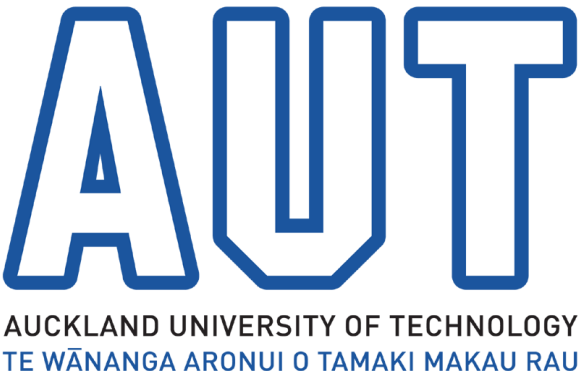
A Generic Platform for the Evolution of Hardware

A thesis submitted to Auckland University of Technology in partial fulfilment of the requirements of the Post Graduate Diploma in Engineering Research

School of Engineering
Auckland University of Technology
By
Abhishek Bedi
Under the supervision of
Dr. John Collins



Acknowledgements

This dissertation is a part of the Post Graduate Diploma at Auckland University of Technology New Zealand.

This piece of work is a result of hard work, patience, sacrifice and unconditional support of many people. I wish to thank everyone who has supported and helped me in completing this research.

Firstly, I wish express my gratitude to Snjezana Soltic for inspiring me to carry out this work. I am thankful to my supervisors Dr. John Collins and Mr. Robert Murphy for keeping patience, being a motivating force and taking care of me during the process. I would like to thank them for the guidance they have given me throughout the period.

I express my honest thanks to the AUT library for their overall support for the entire period of the study at AUT.

At the end, I am grateful to my parents and family members, specially my partner for her extreme sacrifices and providing moral support.

Statement of Originality

‘I hereby declare that this submission is my own work and that , to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for qualification of any other degree or diploma of a university or other institution of higher learning, except where due acknowledgement is made in the acknowledgements.’

Abhishek Bedi

Abstract

Evolvable Hardware is a technique derived from evolutionary computation applied to a hardware design. The term evolutionary computation involves similar steps as involved in the human evolution. It has been given names in accordance with the electronic technology like, Genetic Algorithm (GA), Evolutionary Strategy (ES) and Genetic Programming (GP). In evolutionary computing, a configured bit is considered as a human chromosome for a genetic algorithm, which has to be downloaded into hardware.

Early evolvable hardware experiments were conducted in simulation and the only elite chromosome was downloaded to the hardware, which was labelled as Extrinsic Hardware. With the invent of Field Programmable Gate Arrays (FPGAs) and Reconfigurable Processing Units (RPUs), it is now possible for the implementation solutions to be fast enough to evaluate a real hardware circuit within an evolutionary computation framework; this is called an Intrinsic Evolvable Hardware.

This research has been taken in continuation with project 'Evolvable Hardware' done at Manukau Institute of Technology (MIT). The project was able to manually evolve two simple electronic circuits of NAND and NOR gates in simulation. In relation to the project done at MIT this research focuses on the following:

To automate the simulation by using In Circuit Debugging Emulators (IDEs), and

To develop a strategy of configuring hardware like an FPGA without the use of their company supplied in circuit debugging emulators, so that the evolution of an intrinsic evolvable hardware could be controlled, and is hardware independent.

As mentioned, the research conducted here was able to develop an evolvable hardware friendly Generic Structure which could be used for the development of evolvable hardware. The structure developed was hardware independent and was able to run on various FPGA hardware’s for the purpose of intrinsic evolution. The structure developed used few configuration bits as compared to current evolvable hardware designs.

Table of Contents

List of Figures

List of Tables

Definitions

AMBA protocol: The AMBA® protocol is an open standard, on-chip bus specification that details a strategy for the interconnection and management of functional blocks that makes up a System-on-Chip (SoC).

API: Application Program Interface is a set of routines, protocols, and tools for building software applications. A good API makes it easier to develop a program by providing all the building blocks. A programmer puts the blocks together.

Cell: Here the Cell has been referred to as the building block of the generic cellular structure required for the research. It consists of basic elements required to perform the evolvable hardware functioning.

Cellular Automata: CA evolves in discrete steps with the next value of one site determined by its previous value and that of the neighbour sites

D-type register: A shift register formation consisting of D-type Flip-flops is known as D-type Register.

Flip Flop: A Flip-flop is a simple memory element constructed using logic circuits. It consists of a latch circuit, which can store a state for given input combination.

Generic Platform: The term Generic Platform has been introduced in this research, for the complete generic cellular structure formed by the building block 'Cell'. A generic platform usually consists of 'Cell' in the form of arrays of 1x2, 2x2 or more.

GATE: A gate may consist of one or more inputs and an output depending on the function of its inputs.

Genotypes: The genotype is the specific genetic makeup (the specific genome) of an individual, in the form of DNA. In biology, the genome of an organism is its hereditary information and is encoded in the DNA.

Logical Element (LE): As the name suggest, they are the basic elements that are responsible for logical functioning of programmable logical devices like PAL, FPGA etc.

MPGA: MPGAs are not at all similar to the PLDs in architecture. These devices usually consist of an array of transistors that are pre fabricated into the chips and are customizable by the user into his logic. This customization is done by connecting the transistors with custom wires. In addition, the customization is performed during chip fabrication by specifying the metal interconnect; hence, this requires a lot of manufacturing cost and time.

Multiplexer: A Multiplexer (MUX) is a logic circuit formed together by the combination of a NOT, two AND gates and an OR gate. It is a circuit that generates an output reflecting the state of one of a number of data inputs, based on the value of one or more selection control inputs. A multiplexer can have n number of data inputs with [log2n] select inputs, but only have one output.

NAND GATE: A 'NAND' gate is the combination of an 'AND' followed by a 'NOT' gate.

Shift Register: A flip-flop can store only one bit of information. When a number of flip-flops are joined together with a common clock signal, it is known as a Register. A register that provides the capability to shift the data bits is called a Shift-Register.

Phylogenetics: In biology, phylogenetics is the study of evolutionary relatedness among various groups of organisms (e.g., species, populations).

PLA (Programmable Logic Array): It is a programmable device used to implement combinational logic circuits. The PLA has a set of programmable AND planes, which link to a set of programmable OR planes.

Uniform Crossover: It is a type of crossover in which, each gene of the offspring is randomly selected from the parent gene. This type of crossover can only produce one offspring.

VHDL: It is a hardware description language (HDL) used to design electronic systems at the component, board and system level.

Abbreviations

ANN Artificial Neural Network

API Application Program Interface

ATR Advanced Telecommunications Research Institute International

CAD Computer Aided Design

CHE Complete hardware evolution

CLB Configurable Logic Block

CPLD Complex Programmable Logical Device

DNA Deoxyribonucleic acid

DOD Department of Defence, USA

EC Evolutionary Computation

EHW Evolvable Hardware

EPLD Electrically Programmable Logic Device

ES Evolutionary Strategy

FPD Field-programmable device

GA Genetic Algorithm

GP Generic Platform, this thesis only

GP Genetic Programming

IOBs Input/output blocks -- Sblocks only

LE Logical Element

LUT Lookup Table

MPGA Mask-Programmable Gate Arrays

MUX Multiplexer

NASA National Aeronautics and Space Administration, USA

PAL Programmable array logic

PLA Programmable Logic Array

PLD Programmable Logical Device

POE Phylogenesis Ontogenesis Epigenesis

RAM Random Access Memory

RISC Reduced Instruction Set Computer

RPU Reconfigurable Processing Unit

SPLD Simple Programmable Logical Device

VHSIC Very High Speed Integrated Circuit

VHDL VHSIC Hardware Description Language

Companion CD

The Companion CD provided comes with all the white papers, websites and other electronic sources referred while compiling this thesis. The CD also comes with the experiments done while doing the research both in the Quartus II and MPLab software as well.

1 Introduction

Evolvable Hardware (EHW) is a scheme, inspired by biological evolution, for automatic design of hardware systems. By exploring a large design search space, EHW may find solutions for a problem that is unsolvable using traditional methods or it may find more optimal solutions than those found using traditional methods [1].

Evolvable Hardware involves the same steps as biological evolution. In EHW a Genetic Algorithm (GA) develops a range of circuits (similar to a biological population) in the form of configuration bits (similar to chromosomes), which are downloaded one by one into hardware such as field-programmable gate arrays (FPGA) for fast evolution. The evolved circuits are then fed back to the GA and are compared to the desired circuit. This process keeps on running until the desired circuit is achieved automatically by the system, from the generated population.

A field-programmable gate array or FPGA is a semiconductor device containing up to hundreds of thousands of gates, programmable logic components, switches and programmable interconnects. Early evolvable hardware experiments were conducted in simulation and only the elite chromosome was downloaded to the hardware. Now in modern times, most of the evolution is being done on the hardware.

An Evolvable Hardware System mainly consists of two components, a Genetic Algorithm and Hardware.

As mentioned earlier, in the early evolvable hardware experiments only elite chromosomes were downloaded to the hardware, but now the focus has shifted to generating solutions on the hardware.

This research is a continuation of the Project 'Evolvable Hardware' conducted at Manukau Institute of Technology, Auckland [2]. That project was more oriented towards simulation. The project team was successfully able to generate a simple NAND gate in simulation using a GA. The future work proposed in the project was to get the NAND gate evaluated in the hardware and this became the starting point of this research.

The field of EHW is relatively young but already researchers have not only had to move through different technology platforms such as Xilinx 6200,400 and Virtex® series, but also

evolution friendly features (like, availability of bit stream configuration to the programmer) have disappeared from FPGAs [3]. Due to the new designs of modern FPGAs the bit configuration for an FPGA is not available anymore as it is considered as an intellectual property and hence it is not possible to modify or use the configuration bits for the development of EHW.

Due to this problem, a new approach of developing a 'Generic Cellular Structure' (a high level structure for FPGAs) for EHW has been brought forward to use any available hardware in the market for the development EHW. This research aims to develop such a platform.

Different kinds of hardware available in the market were reviewed for this research, and a sound knowledge was developed of the capabilities of hardware currently available in the global market.

The main requirement of the research was that a simple generic cellular structure with a small chromosome size was to be designed and implemented into FPGA hardware. In addition, this structure was to be verified for the purpose of intrinsic evolution of an electronic circuit. A microcontroller was used for running the genetic algorithm and an FPGA was chosen as the hardware for the generic platform. The research also composed of evolving two basic electronic structures using the Generic Structure with a Genetic Algorithm.

The testing of the generic platform and the genetic algorithm were first to be done in simulation and then they were to be loaded into the hardware for internal evolution as evolvable hardware.

Another requirement for this research, was that the functionality of the evolvable hardware was to be tested using two different Hardware Systems. The two circuits to be evolved were an AND gate and an OR gate.

The circuits evolved by the evolvable hardware system were also to be manually crosschecked for mistakes, to prove the functioning of the developed system.

A literature review including an explanation of evolvable hardware and the history of its elements is given in chapter 2 and chapter 3 respectively. The experiment has been explained in Chapter 4 and Chapter 5 with its solutions analysed in Chapter 6. The final chapter is Chapter 7 where the conclusions of the experiment and future work have been described.

2 Evolvable Hardware Background

In 1992, a new field applying biological evolutionary techniques to hardware design and synthesis was introduced, which gave a new approach for hardware design. The new approach used evolutionary concepts to design innovative and robust circuits automatically. This design scheme was called Evolvable Hardware (EHW) [4].

Higuchi and Furuya [4] first officially proposed the field of Evolvable Hardware at the 2nd International Conference on the Simulation of Adaptive Behaviour. In the words of the proposer, "Evolvable Hardware (EHW) is hardware which is built on software-reconfigurable logic devices (e.g. PLD (Programmable Logic Device) and FPGA (Field Programmable Gate Array)) and whose architecture can be reconfigured by using genetic learning to adapt to the new environment" [5]. The basic idea of EHW is to regard the architecture bits of PLDs as chromosomes of GAs and to find out better hardware structure by GAs, as shown in the figure below:

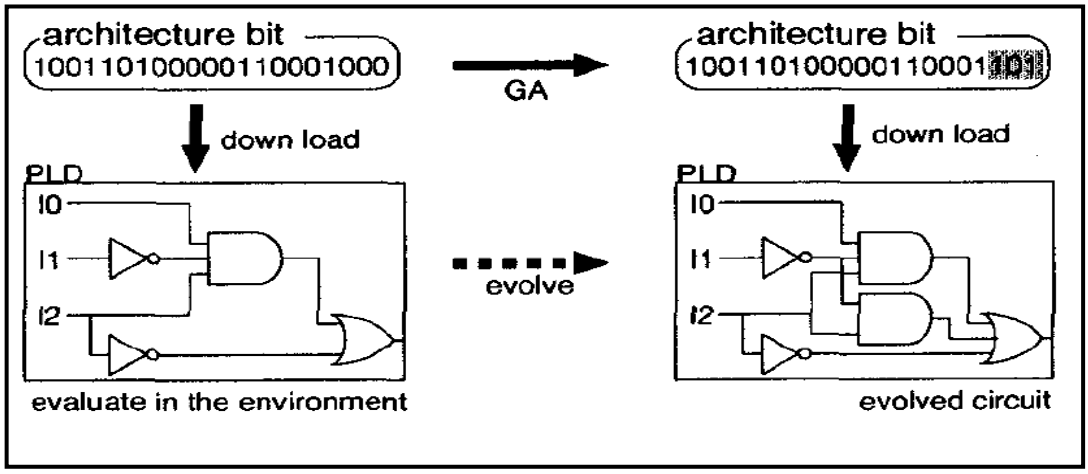


Figure 1 Evolvable Hardware [5]

EHW was considered a system capable of finding solutions to unsolvable problems. The system could also find more optimal solutions than those found using traditional approaches and hence lead to the design of robust systems that found application in the fields of defence, space, automation and fault tolerant systems; this was displayed at the NASA/DOD 2005 conference [6].

After the introduction of the concept many international conferences such as the Genetic and Evolutionary conference, the Congress on Evolutionary Computation, the International

Conference on Evolvable Systems and the NASA/DOD Workshop on Evolvable Hardware were established and since then, new ideas for research in the EHW field have steadily increased [7].

Many research experiments in the field have been carried out around the world including:

- The application to an electro-muscular control artificial arm [8],
- An evolutionary robot navigation system [9],
- Digital Filter design at gate level [10] and
- An Evolved Circuit of a Tone Discriminator [11].

EHW is a young research area and, many organisations around the world are currently working for its further development in various fields of technology. The main research organisations include:

- The Stanford University and NASA in the USA [12], intend to use the evolvable hardware for space research.
- The University of Sussex in Great Britain is working for general research purposes in the field of computers and electronics [13].
- The Electro Technical Laboratory and the ATR (Advanced Telecommunications Research Institute International) in Japan intend to use of EHW in the field of communications [14], and
- The California Institute of Technology in United States of America is one of the other institutes involved in the field [15].
- Although the concept of EHW is relatively new, some EHW applications are already being evaluated for their commercial value [16].

2.2 Definition of Evolvable Hardware

2.2.1 Natural Evolution

Evolvable hardware is a scheme, which was derived from the concept of natural evolution based on Darwin's theory of evolution.

Darwin [17] in his work "The Origin of Species by Means of Natural Selection" has explained the process of natural selection of organisms based on the concept of 'Survival of the Fittest'. The concept of natural selection explains how the weak organisms having more chance of elimination, eventually die and the fit organisms survive and reproduce. In this process of natural selection, the fit individuals produce a new population with their genes crossing over to form new individuals with chromosomes.

The chromosomes developed after crossover, have some different characteristics that may or may not be better than the original chromosomes. This genetic change in a deoxyribonucleic acid (DNA) sequence is known as mutation. These new individuals again go through the process of selection in which the weak are eliminated, and the process carries on producing new population of individuals with steadily improved characteristics.

This concept of natural selection has led to the development of humans and other biological organisms. EHW was invented to design the hardware using the same concept of natural selection.

EHW deals with the designing of analog or digital circuits using the genetic algorithms. This technique acts like an engineer in the design task, and can be used in many different areas.

2.2.2 Hardware Evolution

The field of Evolvable Hardware is a fusion of several different fields. Figure 2 shows the origination of EHW from the intersection of three sciences. As depicted the sciences of biology, computer science and electronic engineering form the basis of fusion for the field of Evolvable Hardware.

As observed by Bentley & Gordon [16], "For many years computer scientists have modelled their learning algorithms on self-organising processes observed in nature. Perhaps the most well known example is the artificial neural network (ANN)" [18]. The work on these learning

algorithms that is based on self-organising processes found in nature is known as bio-inspired software.

Bio-inspired hardware is an established field of electronic engineering that utilises ideas from nature to develop hardware. One recent example of this field is simulated annealing algorithms, which are based on the physical phenomenon of annealing in cooling metals [16].

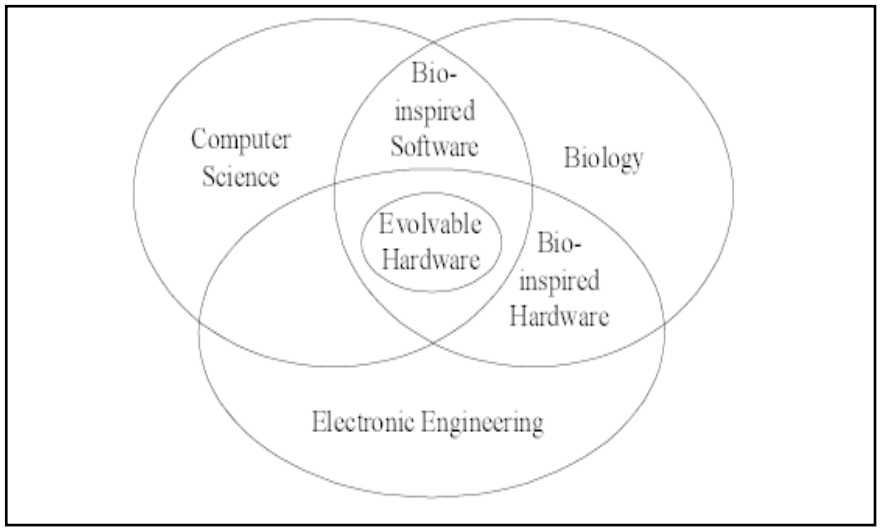


Figure 2 Origination of EHW from the intersection of three sciences [16]

"Evolvable hardware applies techniques derived from Evolutionary Computation (EC), i.e. Genetic Algorithms (GAs), Evolutionary Strategy (ES) and Genetic Programming (GP), to hardware design and synthesis" [19]. These terms are defined below:

Evolutionary Computation (EC) is defined as the field that solves problems using search algorithms inspired by biological evolution [20]. EC involves the same steps as occur in the natural evolution [21].

A Genetic Algorithm (GA) is an algorithm that was adapted from the concept of genes in natural evolution and contains steps like mutation, crossover, reproduction and selection. In this algorithm a population consisting of a lot of circuits i.e. circuit representations, is randomly generated .The behaviour of each circuit is evaluated and the best circuits are combined to generate a new and better population of circuits.

Genetic Programming (GP) is a method for automatically generating computer programs using the process of natural selection [22]. It uses a genetic algorithm to search a computer program

that is nearly most favourable for performing a special task. Even though it is not the first method, but it is so far considered one of the most successful methods of automatic programming [23].

