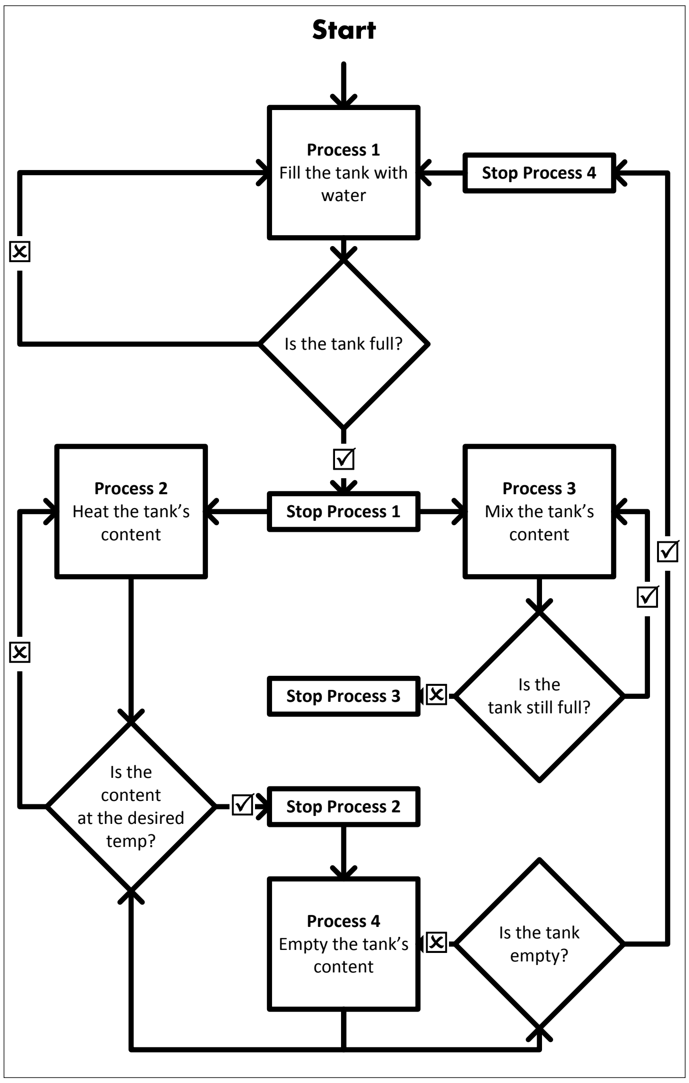


Figure 5.3 Logic flowchart of the mixing tank

To produce the required glue, the mixing tank makes use of four processes:

1. Fill the tank with water
2. Heat the tank's content
3. Mix the tank's content
4. Empty the tank

Some of the above processes may run in parallel, and are not necessarily executed sequentially. For example, the tank's content may be heated and mixed concurrently.

Generally, when operating correctly, the mixing state executes as follows:

1. First, the state starts by filling the mixing tank with water.

2. Then, once filled, both the mixing and heating processes are executed simultaneously. The mixing process is executed on condition that the tank is full; the heating process is executed if the tank's temperature is under the set-point, regardless if the tank is full or empty.
3. The tank's content is only emptied to the holding tank while the content is at the desired temperature—if not, the heater will switch on. The above process is cyclic and can be repeated unconditionally.

Figure 5.4 shows a schematic of the mixing tank's active components during each process.

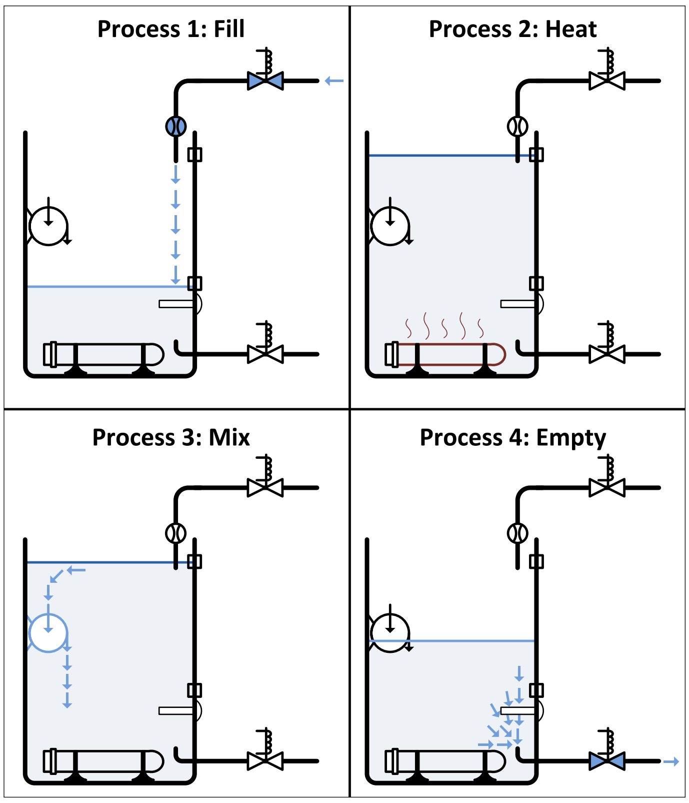


Figure 5.4 Schematic of mixing-tank processes

5.3.2 State Two: The Holding Tank

Now, consider the holding tank's flowchart, shown in Figure 5.5.

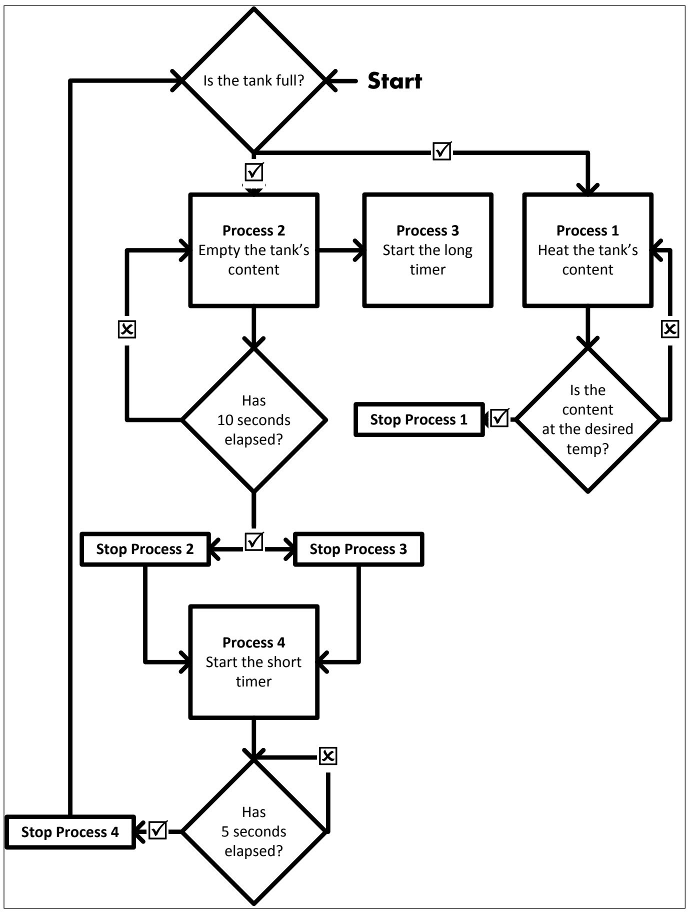


Figure 5.5 Logic flowchart of the holding tank

The holding subsystem also makes use of four processes:

1. Heat the tank's glue
2. Empty the tank
3. Start the long timer
4. Start the short timer

Unlike with the mixing tank, the holding-tank subsystem starts with a full tank since it is first filled by the mixing tank. The control circuit then continually checks that the glue's temperature is at the desired temperature. If the glue is under heated, the heater is activated (process 1), and expulsion is halted. No under-heated glue is allowed to be emptied from the tank.

For process two—empty the tank—the glue is emptied according to a timer-activated cycle. Glue is expelled for ten seconds with five second pauses between the expulsions, i.e. glue will be expelled for ten seconds, followed by a five second break, then expelled again for ten seconds, so on and so forth. Two timers are used to do this—a long timer measuring ten seconds, and a short timer measuring five seconds.

Figure 5.6 shows a schematic of the holding tank's active-components during the first two processes. Processes three and four—activating the timers—have not been included in the figure since the timers have no external components.

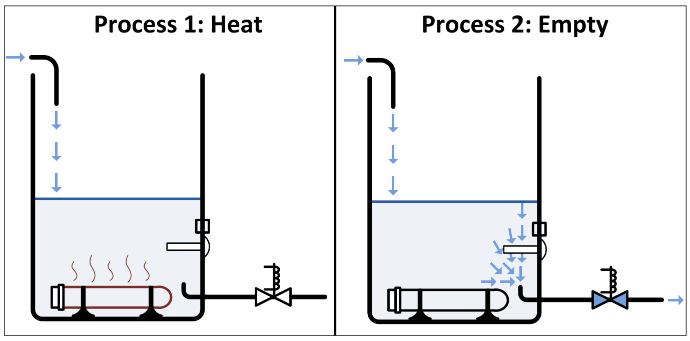


Figure 5.6 Schematic of two holding-tank processes

Finally, it should be noted that state two is also cyclic, in that the glue is continually emptied at timed intervals. However, unlike state one's subsystem which can execute unconditionally, state two's subsystem relies on state one to fill the holding tank. Hence, state two only executes on condition that the holding tank is full, i.e. the minimum water-level sensor needs to be activated.

5.4 Modelling the Finite-State Machine

Now that the operation of both state subsystems has been described, the complete system can now be examined. This will be done by first developing the system's state diagram, and then developing a circuit block diagram.

5.4.1 State Diagram

Figure 5.7 shows a basic state diagram of the tank system with three important sets of data:

1. The sequence in which the states are executed
2. The conditions for the current state to be executed
3. The conditions to transition from one state to the next

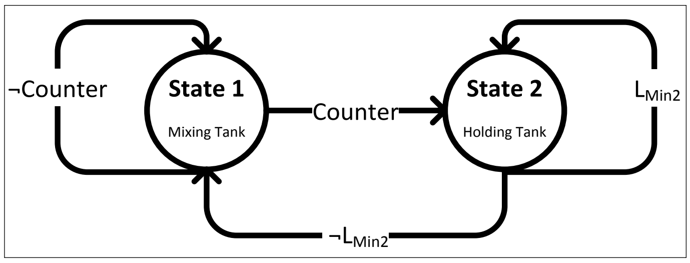


Figure 5.7 State diagram of the tank system

The diagram makes use of two transition variables, namely $Counter$ and L_{Min2}. These transition variables are defined as follows:

- $\boldsymbol{Counter}$: The $Counter$ variable is $true$ if state one has been executed a predefined n number of times. n has arbitrarily been set to one, i.e. state one executes once before setting $Counter$ to $true$ and transitioning to state two.
- $\boldsymbol{L_{Min2}}$: The L_{Min2} variable is $true$ if the minimum glue-level sensor in the holding tank is activated, i.e. the holding tank contains glue. If there is no glue in the holding tank, L_{Min2} becomes $false$ thereby transitioning to state one.

Table 5.3 shows the possible transition-variable combinations, as well as the active state for each combination. In addition, since the FSM's states are activated in a sequence, each transition-variable combination can only transition to the combination shown in the "Next Possible Input" column.

Table 5.3 Possible transition-variable combinations

Decimal Value of Variables (Decimal Input)	$\boldsymbol{Counter}$	$\boldsymbol{L_{Min2}}$	**Active State**	**Next Possible Input**
0	0	0	1	1
1	0	1	1	3
2	1	0	-	0
3	1	1	2	2

Under normal conditions, the FSM executes as follows:

- **Decimal Input 0:** The FSM starts with state one being active. Both the counter and minimum glue-level sensor output *low*.
- **Decimal Input 1:** As state one is executed, glue is pumped into the holding tank, making L_{Min2} $true$.
- **Decimal Input 3**: Finally, once state one has finished executing $n = 1$ times, the $Counter$ variable is set $true$, thereby activating state two. State two executes until all the glue from the holding tank is expelled, i.e. L_{Min2} is $false$.
- **Decimal Input 2:** If $Counter = true$ and $L_{Min2} = false$, the counter can be reset, thereby making the $Counter$ variable $false$ again. Thus, the cycle is repeated, with "Decimal Input 0" being the next combination to execute.

5.4.2 Block Diagram

By using the FSM state diagram as a guide, the system's control circuit can now be modelled. Figure 5.8 shows a block diagram of the control circuit, which can be implemented on an FPGA or ASIC. The system consists of five sub-circuits: combinational logic, sequential logic, counter circuit and two timing circuits.

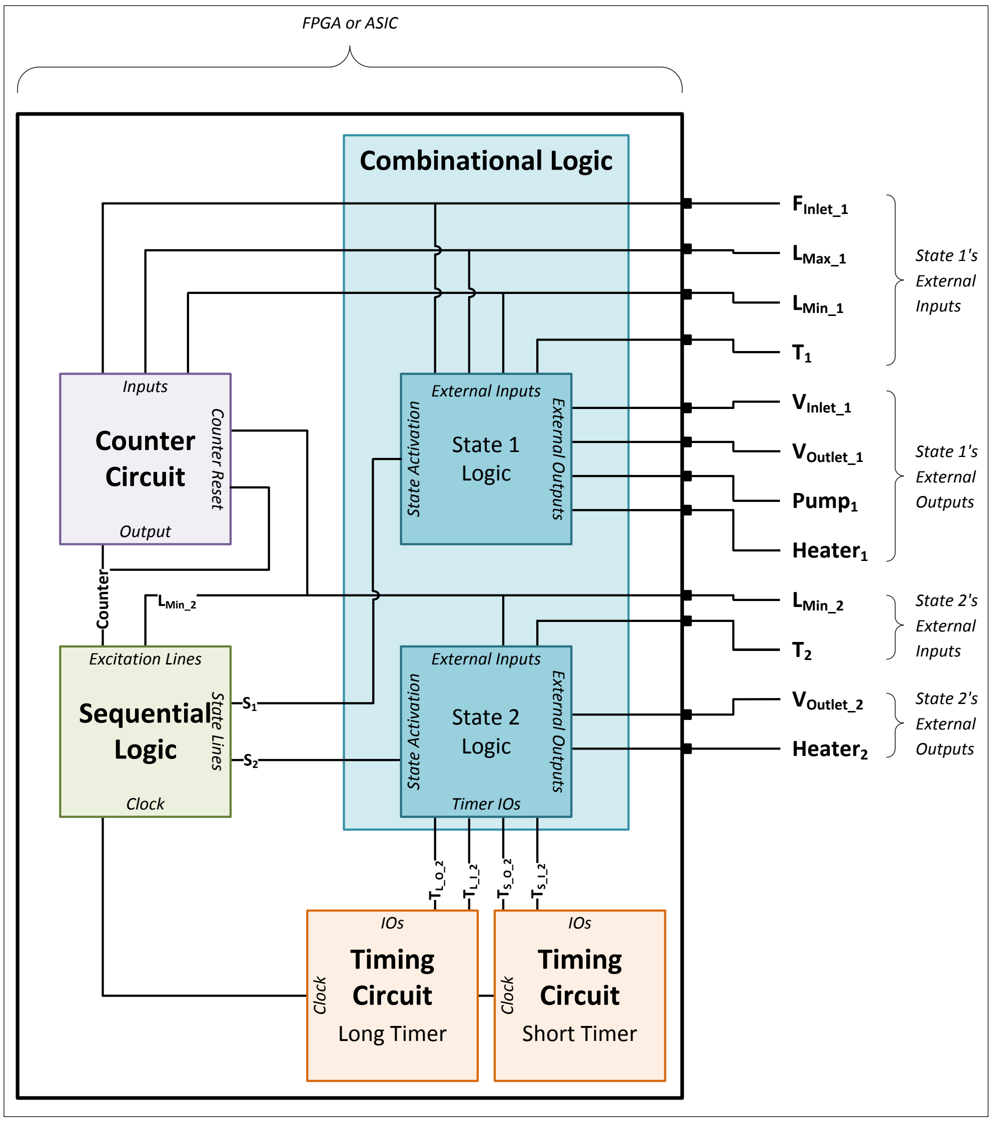


Figure 5.8 Block diagram of the system's control circuitry

The combinational logic portion, implement using Boolean logic, connects directly to the system's sensors and actuators. It controls the system's external outputs according to the logic combination on the external inputs, timer inputs and state lines.