Evolutionary Strategy (ES) is a process that can continuously reproduce new generations, and does trial and selection on the newly generated population. Each new generation is an improvement on the one that went before, thus resulting in systems that are more efficient and more organised than their primitive systems [24]. ES is an important algorithm of GA [25] .It primarily uses real-vector coding, with its search operators being mutation, recombination, and environmental selection. In ES, diversity is not essential because of a greater reliance on mutation, whereas a GA relies more on diversity as crossing over a homogenous population does not yield new solutions.

According to Haddow and Guner [26], Evolvable hardware (EHW) can also be defined as the application of genetic algorithms (GA) and Genetic Programming (GP) to electronic circuits and devices.

Field Programmable Gate Arrays or FPGAs are the electronic devices that are commonly as the platforms for EHW. FPGA are integrated circuit arrays containing of electronic logic hardware that provide designers with reconfigurable logic [27]. It usually contains thousands of programmable elements and interconnects. The interconnects take up a lot of FPGA real estate, resulting in a chip with low gate density compared to other technologies. The programmable logic components can be programmed to duplicate the functionality of basic logic gates (such as AND, OR, XOR, NOT) or more complex combinatorial functions such as decoders or simple math functions.

In most FPGAs, these programmable logic components (or logic blocks, in FPGA terminology) also include memory elements, which may be simple flip-flops or complex blocks of memories.

FPGAs have their historical roots in the complex programmable logic devices (CPLDs) of the early 1970s to mid 1980s. CPLDs and FPGAs include a relatively large number of programmable logic elements. CPLD logic gate densities range from the equivalent of several thousand to tens of thousands of logic gates, while FPGAs typically range from tens of thousands to several million.

In this research, a field-programmable gate array is used as a platform for the technique of EHW and a Genetic Algorithm is used to provide a required design.

The general procedure of the evolvable system is shown below in Figure 3.

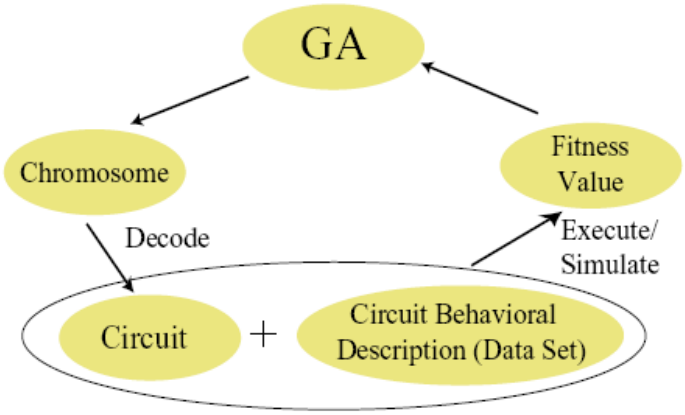


Figure 3 Operation of an Evolvable System [1]

2.3 EHW – Types of Evolution

Early evolvable hardware experiments were conducted by simulation and the best chromosome was downloaded to the hardware for final testing. With the invention of FPGAs, it is now possible for the implementation of solutions to be fast enough to evaluate a real hardware circuit within an evolutionary computation framework; this is termed as an Intrinsic Evolvable Hardware [19].

Hugo de Garis [28] states there are three main methods for achieving evolvable hardware: Extrinsic, Intrinsic and Complete Hardware (on-chip) Evolution. These are shown below in figure 4.

The first method known as Extrinsic EHW is the evolution of electronic circuits through simulation. In this type of evolution, the entire process of evolution including fitness evaluation of the individuals is implemented in software [26] and, at the end of each generation, the best individual is downloaded to the electronic device for final testing.

The second method, Intrinsic EHW, is when each genotype is assessed on the device by downloading the new configuration and testing the device directly.

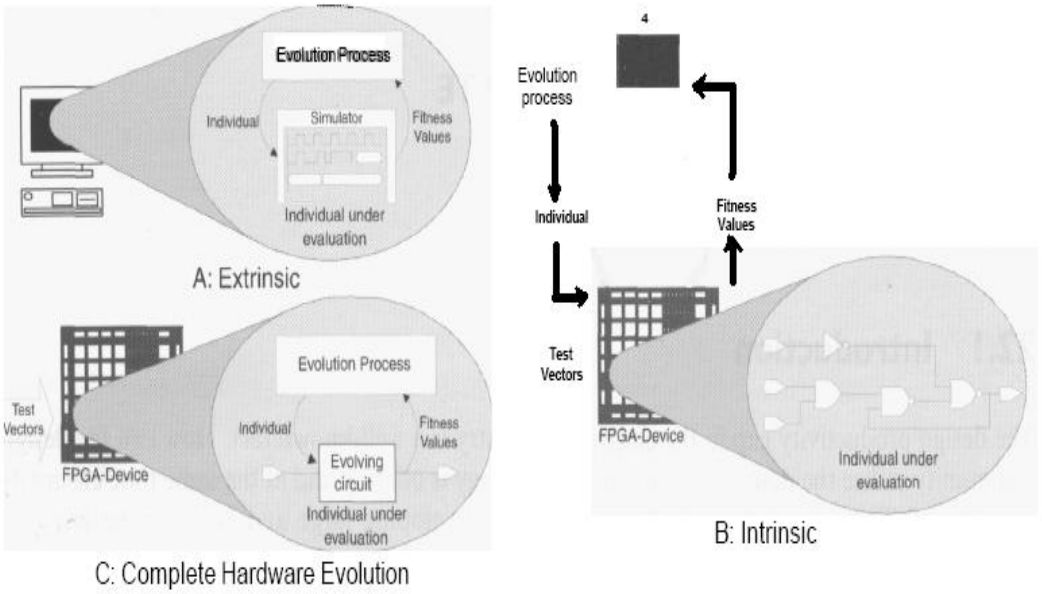


Figure 4 Three types of evolution [26]

The third type and less used form of evolution is an on-chip strategy, also termed complete hardware evolution (CHE) [29, 30]. In this, the complete evolution process is located on the same chip as the evolving circuit. Another similar approach is the use of an on-board processor running the evolutionary algorithm [31].

2.4 Genetic Algorithms

A genetic algorithm (GA) is an algorithm that is capable of finding a solution to a problem by developing a pool of random solutions then working its way towards the an optimum or near optimum solution [32].

The genetic algorithm replicates the same concept of natural selection in computing. In this algorithm a set of circuit representations are first randomly generated, this is the initial population. The behaviour of each circuit is then evaluated (as per the defined fitness function in the genetic algorithm) and the best circuits are combined to generate a new array of circuits that hopefully includes a better circuit solution [1] .

Each individual circuit description is known as a genotype. Genotype describes the genetic constitution of an individual, that is the specific allelic (an allele is a viable DNA coding that occupies a given position on a chromosome) makeup of an individual. The genotypes consist of an array of bits and each bit contained in them is known as a gene.

These newly generated genotypes are passed through the same process, until a new fittest circuit is evolved to behave according to the specification desired by the user. Mutation can also occur in the generated population. This can lead to a chromosome with better fitness. Thus, the final design is based on incremental improvement of a population, initially which was randomly generated. Figure 5 below shows pseudo code for a simple genetic algorithm. The flow chart for a genetic algorithm is shown in figure 42.

```
SimpleGeneticAlgorithm ( )
{
        Initialize the Population;
        Calculate Fitness ;

        While(Fitness Value != Optimal Value )
        {

        Selection;
        Crossover;
        Mutation;

        Calculate Fitness ;
        }
}
```

Figure 5 A Simple example of Genetic Algorithm [2] and [33]

2.4.1 Genetic Algorithm Terminology

Population size: This is the number of chromosomes in one generation. This number should not be too small as this causes only a small part of the search space to be explored. On the other hand, too many chromosomes slow down the GA. Population size is chosen depending on the nature of the evolution being done.

Selection: It is the first operator of a genetic algorithm that selects chromosomes in the population for reproduction. The fitter the chromosome, the more times it is likely to be selected to reproduce. Selection can be done using various selection techniques such as Roulette Wheel selection, Tournament selection, Random selection etc. For example, in the roulette wheel selection also known as stochastic sampling with replacement [34], the

individuals are mapped to contiguous segments of a line, such that each individual's segment is equal in size to its fitness. A random number is generated and the individual whose segment spans the random number is selected (with a possibility of same individuals being selected as well). Hence, the probability of selecting individual is proportional to its fitness. The process is repeated until the desired number of individuals is obtained. This is called mating population. This technique is analogous to a roulette wheel with each slice proportional in size to the fitness, see figure 6.

Number of individual	1	2	3	4	5	6	7	8	9	10	11
fitness value	2.0	1.8	1.6	1.4	1.2	1.0	0.8	0.6	0.4	0.2	0.0
selection probability	0.18	0.16	0.15	0.13	0.11	0.09	0.07	0.06	0.03	0.02	0.0

sample of 6 random numbers:

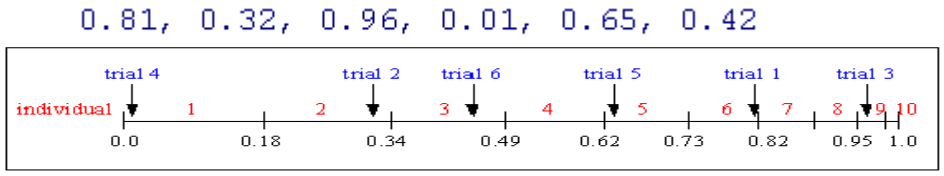


Figure 6 Roulette Wheel Selection [35]

Crossover: It is a GA operator that randomly chooses a locus and exchanges the subsequence before and after the locus between two chromosomes to create two offspring. The crossover method emulates the process of natural crossover by generating offspring that carries forward the important genetic material of the parents, whilst introducing enough variation so that they can potentially become fitter than the parents can. For example as shown in figure 7, the strings 00000000000000 and 11111111111111 could be crossed over after the seventh locus in each to produce the two offspring 00000001111111 11111110000000. Crossover shown below is an example of single point crossover where a single locus has been chosen to crossover the chromosomes, but crossover can also be done at multiple points and is known as multi point crossover.

Mutation: This is also a GA operator, which randomly flips some of the bits in a chromosome. Mutation prevents the GA from being stuck to a local maximum of fitness. For example, as shown below third bit is flipped in first offspring to get 00100001111111 and sixth bit is flipped in the second offspring.

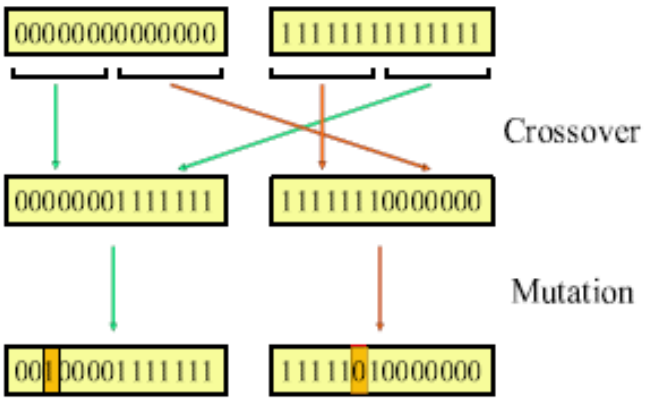


Figure 7 Crossover and Mutation [1]

2.4.2 A Simple Genetic Algorithm[36]

I. [Start] Generate a random population of p *n*-bit chromosomes (candidate solutions to a problem).

II. [Fitness] Calculate the fitness *f(x)* of each chromosome in the population.

III. [New Population] Create a new population by applying the following steps (4 to 7) until *p* offspring have been created:

IV. [Selection] Select a pair of parent chromosomes from the current population according to their fitness (chromosomes with better fitness have a bigger chance to be selected).Selection is done "with replacement," meaning that the same chromosome can selected more than once to become a parent.

V. [Crossover] With the given crossover rate $p_{c,}$ the parent genes are crossed over to form new offspring at a randomly chosen point. If no crossover is performed, then the offspring are an exact copy of the parent.

VI. [Mutation] Mutate the two offspring at each locus with mutation rate $p_{m,}$ and place the resulting chromosomes in the new population.

VII. If *p* is odd, one new population member can be discarded randomly.

VIII. [Replace] Use the new generated population for further processing by replacing the current population.

IX. [Loop] Go to step 2.

3 Hardware Platforms

Field Programmable Gate Arrays (FPGAs) are digital circuits that can be reconfigured, and thus they are excellent candidates for implementing EHW [3]. Commercial FPGAs are based on a 2-dimensional array of cells, in which it is possible to define the cells' functionalities and routing.

As noted by Pauline Haddow [3], current development in FPGA designs have led to disappearance of evolvable hardware friendly features from FPGAs, due to the change in design of new FPGAs which does not give the option of configuration bits to be controlled by a programmer.

The most widely used FPGA for EHW experiments a decade ago was the VIRTEX® XC6200 [11, 37-40] . Hence, this chapter will emphasise the design of evolvable friendly hardware that is available or being developed by different researchers around the world.

The chapter has been mainly divided into five sections, the first two sections describe the history and development of the FPGA. The third section analyses the XC6200 architecture, as it was one of the best evolvable friendly FPGAs and was the basis of many evolvable experiments. The following section discusses recent research being conducted on designing new hardware chips for the purpose of evolvable hardware. The fifth section describes the hardware structure that forms the basis of our design.

3.1 The History and Development of FPGA

The history of the FPGA goes back three decades, to the 1970s when the first Programmable Logic Devices (PLDs) were introduced [41], for implementing logic circuitry in electronic chips. Since then many modifications and developments have been made in the architecture and structure of these devices to meet the needs of electronics. They have been termed as field-programmable devices (FPDs), but we will be referring them as PLDs.

There are several types of PLD available commercially. The three main categories of PLDs are:

- Simple PLDs (SPLDs),
- Complex PLDs (CPLDs), and
- Field-Programmable Gate Arrays (FPGAs)

Simple PLDs usually refer to small types of PLDs, generally containing two planes of logic. The first to be developed in this category was the Programmable Logic Array (PLA) containing two programmable planes, an AND-plane and an OR-plane .Both these planes are programmable by the user and are used to generate logic functions using the 'Sum of Products' form of digital logic.

Typical parameters of a PLA are sixteen inputs, thirty-two product terms and eight outputs. PLAs are efficient in terms of area needed for their implementation on an integrated circuit chip and hence they usually form a part of larger chips such as microprocessors. PLAs were difficult to fabricate and they reduced the speed-performance of circuits implemented in them. This led to the development of a similar device in which the AND- plane was programmable but the OR-plane was fixed [36]. This device was known as programmable array logic (PAL). PAL is a trademark of Advanced Micro Devices Corporation [42].

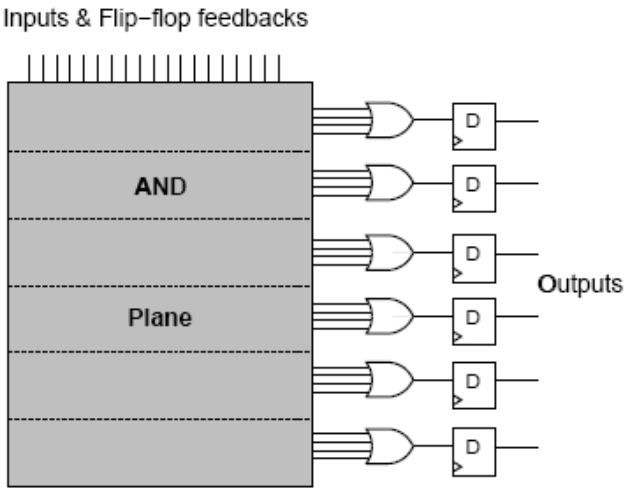


Figure 8 Structure of a PAL [43]

The second category of PLDs, known as *Complex Programmable Logic Devices*, was introduced in the early to mid 1980s. CPLD logic gate densities range from the equivalent of several thousand to tens of thousands of logic gates. A CPLD is also known as Enhanced PLD (EPLD), Super PAL or Mega PAL [43].

A CPLD comprises of PLA or a PAL-like structures together with input-output blocks and interconnection wires. Normally extra circuitry is added to the output of a PAL. This structure as a whole is known as a Macrocell. This macrocell forms the building block of a CPLD; a macrocell is illustrated in figure 9.

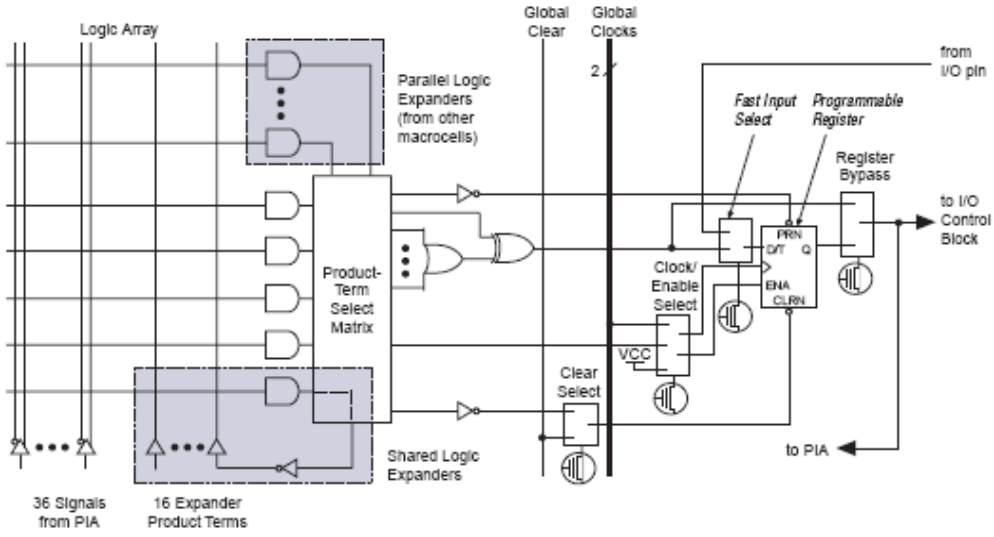


Figure 9 A Macrocell of MAX 7000 [44]

CPLDs were pioneered by Altera, first in their family of chips called Classic EPLDs, and then in three additional series, called MAX 5000, MAX 7000 and MAX 9000. A typical CPLD can provide a functionality equivalent to 50 SPLD devices, but for higher logic capacity, a different approach is required [43].

3.2 General Architecture of an FPGA

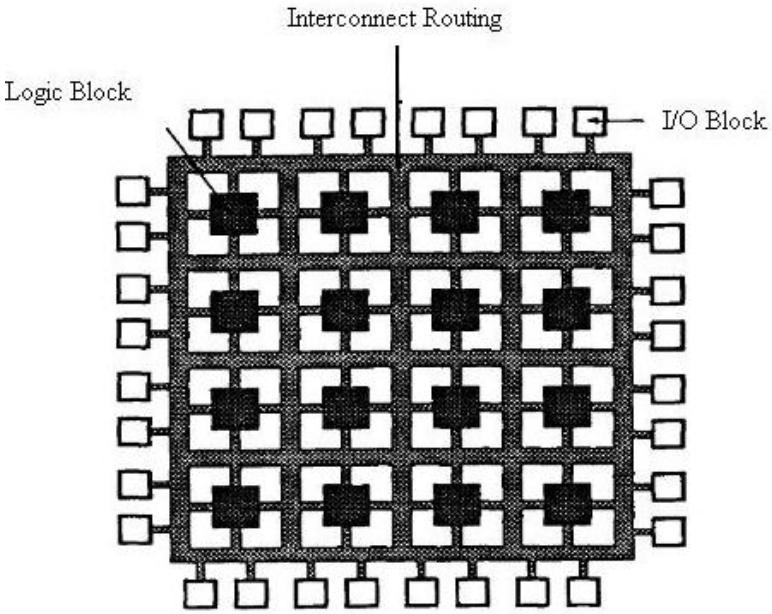


Figure 10 Structure of an FPGA [45]

As illustrated in the above figure, an FPGA consists of an array of logic blocks and interconnect resources, which can be configured through programming to realize different designs.

There is an architectural difference between a PLD and an FPGA. The PLD has a more restrictive structure consisting of one or more programmable sum-of-products logic arrays feeding a relatively small number of clocked registers, leading to predictable timing delays and a higher logic-to-interconnect ratio. The FPGA architectures, on the other hand, are dominated by interconnects which make them more flexible for larger designs, but also far more complex to design for relatively smaller designs. FPGAs can achieve higher level of integration than PLDs, due to more complex routing architectures and logic implementations [45].

An FPGA can be as simple as a transistor or as complex as a microprocessor. It is typically capable of implementing a lot of combinational and sequential logic [45].

FPGAs include a relatively large number of programmable logic elements thus allowing a very high logic capacity [43]. In general, CPLD logic gate densities range from the equivalent of several thousand to tens of thousands of logic gates, while FPGAs typically range from tens of thousands to several million gates.

3.3 Xilinx XC6200 FPGA

FPGAs often took a number of seconds to reconfigure in the past, making them too slow for EHW. This changed with the development of the Xilinx XC6200 series devices. The device held the configuration in Static RAM, meaning that it could be quickly accessed, read and changed, thus making it the favourite choice for EHW.

The main evolvable favourable features of XC6200 were:

- Fast reconfiguration: The device used a parallel interface rather than the conventional serial approach and therefore was able to be configured much faster than previous devices.
- Known data format: The bit-stream format was available so that the user could alter individual parts of the configuration; this was a useful feature for EHW.
- Safe configuration: The device had been designed in such a way, that it restricts the connections between the logic block. Thus, it was safe to load any random configuration into the hardware [46].
- Routing implementation: Its routing implementation was based on multiplexers rather than on anti-fuse or memory bits, (short circuits could be generated in almost every other FPGA) [47].

Other features of the XC6200 were:

- Microprocessor interface: A standard microprocessor interface to static RAM was used to configure the device.
- Partial reconfiguration: Configuring the device was easier and was not interlinked between different areas of the device; hence, configuration in one area could be changed without affecting another area of the device [46].

- However, the production of XC6200 was stopped, leaving the EHW world with no choice except to develop its own evolvable friendly hardware.
- Most of these EHW hardware chips were based on the architecture of the XC6200 due to its reconfigurable characteristics, which prevented the outputs of a structure from being connected together. A brief description of this architecture is given below.

3.3.1 Logical and Physical Organisation

The XC6200 was a second-generation fine-grain architecture, employing a hierarchical cellular array structure[48]. The XC6200 architecture may be viewed as a hierarchy. A large array of simple cells lies at the lowest level of the hierarchy (Figure 11) and has been termed as a 'sea of gates'. Each cell in this array is individually programmable to implement a D-type register and a logic function such as a multiplexer or gate. Any cell may be configured to implement a purely combinatorial function, with no registers involved.

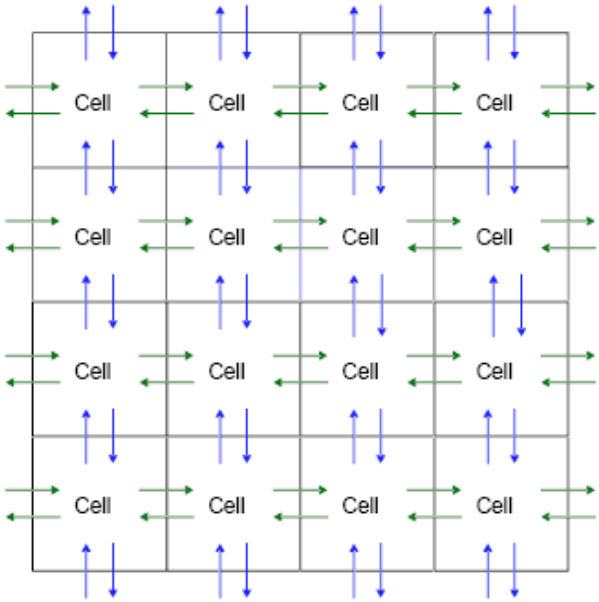


Figure 11 Nearest Neighbour interconnect array structure [48]

The structure of the XC6200 is a combination of an array structure formed by connecting neighbouring cells, which are then grouped into different levels of hierarchy such as a unit cell, 4x4 cellblocks, 16x16 cellblocks, 64x64 cellblocks etc.

In addition, each level has its own routing resources. Wires of length 1 are provided to allow basic cells to route across themselves and length 4 wires allow 4x4 cells to route amongst themselves. Larger XC6200 products extend this process of routing wires, by using a scaling factor of four at each hierarchical level. There are also long wires at each level, which are of chip-length and are termed as 'FastLANEs ™'. The structure is shown below.

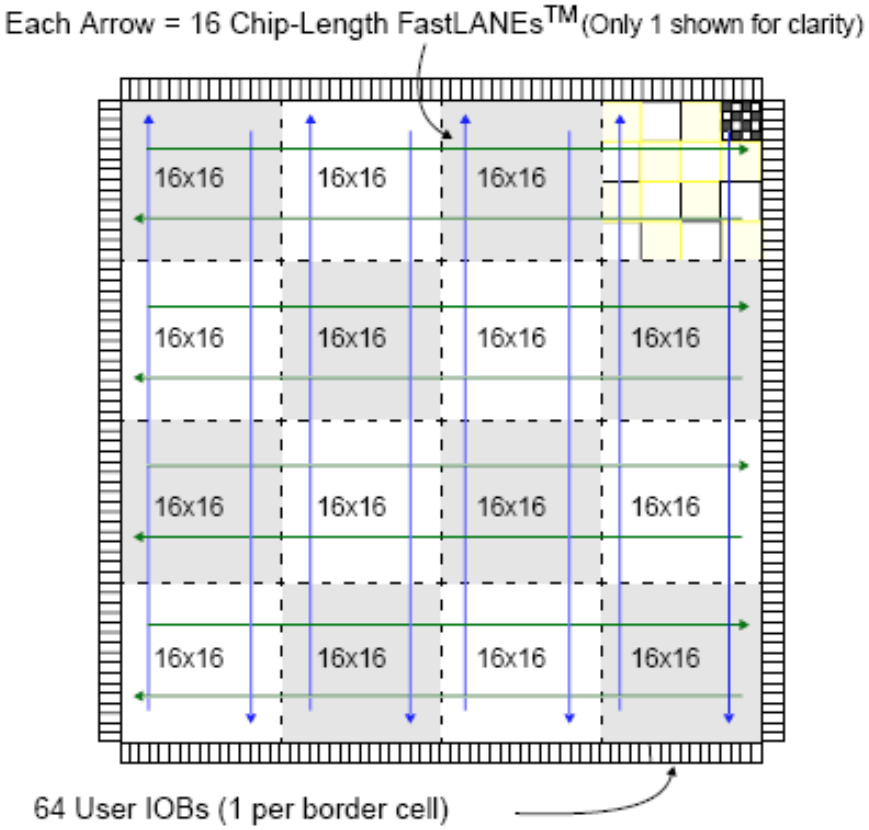


Figure 12 XC6200 structure [48]

3.3.2 Basic Cell

The basic cell of the XC6200 is shown in figure 13 below. Here the inputs from neighbouring cells are labelled N, S, W, E and those from the length 4 wires N4, S4, W4 and E4 according to their signal direction. Additional inputs include clock and asynchronous clear for the functional unit D-type register. Here, the output of the cell function unit implementing the gates and registers has been labelled as F. There is another output, labelled as 'Magic'. The magic output is an additional routing resource located in each cell, but is not always available for routing. The availability of the magic output is dependent on the logic function

implemented in the cell. Bits within the configuration memory control the multiplexers within the cell.

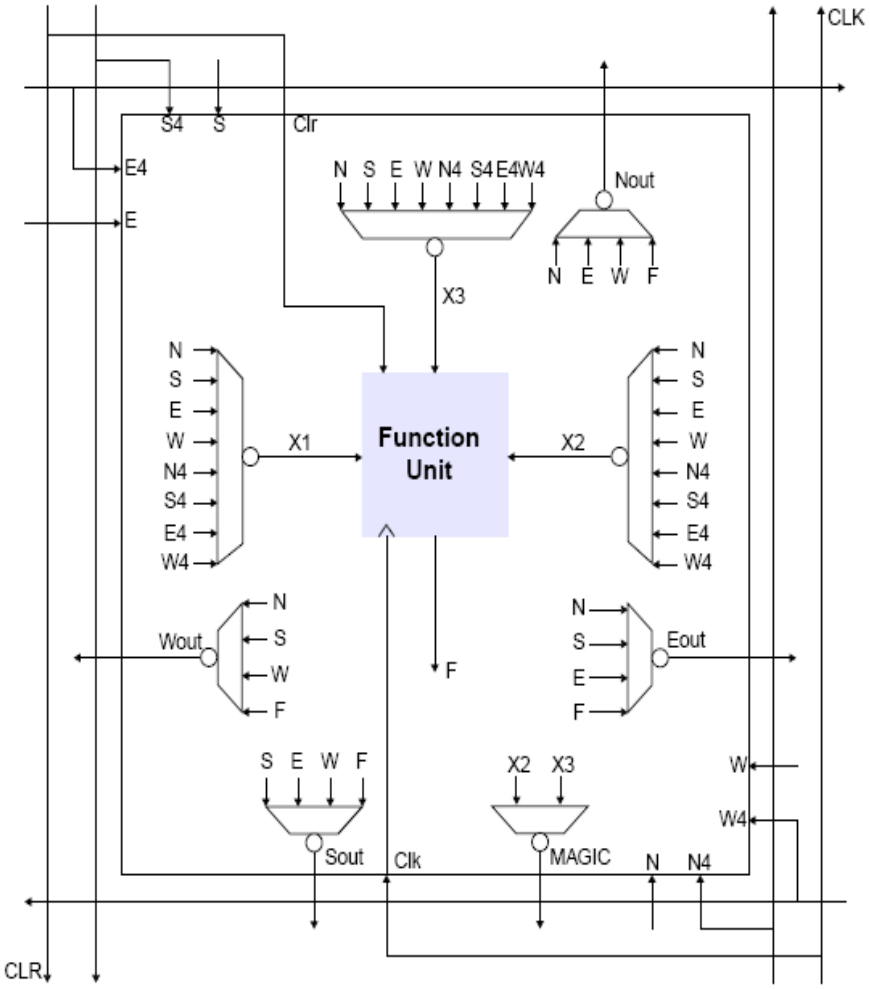


Figure 13 XC6200 Basic Cell [48]

3.3.3 The Configurable Logic Block Structure

The Function Unit is also known as the Configurable Logic Block (CLB).The implementation of the XC6200 CLB is shown in figure 14 below. Y2 and Y3 are the input multiplexers and provide the conditional inversion of the X2 and X3 inputs. The CS output multiplexer selects a combinational or sequential output based on the programming. The RP multiplexer allows the contents of the register (D flip-flop) to be 'protected'.

Hence, there are two paths in the structure:

- *The Sequential Path*: Passing through the input multiplexers and then through the flip-flop [49].
- *The Combinational* Path: Bypasses the flip-flop.

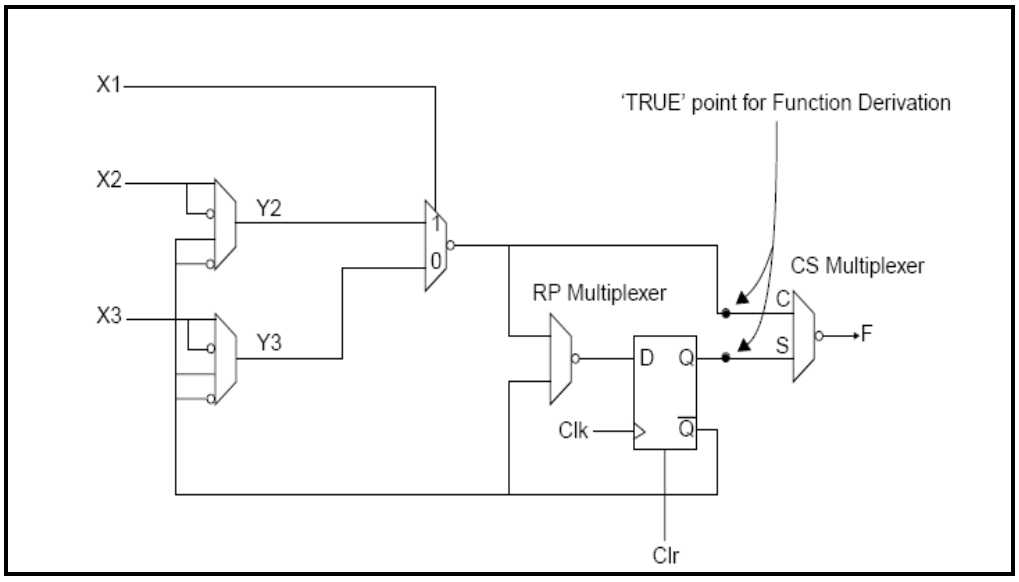


Figure 14 The Configurable Logic Block / The Function Unit [48]

3.4 EHW friendly Structures

Based on the above features and the design of the XC6200, researchers have tried to create different evolvable friendly designs like POEtic chip [47] , evolvable motherboard [50] , Processing Integrated Grid (PIG) [51] and ETLs GRD chip [31]. Some researchers have also tried to improve the XC6200 design for more power and performance, e.g. Gigahertz FPGAs with new power saving techniques.

This section will describe two designs, which may have an effect on EHW development in the future. In addition, the inefficiencies of these designs that lead to this research have been explained later in this chapter.

3.4.1 POEtic Chip

The POEtic chip is a new system on chip platform, which is intended to compensate for the unavailability of the Xilinx XC6200, for the field of EHW. The POEtic chip has been specifically designed to ease the development of bio-inspired applications. The composition of the chip consists of a microprocessor in the environmental subsystem and a 2D reconfigurable array called the Organic Subsystem. The reconfigurable array consists of basic elements called molecules, that are mainly 4 input look up tables (LUT) and flip flops. There is a second layer in the organic subsystem that implements a dynamic routing algorithm that is intended to allow multi chip designs, letting the user work with a bigger reconfigurable virtual array.

The name of the POEtic chip was inspired from the three-life axis of nature:

- Phylogenesis is the way species are evolving, by transmitting genes from parents to children.
- Ontogenesis corresponds to the growth of an organism and self-healing in living beings.
- Epigenesis deals with learning capabilities like the brain.

3.4.1.1 The Microprocessor

The microprocessor is a 32-bit RISC processor, specially designed for the POEtic chip. It exposes 57 instructions, two of which give access to a hardware pseudo random number generator for the evolutionary process. There is an AMBA bus [52], which is used for communication with all the internal elements, (as shown in figure15) as well the external

world. It also can be used to connect several POEtic chips to create a bigger array. The microprocessor can configure the array, and retrieve its state. Access to the array is made in a parallel manner because the array is mapped onto the microprocessor address space. As a result, it is fast to configure or reconfigure the array.

The researchers have also developed a C compiler and an assembler, making it easy for the user to write programs. In addition, they plan to supply an API that can help the user to build a genetic algorithm by choosing the type of crossover, the selection process, and so on.

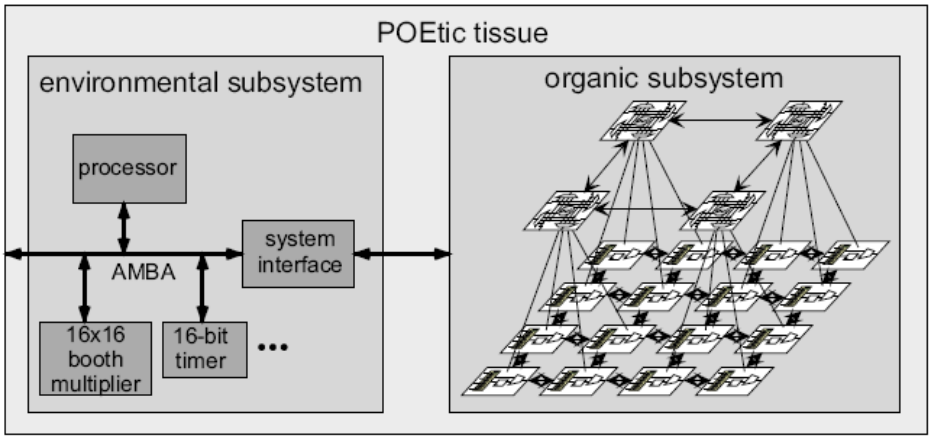


Figure 15 The POEtic chip showing the Microprocessor, and the Reconfigurable Array [47].

3.4.1.2 The Reconfigurable Array

The reconfigurable array of the chip comprises of two planes: The first plane is a grid of basic elements, called molecules, mainly consisting of a 4-input look-up table, and a flip-flop. The second plane is a grid of routing units that can dynamically create paths at runtime between points of the circuit. These routing units implement a distributed dynamic routing algorithm, based on addresses. The array can be used to create connections between cells in a cellular system to connect chips together or can be used to create long distance connections at runtime. The molecules execute a function, according to an operational mode defined for each molecule by three configuration bits (refer [53] for details). There are eight operational modes for the molecule.

3.4.1.3 Molecular communication

As in the XC6200, inter-molecular communication is implemented with multiplexers. This feature avoids short circuits that could happen when partially reconfiguring a molecule, or during an unconstrained evolution process. As shown in figure16, every molecule is directly connected to its four neighbours, sending them its output while long-distance connections are implemented by the way of switch boxes. Each cardinal direction provides two input lines and two corresponding outputs. The six input lines from the cardinal directions or from the output of the molecule (or the inverse of both) can be used to select each output.