The timing circuits act as on-delay timers, i.e. when the $T_{L\_O\_2}$ or $T_{S\_O\_2}$ lines are set to *high*, a predetermined amount of time elapses before the $T_{L\_I\_2}$ or $T_{S\_I\_2}$ lines are also set to *high.* To reset the timers, the $T_{L\_O\_2}$ or $T_{S\_O\_2}$ lines are set to *low*. There are various ways in which the long and short timers can be implemented. Both digital and analogue circuit configuration exist, which make use of components such as 555 ICs, transistors, capacitors, oscillators and MOSFETS (see (CircuitoZ, n.d.) and (Electronic Project Circuits, 2012)).

The counter circuit, implemented using flip-flops, counts the number of times the mixing tank has been filled and emptied, i.e. the number of cycles the mixing tank has executed as explained in Section 5.3.1 . The mixing tank only needs to complete one cycle for the counter's $Counter$ line to be set *high*. To do this, the counter is activated according to three sensor inputs: F_{Inlet1}, L_{Max1} and L_{Min1}; and reset with lines $Counter$

and L_{Min2}. The three input lines need to be *low* in order to increment the counter (discussed further in Section 5.5.1.1), while the reset lines, $Counter$ and L_{Min2}, need to be *high* and *low* respectively to reset the counter (decimal 2 in Table 5.3).

Finally, the sequential circuit forms the central control of the FSM. Unlike the combinational circuit, the sequential circuit consists of a memory section (flip-flops) which stores the current state. To determine whether to transition to the next state, two excitation lines are used. These excitation lines represent the transition variables in the state diagram (Figure 5.7). Thus, they are connected to the counter and minimum glue-level sensor. According to the active state, either state line S_1 or S_2 is *high*. (Note: S_1 or S_2 cannot be *high* simultaneously.) S_1 controls state one's combinational logic, while S_2 controls state two's combinational logic.

5.5 Evolving the Finite-State Machine's Combinational Logic

In Chapter 4 it was shown how the V-FPGA and GA had been setup and optimised to evolve a four-input/four-output circuit. One of the evolution constraints was that no memory elements were allowed in the evolved phenotype, resulting in feedback loops being prohibited. Thus, only combinational logic (and not sequential) could be evolved.

Following on from the previous chapters, the same EHW setup, used to evolve the $1+\lambda$ multiplier, can now be used to evolve the combinational logic of the FSM. In addition, the reader is reminded about the following: The $1+\lambda$ GA will make use of the same evolutionary constraints, elitism, mutation and $Fitness_{Overall}$ fitness function (from Equation 4.5).

5.5.1 The States' Combinational Logic

To evolve the FSM's combinational circuit, a truth table used by the fitness function, representing the outputs for each possible external-input combination, is required for each state.

5.5.1.1 State One

Consider Table 5.4, which shows the truth table used to control the mixing tank's inlet and outlet valves, heater and pump. Table 5.4 has been designed in such a manner as to allow the processes to be executed in the desired sequence.

Table 5.4 State one's truth table

Decimal Value of Inputs	L_{Max1}	L_{Min1}	F_{Inlet1}	T_1	V_{Inlet1}	$V_{Outlet1}$	$Heater_1$	$Pump_1$
0	0	0	0	0	1	0	1	0
1	0	0	0	1	1	0	0	0
2	0	0	1	0	1	0	1	0
3	0	0	1	1	1	0	0	0
4	0	1	0	0	0	0	1	0

5	0	1	0	1	0	1	0	0
6	0	1	1	0	1	0	1	0
7	0	1	1	1	1	0	0	0
8	1	0	0	0				
9	1	0	0	1				
10	1	0	1	0				
11	1	0	1	1				
12	1	1	0	0	0	0	1	1
13	1	1	0	1	0	1	0	1
14	1	1	1	0	0	0	1	1
15	1	1	1	1	0	0	0	1

A total of sixteen input combinations are shown, of which four input combinations are not possible. These not-possible inputs occurs when L_{Max1} is $true$ while L_{Min1} is $false$; and are invalid because the water cannot activate the maximum water-level sensor without also activating the minimum water-level sensor. Thus, the outputs of the not-possible inputs are irrelevant, i.e. they can be $true$ or $false$ (indicated using blank cells in Table 5.4).

The sequence of processes that occur in Table 5.4 is clarified in Table 5.5. Table 5.5 shows the active process as well as the next-possible input combinations to which each input can transition. For example, input twelve will turn both $Pump_1$ and $Heater_1$ *on* (process two and three), and can transition to input thirteen.

Table 5.5 Next-possible-input-combination table for state one

Decimal Value of Inputs	Current Process	Next-Possible Input	Description
0	1 and 2	1 or 2	V_{Inlet1} and $Heater_1$ *on*
1	1	0 or 3	V_{Inlet1} *on*
2	1 and 2	3 or 6	V_{Inlet1} and $Heater_1$ *on*
3	1	2 or 7	V_{Inlet1} *on*
4	2	5	$Heater_1$ *on*
5	4	1 or 4	$V_{Outlet1}$ *on*
6	1 and 2	7 or 14	V_{Inlet1} and $Heater_1$ *on*
7	1	6 or 15	V_{Inlet1} *on*
12	2 and 3	13	$Pump_1$ and $Heater_1$ *on*
13	3 and 4	5 or 12	$Pump_1$ and $V_{Outlet1}$ *on*
14	2 and 3	12 or 15	$Pump_1$ and $Heater_1$ *on*
15	3	13 or 14	$Pump_1$ *on*

The above sequence of processes is graphically shown in Figure 5.9. The figure draws similarities from Figure 5.7's state diagram. Each input has a transition variable which, when toggled, allows the current input to transition to the next input. For example, for input seven to transition to six, T_1 has to be negated, i.e. the water temperature needs to drop below the set-point. Likewise, for input seven to transition to fifteen, L_{Max1} has to be $true$, i.e. the tank must be filled to its maximum level.

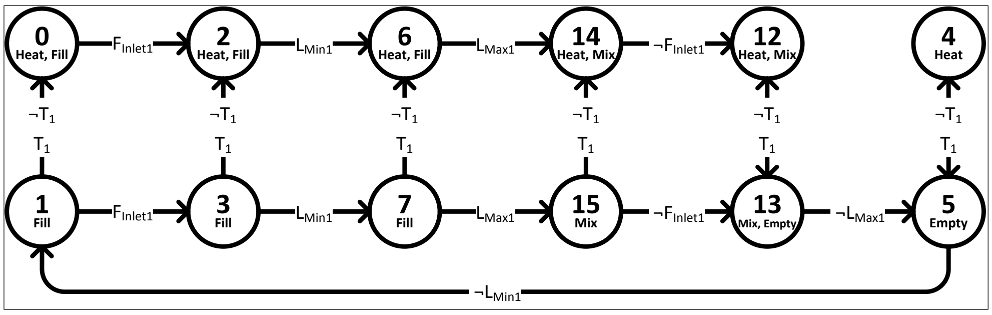


Figure 5.9 State one's input-combination diagram

The above input-combination diagram shows, as previously discussed, that the mixing tank's processes are indeed cyclic. According to the T_1 variable, either input zero or one will be the first active input; with the final active input being input five.

Finally, recall from Figure 5.8's block diagram that the counter tallies the number of cycles the mixing subsystem executes. To do this, the counter must incremented every time input zero or one is activated. This occurs when F_{Inlet1}, L_{Max1} and L_{Min1} are $false$. (T_1 is irrelevant since the subsystem can start on either input zero or one.) Hence, F_{Inlet1}, L_{Max1} and L_{Min1} form the counter's inputs and are used to identify the start of a new cycle.

5.5.1.2 State Two

Table 5.6 shows the holding tank's truth table. As for the mixing tank, different input combinations result in different processes being executed.

Table 5.6 State two's truth table

Decimal Value of Inputs	L_{Min2}	T_2	$T_{L\_I\_2}$	$T_{S\_I\_2}$	$V_{Outlet2}$	$T_{L\_O\_2}$	$T_{S\_O\_2}$	$Heater_2$
0	0	0	0	0				
1	0	0	0	1				
2	0	0	1	0				
3	0	0	1	1				
4	0	1	0	0				
5	0	1	0	1				
6	0	1	1	0				
7	0	1	1	1				
8	1	0	0	0	0	1	0	1
9	1	0	0	1	0	0	0	1
10	1	0	1	0	0	1	1	1
11	1	0	1	1	0	0	1	1
12	1	1	0	0	1	1	0	0
13	1	1	0	1	0	0	0	0
14	1	1	1	0	0	1	1	0
15	1	1	1	1	0	0	1	0

A total of eight not-possible input combinations occur. These are all based on L_{Min2} being $false$. Since L_{Min2} is also the FSM's transition variable (see Figure 5.7), when L_{Min2} toggles to $false$, the FSM immediately transition to state one. Thus, state two only executes while L_{Min2} is $true$.

Table 5.7 shows the next-possible-input combination of each input. Again, each input activates certain processes in the tank. For example, input fourteen turns both the short and long timers *on* (process three and four).

Table 5.7 Next-possible-input-combination table for state two

Decimal Value of Inputs	Current Process	Next-Possible Input	Description
8	1 and 3	10 or 12	$Heater_2$ and $T_{L\_O\_2}$ *on*
9	1	8 or 13	$Heater_2$ *on*
10	1 and 3 and 4	11 or 14	$Heater_2, T_{L\_O\_2}$ and $T_{S\_O\_2}$ *on*
11	1 and 4	9 or 15	$Heater_2$ and $T_{S\_O\_2}$ *on*
12	2 and 3	8 or 14	$V_{Outlet2}$ and $T_{L\_O\_2}$ *on*
13	Reset	9 or 12	Reset all processes
14	3 and 4	10 or 15	$T_{L\_O\_2}$ and $T_{S\_O\_2}$ *on*
15	4	11 or 13	$T_{S\_O\_2}$ *on*

The above sequence of processes is graphically shown in Figure 5.10. Notice, in the figure, that the holding tank's subsystem is also cyclic on condition L_{Min2} is $true$.

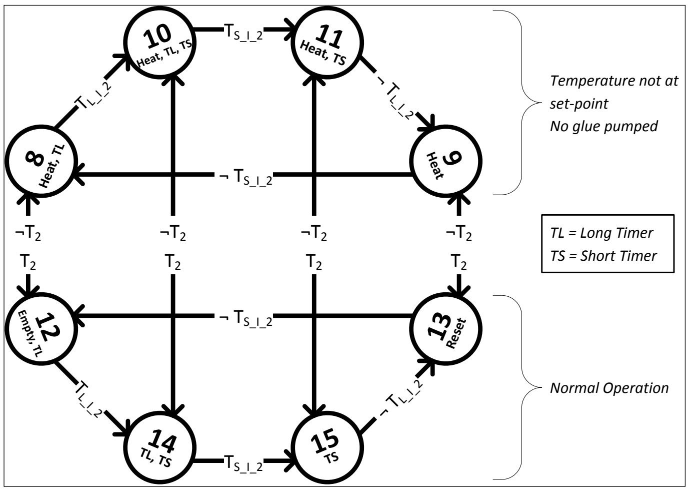


Figure 5.10 State two's input-combination diagram

There are two sub-cycles shown in Figure 5.10: The one sub-cycle—inputs eight, ten, eleven and nine—does not allow glue to be pumped, while the other cycle—inputs twelve, fourteen, fifteen and thirteen—does. The latter cycle executes under normal conditions, i.e. when the glue's temperature is above the set-point. Thus, it is possible to transitions between the two sub-cycles by toggling the T_2 variable.

5.6 Results

The successful evolution results, for each state, are discussed below. Like with the evolved multipliers, each evolution attempt was limited to 3000 generations. Not all evolution attempts were successful: State one was evolved seven times, while state two was evolved four times before a finding a fit phenotype.

As discussed in Chapter 4, the $1 + \lambda$ GA is more prone to downloading errors (see Section 4.5.4.3 , page 71), hence there are spikes in the graphs (Figure 5.11 and Figure 5.15) which have been demarcated with **×**. In addition, there are large jumps in the graphs due to the F_{CP} variable in the $Fitness_{Overall}$ fitness evaluation.

For referencing purposes, the independent CPs have been coloured red.

5.6.1 State One

Figure 5.14 shows the evolution of state one's 100%-fit phenotype. Within the first 39 generations, two of the four CPs were evolved, with the third and fourth CPs taking 1890 and 2964 generations respectively.

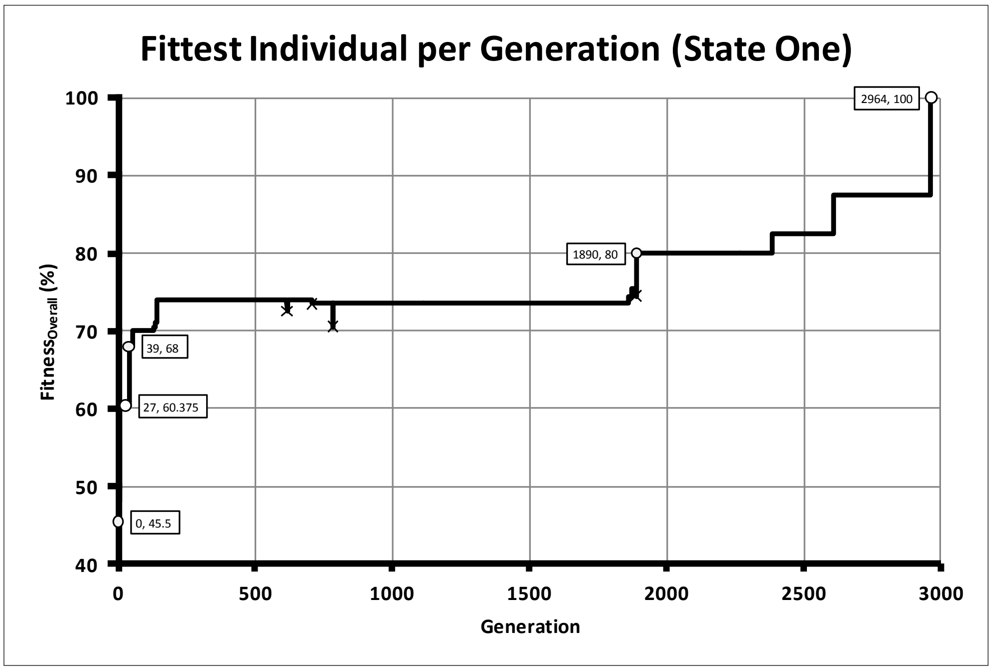


Figure 5.11 Results of state one's evolution

The following milestones are also noted in Figure 5.11's graph:

- At fitness values 60.4%, 68%, 80% and 100%, the outputs $Pump_1$, $Heater_1$, $V_{Outlet1}$ and V_{Inlet1} respectively are correctly evolved.
- It took 17 hours and 44 minutes to execute all 2964 generations.

As expected, Figure 5.12 shows why the first two CPs, $Pump_1$ and $Heater_1$, evolved quickly—the CPs only pass one LE. In contrast, $V_{Outlet2}$ and V_{Inlet2} pass five and six LEs respectively. In total, ten of the twenty available LEs were used, with five of the ten LEs placed in column zero.