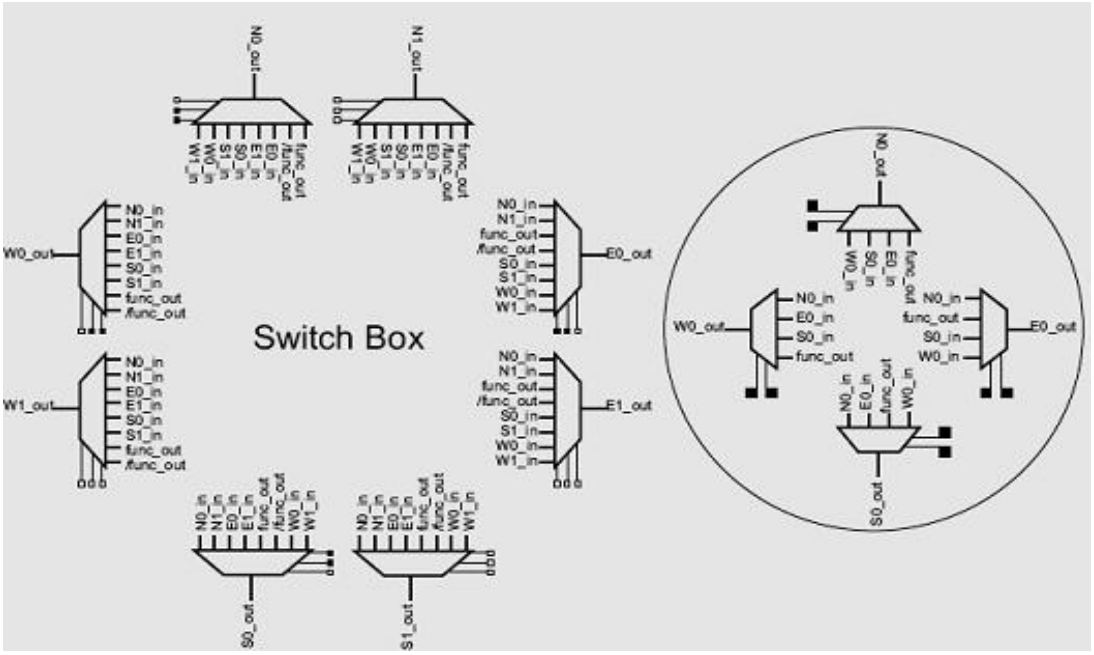


Figure 16 POEtic chip -- Switch Box [47]

3.4.1.4 Configuration Bits

The POEtic chip has seventy-six configuration bits. The bits are split into five blocks as shown in Table 1. The first bit is used to indicate whether reconfiguration is required in a block or not. In terms of execution, the microprocessor has a 32-bit bus to access these bits. As there are only two clock cycles needed to write and three words of 32 bits define a molecule, the configuration of the entire array is very fast. The reconfiguration is made in parallel as compared to Xilinx's JBits ([46]and [54]), in which the entire bit stream is sent each time in serial.

Table 1. THE FIVE BLOCKS OF CONFIGURATION BITS FOR POETIC HIP [47]

Number of bits	Description
1	global partial configuration enable
2	configuration input origin
1	lut partial configuration enable
16	lut(15 downto 0) (cf. figure 2)
1	lut inputs configuration enable
14	selection of the lut inputs (cf. figure 4)
1	switchbox partial configuration enable
8x3	3 bits for each of the 8 multiplexers (cf. figure 3)
1	mode partial configuration enable
3	operational mode (cf. section 3.2)
1	other bits partial configuration enable
1	sequential or combinational output
1	flip-flop reset value
1	dff enable used or not
1	clock edge
3	local reset origin
1	local reset enable
1	asynchronous/synchronous reset
1	molecule enable
1	value of the flip-flop

3.4.1.5 Conclusion of the Design

The POEtic chip is a useful EHW platform. It has dynamic routing capabilities that allow functional level evolution using sine generators, adders, multipliers as building blocks [47]. The entire bit stream can be used to execute an unconstrained evolution. At present, a test chip containing around 12 molecules is being fabricated. After the functional test of this chip, the researchers intend to manufacture a final POEtic chip, containing about 200 molecules.

3.4.2 EHW Chip

This section discusses the second chip known as the EHW Chip. Generally, there are two main restrictions associated with the EHW:

- Slow learning speed of the systems and
- Large size of the EHW systems

To overcome these problems, a gate-level Evolvable Hardware Chip has been described [55]. The chip intends to integrate both the GA hardware and the reconfigurable hardware within a single LSI chip. This chip was initially proposed in 1998 [56] and now the same team has developed an improved design. A block diagram of the EHW chip is shown in fig 15. The chip consists of a genetic algorithm (GA) unit, a PLA (Programmable Logic Array) unit as the reconfigurable hardware logic, registers, and control logic. The main advantage of the chip design is that it has two ports for parallel access connected to the external two-port RAM and hence, it can process two chromosomes at a time in parallel. The GA unit and PLA unit also have a parallel accessing architecture for the data stream from the two ports. The chip has to be connected to an external memory and a CPU.

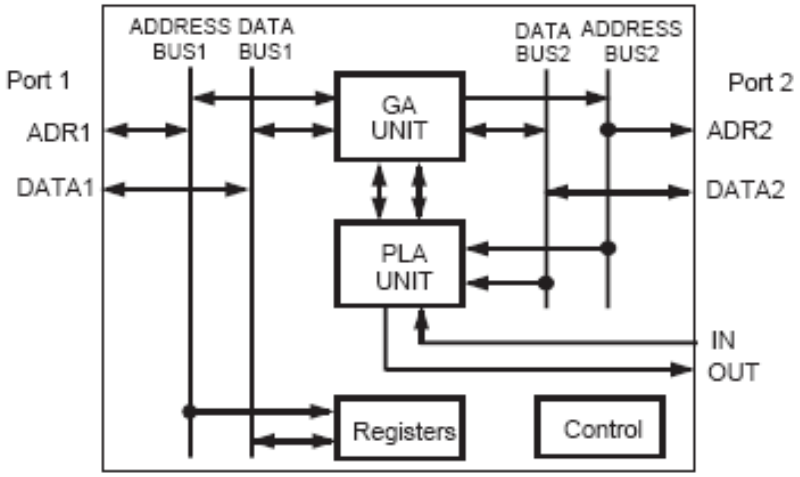


Figure 17 Block Diagram of EHW chip [55]

3.4.2.1 The GA Unit

This unit executes the GA learning operations using the steady state GA (In a steady-state genetic algorithm one member of the population is changed at a time) and elitist recombination (in this children compete with their parents to be included in next population)

[57]. The block diagram of the GA unit is shown in figure 18. Here the GA unit is used to select two chromosomes in units of 32 bits, from the chromosome memory in parallel. Then it carries out uniform crossover and mutation on these to make two chromosome segments of 32 bits. Uniform crossover is carried out using a random 32-bit string. A mutation rate of zero, 1/256, 2/256, or 3/256 can be selected. The two new chromosome segments are then sent to the PLA. After all chromosome segments have been sent, their fitness values are calculated using training patterns. In addition, this chip has an on-line editing mode for the training-pattern memory and it allows the changing of training-pattern memory during learning, to provide online learning. This helps to ensure a smooth adaptation process[8].

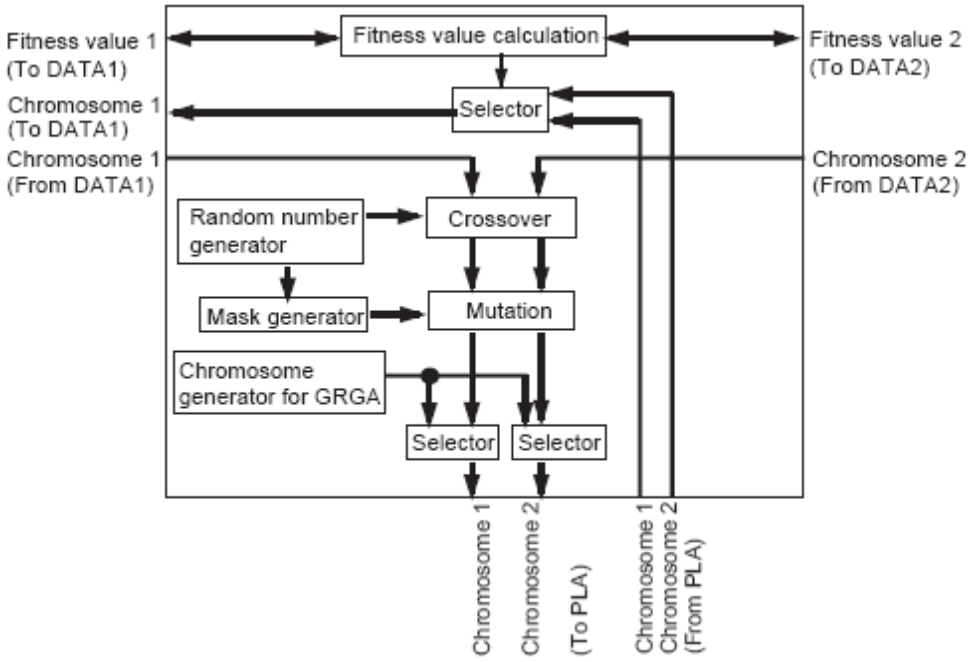


Figure 18 GA unit Block diagram [55]

3.4.2.2 The PLA unit

There are two PLA blocks for parallel evaluation of two circuits in the PLA unit. These blocks read two chromosomes from the GA unit in parallel in units of 32 bits to implement two circuits in parallel. The evaluation of two cells is the done by using the training data. There are two input-output modes available for selection: an 8-bit input/8-bit output mode or a 12-bit input/4-bit output mode.

The architecture of the PLA is shown in Figure 19. In the PLA, 32 product term lines are divided into two groups of 16 lines. Each bit of the output (8-bits) from the two groups can be connected to either an OR, or an XOR gate. The operation is the same as a conventional PLA if the OR gate is selected. If the XOR gate is selected, then the XOR operation is executed on each bit of the two outputs from the two groups. If XOR is used with 'AND' and 'OR' gates, the PLA can generate a circuit with less product term lines [58]. This option is useful for circuits that need many product term lines.

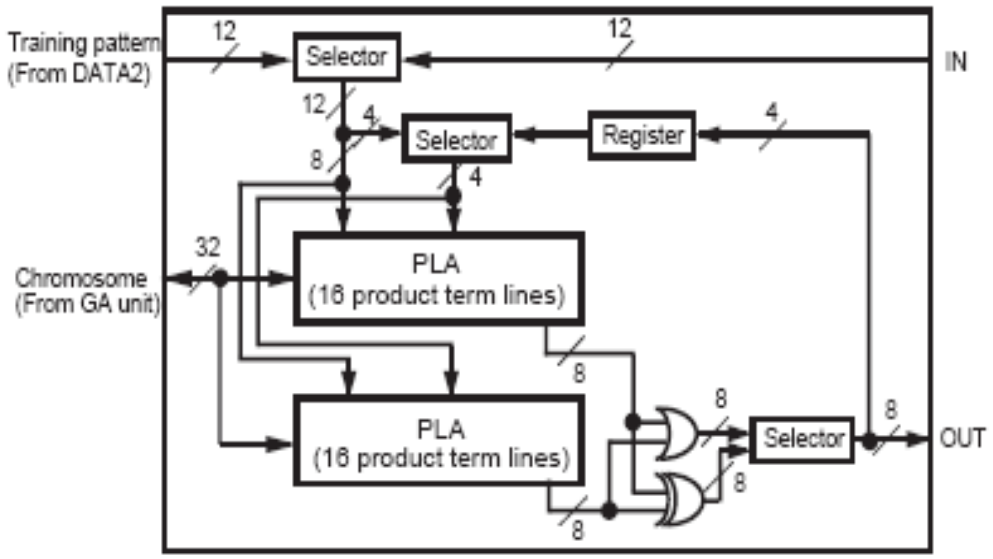


Figure 19 Block diagram of one PLA [55]

An option for selecting a feedback loop from the output to the input is provided to each PLA. If this option is selected, the upper 4 bits of the PLA output is connected to the upper 4 bits of the PLA input via a register (Figure 19). As the feedback loop can store the state of the circuit, it can be used when the EHW has to learn a sequential circuit.

3.4.2.3 Random Number Generator:

A parallel random number generator using cellular automata [59] had been selected for implementation on the EHW chip, and could produce a 560 bit random bit-string at every clock cycle.

3.4.2.4 The RAM

The EHW chip works with an external 2-port RAM on the board. It has been divided into three memories: a chromosome memory, a training pattern memory, and a memory for the fitness value. All the individuals in 16 bits x 2048 words are stored in the chromosome memory. The chromosome length is 1024 bits, and the population number is 32. This memory has two input/output ports. Two chromosomes of 16 bits can be read or written in parallel from the GA unit using these two ports. The training data memory can store a maximum of 256 training data set of 16 bits each. The memory for the fitness data stores all the fitness values for all 32 chromosomes with an 8-bit integer value.

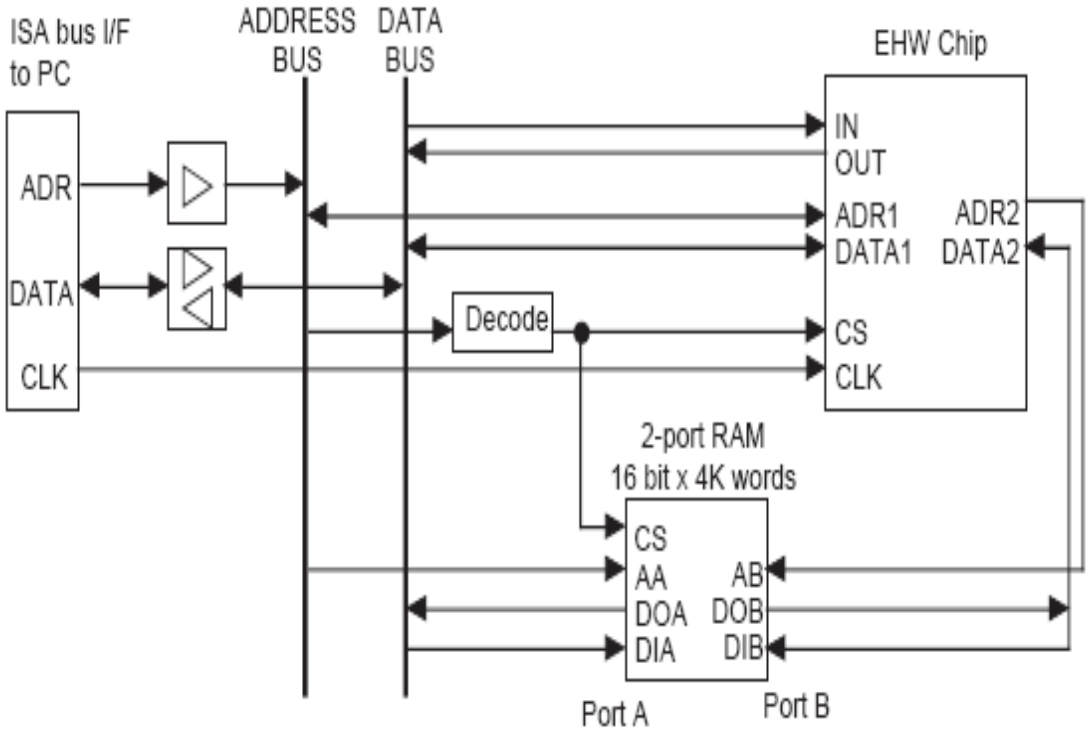


Figure 20 Block Diagram of the complete EHW chipboard [55]

3.5 The Virtual Sblock FPGA

Another type of design that has been proposed by a group of researchers [3] is known as the Virtual Sblock FPGA. This design is a virtual evolution friendly reconfigurable platform that can be mapped onto a given technology, and thus was chosen as the starting point for the development of a generic platform in this research.

The Virtual Sblock FPGA is a technology independent platform for evolvable hardware. The key feature is that it is a more evolution friendly hardware platform for evolving digital circuits than commercial FPGAs. However, the platform may be mapped onto today's FPGAs.

3.5.1 Architecture

This virtual EHW platform consists of blocks (named Sblocks) that have been laid out as a symmetric grid. Each Sblock connects to the Sblocks on its four sides (north N, east E, south S and west W). The output value of a Sblock is synchronously updated and sent to all its four neighbours (its von Neumann neighbourhood), and as a feedback signal to itself.

Each Sblock consists of both a simple logic/memory component and routing resources. The Sblock can be configured as a logic or memory element or it can be configured as a routing element to connect one or more neighbours to non-local nodes. Several nodes of Sblocks can be connected together as routing elements to realise longer connections.

The internal connections of the Sblocks include four pairs of unidirectional wires attached through routing logic to a routing channel. Each pair includes one input and one output connection.

The routing logic chooses the appropriate wire of the routing channel and forwards this to the neighbouring Sblock. Incoming data is forwarded to the appropriate channel wire for either forwarding to logic/memory or to the given output channel. The Sblock is illustrated in figure 21. As each Sblock can communicate on each of its edges to neighbouring Sblocks, this provides a symmetrical and scalable architecture.

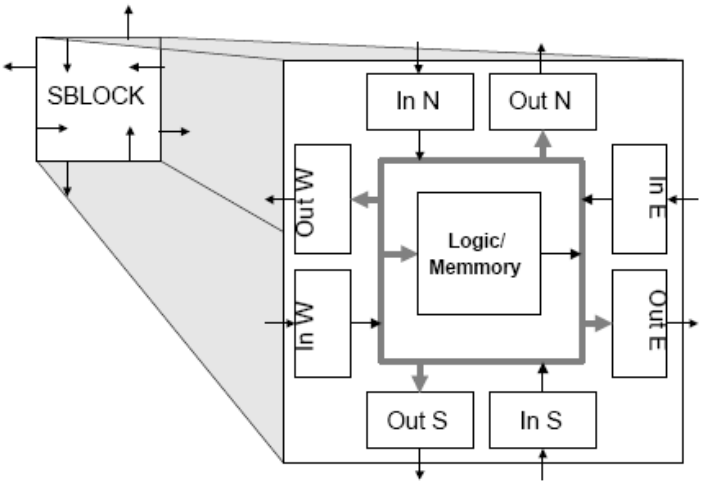


Figure 21 Sblock – Routing and Logic/Memory Block [60]

The internal routing of each Sblock only allows inputs to be routed to outputs and the interface between Sblocks only allows outputs to be routed to the inputs. This way, generation of illegal configurations by evolution is prevented. IOBs (input/output blocks) are placed at the perimeter of the chip. Researchers have also proposed an on-board oscillator for their architecture [24]. A more detailed view of the Sblock logic is shown in figure below.

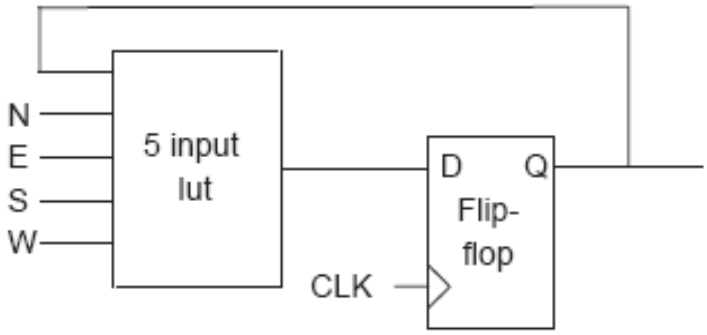


Figure 22 Sblock Logic [3]

Inputs from the neighbouring Sblock and a feedback from the output are connected to a five input lookup table (LUT). The LUT can be configured to hold a function. If Don't Care (DC) bits are placed by a GA at a given input, then that neighbouring Sblock is not connected to it. In this way the LUT is programmed not only for desired functionality but external connectivity of sblocks [3].

3.5.2 Configuration

The Sblocks have two global configuration buses, an address and a data bus that traverses the grid. Each bus is 16-bits wide. The address bus can thus address approximately 64000 CLBs. The buses traverse each row in the grid thus allowing both serial and column parallel configuration of the complete grid. Addressing each CLB individually opens up the possibility for partial reconfiguration. The number of frames needed to address a complete column is 48 frames, as 48 bits are needed to configure a single CLB. This restricts partial configuration to one or more columns.

3.5.3 Configuration Data

In the current structure of Sblock, routing is an interdependent resource of logic/memory. Each Sblocks' configuration data includes its necessary routing and logic data.. This also decreases the speed of evolution, since the process handles more individuals and the configuration time is higher with a lot of routing configuration data to handle.

3.5.4 Feedback of Information

In the Sblock, any CLB, IOB, or range of CLBs/IOBs may be accessed through the configuration buses to read back the data lying in these blocks.

3.6 Analysis of These Designs

Although the designs presented are evolution friendly they all (except Sblock) have a common drawback of being confined to one FPGA design or one range of FPGAs .The POEtic chip is a chip itself with limited capabilities going up to a maximum of 256x256 molecules, also it is still under trial and has not been manufactured. Whereas, the EHW chip is again a supporting chip for the FPGA and one can only use the design board for any type of EHW experiments.

Only the Sblock seems to be a generalised solution but it tends to use a long configuration bit stream (32 bits) for one chromosome, which could be cumbersome for larger designs for EHW.

To avoid these problems a new design for a 'Generic Cellular Structure' for EHW has been developed so that any available FPGA hardware in the market can be used for the development of EHW. This research aims at developing a more generalised virtual EHW

platform that can be used on any past or present FPGA chip of any type. In addition, this design will require fewer configuration bits than the Sblock.

4 Generic Platform

4.1 Introduction

A “Generic Platform” is a generic cellular structure designed and implemented into FPGA hardware for the purpose of intrinsic evolution of an electronic circuit.

The Generic Platform is a general design implemented in Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) or any other High Level Language.

An evolvable hardware system mainly consists of two components:

- *Genetic Algorithm:* An algorithm used to evolve and test a desired circuit
- *Hardware:* This may be a field programmable gate array, which can be used to evaluate the evolved circuit.

The genetic algorithm being a program, the platform required for this program could be any personal computer, a microcontroller, or an FPGA depending on the type of evolution required.

In EHW, FPGA’s may be seen to be the target technology as they provide a commercially available re-configurable platform with fast processing speeds. The main elements of FPGA chips are configurable logic blocks (CLBs) connected together in a grid format and configurable routing resources. In addition, configurable input/output blocks (IOBs) are connected to the grid at the perimeter of the chip, making FPGA’s really suitable as hardware for EHW.[60]

The most vital feature required in the hardware is that it should be evolution friendly, which as stated by Haddow and Tufte [3] has disappeared from the latest FPGA models available.

Therefore, there is a demand for a generic cell structure specifically EHW friendly, which could be easily loaded onto any FPGA hardware available now or in future.

4.2 Experiment- An Introduction

The main requirement of the research was to develop an Evolvable Hardware friendly generic cellular structure that could be created using a VHDL compiler or VHDL design tools. This structure was to be implemented on an FPGA and a microcontroller was used to run the genetic algorithm. Computer Aided Design (CAD) software was to be used to check the actual behaviour of the generic structure in simulation on a PC.

A cell structure was designed in CAD software using VHDL Tools. This cell was the basic element of the Generic Platform.

The cell structure was initially inspired from the design of the Xilinx 6200, where the multiplexers are used for selection of inputs, which was discussed (in section 3.3) as one of the evolvable friendly FPGAs.

The Generic Platform was constructed using an array of two by two cells, giving a minimum usable platform for a basic circuit to be developed by the EHW scheme. The two basic structures evolved using the EHW scheme were the 'AND' and the 'OR' gates.

The Altera® MAX® 7000S (FPGA) was chosen as the hardware for the generic platform. The selected hardware was chosen, as it was readily available, where it was being used for various electronics projects.

The cell and the Generic Platform were developed in VHDL, using the Quartus-II®- Computer Aided Design (CAD) software, provided by Altera®, manufacturer for the FPGA.

Once designed, the generic structures were tested in Quartus-II®, by using the simulation facility provided in the software. The generic platform structure was then loaded into the hardware (FPGA) for the evaluation of the circuits developed by the genetic algorithm.

A PIC® microcontroller (16F877) was used to run the GA program, as it was readily available with university, to complete the intrinsic evolution process. The program was used to generate the chromosome population for a particular cell structure. These chromosomes were then sent to the microcontroller for evaluation. The correct chromosomes were selected by the program and displayed on a personal computer, connected to the microcontroller through a serial port.

All the solutions were then stored into a file for manual verification of the circuits. The circuits formed by the program were manually verified as correct solutions.

Once the working of the platform was verified, it was loaded into another FPGA to test the generalised nature of the design. The GA program was loaded into the ATMEL® ATMega128 microcontroller, the other FPGA used to test was Altera's FLEX®, as it was readily available. The obtained results were verified manually to confirm the correct operation of the GA.

4.3 The Cell Design

The cell structure as described earlier was the building block of the complete generic platform. This structure was to be a simple evolvable friendly structure, which would require the minimum amount of resources on modern FPGAs. In addition, it was required that the cell structure could be scalable and could be accommodated into almost any available FPGA/CPLD in the market.

Hence, the cell design was based on one of the basic elements of digital electronics and the foundation of almost all digital logic circuits, a NAND gate. The design was inspired by the Xilinx® 6200 architecture, but it is much simpler than that architecture.

The main elements of the cell were:

- A NAND gate – It is the core component of the cell.
- Multiplexers – These have been used to select the inputs and the final output of the cell.
- Flip-flops – Flip-flops are used to create a 5 bit serial in parallel out shift register to store the cell configuration (chromosome) produced by the genetic algorithm.

4.3.1 NAND Gate

The NAND gate is treated as the basic gate of many combinational circuits as they can be used to implement any combinational circuit. A few possible combinational circuits using NAND gates are shown below.

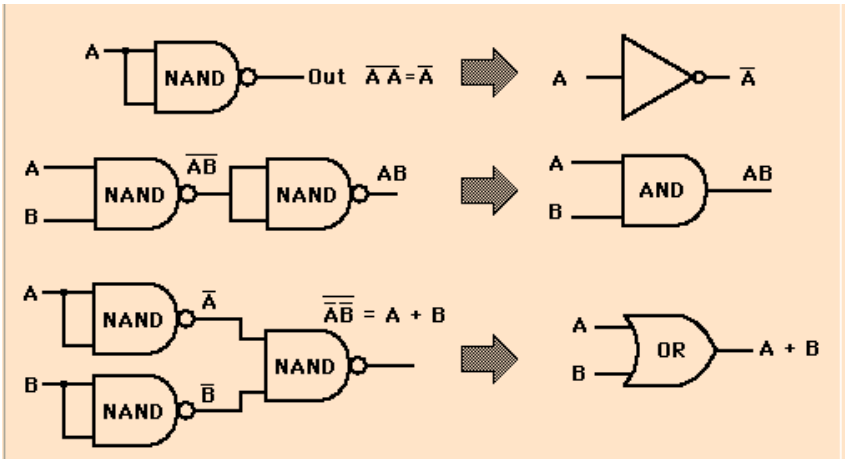
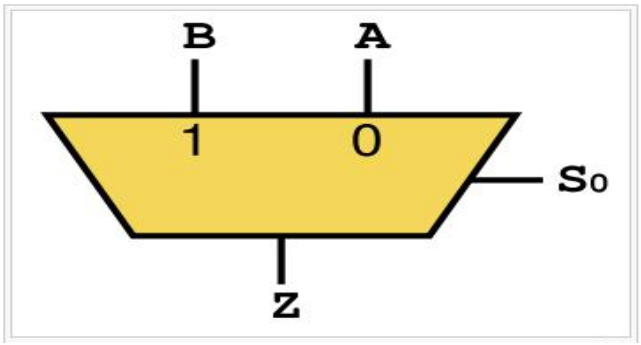


Figure 23 Combinational logic with NAND gate

4.3.2 Multiplexer (MUX)

The most basic 2-to-1 multiplexers have 2 data inputs, 1 select input, and 1 output. The figure below shows, the graphical symbol and truth table of a two input multiplexer. The output equals one of the data inputs, depending on the state of the select input. Larger multiplexers behave the same, having 2^n inputs and n select bits.



S	A	B	Z
0	1	1	1
	1	0	1
	0	1	0
	0	0	0
1	1	1	1
	1	0	0
	0	1	1
	0	0	0

Figure 24 2-to-1 MUX and truth table [61]

Three multiplexers have been used in the construction of the cell. At the input, two 4-to-1 multiplexers have been used to select the desired inputs to the cell and a 2-to-l MUX selects the output of the cell. The select inputs of the multiplexers are driven by the shift register, described in the next section.

4.3.3 Shift Register

The Preset and Clear inputs to the flip-flops have been set high (‘1’), to allow synchronous operation of the shift register, using the clock inputs. The flip-flops have a common clock signal for loading the flip-flops in series. Five clock pulses are required to load the shift register.

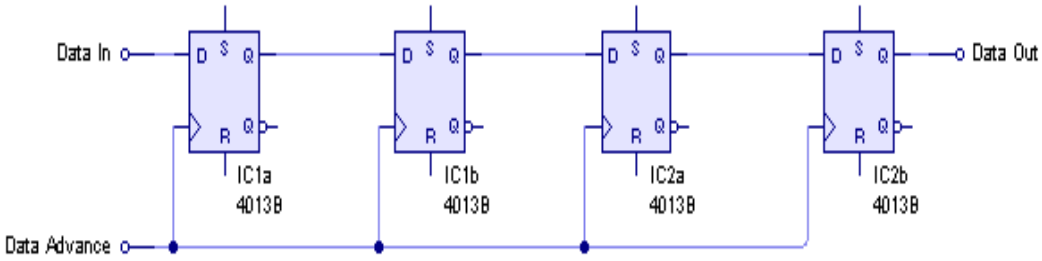


Figure 25 A Shift Register [62]

4.4 The Complete Cell Structure

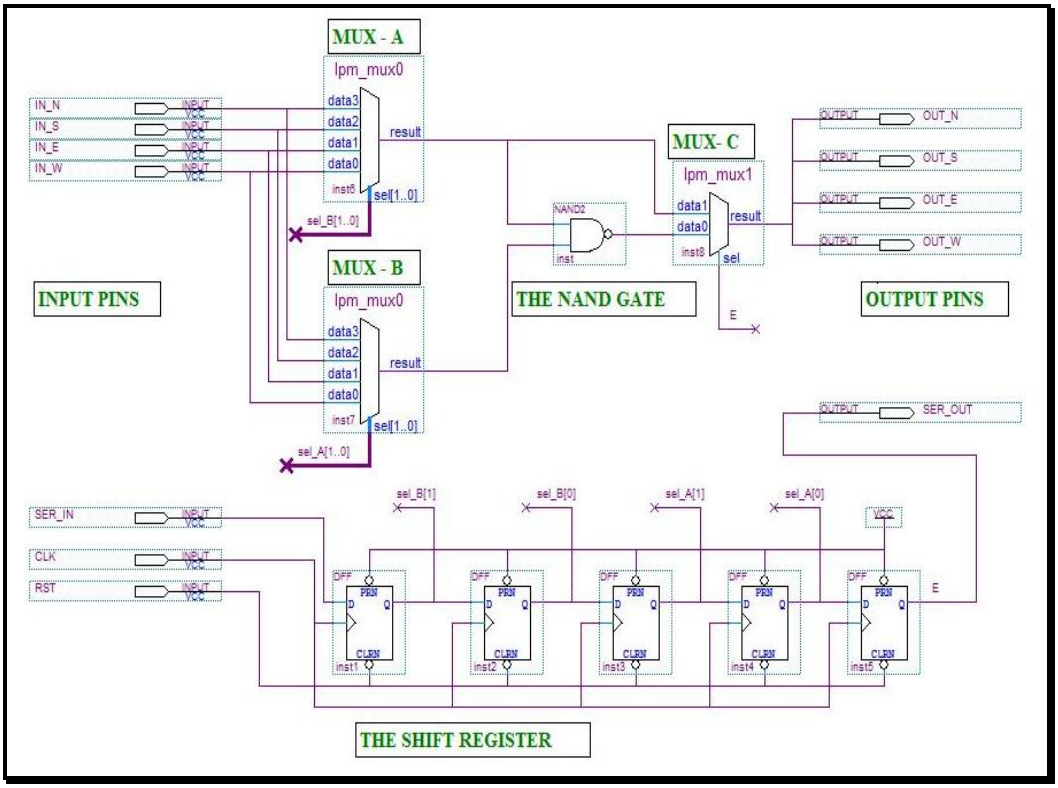


Figure 26 The Cell

As shown above, the 2 input NAND gate forms the heart of the cell structure. Each of the two inputs of the NAND gate is connected to a 4-to-1 multiplexer that selects one of the inputs as

the NAND gate input. The two input multiplexers have four input pins labelled IN_N (north), IN_S (south), IN_W (west) and IN_E (east). Similarly, the output multiplexer has one final output connected to all four output pins OUT_N, OUT_S, OUT_W, and OUT_E.

In addition, the structure has a shift register consisting of five-flops, which are controlled through a clock signal (CLK pin) driven by the genetic algorithm. The SER_IN pin is also controlled by the GA, and is used to send the configuration stream into the shift register. The RST pin is the reset pin used to clear the shift register if required. SER_OUT pin is the output of the last flip-flop of the shift register.

4.5 Cell Operation

The cell structure has been divided into three sections to explain its operation.

4.5.1 The Input Multiplexers

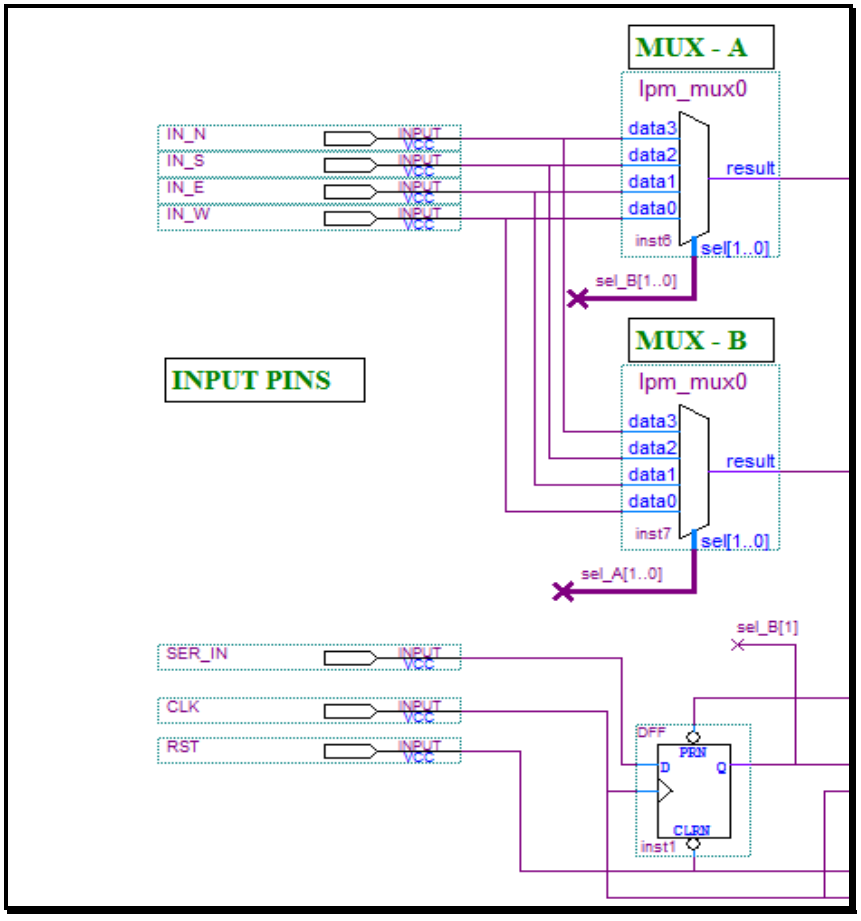


Figure 27 INPUT Section of the 'Cell'

As shown above, the input section of the cell consists of four input pins connected to the two multiplexers (MUX - A and MUX - B), which are used to select the input values for the NAND gate. The input pins are connected to the input pins of the FPGA or they connect to the output pins of other cells as per the configuration loaded (described later).

4.5.2 Shift Register

The other input pins are:

The SER_IN pin is connected to the input of the shift register internally in the cell. Externally it could be connected to the micro controller so that chromosome bits can be loaded serially in to the Generic Platform. Alternatively, it is connected to the serial output of the other cells in the Generic Platform.

Other input pin is the CLK pin that is an external clock. This clock is provided by the microcontroller through the genetic algorithm and is used to control the loading of the cell shift registers.

The RST pin is used to clear the shift-register if required; this pin was initially used to test the cell structure and then was removed from the design as this was later initiated using the GA.

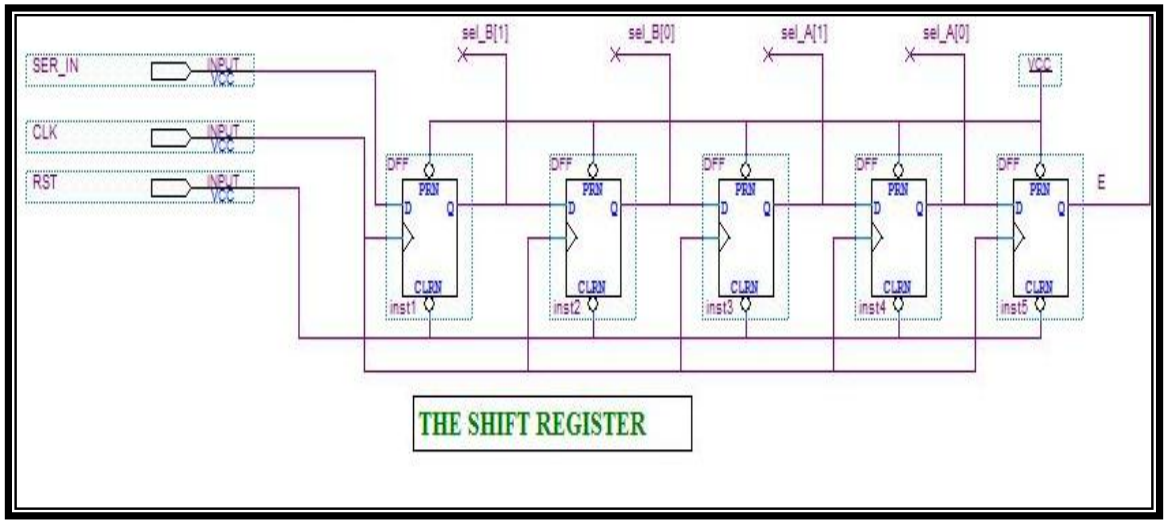


Figure 28 The Functional section

The shift register is used to store the configuration of the chromosome that is further used to set up the whole cell. As described the shift register consists of five flip-flops, out of which the first two flip-flops set the first multiplexer i.e. MUX-A (refer to Table2).

The third and fourth flip-flops control the second multiplexer MUX-B (the table below shows the selection bits for the flip flop) and the last flip-flop controls the last multiplexer which is used to decide whether the cell will act as a gating structure or a routing structure.

Table 2. **TABLE FOR THE SHIFT REGISTER SEQUENCE**

Flip-Flop Number	Bit for Select Bus of Multiplexers	Multiplexer Selected
1	E	MUX-C
2	Sel_A[0]	MUX-B
3	Sel_A[1]	MUX-B
4	Sel_B0]	MUX-A
5	Sel_B[1]	MUX-A

The shift register also controls the functioning of the heart of the cell – the NAND gate. The NAND gate can act as a NAND or a NOT gate depending on the inputs selected by the configuration of first four flip- flops of the shift register. If the same input is selected to the NAND gate by the four flip-flops the NAND gate act as a NOT gate otherwise it acts as a NAND gate with the two inputs selected.