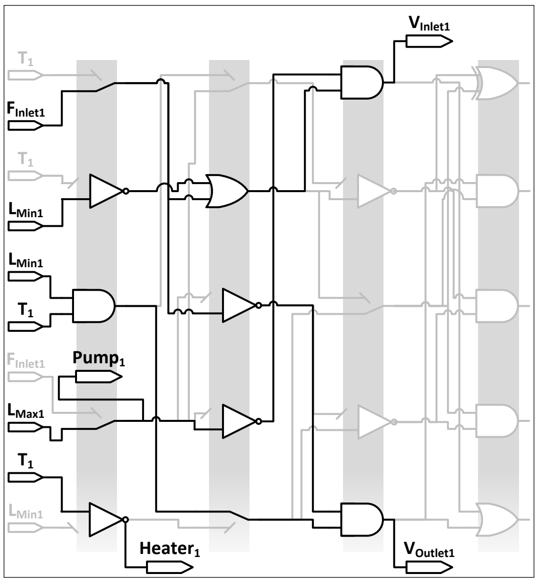


Figure 5.12 State one's final phenotype

The final simplified phenotype is shown in Figure 5.13, using only eight of the original ten LEs. Notice that $Pump_1$ and $V_{Outlet1}$ are independent of each other, but dependent on V_{Inlet1}, i.e. $Pump_1$ and $V_{Outlet1}$ were independent CPs until V_{Inlet1} was evolved. The significance of this is discussed in later in Section 5.7.1 .

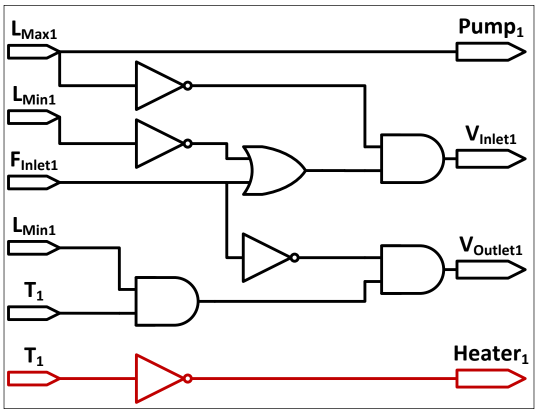


Figure 5.13 State one's simplified phenotype

5.6.2 State Two

Figure 5.14 shows the evolution of state two's 100%-fit phenotype.

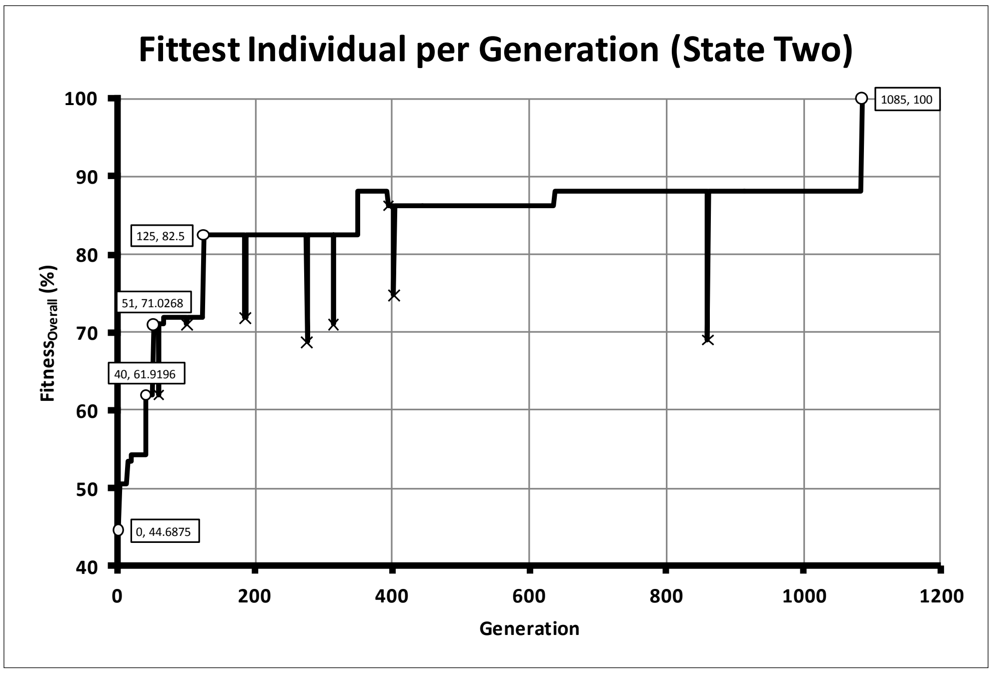


Figure 5.14 Results of state two's evolution

Again, the following milestones are noted in Figure 5.14's graph:

- At fitness values 61.9%, 71%, 82.5% and 100%, the outputs $Heater_2$, $T_{L\_O\_2}$, $T_{S\_O\_2}$ and $V_{Outlet2}$ respectively are correctly evolved. The $Heater_2$, $T_{L\_O\_2}$ and $T_{S\_O\_2}$ CPs pass one LE while $V_{Outlet2}$ CP passes six.
- It took a total of 1085 generations to evolve the phenotype. This amounted to an evolution time of approximately 6 hours and 29 minutes.

The evolved phenotype, shown in Figure 5.15, makes use of eight LEs. As with state one, note the distribution of used LEs: All five LEs in column zero, but no LEs in column three, have been used.

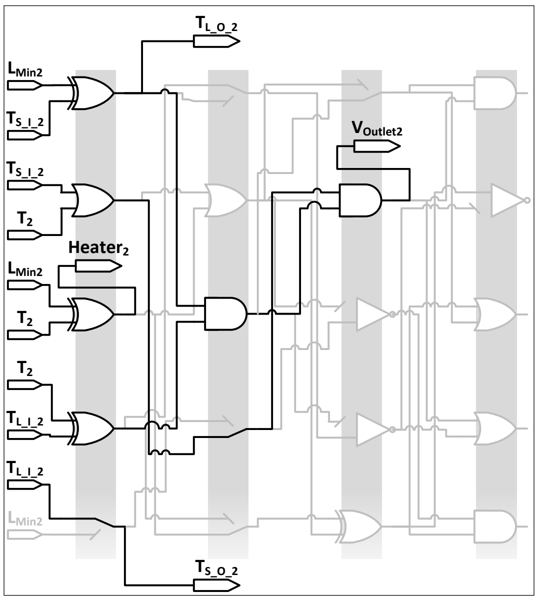


Figure 5.15 State two's final phenotype

Finally, Figure 5.15 can be simplified by removing the wire LEs, as shown in Figure 5.16. The simplified phenotype uses only six of the eight LEs.

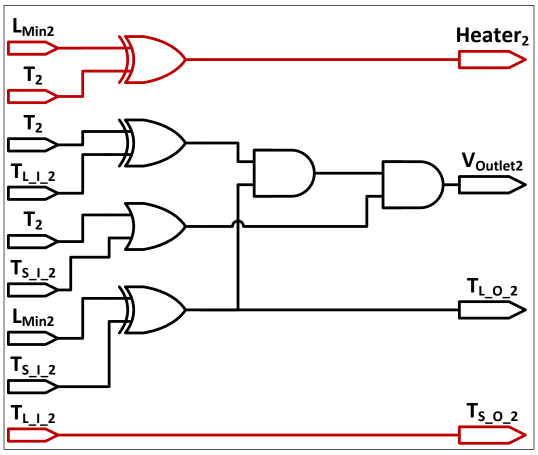


Figure 5.16 State two's simplified phenotype

5.7 Discussion

5.7.1 The Evolved Critical Paths

As done in Chapter 4, Table 5.8 shows the net number of LEs that were evolved for each CP. Unlike with the evolved multipliers, notice that the state-one CPs were not all evolved in ascending order of the net number. In particular, $V_{Outlet1}$ was evolved before V_{Inlet1} even though their net numbers are five and four respectively. This is because the $Pump_1$ and $V_{Outlet1}$ CPs are independent of each other, but are both dependent to V_{Inlet1}.

During evolution, two or more initially independent CPs, or pre-independent CPs, can be linked together to form a third CP, thereby making all the CPs dependent. The third CP will take longer to evolve since the pre-independent CPs first need to be evolved; but will require fewer net LEs because the CP will be dependent. This explains why the V_{Inlet1} CP, although last to evolve, has a lower net-number.

Table 5.8 Summary of the net number-of-LEs evolved by each CP

	State-One's Phenotype				State-Two's Phenotype			
	$\boldsymbol{Pump_1}$	$\boldsymbol{Heater_1}$	$\boldsymbol{V_{Outlet1}}$	$\boldsymbol{V_{Inlet1}}$	$\boldsymbol{Heater_2}$	$\boldsymbol{T_{L\_O\_2}}$	$\boldsymbol{T_{S\_O\_2}}$	$\boldsymbol{V_{Outlet2}}$
Number of evolved LEs used by the CP	1	1	5	6	1	1	1	6
Number of LEs, used by the CP, that were previously evolved by other CPs	0	0	0	2	0	0	0	1
Net number of LEs that were evolved (Net evolved LEs)	1	1	5	4	1	1	1	5

All CPs with one net LE were evolved within the first 200 generations. In contrast, the $V_{Outlet1}$ and $V_{Outlet2}$ CPs both have a net number of five—the highest of all the CPs evolved in this study. Both took over 1000 generations to evolve, with the $V_{Outlet1}$ CP taking 1890 generations.

The above results are in concurrence with the results of the multiplier CPs in Chapter 4—the more net LEs a GA has to evolve, the longer the evolution takes (with the exception of pre-independent CPs).

5.7.2 Modular Evolution and Scalability

From the simplified phenotypes, the complete combinational-logic circuit for the FSM can now realised by adjoining the sub-circuits as shown in Figure 5.17. The complete circuit makes use of 10 inputs, 8 outputs and 22 LEs.

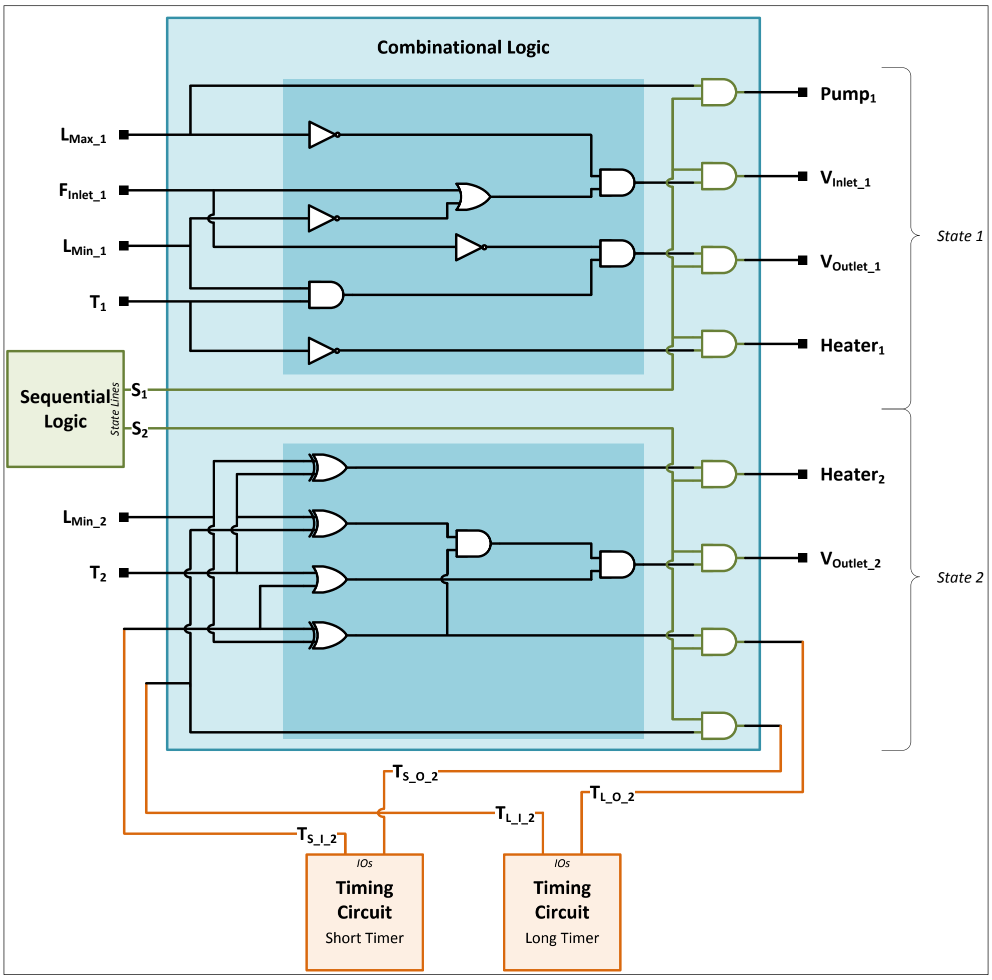


Figure 5.17 Complete combinational-logic of the FSM

Of the 22 LEs, eight are AND-gates (coloured green in Figure 5.17) connected to the sequential logic's state lines. These AND-gates have direct control over the external outputs. For example, if state one is active, i.e. S_1 is *high* and S_2 is *low*, then all of state one's external outputs will be *on*/*off* according state one's combinational logic, while state two's external outputs will all be *off*.

Comparing the complete combinational circuit and the state's evolved circuits, the complete circuit is much larger, having over double the number of inputs, external outputs and LEs. However, even so, the majority of the circuit has been autonomously evolved, with only minimal human expertise needed only to adjoin the sub-circuits. Had the complete circuit been evolved using the 20-LE V-FPGA, no solution would be evolved due to there not being enough available LEs, or IOs. A larger V-FPGA could be employed, with more LEs, external outputs and inputs. However, this would substantially enlarge the GA's search space. Both examples would result in an increase in scalability problems, with inevitable long evolution-times and poor repeatability.

Consequently, the modular evolution has successfully demonstrated a method of dealing with scalability—by decomposing circuits into sub-circuits and evolving the sub-circuits at gate-level.

Nevertheless, future work will still need the designer to have insight into the EHW setup's capabilities in order to select the size of each sub-circuit and V-FPGA. Sub-circuits can, of course, be any size. For example, it is possible to evolve each CP independently; or larger state combinational circuits. But large sub-circuits should, again, be used cautiously, since scalability problems can occur even when using modular evolution.

Finally, the above modular evolution demonstrated state decomposition, i.e. each state's sub-circuit is independently evolved. Although decomposition strategies, as discussed in Chapter 2, have proven successful (Kalganova, 2000), a major problem is defining the decomposition, i.e. defining how the circuit should be dividing into sub-circuits. The size of each sub-circuit in this research is primarily based on each state's requirements. However, more complex states would require further decomposition.

Stomeo, et al (2006) have proposed a new method called "Generalised Disjunction Decomposition", or GDD, where a circuit is decomposed according to the inputs. This is based on the fact that the number of generations required to evolve a circuit is directly influenced by the number of external inputs. In fact, Stomeo, et al (2006) showed that a 15-input circuit can take ten times longer to evolve than a 10-input circuit.

Future work can consider using GDD when evolving larger state sub-circuits, especially since the GDD research has shown usefulness in combinational-logic evolution. In addition, Stomeo, et al (2006) have concluded that GDD-evolved circuits have reached "higher values of fitness during optimisation", and are thus more efficient.

5.7.3 The Distribution of Logic Elements and External Outputs

CPs make use of multiple external inputs, but only one external output, i.e. CPs taper towards their outputs. This is evident when analysing Table 5.9. The table summarises the distribution of the external outputs and the used LEs for all the evolved CPs when using the 20-LE CGP layout.

Table 5.9 The distribution-of-external-outputs and used-LEs per CGP column

	Distribution of External Outputs				**Number of used LEs per Column**			
CGP Column	**0**	**1**	**2**	**3**	**0**	**1**	**2**	**3**
Trial-Canonical Phenotype	0	3	0	1	4	4	1	1
Final-Canonical Phenotype	1	2	1	0	4	4	1	0
$1 + \lambda$ Phenotype	1	1	1	1	4	4	2	1
State One's Phenotype	2	0	2	0	5	3	2	0
State Two's Phenotype	3	0	1	0	5	2	1	0
Total	7	6	5	2	22	17	7	2
Percentage	35%	30%	25%	10%	88%	68%	28%	8%

From the first set of data, i.e. the distribution of external outputs, 35% of external outputs were placed in column zero. This means that 35% of the CPs used only one LE. The percentage steadily decreases as the column number increases, with only 10% of the evolved CPs having external outputs in column three. Column three's external outputs used LEs from all four columns.