4.5.3 The Output Section

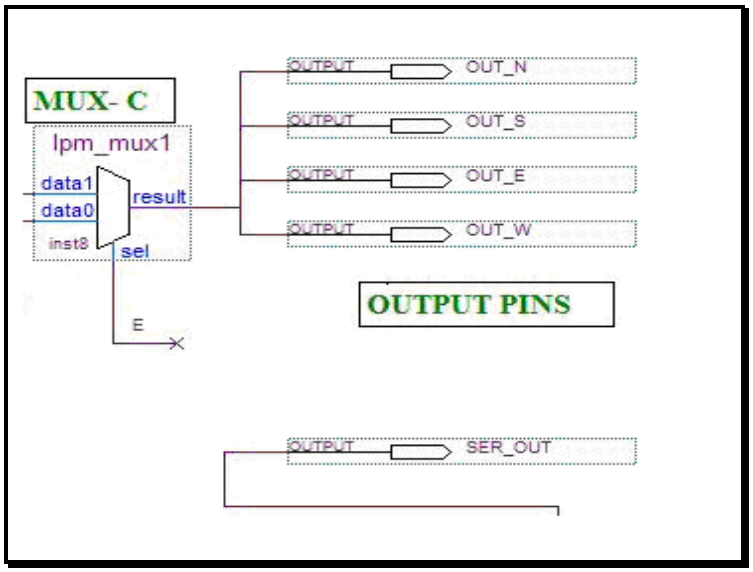


Figure 29 Output Section

The output section consists of a multiplexer MUX- C that has 2 inputs, ‘data 0’ and ‘data 1’. The ‘data 0’ is connected to the NAND gate output and ‘data1’ is connected directly to the MUX-A output. The input from the NAND gate is chosen if the circuit is selected as a gating structure otherwise the MUX-A output is selected as the output of the cell, if it is selected as a routing structure.

The output of MUX-C is directed either as the output to other cells connected or as the final output to an output pin of the FPGA. The output also consists of a final output pin from the shift register, which transfers the configuration bits from one cell to the next.

4.6 The Generic Platform

The Generic Platform (GP) is the basic evolvable friendly structure based on the cell.

GP is an expanded version of the cell, which can be used to evolve electronic circuits with the help of a genetic algorithm. Depending on the need of the design, various numbers of cells can be used and the GP can be designed (by changing number of cells) for evolving a particular function.

To check the integrity of the design, the functionality of the cell was tested alone by itself, and then in a 1 x 2-array .Then the GP structure of 2 x 2 Cells, was used to evolve two basic structures, an AND gate and an OR gate.

A few standards were developed for designing the GP. The main reason for these standards was to maintain uniformity while expanding the GP design to a larger scale. The other important reason was to make sure the inputs and the outputs of the structure were not accidentally connected to each other while the design was generated, as this could be harmful for the hardware. In addition, the GP was meant to be designed in such a manner that it would be very flexible and could be used with any available hardware. Hence, to satisfy all these conditions, the following standards were applied:

- Unused pins were grounded, so that they had a definite known value at all times. This standard applied to cell inputs on the edge of the GP that had no adjacent cell.
- All cell clock signals were connected together and driven from the same the clock source.
- For a particular application, the number of cells in the GP was fixed and could not be increased dynamically while the GA was running.
- The cell shift registers were connected in a sequence across each row of the array of cells with the last cell of the row being connected to the first cell of next row as shown in Figure 30.

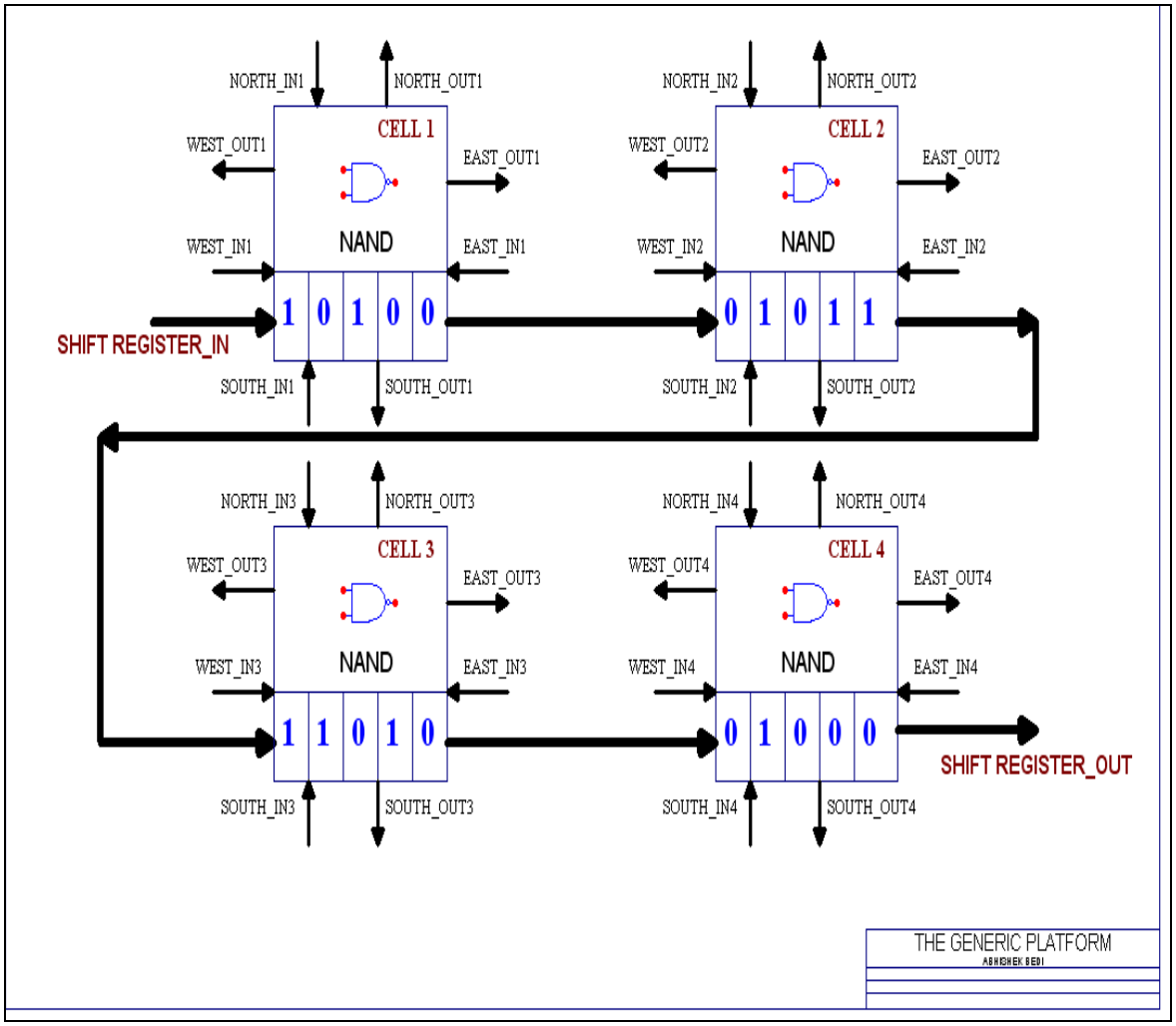


Figure 30 Shift register connection between cells

4.6.1 1x2 Cell Generic Platform for Test

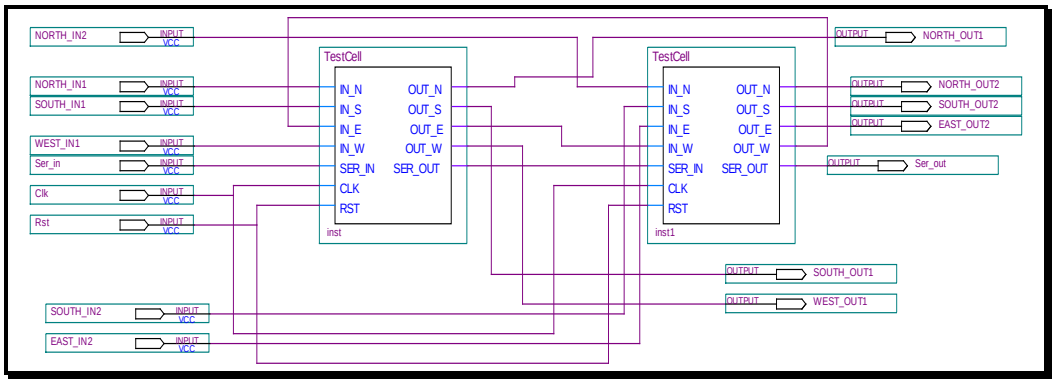
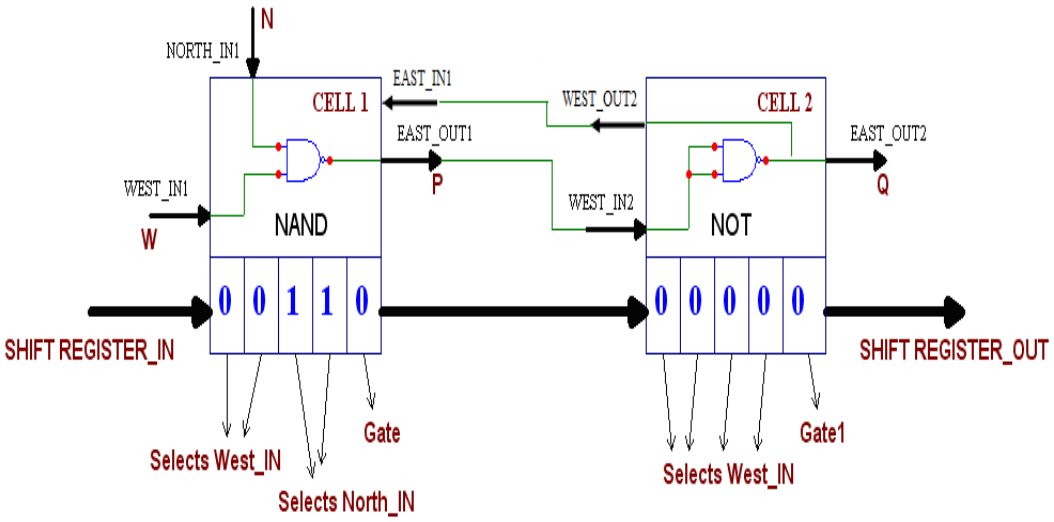


Figure 31 The Generic Platform for 1x2 cells

As shown above, this GP has been designed using an array of 1x2 cells. This structure was only used for testing the functionality of the GP. It was designed to generate an AND gate, using a NAND in the first cell and an Inverter in the second (shown below). The external inputs to the GP are the N (north) and W (west) inputs to the cell1. The output of the GP is Q (east) from the cell2.



All the Unused pins have been Grounded and hence have not been shown .

Figure 32 NAND + NOT gate diagram

Here the two cells are joined together sideways, with the first cell acting as a NAND gate joined to the second cell through its East output and East input. The first cell has two inputs from the micro connected to its North and West inputs. The second cell, acting as an inverter, has its West input connected to the East output of the first cell.

As described in the standards, other inputs are not used for the design and all the unused pins are grounded. Hence, in the first cell the South input pin is grounded, whereas, in the second cell the North, South and East input pins are grounded. The truth table based on figure 32 describing the working of 1x2 cells is shown below:

Table 3. **TRUTH TABLE FOR NAND + NOT = AND GATE**

N	W	P	Q
0	0	1	0
0	1	1	0
1	0	1	0
1	1	0	1

4.6.2 Generic Platform for 2x2 Cell array

After the successful development of the 1x2 array cells, the generic platform was developed into the final desired structure of a 2x2 array. Therefore, there are four cells instead of two in the Generic Platform. To explain the structure of the 2x2 array, the GP is divided into two sections; the *Upper Section* and the *Lower Section*. Both sections have two cells joined together sideways, similar to the 1x2-array structure. Vertically adjacent cells in these two segments are also interconnected through input and output pins.

As shown below, the upper section has both the cells 1and 2 connected to the lower section cells 3 and 4 through the South input and output pins. Similarly, the lower segment is connected to the upper segment with the North input and output pins. In addition, the shift registers in the cells are linked in series following a path from Cell 1to Cell 4, with the connection between Cell 2 and Cell 3 acting as the link between the two layers, shown in figure 33.

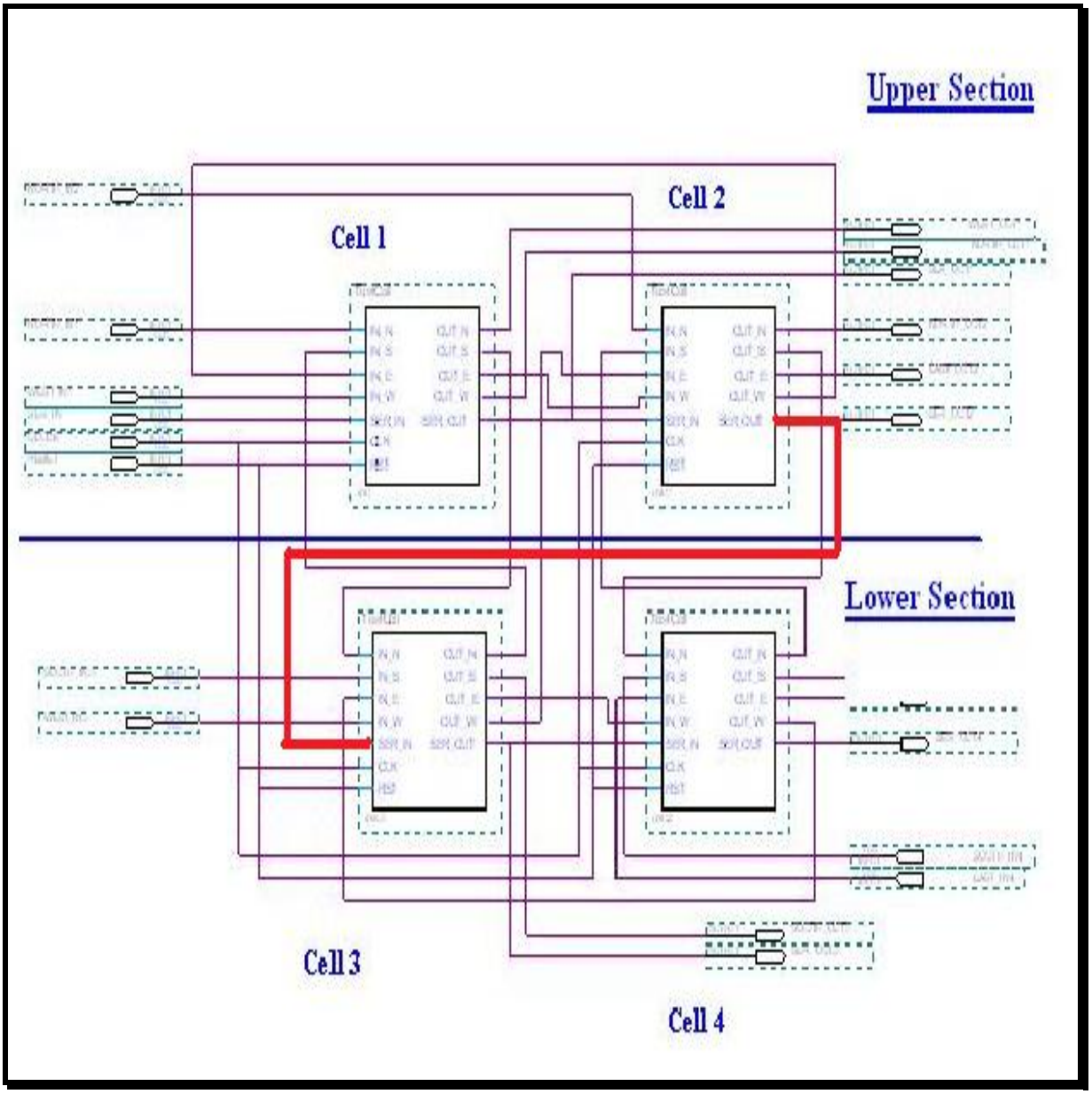


Figure 33 GP 2x2 structure; showing the two sections

4.7 Development in CAD Software

Once the generic platform was designed, it was developed in a VHDL Tool. Quartus-II® was used as the VHDL tool for the generic platform, because it was freely available CAD software consisting of VHDL tools for the Altera CPLD MAX 7000S used in this research.

As the design consists of basic elements of digital electronics like flip-flops, multiplexer and the logic gates, the intellectual property (IP) designs/structures from Altera® were used to design the Generic Platform.

All the desired elements were assembled together to design a simple cell structure schematic in the Quartus software. The entity cell was compiled and the symbol generated for the cell was further used to develop the top-level entity, the generic platform, in a schematic form. To implement the logic circuit of the Generic Platform onto an FPGA, Quartus was used as the Computer Aided Design software. CAD software makes it easy to implement a desired logic circuit using a programmable logic device, such as a field-programmable gate array (FPGA) chip. A typical FPGA CAD flow is illustrated below [63]:

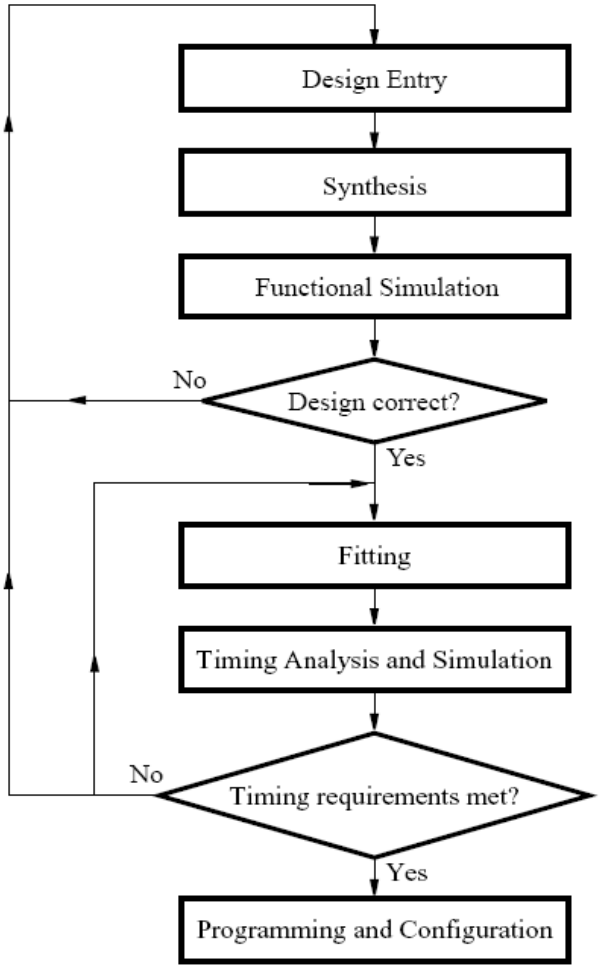


Figure 34 Flow chart for CAD software [63]

Following the design flow, the design entities of the cell and the GP were first analysed and synthesised in Quartus. The designs were then tested in simulation, as described next.

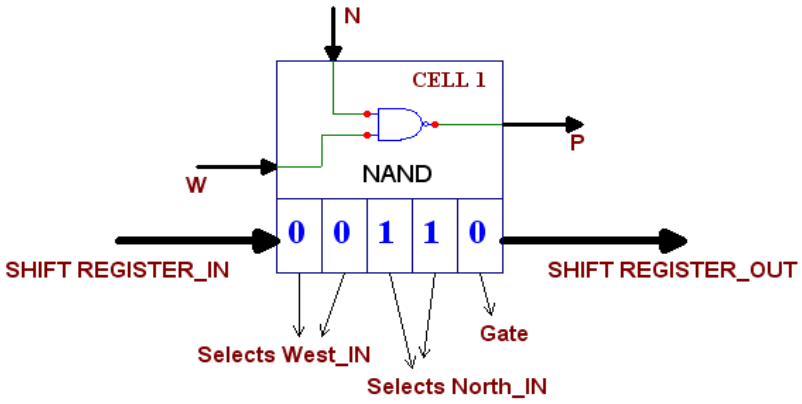
4.7.1 Section D: Testing in Quartus-II®

The cell being the building block of the generic platform, the functional testing of the cell was done first. After successful testing of the cell, the GP was tested for its desired functioning.

Depending on the configuration bits, the cell could behave as a NAND gate, NOT gate or a ROUTER. The cell was tested for all of these three functioning in the Quartus simulation. Then the timing simulation was performed for the same logical circuits, using output pins as various test outputs. The timing simulation allowed observation of the expected behaviour of the structures, when loaded onto the FPGA. Once the cell was tested, the generic platform was tested in the forms of one by two array and two by two arrays.

4.7.1.1 The testing of Cell as a NAND gate

To test the functioning of NAND structure, two the cell inputs (North and West) were selected to be the inputs to the NAND gate. Other two cell inputs (South and East) were grounded to have a definite state during testing. To configure cell for these inputs, the shift register was loaded with a binary value of "00110" or decimal value of "06" (in figure 35).



All the Unused pins have been Grounded and hence have not been shown .

Figure 35 Behaviour of cell as a NAND gate and the shift register values explained

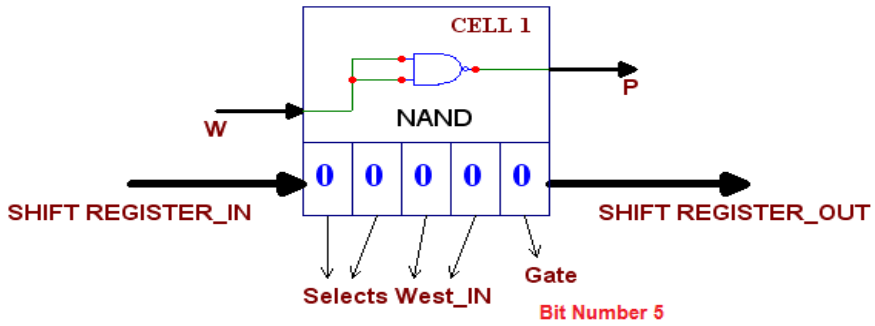
The state of the input pins (north and west) was represented by waveforms in Quartus. The results produced by the circuit on the output (EAST_OUT) waveform, were crosschecked with the NAND gate truth table. Other outputs (NORTH_OUT, WEST_OUT and SOUTH_OUT) got the same result, as the output of the cell is same for each direction. The simulation of the NAND gate is given in appendices. Table 4, below shows the how the inputs are selected by the appropriate bits in the shift register.

Table 4. TABLE FOR THE INPUT SELECT BITS IN MUX-A AND MUX-B

Bits in the '**Select Bus**' of Multiplexers	Input Selected
00	West
01	East
10	South
11	North

4.7.1.2 The Testing of Cell as a NOT gate:

The NOT gate was tested in a similar fashion, except in this case the North input to the Cell was also grounded. To test the structure as a NOT gate the West input was selected by both the multiplexers and the cell was selected as a gating structure by bit number 5. Therefore, the shift register was loaded with a Binary value of "00000" or a Decimal value of "0" (depicted below).

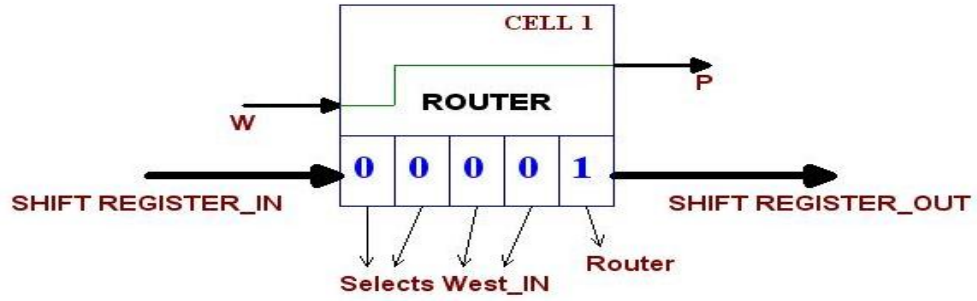


All the Unused pins have been Grounded and hence have not been shown .

Figure 36 The circuit diagram with other pins grounded, functional simulation shown in appendix

4.7.1.3 The Testing of Cell as a ROUTER:

The cell structure this time was tested for routing functionality, by selecting the West pin as the input to be passed on to the output. Also the output multiplexer,(MUX-C) was selected to '1', to cause the cell structure to function as a router. The shift register was loaded with a Binary value of "00001" or a Decimal value of "01". The functional testing of the router is shown below:



All the Unused pins have been Grounded and hence have not been shown .

Figure 37 Cell as a router waveform

After the successful functional testing of the cell, the Generic Platform was tested for functioning as a 1x2-array structure and a 2x2-array structure. As explained earlier, the 1x2 GP was tested for an AND gate and the 2x2 GP was tested for both 'AND' and 'OR' gate.

4.7.1.4 The Testing of Generic Platform in a 1x2-array:

The top-level design Generic Platform consisting of two cells in a 1x2 form was tested for functioning as an AND gate. The design generated had the unused pins grounded as described by the standard in section 4.6. Here again, in the 1x2 cell structure the north and west input pins to the cell 1 were selected as input pins and EAST_OUT2 from cell 2 was selected as an AND output. A point to note is that the shift register in this design was turned into a shift register chain i.e. two 5-bit shift registers were joined together. The testing of shift register was done by using a test point in-between the two cells 1 and 2 (TP1, diagram in Appendix II). The configuration loaded in the register was "0011000000" and was decided in accordance with Table 4 on a per cell basis.

The explanation of the Configuration Bit loaded in the 1x2 Array is as follows:

0 0	**1 1**	**0**	**0 0**	**0 0**	**0**
West	**North**	**Gate**	**West**	**West**	**Gate**
CELL 1			**CELL 2**		

Figure 38 Bit configuration explanation for 1x1 array

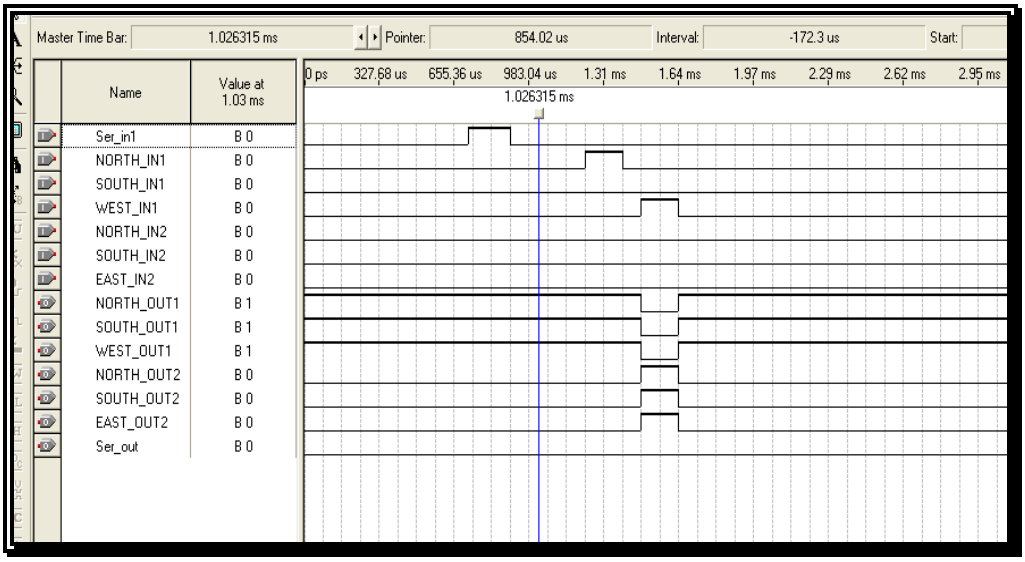


Figure 39 Functional simulation for 1x2 array

4.7.1.5 Testing of GP for 2x2-array:

The Generic Platform for the 2x2-array was tested for functioning as an OR gate. Here the two inputs for the OR gate were chosen from two different cells; cell 1 and cell 3, but the output was still fixed on to the cell 2 output pin EAST_OUT2. In addition, the four cells had test points between four shift registers, to check the correct functioning of the shift registers. All the unused pins were again grounded. For this GP the size of the shift register was 20 bits. Hence, the following configuration was loaded into the shift register chain, binary "11110001000000000111". The configuration bit loaded has been explained in the figure 40:

11 11 0	**00 10 0**	**00 00 0**	**00 11 1**
Noth North Gate	**West South Gate**	**West West Gate**	**West North Router**
CELL 1	**CELL** 2	**CELL** 3	**CELL** 4

Figure 40 Bit configuration explanation for 2x2-array Functional simulation in appendix

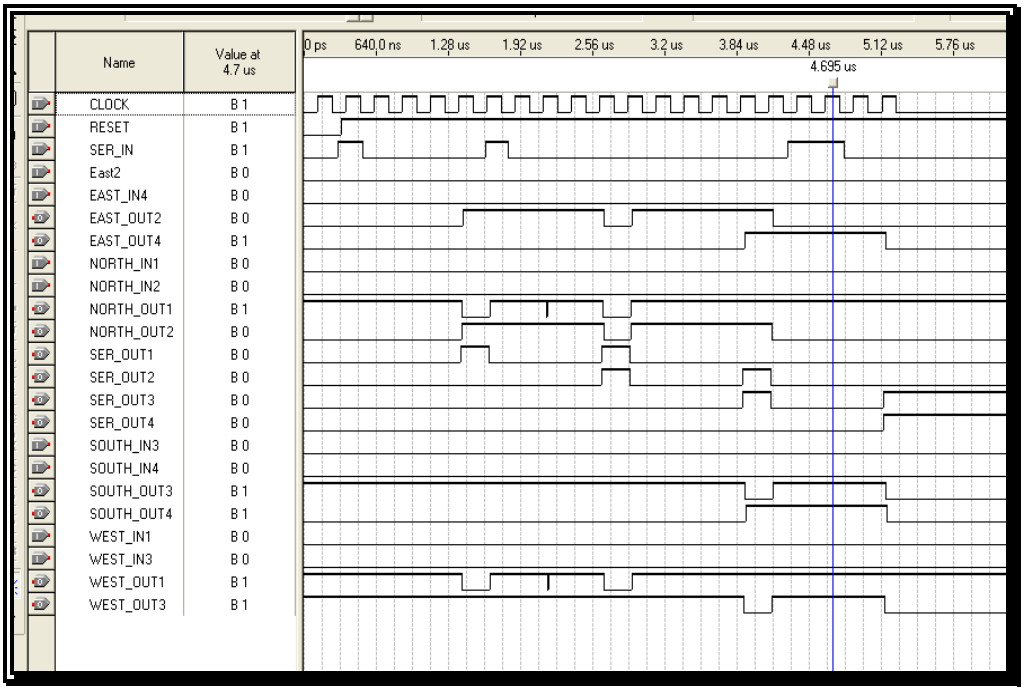


Figure 41 Functional simulation for 2x2 array

After the functional simulation of the design as shown above, the design was passed through the CAD Fitter tool, which determined the placement of the Logical Elements (LEs) in an actual FPGA chip. Fitter also chose routing wires in the chip to make the required connections between specific LEs. After this, the Timing Analysis was done, where propagation delays along the various paths in the fitted circuit were analysed to provide an indication of the expected performance of the circuit.

Once the timing analysis was finished, the Timing Simulation came into effect. Here the fitted circuit was tested to verify both its functional correctness and timing. Once the circuit behaved as required in the timing simulation, it was ready to be programmed into the FPGA.

Finally, the programming and configuration of the designed circuit was implemented in a physical FPGA chip by programming it through the CAD software (the complete process of designing a circuit and programming onto an FPGA is provided in Appendix I).

5 The Genetic Algorithm Code

5.1 Introduction

This chapter will discuss the genetic algorithm code that was written in the high-level programming language C and was loaded into the microcontroller as described earlier. In addition, here the expected results will be discussed.

The GA program was distributed in four main functions, named as follows:

- Fitness
- Selection,
- Crossover and
- Mutation

The diagram in figure 42 shows the structure of the GA program.

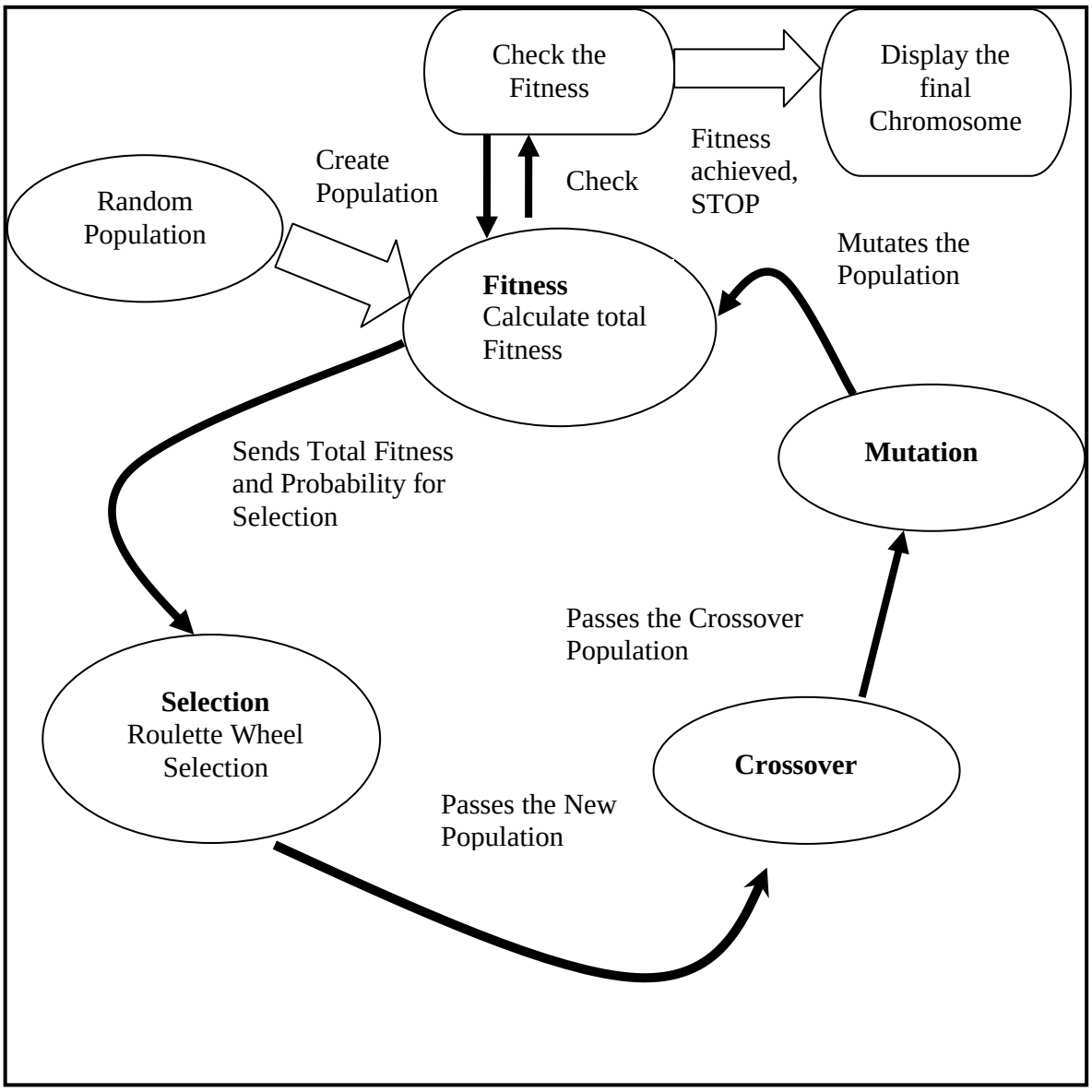


Figure 42 Diagram showing the various steps of Genetic Algorithm

At the beginning of the program, various parameters required for the execution of the GA are defined, based on the design of the Generic Platform being tested. These parameters formed the base of the GA run.

5.2 Parameters of GA

The parameters decided by the user, before the beginning of the algorithm were:

Population Size – Could be anything between 5 and 10 chromosomes to start as parent population. It has been represented by 'p' in the program.

Length of Chromosome – Depends on the design of the GP. The chromosome requires 5 bits for each cell in the GP. The variable used to represent length of chromosome is 'n'.

Crossover Rate – This parameter is the rate of chromosomes to be selected for crossover. The rate was generated randomly, up to a maximum of 50 percent of the total population.

Mutation Rate – The mutation rate was generated randomly, to mutate upto half the chromosomes in the new population generated after the crossover. For each of the chromosomes, one randomly selected bit was mutated.

Maximum Generation Number – It defines the maximum number of times the GA performs the optimization of the parent population.

5.2.1 Fitness Function

This function determines the fitness of each individual (chromosome) in the population. The function reads each individual chromosome in the population and passes it onto the FPGA to configure the GP for evaluation. It then calls a function Test to evaluate the fitness of the chromosome for the desired circuit behaviour e.g. an 'AND' gate. After the test is performed, the fitness value for each chromosome is stored in an array named *fu[i]*.

```
//====================== (6) FITNESS============================

void fitness(char p)
  {   char j;
      for (j=0;j<p;j++)            //fills the array from 0-9
      {
         config_data=c[j];         // Start the fitness function
         load_register();          // Sends the configuration to the FPGA
         test();                   // Tests the configuration
         if (flag2==0)             //
         checkpattern();           // Only check for answer if passes the test.
         fu[j]=score;              // Reads the Fitness
      }
 }
```

Figure 43 Code for fitness

5.2.2 Test Function:

The test function checks the behaviour of the Generic Platform for a particular chromosome.

```
void test(void)
{       score = 1;              // test 1
        RD2=0;                  // SET INPUT 1
        RD3=0;                  // SET INPUT 2
        if(RD4==1)              //Read Output
        {       score = 0;      // Discard the Chromosome
                return;         // Return out of test function
        }
        else if (RD4==0)        // Read Output
        score=score+1;          // Inc. the score from FPGA
                // Fitness is 0 to 4
}
```

Figure 44 Code for Test function showing only the first test

As shown, in this function, the inputs to the GP (RD2 and RD3) are set and then the output (RD4) is read from the output pin of the generic platform on the FPGA. This function calculates the fitness value for each individual chromosome and passes it back to the fitness function. The variable score is incremented every time an expected result is generated on the output to a maximum limit of four for an AND/OR gate, as they can only have four possible conditions as per their truth table. Hence, for every chromosome configuration, the test function is run and then the function calculates how many outputs match the outputs of the required circuit function.

Condition: After every run of the fitness function, the condition for desired score is checked. If the desired score is achieved, the GA stops and displays the results, or else it continues to further tasks for optimisation of the parent population.