A similar trend is noted in the second set of data, which shows the number of used LEs per column. The reader is reminded that there are five available LEs per column. For column zero, 88% of the available LEs were used by the CPs. On the contrary, only 8% of the LEs in column three were utilised.

Overall, only 48% of the available LEs were used. This is not necessarily a negative point, as the evolution process needs unused LEs, when evolving novel solutions, in order to prevent scalability. However, the distribution of these unused LEs can be improved upon by assigning more LEs to the columns that are likely to use the LEs.

Now, consider the state-one and state-two phenotypes, which both made use of all five LEs in column zero, but no LEs from column three. There are a number of consequences to consider for using the 20-LE CGP layout:

- Since there are no free LEs in column zero, it remains unproven, but possible, that more efficient CPs may be evolved if more LEs are made available in these columns. Column zero's LEs are also important since the evolution constraints only allow external inputs to be connected to these LEs.
- Since more of the column-zero LEs are utilised, there is a higher probability of the GA erroneously altering a correctly evolved column-zero LE, while trying to evolve the remaining CPs.
- Since the last column does not feature in the either of the evolved phenotypes. Thus, the column is unnecessary and only adds to the search space and evolution time.

Consequently, the standard CGP layout of LEs is questionable. One solution to this problem could be to use the proposed pyramid layout of LEs, as shown in Figure 5.18.

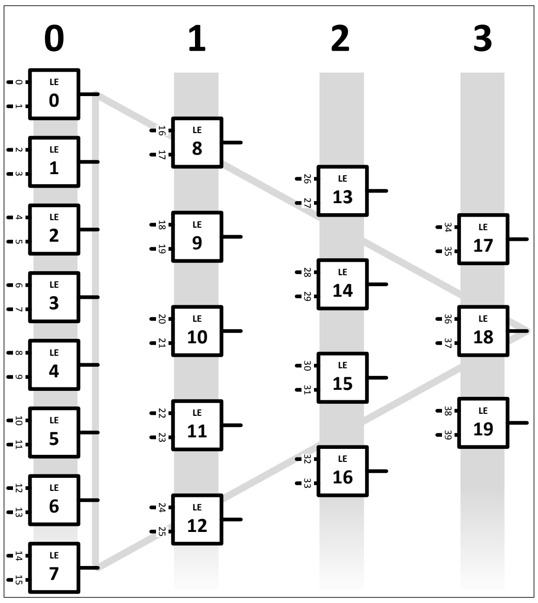


Figure 5.18 Proposed pyramid layout of LEs

The exact size of each column in the pyramid is left to the designer, but the idea is to arrange the LEs in a pyramid or triangular formation. The layout does not reduce the GA's search space, but rather places more emphasis on the external-input LEs in column zero. Since there are more LEs in the first column, there is a higher probability, i.e. eight in twenty chance, of an external output being connected to LE 0 to 7. Likewise, there is a low probability, i.e. only three in twenty chance, of LE 17 to 19 being connected to an external output. Thus, the GA is more likely to connect the external outputs to correctly evolved CPs. In addition, each column will proportionally have more available LEs, thereby preventing scalability problems.

Chapter 6

CONCLUSION

6.1 Introduction

This study set out to explore the implementation of an EHW system as an alternative method of synthesising digital circuits. Three key issues have been investigated: the design and implementation of a custom VLSI architecture; the effectiveness of optimising the GA and its genetic operators; the evolution of a real-world control circuit. More specifically, from the investigated issues, the research sought to answer the following questions:

1. Although theoretically proven, practically could the V-FPGA be a viable evolution platform?
2. Do $1 + \lambda$ GAs provide advantages over canonical GAs when used in EHW systems?
3. Could a simple FSM be scaled and evolved if modular evolution was used?

The above questions related closely to two pressing issues found in EHW systems: portability and scalability.

6.2 Contributions

According to the literature, portability was mainly being solved using VRCs. Although the VRC concept was first introduced in the early 2000s, it was evident in many current research papers that VRCs were still popular and relevant due to their device independence, cost minimisation and flexibly. But flexibility also meant that VRC architectures were not standardised, differing according to their design, granularity, implementation and size.

The V-FPGA architecture is one example of a VRC, which before this research, had not yet been implemented on an FPGA. Generally it was found that although it was advantageous to understand how the V-FPGA's LCM, RCM, RM and multiplexers operated, it was more critical for the designer to understand how to program the LCM and RCM using the write-enable, write-address, data, routing and clock lines. Through the successful simulation of a multiplier, it was shown that: the LCM and RCM are independent entities, and could thus be programmed individually using parallel communication; the clock and write-enable lines should only be active while programming the memories.

Overall, the simulation proved the architecture to be a viable platform on which to perform hardware evolution. Thus, the first question of the study has been shown to be true.

The analysis of Chapter 3 led to a new genotype representation in LabVIEW software. This representation could serve as a model for future V-FPGA studies, since it was able to represent all the necessary $WA_{Routing}$,

$D_{Routing}$, WA_{Logic} and D_{Logic} data needed for genotype-phenotype mapping, while still placing special attention on the prohibition of illegal connections.

Besides portability, it was evident from numerous sources that scalability was an even more pressing problem prohibiting real-world EHW applications, with side-effects including poor performance, evolution-times and repeatability. The literature suggested that scalability be overcome using a multifaceted approach, which was partly done in Chapters 4 and 5 by optimising the GAs' operators, using evolution constraints and decomposing the solution circuit.

One noteworthy contribution this research has made is the in-depth analysis of the phenotypes' CPs. Through analysing the CPs, it has been shown that a great amount of insight can be gained into a phenotype's fitness. Particularly, the identification of the CP's dependence is valuable, since dependent CPs reduced the required net number of evolved LEs.

Generally, in both the multiplier and state phenotypes, the CPs were evolved in ascending order of the net LEs. This suggests that evolution always favoured CPs with lower net numbers. However, we have seen that in one special case, if two independent CPs are used by a third CP, the resulting third CP has a lower net number than both independent CPs.

The CP analysis also led to the development of the $Fitness_{Overall}$ fitness function, which had a distinctive way of not only rewarding correct out elements, but also encouraging more efficient evolution through sustaining evolved CPs, and further developing partially-evolved CPs.

Like the fitness function, the development of the GAs' parameters was largely achieved on a trial-and-error basis. The empirical findings in this study provided a new understanding of:

- The scalability consequences for configuring V-FPGAs that were too large or small.
- Constrained evolution, and how it could be used to minimise the number of routing and logic permutations, thereby reducing the GA's search space.
- How poor repeatability, of which the stalling effect was an indicator, could be linked to scalability, erroneously-evolved chromosomes and inaccurate downloads.

By comparing the final-canonical and $1 + \lambda$ results, the study confirmed previous findings that suggest the $1 + \lambda$ GA is more effective in EHW. It was shown that $1 + \lambda$ GA was substantially quicker to evolve a solution, and found a solution within a fewer number of attempts, when compared to the canonical GA. Thus, although the study was based on a small sample of evolution attempts, the results imply the $1 + \lambda$ GA is more suitable and efficient for EHW. It was discussed that this was possibly due to crossover reducing the canonical system's evolvability, and that the $1 + \lambda$ GA's smaller population size was better suited for EHW systems. Having argued that the $1 + \lambda$ GA has shown more favourable results, the second question of the study has thus been addressed and shown true.

An interesting and valuable finding to emerge from the study is that all five evolved phenotypes had unique topologies. Further analysis of the multiplier phenotypes also revealed that none of the evolved phenotypes were less efficient than the conventional multiplier. In fact, it was shown that two of the three phenotypes reduced the total number of LEs, with the third phenotype reducing the number of used LEs in two CPs. Thus, it has been experimentally shown that hardware evolution was able to optimise a conventional multiplier. Even so, considerably more work will need to be done to determine if the previous statement can be generalised to all circuits, and not just multipliers.

Although all the phenotypes were all unique, similarities in the placement of external outputs and the number of used LEs were found. We have seen that as the number of columns in the CGP array increases, so the likelihood of an external output being placed in the column decreases. Furthermore, the number of used LEs per column also substantially decreases per added column. Thus, increasing the number of columns in a CGP array should always be done with care.

Finally, Chapter 5 demonstrated the evolution of a state-decomposed control circuit. At this point, we need to consider the following objection: The research only evolved the combinational logic, thus excluding the control circuit's sequential, timing and counter logic. Although true, the aim of the chapter was to only demonstrate modular evolution, even if only on a sub-section of the control circuit. The development of the state diagram, block diagram and state truth tables proved to be critical to developing the FSM. It was shown that the evolution of each state's sub-circuit was possible; and that the final control circuit made use of 22 LEs, 10 inputs and 8 outputs—more than double the available resources on the 20-LE V-FPGA. Thus, modular evolution has been shown to be a successful tool when dealing with scalability, thereby affirming the third question.

6.3 The Way Forward: Future Work

A number of important limitations need to be considered. Firstly, while recognising that the hardware setup produced successful results, it is hesitantly recommended for future work. Chapter 4 showed that data communication between LabVIEW and the V-FPGA was mostly slow, inaccurate and detrimental. This was evident in: the spikes seen in the "Results" sections; the long evolution times; the poor repeatability. Future research should therefore investigate completely eliminating LabVIEW and the DAQ interfacing hardware, and instead concentrate on implementing the GA using a soft-processor, thereby creating an on-chip solution.

Secondly, an issue that was not addressed in this study was whether sequential logic could be evolved. The evolution constraints prohibited this possibility. For now, these constraints allowed this study to focus on minimising scalability. But eventually, research into evolving sequential logic—or specifically unconstrained evolution—using the V-FPGA will be necessary, since real-world control circuits require more than just combinational functions.

Thirdly, it would be interesting to compare the results of a GA using a pyramid LE-array with that of a GA using a CGP LE-array. The pyramid array could present further GA enhancements.

Fourthly, the current study has only examined modular evolution using state decomposition, which relied on each state's sub-circuit being evolvable. However, this will not always be the case, as complex states will require further decomposition. Thus, in future studies, better decomposition techniques, such as GDD, will need to be investigated to ensure the successful evolution of complex sub-circuits.

Fifthly, whilst this study did focus on scalability, there is still a need to further examine multi-VRCs and parallel evolution as possible solutions to scalability. It is hoped that, if all scalability solutions, namely multi-VRCs, $1 + \lambda$ GAs, SOPCs, parallel evolution and modularisation, are to used in one system, the evolution of real-world circuits will be attainable.

Finally, though recognising the research documented here and by others involved small and simple circuits, these circuits should not be dismissed, as they still play a major role in fine-tuning system parameters. Small circuits are far more practical to analysis, and provide important insight into scalability and the unusual ways in which GAs synthesise circuits.

Until scalability is overcome, and evolution can provide solutions to real-world applications, further progress in the field will be required to make EHW a credible engineering tool.

BIBLIOGRAPHY

Altenberg, L., 1994. The evolution of evolvability in genetic programming. In: L. Spector, W. B. Langdon & U. O'Reilly, eds. *Advances in genetic programming.* Cambridge: MIT Press, pp. 47-74.

Altera, 2006. *DE2 Development and Education Board - User Manual Version 1.4.* [Online]
Available at: ftp://ftp.altera.com/up/pub/Webdocs/DE2_UserManual.pdf
[Accessed 14 June 2012].

Altera, 2008. *History of Boards in the Altera University Program.* [Online]
Available at: ftp://ftp.altera.com/up/pub/Webdocs/board-history.pdf
[Accessed 20 July 2013].

Altera, 2013. *FPGAs.* [Online]
Available at: http://www.altera.com/products/fpga.html
[Accessed 19 September 2013].

Antola, A., Castagna, M., Gotti, P. & Santambrogio, M. D., 2007. *Evolvable Hardware: A Functional Level Evolution Framework Based on ImpulseC.* Paris, ERSA.

Barlow, G. J. & Edwards, M. A., 2001. *Self-Evolving Hardware,* s.l.: North Carolina State University.

Bedi, A., 2009. *A Generic Platform for the Evolution of Hardware,* Auckland: Auckland University of Technology.

Black, P. E., 2008. *finite state machine.* [Online]
Available at: http://xlinux.nist.gov/dads//HTML/finiteStateMachine.html
[Accessed 7 August 2012].

Cancare, F. et al., 2011. *A bird's eye view of FPGA-based Evolvable Hardware.* San Diego, IEEE.

Cancare, F., Santambrogio, M. D. & Sciuto, D., 2010. *A direct bitstream manipulation approach for Virtex4-based evolvable systems.* Paris, IEEE.

CircuitoZ, n.d. *Switch-On Delay Circuit.* [Online]
Available at: http://circuitos.cl.tripod.com/schem/r67.gif
[Accessed 24 September 2013].

Daciuk, J., 1998. *Finite State Automata.* [Online]
Available at: http://www.eti.pg.gda.pl/katedry/kiw/pracownicy/Jan.Daciuk/personal/thesis/node12.html
[Accessed 3 November 2013].

Darwin, C., 1859. *On the Origin of the Species by Means of Natural Selection: Or, The Preservation of Favoured Races in the Struggle for Life.* 1st ed. London: W.Clowes and Sons.

de Garis, H., 1993. Evolvable Hardware: Genetic Programming of a Darwin Machine. In: *Artificial neural nets and genetic algorithms.* Vienna: Springer, pp. 441-449.

de Garis, H. et al., 1997. “CAM-brain” ATR's billion neuron artificial brain project: A three-year progress report. *Artificial Life and Robotics,* 2(2), pp. 56-61.

Dobai, R. & Sekanina, L., 2013. *Image filter evolution on the Xilinx Zynq Platform.* Torino, IEEE.

Dobai, R. & Sekanina, L., 2013. *Towards Evolvable Systems Based on the Xilinx Zynq Platform.* Orlando, IEEE.

Eberhart, R. C. & Kennedy, J., 1995. *A new optimiser using particle swarm theory.* Nagoya, IEEE.

Electronic Project Circuits, 2012. *The Basic Delay Timer Circuits using IC-555 Base.* [Online]
Available at: http://www.eleccircuit.com/the-basic-delay-timer-circuits-using-ic-555-base/
[Accessed 24 September 2013].

Engelbrecht, A. P., 2007. *Computational Intelligence: An Introduction.* 2nd Edition ed. Chichester: John Wiley & Sons Ltd.

EngineersGarage, 2012. *VLSI Technology.* [Online]
Available at: http://www.engineersgarage.com/articles/vlsi-design-future
[Accessed 29 July 2013].

Floyd, T. L., 2009. *Digital Fundamentals.* 10th ed. Upper Saddle River: Pearson Education International.

Gers, F., de Garis, H. & Korkin, M., 1998. CoDi-1Bit: A simplified cellular automata based neuron model. *Artificial Evolution,* pp. 315-333.

Gordon, T. G. & Bentley, P. J., 2005. *Development Brings Scalability to Hardware Evolution.* s.l., IEEE.

Gordon, T. G. W., 2005. *Exploiting Development to Enhance the Scalability of Hardware Evolution,* London: University of London.

Gordon, T. G. W. & Bentley, P. J., 2002. On Evolvable Hardware. In: *Soft Computing in Industrial Electronics.* s.l.:Physica-Verlag HD, pp. 279-323.

Greenwood, G. W. & Tyrrell, A. M., 2006. *Introduction to Evolvable Hardware.* 1st E-Book ed. International: Wiley.

Guccione, S. A. & Levi, D., 1999. *JBits: A Java-Based Interface to FPGA Hardware.* [Online]
Available at: http://www-inst.eecs.berkeley.edu/~cs294-59/fa10/resources/Xilinx-history/jbits.pdf
[Accessed 28 July 2013].

Haddow, P. C. & Tufte, G., 2001. *Bridging the genotype-phenotype mapping for digital FPGAs.* Long Beach, IEEE.

Harel, D., 1986. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming,* 8(1), pp. 231-274.

Hauschildt, K., n.d. *Evolvable Hardware in Theory and Implementation.* San Fancisco, Creative Commons.

Higuchi, T. et al., 1999. Real-World Applications of Analog and Digital Evolvable Hardware. *IEEE Transactions on Evolutionary Computation,* 3(3), pp. 230-235.

Higuchi, T. et al., 1997. *Evolvable Hardware at Function Level.* New York, IEEE Press.

Holland, J. H., 1975. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.* 1st ed. Michigan: University of Michigan Press.

Hollingworth, G., Smith, S. & Tyrrell, A., 2000. *Safe intrinsic evolution of Virtex devices.* Palo Alto, IEEE.

Hubert, C., n.d. *Fitness Landscape.* [Online]
Available at: http://www.christianhubert.com/writings/fitness_landscape.html
[Accessed 9 October 2013].

Institute of Electrical and Electronics Engineers, 2013. *2014 IEEE International Conference on Evolvable Systems - From Biology to Hardware.* [Online]
Available at: http://www.ieee-ssci.org/ICES.html
[Accessed 23 July 2013].

Iwata, M. et al., 1996. A pattern recognition system using evolvable hardware. In: H. Voigt, W. Ebeling, I. Rechenberg & H. Schwefel, eds. *Parallel Problem Solving from Nature — PPSN IV.* Berlin: Springer-Verlag, pp. 761-770.

Kalganova, T., 2000. *Bidirectional incremental evolution in extrinsic evolvable hardware.* Palo Alto, IEEE.

Katz, R. H., 1993. *Contemporary Logic Design.* 1st ed. California: Addison Wesley Publishing Company.

Keymeulen, D., Stoica, A., Zebulum, R. S. & Ferguson, M. I., 2003. *Scalability Issues in Evolutionary Synthesis of Electronic Circuits: Lessons Learned and Challenges Ahead.* Palo Alto, American Association for Artificial Intelligence.

Keymeulen, D. et al., 2004. *High Temperature Experiments for Circuit Self-Recovery.* Seattle, Springer Verlag.

Ko, M., Kang, T. & Hwang, C., 1997. Function optimisation using an adaptive crossover operator based on locality. *Engineering Applications of Artificial Intelligence,* 10(6), pp. 519-524.

Koza, J. R., Keane, M. A. & Streeter, M. J., 2003. Genetic Programming's Human-Competitive Results. *IEEE Intelligent Systems,* 18(3), pp. 25-31.

Krohling, R. A., Zhou, Y. & Tyrrell, A. M., 2002. Evolving FPGA-based robot controllers using an evolutionary algorithm. pp. 41-46.

Kryvyi, S. L., 2011. Finite-state automata in information technologies. *Cybernetics and Systems Analysis ,* 47(5), pp. 669-683.

Lambert, C., Kalganova, T. & Stomeo, E., 2009. FPGA-based Systems for Evolvable Hardware. *International Journal of Electrical and Electronics Engineering,* 3(1), pp. 62-68.

Lazzaro, J., 2010. *Xilinx Part Family History.* [Online]
Available at: http://www-inst.eecs.berkeley.edu/~cs294-59/fa10/resources/Xilinx-history/Xilinx-history.html
[Accessed 12 October 2013].

Macias, N. J., 1999. *The PIG paradigm: the design and use of a massively parallel fine grained self-reconfigurable infinitely scalable architecture.* Pasadena, IEEE.

Majzoobi, M., Koushanfar, F. & Potkonjak, M., 2012. Trusted design in FPGAs. In: H. Tehranipoor & C. Wang, eds. *Introduction to Hardware Security and Trust.* New York: Springer, pp. 195-229.

Martin, P. & Poli, R., 2002. *Crossover Operators for a Hardware Implementation of GP using FPGAs and Handel-C.* New York, GECCO.

Mataric, M. & Cliff, D., 1996. Challenges in Evolving Controllers for Physical Robots. *Robotics and Autonomous Systems,* 19(1), pp. 67-83.

Math is Fun, 2011. *Combinations and Permutations.* [Online]
Available at: http://www.mathsisfun.com/combinatorics/combinations-permutations.html
[Accessed 4 December 2013].

Mead, P., 2001. *Systems on Programmable Chips – Will SOPC eclipse SoC?,* Cambridge: Altera Corporation.

Miller, J. F., Job, D. & Vassilev, V. K., 2000. Principles in the evolutionary design of digital circuits. *Journal of Genetic Programming and Evolvable Machines,* 1(1-2), pp. 7-35.

Miller, J. F., Kalganova, T., Lipnitskaya, N. & Job, D., 1999. *The genetic algorithm as a discovery engine: Strange circuits and new principles.* Edinburgh, Society for the Study of Artificial Intelligence and Simulation of Behaviour.

Miller, J. F. & Thomson, P., 1998. Aspects of digital evolution: Geometry and learning. In: M. Sipper, D. Mange & A. Perez-Uribe, eds. *Evolvable Systems: From Biology to Hardware.* Berlin: Springer, pp. 25-35.

Miller, J. & Job, D., 1999. *Principles in the Evolutionary Design of Digital Circuits - Part I,* Scotland: Napier University.

Miller, J. & Thomson, P., 2000. *Cartesian Genetic Programming.* Berlin, Springer Verlag.

Moore, G. E., 1965. Cramming more components onto integrated circuits. *Electronics,* 38(8).

Moreno, J. M. et al., 1998. *Feasible Evolutionary and Self-Repairing Hardware by means of the dynamic reconfiguration capabilities of the FIPSOC devices.* Lausanne, Springer Berlin Heidelberg.

Murakawa, M. et al., 1999. The GRD Chip: Genetic Reconfiguration of DSPs for Neural Network Processing. *IEEE Transactions on Computers,* 48(6), pp. 628-639.

National Instruments, 2012. *NI USB-6501 (24-ch, 8.5 mA).* [Online]
Available at: http://sine.ni.com/nips/cds/view/p/lang/en/nid/201630
[Accessed 20 June 2013].

Nolfi, S. & Floreano, D., 2000. *Evolutionary Robotics: The Biology, Intelligence and Technology of Self-Organizing Machines.* 1 ed. Cambridge: MIT Press.

Popa, R., Nicolau, V. & Epure, S., 2006. *Evolvable Hardware in Xilinx PLDs.* Galati, ISEEE.

Rustem, P., 2012. *Genetic Algorithms: An Overview with Applications in Evolvable Hardware.* [Online]
Available at: http://cdn.intechopen.com/pdfs/30231/InTech-Genetic_algorithms_an_overview_with_applications_in_evolvable_hardware.pdf
[Accessed 11 February 2013].

Schnee, J. E., 1978. Government programs and the growth of high-technology industries. *Research Policy,* 7(1), pp. 3-24.

Sekanina, L., 2002. *Evolution of Digital Circuits Operating as Image Filters in Dynamically Changing Enviroments.* Brno, Brno University of Technology.

Sekanina, L., 2003. Virtual reconfigurable circuits for real-world applications of evolvable hardware. In: *Evolvable Systems: From Biology to Hardware.* Berlin: Springer Berlin Heidelberg, pp. 186-197.

Sekanina, L. & Azeddien, S. M., 2000. Toward uniform approach to design of evolvable hardware based systems. In: *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing.* Berlin: Springer Berlin Heidelberg, pp. 814-817.

Sekanina, L. & Freidl, S., 2005. An Evolvable Combinational Unit for FPGAs. *Computing and Informatics,* 23(5), pp. 461-486.

Slorach, C. & Sharman, K., 2000. The design and implementation of custom architectures for evolvable hardware using off-the-shelf programmable devices. In: *Evolvable Systems: From Biology to Hardware.* Berlin: Springer Berlin Heidelberg, pp. 197-207.

Smith, F., 2010. *A Virtual VLSI Architecture for Computer Hardware Evolution.* Bela Bela, ACM.

Stocia, A. et al., 2004. Taking evolutionary circuit design from experimentation to implementation: some useful techniques and a silicon demonstration. *EE Proc.-Comput. Digit. Tech.,* 151(4), pp. 295-300.

Stomeo, E., Kalganova, T. & Lambert, C., 2005. *Mutation Rate for Evolvable Hardware.* s.l., s.n.

Stomeo, E., Kalganova, T. & Lambert, C., 2006. Generalized disjunction decomposition for evolvable hardware. *Systems, Man, and Cybernetics, Part B: Cybernetics,* 5(36), pp. 1024-1043.

Thompson, A., 1995. *Evolving Electronic Robot Controllers that Exploit Hardware Resources.* s.l., Springer Verlag.

Thompson, A., 1996. *An Evolved Circuit: Intrinsic in Silicon, Entwined with Physics.* [Online]
Available at: http://www.sussex.ac.uk/Users/adrianth/ices96/paper.html
[Accessed 23 January 2013].

Thompson, A., 1998. On the automatic design of robust electronics through artificial evolution. In: *Evolvable Systems: From Biology to Hardware.* Berlin: Springer, pp. 13-24.

Thompson, A., 1999. *The Evolvatron.* [Online]
Available at: http://www.sussex.ac.uk/Users/adrianth/TEC99/node20.html
[Accessed 14 October 2013].

Thompson, A., Harvey, I. & Husbands, P., 1996. *The Natural Way to Evolve Hardware.* Atlanta, IEEE.

Torresen, J., 1998. A divide-and-conquer approach to evolvable hardware. In: *Evolvable Systems: From Biology to Hardware.* Berlin: Springer, pp. 57-65.

Torresen, J., 1999. *Increased complexity evolution applied to evolvable hardware.* St. Louis, ASME Press.

Torresen, J., 2004. *An Evolvable Hardware Tutorial.* Antwerp, FPL.

University of Heidelberg, 2011. *Evolvable Hardware.* [Online]
Available at: http://www.kip.uni-heidelberg.de/cms/vision/projects/recent_projects/evolvable_hardware/?L=0
[Accessed 31 July 2013].

van den Berg, A. E. & Smith, F., 2013. *Hardware evolution of a digital circuit using a custom VLSI architecture.* East London, ACM.

Vassilev, V. K. & Miller, J. F., 2000. *Embedding landscape neutrality to build a bridgefrom the conventional to a more efficient three-bit multiplier circuit.* San Francisco, Morgan Kaufmann.

Vassilev, V. K. & Miller, J. F., 2000. *Scalability problems of digital circuit evolution evolvability and efficient designs.* s.l., IEEE.

Vassilev, V. K. & Miller, J. F., 2000. *The Advantages of Landscape Neutrality in Digital Circuit Evolution.* Edinburgh, Springer-Verlag.

Vassilev, V. K., Miller, J. F. & Fogarty, T. C., 1999. *On the Nature of Two-Bit Multiplier Landscapes.* Pasadena, IEEE ComputerSociety.

Wang, J., Piao, C. H. & Lee, C. H., 2007. Evolvable Systems: From Biology to Hardware. In: *Implementing multi-VRC cores to evolve combinational logic circuits in parallel.* Berlin: Springer Berlin Heidelberg, pp. 23-34.

Wright, D. R., 2005. *Finite State Machines.* [Online]
Available at: http://www4.ncsu.edu/~drwrigh3/docs/courses/csc216/fsm-notes.pdf
[Accessed 7 August 2013].

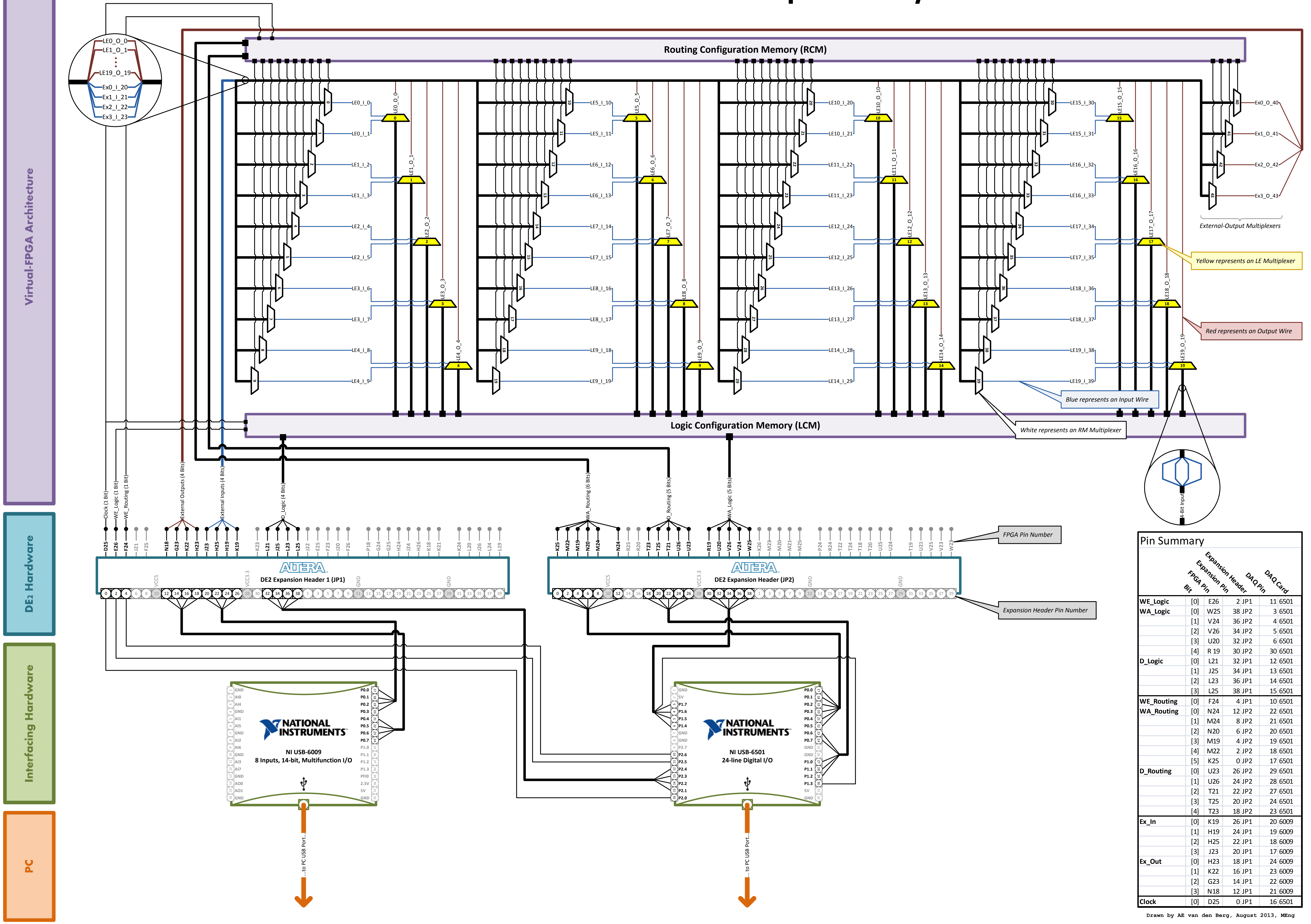
Xilinx, 2013. *FPGA vs. ASIC.* [Online]
Available at: http://www.xilinx.com/fpga/asic.htm
[Accessed 31 October 2013].

Yannou, J.-M., 2011. Xilinx’s 3D (or 2.5D) packaging enables the world's highest capacity FPGA device, and one of the most powerful processors on the market. *3D Packaging*, November, pp. 21-23.

Yasunaga, M., Yoshihara, I. & Kim, J. H., 2003. Gene finding using evolvable reasoning hardware. In: *Evolvable Systems: From Biology to Hardware.* Berlin: Springer Berlin Heidelberg, pp. 198-207.

APPENDICES

APPENDIX A: Electrical Schematic of Complete EHW System



Virtual-FPGA Architecture
DE2 Hardware
Interfacing Hardware
PC
Routing Configuration Memory (RCM)
Logic Configuration Memory (LCM)
External-Output Multiplexers
Yellow represents an LE Multiplexer
Red represents an Output Wire
Blue represents an Input Wire
White represents an RM Multiplexer
4-Bit Input
FPGA Pin Number
Expansion Header Pin Number
DE2 Expansion Header 1 (JP1)
DE2 Expansion Header 2 (JP2)
Clock (1 Bit)
WE_Logic (1 Bit)
WE_Routing (1 Bit)
External Outputs (4 Bits)
External Inputs (4 Bits)
D_Logic (4 Bits)
WA_Routing (6 Bits)
D_Routing (5 Bits)
WA_Logic (5 Bits)
NATIONAL INSTRUMENTS
NI USB-6009
8 Inputs, 14-bit, Multifunction I/O
NI USB-6501
24-line Digital I/O
... to PC USB Port ...

Pin Summary

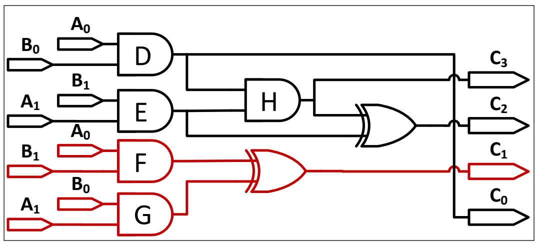
	Bit	FPGA Pin	Expansion Pin	Expansion Header	DAQ Pin	DAQ Card
WE_Logic	[0]	E26	2	JP1	11	6501
WA_Logic	[0]	W25	38	JP2	3	6501
	[1]	V24	36	JP2	4	6501
	[2]	V26	34	JP2	5	6501
	[3]	U20	32	JP2	6	6501
	[4]	R 19	30	JP2	30	6501
D_Logic	[0]	L21	32	JP1	12	6501
	[1]	J25	34	JP1	13	6501
	[2]	L23	36	JP1	14	6501
	[3]	L25	38	JP1	15	6501
WE_Routing	[0]	F24	4	JP1	10	6501
WA_Routing	[0]	N24	12	JP2	22	6501
	[1]	M24	8	JP2	21	6501
	[2]	N20	6	JP2	20	6501
	[3]	M19	4	JP2	19	6501
	[4]	M22	2	JP2	18	6501
	[5]	K25	0	JP2	17	6501
D_Routing	[0]	U23	26	JP2	29	6501
	[1]	U26	24	JP2	28	6501
	[2]	T21	22	JP2	27	6501
	[3]	T25	20	JP2	24	6501
	[4]	T23	18	JP2	23	6501
Ex_In	[0]	K19	26	JP1	20	6009
	[1]	H19	24	JP1	19	6009
	[2]	H25	22	JP1	18	6009
	[3]	J23	20	JP1	17	6009
Ex_Out	[0]	H23	18	JP1	24	6009
	[1]	K22	16	JP1	23	6009
	[2]	G23	14	JP1	22	6009
	[3]	N18	12	JP1	21	6009
Clock	[0]	D25	0	JP1	16	6501

Drawn by AE van den Berg, August 2013, MEng

APPENDIX B: Truth Tables of the Evolved Phenotypes

By using Boolean algebra, the following truth tables prove that the evolved phenotypes are indeed logically valid, i.e. the phenotypes produce the correct outputs.

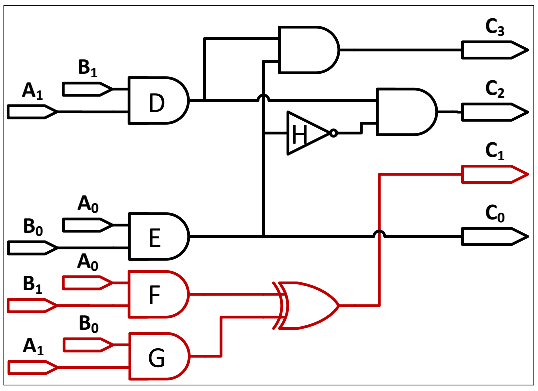
Multiplier Truth Tables



Simplified trial phenotype

Truth table of the simplified trial phenotype

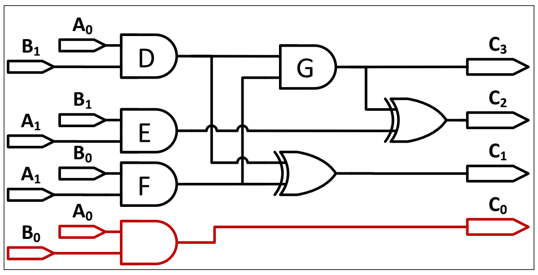
A_1	A_0	B_1	B_0	$D = A_0.B_0$	$E = A_1.B_1$	$F = A_0.B_1$	$G = A_1.B_0$	$H = D.E$	$C_3 = H$	$C_2 = E\oplus H$	$C_1 = F\oplus G$	$C_0 = D$
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0
0	1	0	1	1	0	0	0	0	0	0	0	1
0	1	1	0	0	0	1	0	0	0	0	1	0
0	1	1	1	1	0	1	0	0	0	0	1	1
1	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	1	0	0	0	1	0
1	0	1	0	0	1	0	0	0	0	1	0	0
1	0	1	1	0	1	0	1	0	0	1	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0
1	1	0	1	1	0	0	1	0	0	0	1	1
1	1	1	0	0	1	1	0	0	0	1	1	0
1	1	1	1	1	1	1	1	1	1	0	0	1



Simplified canonical phenotype

Truth table of the simplified canonical phenotype

A_1	A_0	B_1	B_0	$D = A_1 \cdot B_1$	$E = A_0 \cdot B_0$	$F = A_0 \cdot B_1$	$G = A_1 \cdot B_0$	$H = \overline{E}$	$C_3 = D.E$	$C_2 = D.H$	$C_1 = F \oplus G$	$C_0 = E$
0	0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0	1	0	0	0	0
0	0	1	1	0	0	0	0	1	0	0	0	0
0	1	0	0	0	0	0	0	1	0	0	0	0
0	1	0	1	0	1	0	0	0	0	0	0	1
0	1	1	0	0	0	1	0	1	0	0	1	0
0	1	1	1	0	1	1	0	0	0	0	1	1
1	0	0	0	0	0	0	0	1	0	0	0	0
1	0	0	1	0	0	0	1	1	0	0	1	0
1	0	1	0	1	0	0	0	1	0	1	0	0
1	0	1	1	1	0	0	1	1	0	1	1	0
1	1	0	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	1	0	1	0	0	0	1	1
1	1	1	0	1	0	1	0	1	0	1	1	0
1	1	1	1	1	1	1	1	0	1	0	0	1

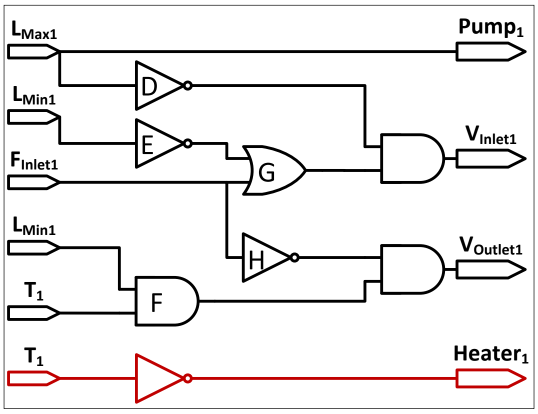


Simplified $1 + \lambda$ phenotype

Truth table of the simplified $1 + \lambda$ phenotype

A_1	A_0	B_1	B_0	$D = A_0 \cdot B_1$	$E = A_1 \cdot B_1$	$F = A_1 \cdot B_0$	$G = D.F$	$C_3 = G$	$C_2 = E \oplus G$	$C_1 = D \oplus F$	$C_0 = A_0 \cdot B_0$
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0
0	1	0	1	0	0	0	0	0	0	0	1
0	1	1	0	1	0	0	0	0	0	1	0
0	1	1	1	1	0	0	0	0	0	1	1
1	0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0	0	0	1	0
1	0	1	0	0	1	0	0	0	1	0	0
1	0	1	1	0	1	1	0	0	1	1	0
1	1	0	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	1	0	0	0	1	1
1	1	1	0	1	1	0	0	0	1	1	0
1	1	1	1	1	1	1	1	1	0	0	1

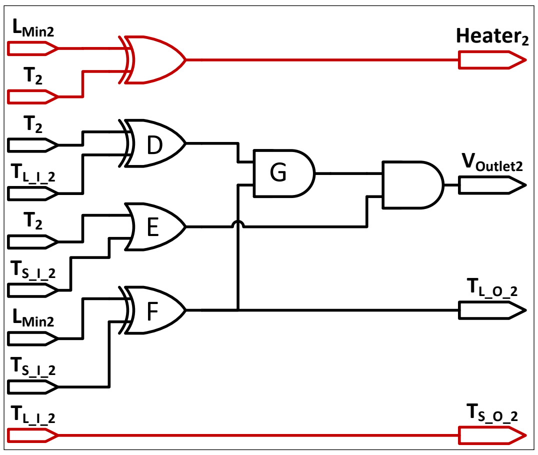
Finite-State Machine Truth Tables



State one's simplified phenotype

Truth table of state one's simplified phenotype

L_{Max1}	L_{Min1}	F_{Inlet1}	T_1	$D = \overline{L_{Max1}}$	$E = \overline{L_{Min1}}$	$F = T_1.L_{Min1}$	$G = D + F_{Inlet1}$	$H = \overline{F_{Inlet1}}$	$V_{Inlet1} = D.G$	$V_{Outlet1} = F.H$	$Heater_1 = \overline{T_1}$	$Pump_1 = L_{Max1}$
0	0	0	0	1	1	0	1	1	1	0	1	0
0	0	0	1	1	1	0	1	1	1	0	0	0
0	0	1	0	1	1	0	1	0	1	0	1	0
0	0	1	1	1	1	0	1	0	1	0	0	0
0	1	0	0	1	0	0	0	1	0	0	1	0
0	1	0	1	1	0	1	0	1	0	1	0	0
0	1	1	0	1	0	0	1	0	1	0	1	0
0	1	1	1	1	0	1	1	0	1	0	0	0
1	0	0	0	0	1	0	1	1	0	0	1	1
1	0	0	1	0	1	0	1	1	0	0	0	1
1	0	1	0	0	1	0	1	0	0	0	1	1
1	0	1	1	0	1	0	1	0	0	0	0	1
1	1	0	0	0	0	0	0	1	0	0	1	1
1	1	0	1	0	0	1	0	1	0	1	0	1
1	1	1	0	0	0	0	1	0	0	0	1	1
1	1	1	1	0	0	1	1	0	0	0	0	1



State two's simplified phenotype

Truth table of state two's simplified phenotype

L_{Min2}	T_2	$T_{L\_I\_2}$	$T_{S\_I\_2}$	$D = T_2 \oplus T_{L\_I\_2}$	$E = T_2 + T_{S\_I\_2}$	$F = L_{Min2} \oplus T_{S\_I\_2}$	$G = D.E$	$V_{Outlet2} = E.G$	$T_{L\_O\_2} = F$	$T_{S\_O\_2} = T_{L\_I\_2}$	$Heater_2 = L_{Min2} \oplus T_2$
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	1	1	0	0	1	0	0
0	0	1	0	1	0	0	0	0	0	1	0
0	0	1	1	1	1	1	1	1	1	1	0
0	1	0	0	1	1	0	0	0	0	0	1
0	1	0	1	1	1	1	1	1	1	0	1
0	1	1	0	0	1	0	0	0	0	1	1
0	1	1	1	0	1	1	0	0	1	1	1
1	0	0	0	0	0	1	0	0	1	0	1
1	0	0	1	0	1	0	0	0	0	0	1
1	0	1	0	1	0	1	1	0	1	1	1
1	0	1	1	1	1	0	0	0	0	1	1
1	1	0	0	1	1	1	1	1	1	0	0
1	1	0	1	1	1	0	0	0	0	0	0
1	1	1	0	0	1	1	0	0	1	1	0
1	1	1	1	0	1	0	0	0	0	1	0

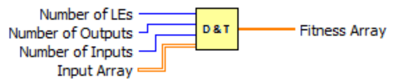
APPENDIX C: Raw Download Data

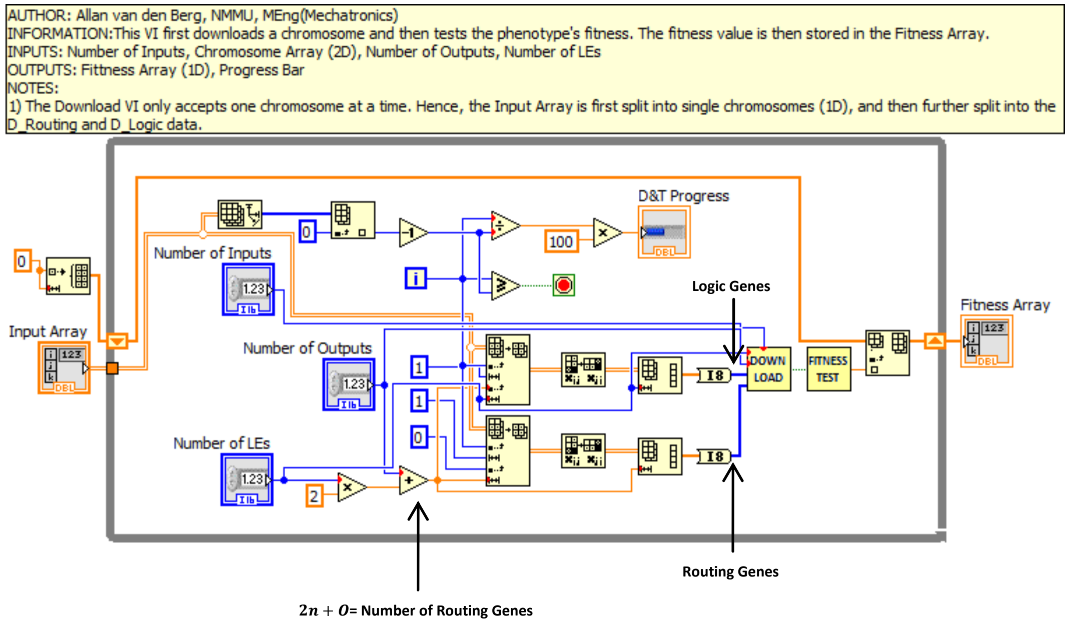
	100 Hz		71.43 Hz		55.56 Hz		50 Hz		33.33 Hz		25 Hz		20 Hz		16.67 Hz		14.29 Hz		12.5 Hz		Testing VI
1	0	0.582033	0	0.672039	0	0.853049	1	0.962055	1	1.38708	1	1.83711	1	2.30113	1	2.76016	1	3.22118	1	3.68121	0.119007
2	0	0.626036	0	0.670038	0	0.834048	0	0.937054	1	1.38108	1	1.83811	1	2.29913	1	2.76316	1	3.22418	1	3.68221	0.123007
3	0	0.634036	0	0.677039	0	0.831048	0	0.924053	1	1.38608	1	1.8361	1	2.30513	1	2.76016	1	3.22818	1	3.68121	0.118007
4	0	0.640037	0	0.69704	0	0.828047	0	0.926053	0	1.38008	1	1.84511	1	2.31013	1	2.75916	1	3.23018	1	3.68421	0.124007
5	0	0.595034	0	0.666038	0	0.838048	1	0.919053	1	1.38308	1	1.84011	1	2.30113	1	2.76016	1	3.22618	1	3.68321	0.121007
6	0	0.629036	0	0.70704	1	0.829047	0	0.929053	1	1.38908	0	1.84711	1	2.30213	1	2.74416	1	3.19418	1	3.68021	0.120007
7	0	0.642037	0	0.674038	0	0.830048	0	0.928053	1	1.41708	1	1.84111	1	2.30013	1	2.76216	1	3.22118	1	3.68021	0.120007
8	0	0.618036	0	0.681039	0	0.830048	1	0.926053	0	1.38508	1	1.8291	1	2.30913	1	2.77116	1	3.23118	1	3.68121	0.123007
9	0	0.634037	0	0.668038	0	0.830048	0	0.921053	0	1.38108	1	1.8351	1	2.30013	1	2.76116	1	3.23418	1	3.67921	0.122007
10	0	0.643345	0	0.677039	0	0.842048	0	0.917053	1	1.39008	1	1.8291	1	2.30113	1	2.76316	1	3.22018	1	3.69221	0.120007
11	0	0.639093	0	0.71214	0	0.838048	0	0.936054	0	1.38308	1	1.84111	1	2.31313	1	2.76516	1	3.22118	1	3.66221	0.134007
12	0	0.669833	0	0.678039	0	0.829047	0	0.920053	1	1.37708	0	1.84011	1	2.28813	1	2.75916	1	3.22018	1	3.68221	0.139008
13	0	0.658811	0	0.679039	0	0.830047	0	0.919053	0	1.37008	1	1.84011	1	2.28913	1	2.76316	1	3.33254	1	3.68021	0.120007
14	0	0.645037	0	0.678039	0	0.830048	0	0.920053	0	1.37908	1	1.83711	1	2.35013	1	2.76916	1	3.18559	1	3.68421	0.130008
15	0	0.639929	0	0.689039	0	0.841048	0	0.921053	0	1.37308	1	1.84211	1	2.30213	1	2.76216	1	3.22018	1	3.68121	0.120007
16	0	0.662922	0	0.672038	0	0.831048	0	0.917053	1	1.37508	1	1.85811	1	2.29713	1	2.75316	1	3.22118	1	3.68021	0.122007
17	0	0.679677	0	0.684039	0	0.841048	0	0.921053	1	1.37308	0	1.86711	1	2.30313	1	2.76916	1	3.22118	1	3.64921	0.121007
18	0	0.640961	0	0.681039	0	0.831048	0	0.941053	1	1.38708	1	1.84011	1	2.30513	1	2.77016	1	3.19918	1	3.68221	0.124007
19	0	0.648037	0	0.680039	0	0.845048	0	0.932054	1	1.37608	1	1.8301	0	2.30213	1	2.76316	1	3.19418	1	3.65421	0.123007
20	0	0.656037	0	0.666039	0	0.841048	0	0.927053	0	1.38008	0	1.8261	1	2.32113	1	2.76216	1	3.22018	1	3.68721	0.119007
	0	0.6392	0	0.680444	5	0.835148	15	0.927203	60	1.38268	80	1.840007	95	2.30503	100	2.76206	100	3.223319	100	3.67841	0.123107
		0.022879		0.012506		0.006859		0.010704		0.009848		0.009556		0.012896		0.006034		0.028978		0.010631	0.00533

🟦	Accuracy Percentage
🟥	Mean Download-Time
🟩	Standard Deviation of the Mean

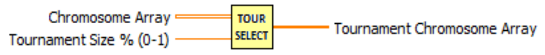
APPENDIX D: Selected LabVIEW Code

Download-and-Test VI

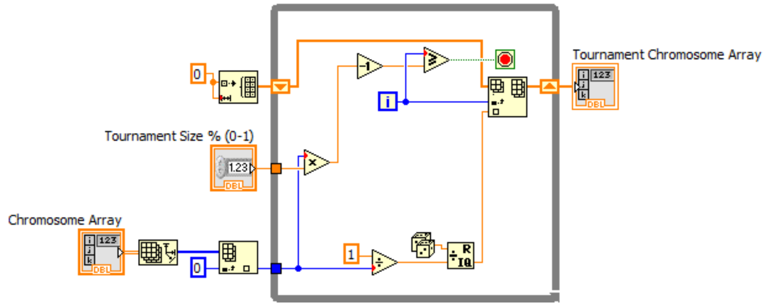



AUTHOR: Allan van den Berg, NMMU, MEng(Mechatronics)
INFORMATION:This VI first downloads a chromosome and then tests the phenotype's fitness. The fitness value is then stored in the Fitness Array.
INPUTS: Number of Inputs, Chromosome Array (2D), Number of Outputs, Number of LEs
OUTPUTS: Fittness Array (1D), Progress Bar
NOTES:
1) The Download VI only accepts one chromosome at a time. Hence, the Input Array is first split into single chromosomes (1D), and then further split into the D_Routing and D_Logic data.

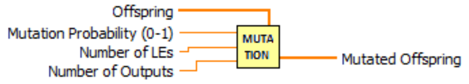
Tournament-Selection VI



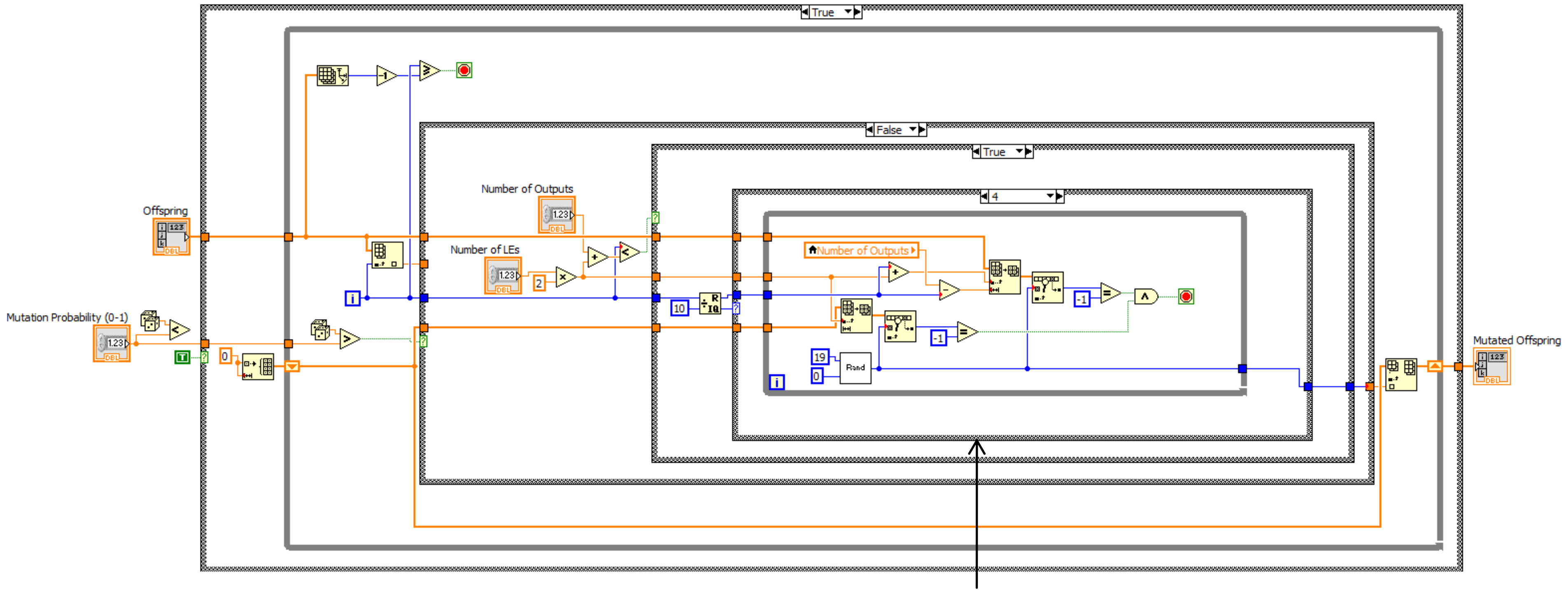
AUTHOR: Allan van den Berg, NMMU, MEng(Mechatronics)
INFORMATION:The Tournament Selection VI selects a random percentage of chromosomes from the population.
INPUTS: Tournament Size, Chromosome Array (2D)
OUTPUTS:Tournament Chromosome Array (2D)
NOTES:
1) Outputs the position of the selected chromosomes.
2) Chromosomes can be selected twice.



Mutation VI

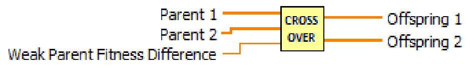


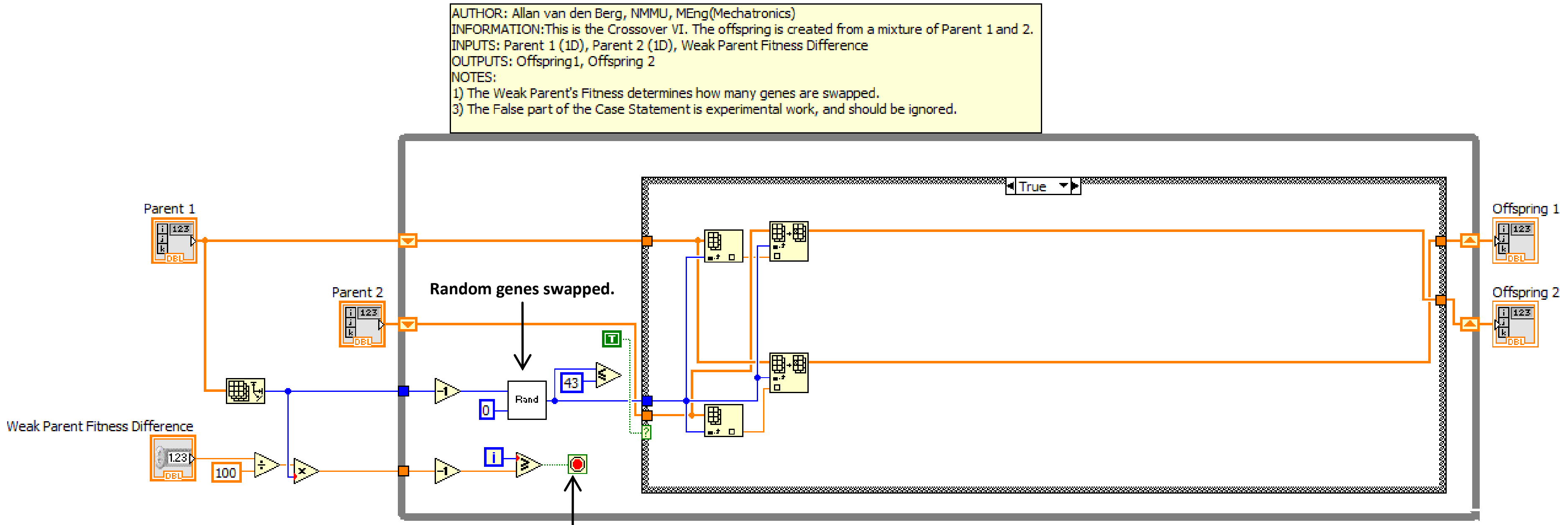
AUTHOR: Allan van den Berg, NMMU, MEng(Mechatronics)
INFORMATION: The Mutation Probability determines whether the Offspring will be mutated.
INPUTS: Mutation Probability (0-1), Offspring (1D), Number of Outputs, Number of Inputs, Number of LEs
OUTPUTS: Mutated Offspring (1D)
NOTES:
1) First case statement determines whether the offspring will be mutatued. False = Offspring passes though without mutation. True = While-loop runs through every gene of the offspring's chromosome.
For the non-canonical GA, the first case statement was kept true to ensure mutation always took place.
3) Second case statement determines whether a particular gene will be mutated.
4) If the second case statement is false, a new mutation gene needs to be created.
5) The third case statement makes sure the created gene is valid. True = the selected gene forms part of the D_Routing array
and thus, cannot be larger than LE+O-1. False (ie gene index > 2LE+O-1) = D_Logic gene needs to be made using fundamental gates.



Case statement ensures the new mutation-gene complies with the evolution constraints.

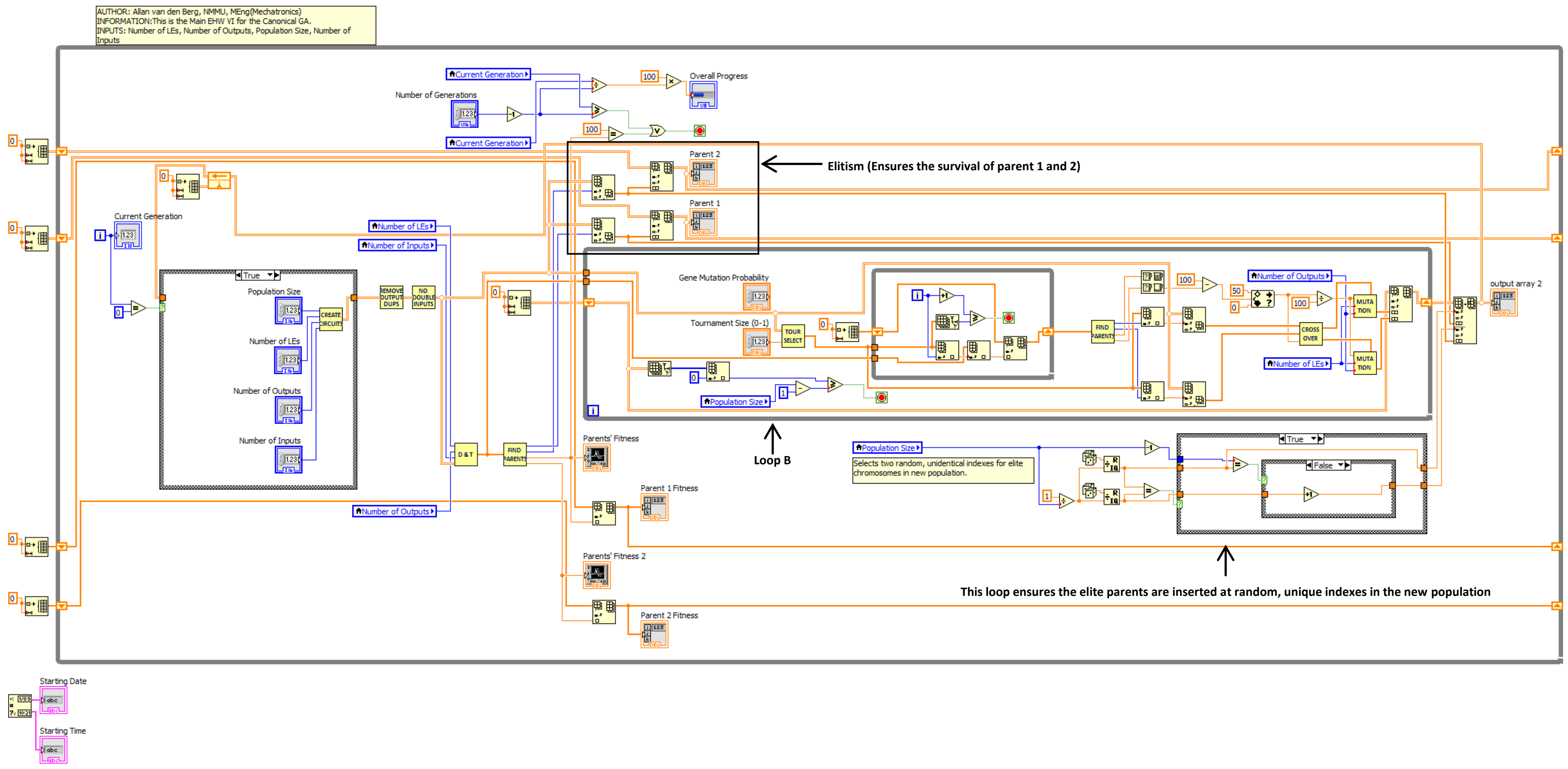
Crossover VI




AUTHOR: Allan van den Berg, NMMU, MEng(Mechatronics)
INFORMATION:This is the Crossover VI. The offspring is created from a mixture of Parent 1 and 2.
INPUTS: Parent 1 (1D), Parent 2 (1D), Weak Parent Fitness Difference
OUTPUTS: Offspring1, Offspring 2
NOTES:
1) The Weak Parent's Fitness determines how many genes are swapped.
3) The False part of the Case Statement is experimental work, and should be ignored.

Loop swaps a percentage of genes. The stop condition is determined by the Weak-Parent-Fitness-Difference variable.

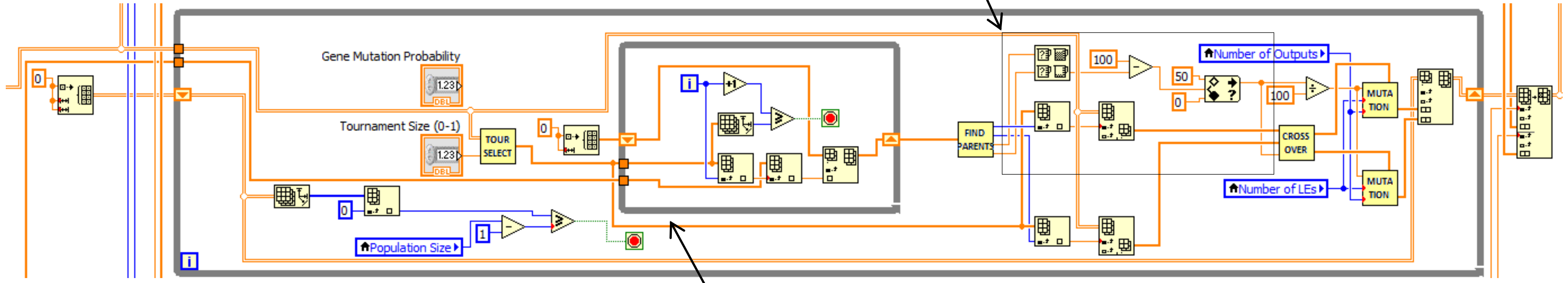
Main VI: Canonical Overview


AUTHOR: Allan van den Berg, NMMU, MEng(Mechatronics)
INFORMATION:This is the Main EHW VI for the Canonical GA.
INPUTS: Number of LEs, Number of Outputs, Population Size, Number of Inputs

Selects two random, unidentical indexes for elite chromosomes in new population.

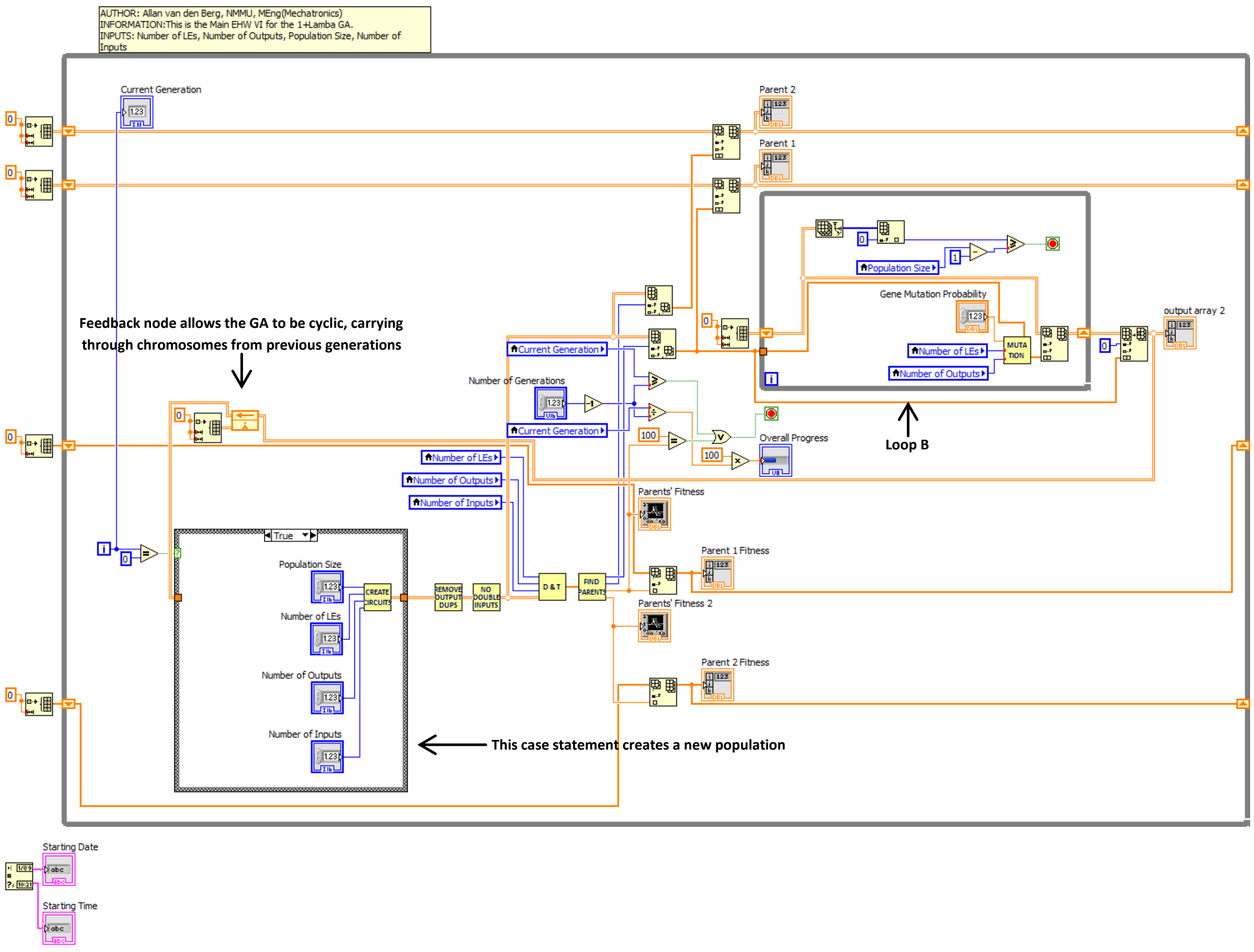
Main VI: Detailed View of Loop B in Canonical GA

Calculates weaker parent's fitness

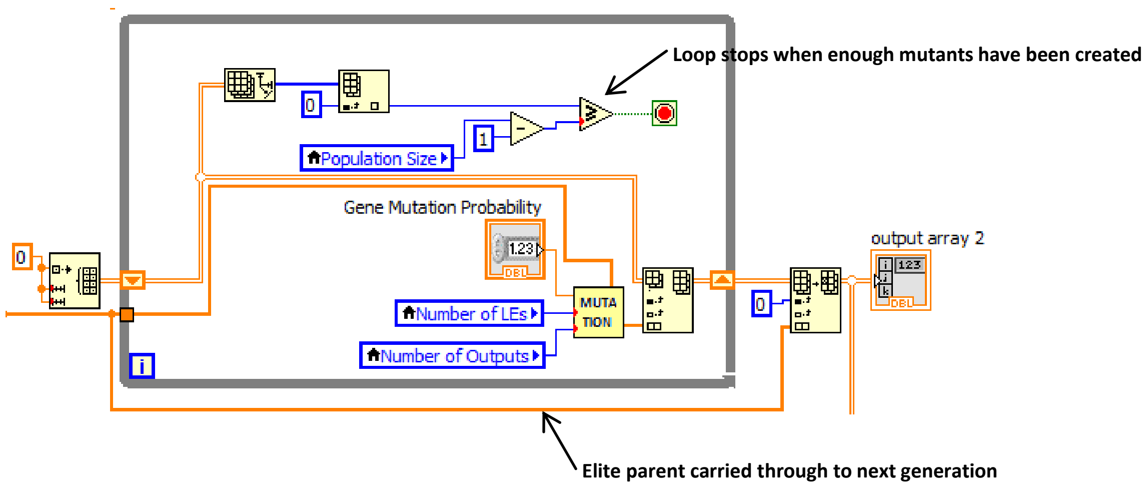


This loop places the selected tournament individuals into a separate array

Main VI: $1 + \lambda$ Overview


AUTHOR: Allan van den Berg, NMMU, MEng(Mechatronics)
INFORMATION:This is the Main EHW VI for the 1+Lamba GA.
INPUTS: Number of LEs, Number of Outputs, Population Size, Number of Inputs

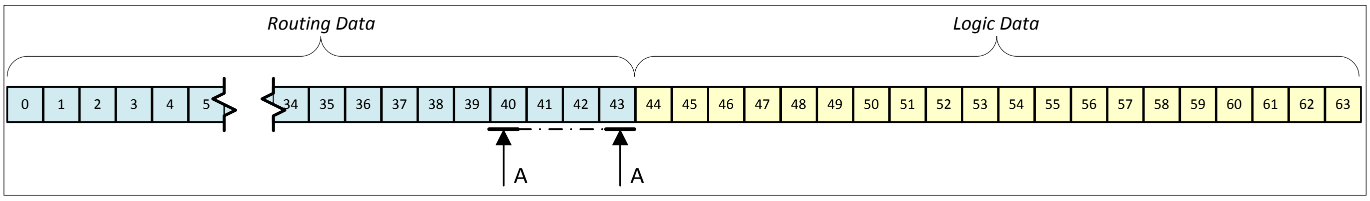
Main VI: Detailed View of Loop B in $1 + \lambda$ GA


Loop stops when enough mutants have been created
Population Size
Gene Mutation Probability
Number of LEs
Number of Outputs
MUTA TION
output array 2
Elite parent carried through to next generation

APPENDIX E: Search-Space Permutations

Summary of the Hardware Chromosome

Consider the 64-gene chromosome shown below, as used in the research (see Section 4.3.4.).



64-Gene Chromosome

Genes 0 to 43 are Routing Genes, and genes 44 to 63 are Logic Genes. Genes 40 to 43—Section A-A in the above figure—define the four external outputs.

Permutations and Combinations

In the mathematics of counting, two concepts are used: permutations and combinations.

Permutations are used when the order of the selected items are important. For example, an ATM pin 1234 will not be the same as 4321.

Combinations are used if the order does not matter. For example, for lottery numbers, the order of the six winning numbers is irrelevant, as long as the correct six numbers are chosen.

Now, consider the hardware chromosome again. The genes are arranged in a specific order, with each gene configuring a specific LE or external IO. For example:

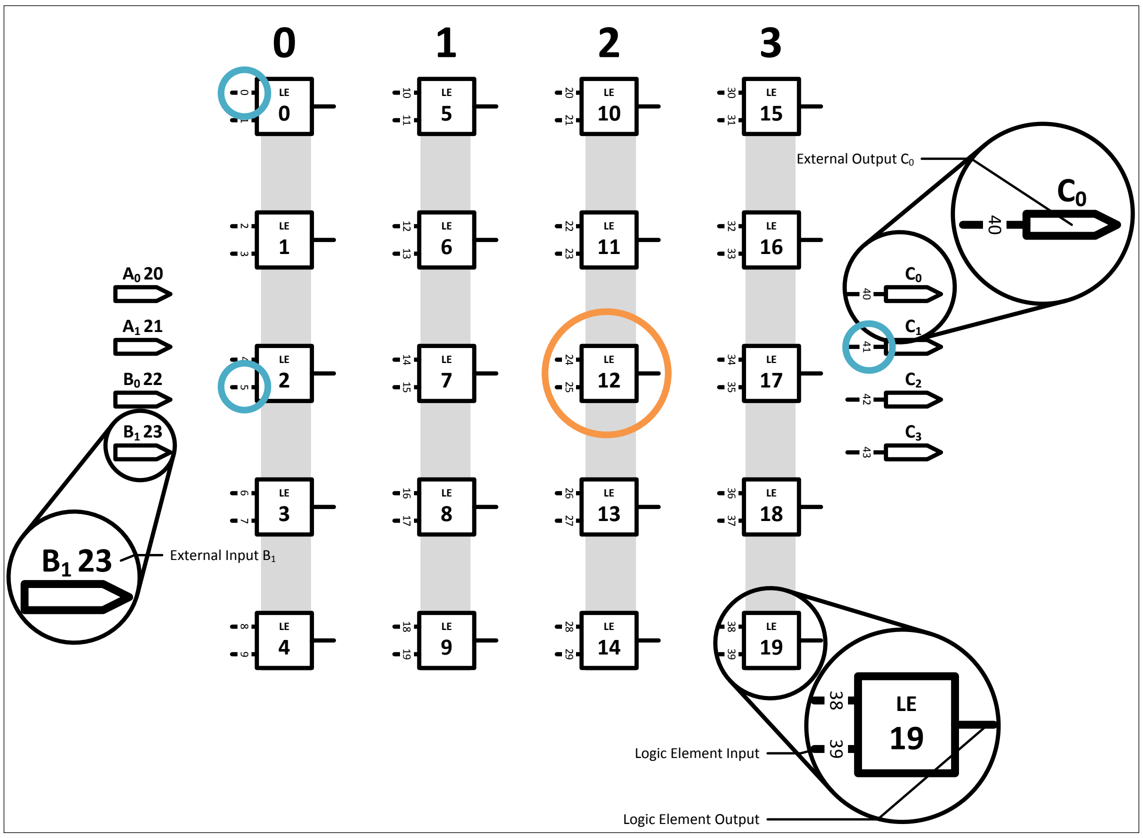
- Gene 0 configures the first input to LE 0
- Gene 5 configures the second input to LE 2
- Gene 41 configures the second external output
- Gene 56 configures LE 12's logic

(The above examples have been highlighted in the "Layout of LEs" figure below.)

Hence, in order to find the number of routing and logic possibilities—and since order is important—permutations (and not combinations) have to be used.

An important factor to consider when calculating permutations is whether or not repetition is allowed. For the unconstrained chromosome, repetition is allowed. For example:

- Any of the routing genes could have the same values. For example, gene 0 and gene 5 could have the same value, thus the first input of LE 0 and the second input of LE 2 will be routed from the same source.
- Any of the logic genes could have the same values, i.e. configure the same logic gate. For example, gene 44 and 56 could both be configured as NOT-gates.



Layout of LEs

For permutations with repetition, the following formula is used (Math is Fun, 2011):

$$n^r$$

where there are n things to choose from, and r things are chosen. For example, a combination lock requires four numbers in a specific order. If there are ten numbers to choose from (0, 1, 2, 3...9), and we choose four of them, then there are $n^r = 10^4 = 10000$ permutations.

Now, recall from Section 4.3.4. that the value of each gene is determined by $D_{Routing}$ and D_{Logic} . Also, recall from Section 3.2.1. and 3.3.2. that:

- the maximum number that $D_{Routing}$ can equal is $n + I - 1 = 20 + 4 - 1 = 23$, i.e. 24 possible values
- the maximum number that D_{Logic} can equal is 15, i.e. 16 possible gates

Hence, each Routing Gene can be defined as an integer ranging from 0 to 23, and each Logic Gene an integer from 0 to 15.

The Unconstrained Permutations

Routing the LEs: Genes 0 to 39

For the routing of the LEs, there are forty LE inputs (two inputs per LE) which can be connected to one of any of the other twenty LEs' outputs, or any of the four external inputs. Hence, stated differently, there are 24 values to choose from, and we choose forty of them for each gene. Thus, there are $24^{40} \approx 1.62 \times 10^{55}$ possible LE permutations.

Routing the External Outputs: Genes 40 to 43

Similarly, for the routing of the external outputs, there are four external outputs which can be connected to the output of any of the twenty LEs, or any of the four external inputs. Thus, there are $24^4 \approx 331 \times 10^3$ possible external-output permutations.

Defining the LEs' Logic: Genes 44 to 63

For the logic section of a chromosome, there are twenty LEs, each of which can be configured into one of 16 different logic functions. Again, stated differently, there are 16 possible logic values to choose from, and we choose twenty of them for each Logic Gene. Thus, there are $16^{20} \approx 1.21 \times 10^{24}$ possible logic permutations.