5.2.3 Selection Function

This function selects chromosomes that become the parents of the next generation using different selection criteria. Due to memory constrains of the microcontroller, instead of using any particular method for selection, the GA, sorts out the fitness array fu[j] in descending order. The elite chromosomes (chromosomes that are closest to the expected result) are placed on top of the newly generated population, as they have the highest fitness. It then stores the sorted population as a selected population.

```
void selection ( char p)                    // Selection function
{

        char i,j,temp1;
unsigned long tem;
        for (i=0;i<p;i++)
        {
                        for (j=i+1;j<p;j++)
                {
                        if ( fu[i]<fu[j])    // Checks for the Fitness of the population
                                {                       // arranges array according to fitness
                        tem=c[i];           // SORTING IN DESCENDING ORDER !!!
                        temp1=fu[i];
                                c[i]=c[j];              // Highest Fitness
                        fu[i]=fu[j];
                        c[j]=tem;           // Lowest Fitness
                        fu[j]=temp1;
                        }
                }

        }

}
```

Figure 45 Code for Selection of population using sorting

After selection, this newly generated population is treated as an initial population by the next function, the crossover function.

5.3 Crossover Function:

This function creates new individuals (chromosomes) from the mating population produced by the selection function. Pairs of chromosomes are selected at random from the mating population and single point crossover is used to create new chromosomes. In single point crossover, a crossover point on the parent chromosome is selected randomly. All genes beyond that point in the chromosome are swapped between the two parent chromosomes. The resulting chromosomes are the children chromosomes. If the parent chromosomes are same then there is no change in the offspring.

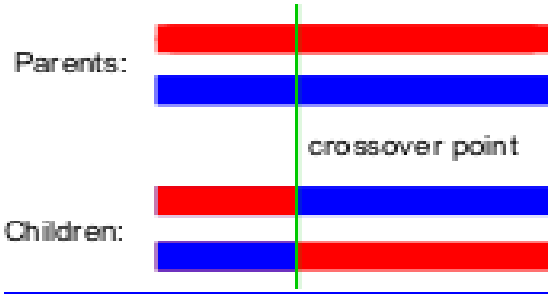


Figure 46 Single Point Crossover [64]

The following code implements single point crossover:

```
                                    // Find the Point
                        // for Single Point Crossover
 j=random(n+1);                                  //Random point for Crossover
p1=0b11111111111111111111>>(20-j); //convert the no. to binary equivalent
a=(v1&(~p1))|(v2&p1);              // Single point crossover
b=(v1&p1)|(v2&(~p1));
v1=a;                              //Read variable1
v2=b;                              //Read variable2
c[k]=v1;                                    //Store variable1
c[k+1]=v2;                         //Store variable2
```

Figure 47 Lines of code showing crossover

A random variable is generated to decide the point of crossover, which is denoted by 'j' (n is the number of the bits in the chromosome). This is converted into a binary mask 'p1', which has 1's for bits that will be crossed over. This number 'p1'is used to perform crossover, by taking the complement of 'p1'. The complement of p1 is ANDed with one of the chromosomes (v1) and 'p1' is 'ANDed' with other chromosome (v2) as shown above.

Once the crossover is complete, these newly generated chromosomes 'a' and 'b' replace the parent chromosomes in the population.

5.4 Mutation Function

Here the population generated by the selection and crossover function is mutated, depending on the mutation probability 'Pm' defined by the user. A chromosome is randomly chosen and then a bit in the chromosome is randomly chosen and its value changed. As shown below, the mutation is done using the boolean operator 'XOR'. This process is repeated for a number of times, depending on by the mutation probability 'Pm'.

```
Variable = c[k];              //select the chromosome from the population array
r = random (n);               //Find the Bit to be Mutated
I = 1L<<r;              //Create a 20 bit long variable
Var  = variable ^ I;    //XOR the 20 bits
C[k]  = var;            //Save the mutated variable
                        //Chromosome to be mutated entered in original array
```

Figure 48 Code for mutation

Mutation stops the GA (best fitness) from being stuck at a local maximum that is not an ideal solution.

Replacing and Testing:

After the mutation, the fitness of the new optimised population is again checked for the desired value. If the total fitness desired is achieved, the GA stops or else it again continues with these tasks (as shown in state diagram).

5.5 Output of the GA

For the second and any other consecutive runs, the randomly generated parent population is replaced by the new optimised chromosomes. This modified population now acts as the parent population for further optimisation.

If an ideal fitness is achieved the program returns the number of iterations and the ideal chromosome with its fitness. This is displayed on the PC screen. In addition, it displays the final population of chromosomes.

6 Hardware Testing

6.1 Introduction

The hardware involved in the process was a microcontroller and an FPGA. The genetic algorithm was loaded onto the microcontroller and the Generic Platform was loaded into the FPGA. The hardware was tested in the same way as using the simulator. First the GP consisting of a 1x2 array of cells was used to evolve an 'AND' gate. Then a 2x2 array of cells was evolved for an 'OR' and an 'AND' gate.

The functionality of the evolvable hardware was tested by using two different hardware systems:

- The MAX CPLD with a PIC 16F877 microcontroller, and
- The FLEX FPGA with an AT Mega 128 microcontroller

In this section, the intrinsic evolution will be discussed further in details. In addition, the GA code for the hardware system can be found in appendices.

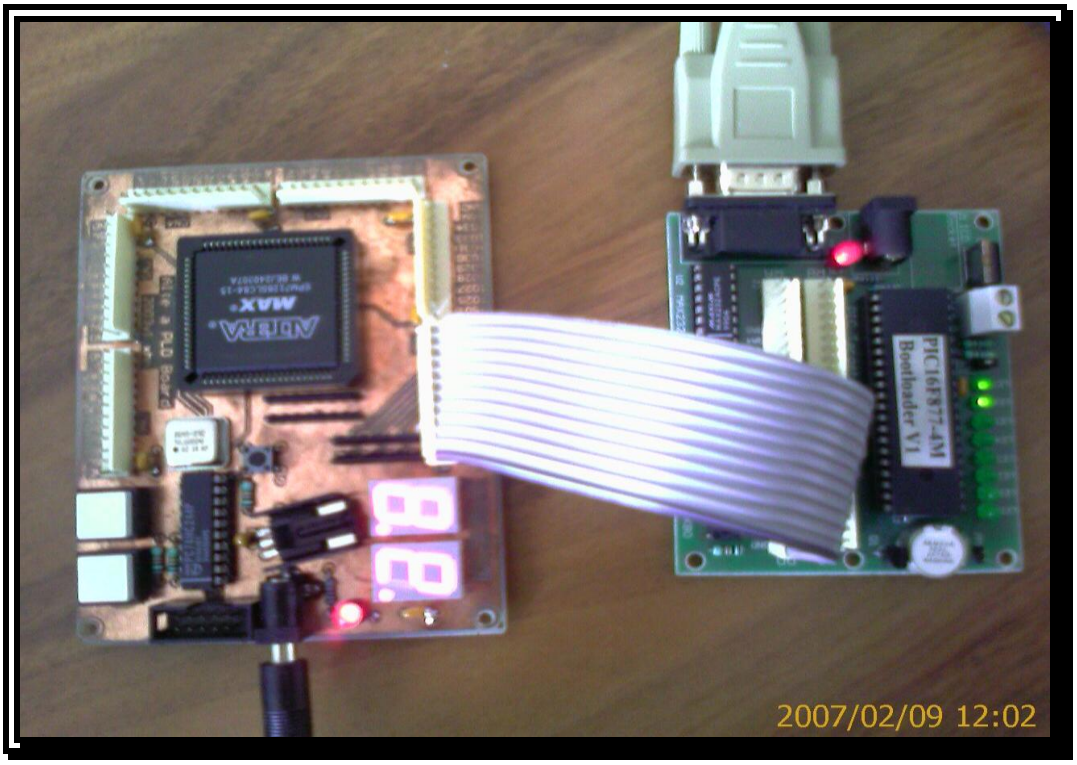


Figure 49 Jpeg photo of the EHW system showing ports and development boards

6.2 Hardware Tests

6.2.1 1x1 Array Test

Before conducting the intrinsic evolution on the hardware, a simple testing of the cell structure was performed on the hardware. In the experiment, a single cell structure was loaded into the FPGA and a test program was loaded into the microcontroller to confirm the functioning of the cell in hardware. The test program loaded all the possible chromosomes into the cell. As the length of chromosome for the cell is five bits, the program generated configuration bit streams from "00000" binary (0 decimal) to "11111" binary (31 decimal). This generated bit stream was passed onto the FPGA for testing if the design behaved as a simple NAND gate, using the North and West as the two inputs. The cell behaved as a NAND gate with configuration bit values decimal 6 and decimal 24.

Interpretation of the result:

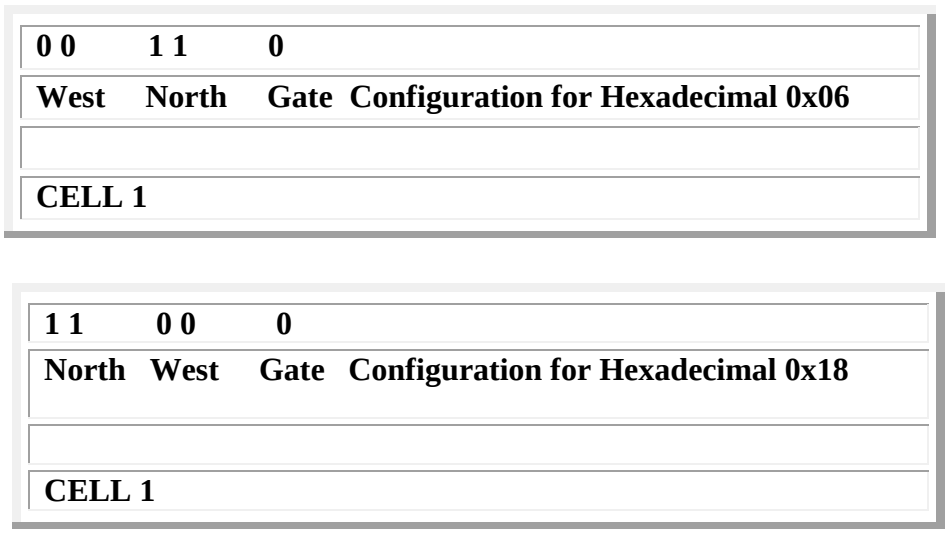


Figure 50 Interpretation of results achieved for NAND gate

6.2.2 1x2 array Test

The 1x2 array was used to generate an 'AND' gate, a 'NOT' gate, and a router. The GP, being the combination of two cells, had 10-bit long chromosomes. The first cell had the two inputs going to it and the second cell had the final output coming out of it.

The microcontroller and the FPGA were connected through their respective development boards, using an 8-bit serial port. The control pins on the micro and the FPGA were different for both the hardware setup as two different development boards were used. It has been shown in the Appendix II.

First, the testing was done by running the program in a controlled way by using few known 10-bit chromosomes for ‘AND’ gate, as given below.

The known Hexadecimal value of 0xC0 was fed into the Structure.

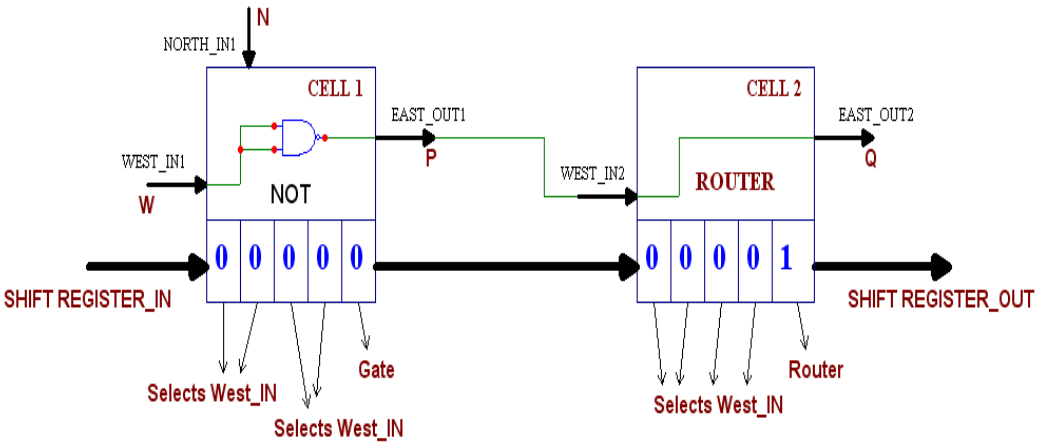
0 0	**1 1**	**0**	**0 0**	**0 0**	**0**
West	**North**	**Gate**	**West**	**West**	**Gate**
CELL 1			**CELL 2**		

Figure 51 Known 10-Bit configuration for AND gate, using 10-bits.

Similarly, the behaviour of a ‘ROUTER’ and a ‘NOT’ gate were checked using a 10-bit known configuration. The configuration bits used for the ‘NOT’ gate were as follows:

0 0	**0 0**	**0**	**0 0**	**0 0**	**1**
West	**West**	**Gate**	**West**	**West**	**Router**
CELL 1			**CELL 2**		

Figure 52 NOT gate 10- bit known configuration



All the Unused pins have been Grounded and are not shown .

Figure 53 Behaviour of 1x2 Structure as a NOT gate with 10- bit known configuration

Bits used for the behaviour of GP as a router are:

0 0	**1 1**	**0**	**0 0**	**0 0**	**1**
West	**North**	**Gate**	**West**	**West**	**Router**
CELL 1			**CELL 2**		

***West is routed as Output*.**

Figure 54 Router 10- bit known configuration

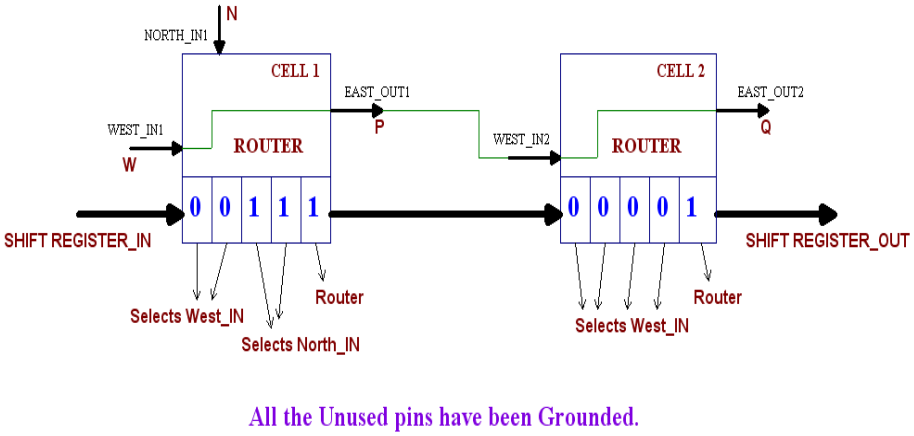


Figure 55 Router gate 10- bit known configuration

6.2.3 2x2 Array Test

The GP was also tested as a 2x2 array. In this instance, the GP, contained four cells and had a 20-bit long chromosome. In this test, the first cell and the third cell each had one input and the second cell produced the output. The selection of inputs for the 'OR' gate was different from the 'AND' gate design as it had Cell -1 and Cell -3 with inputs and Cell -2 had the output connected to it.

The known 20-bit chromosome configurations that were used to test the behaviour of 'AND' and 'OR' gates, are given below.

Table 5. KNOWN 20-BIT CONFIGURATION FOR 'AND' USING 20-BITS

1 1	0 0	0	0 0	0 0	0	1 0	0 0	1	0 0	1 1	1
Noth	West	Gate	West	West	Gate	South	West	Router	West	North	Router
CELL 1			CELL 2			CELL 3			CELL 4		

Table 6. KNOWN 20-BIT CONFIGURATION FOR 'OR' USING 20-BITSITS

1 1	1 1	0	0 0	1 0	0	0 0	0 0	0	0 0	1 1	1
Noth	North	Gate	West	South	Gate	West	West	Gate	West	North	Router
CELL 1			CELL2			CELL3			CELL 4		

7 Genetic Algorithm Results

7.1 Initial Results of Evolution

All results in this chapter were obtained using the 2x2 array on the Generic Platform. Only few evolved results are discussed here and other evolved results are outlined in the Appendix III.

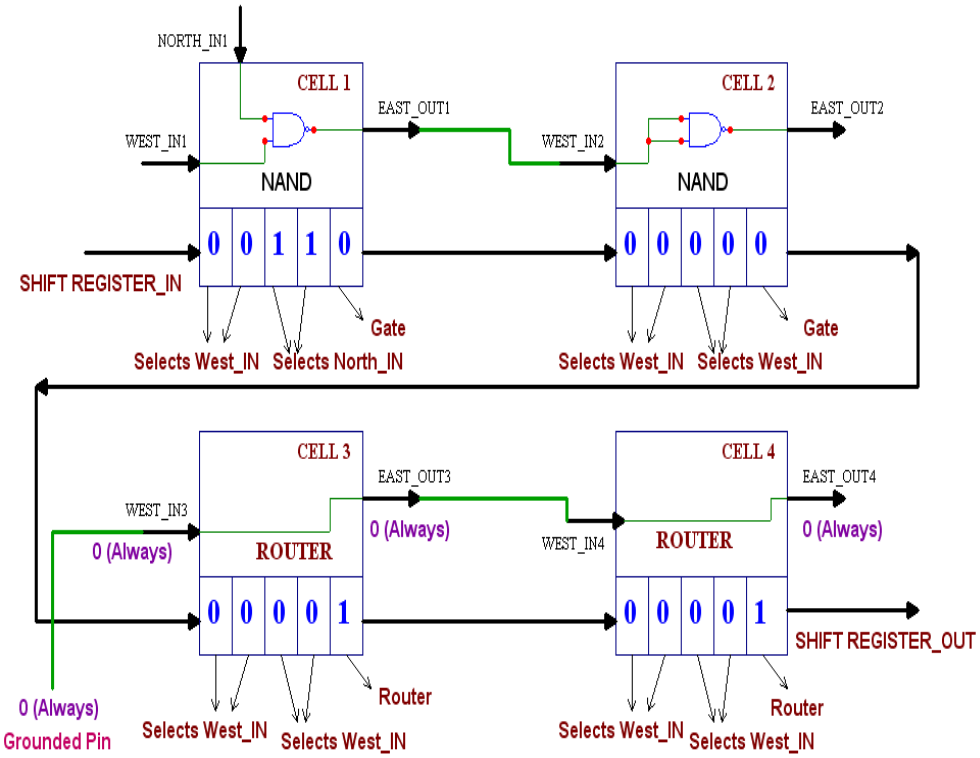
The first run of the experiment was a complete evolvable hardware run using 20-bit configuration streams as the chromosomes. The EHW system was run to evolve an 'AND' gate and an 'OR' gate. The GA was run until a correct solution was evolved or for a maximum of five thousand generations. The final evolved configuration stream was displayed in hexadecimal form by the GA. The solutions evolved by the GA, that it found to behave as an 'AND' gate and an 'OR' gate, are as follows:

- **0x30021 ('AND' gate)**
- **0x35339 ('AND' gate)**
- **0xE10C7 ('OR' gate)**
- **0xF1005 ('OR' gate)**

Figure 56 Results evolved after the First Run

The GA found 4940 correct solutions for 'AND' gate in 5000 iterations, this meant that the GA would find a correct solution for 98.8 percent of the generated population. This result became a point of concern, as expected correct solutions for an 'AND' gate should have been less than 98.8 percent of the population generated. Hence, there was the need for manual analysis of the derived solutions. Most of the "correct solutions" as found by the GA were manually analysed. The manual analysis of the 'AND' gate configurations is given here. The 'OR' gate analysis are given in the appendix. In addition, behaviour of these configurations in simulation is depicted after the manual analysis.

I. Configuration value, 0x30021:



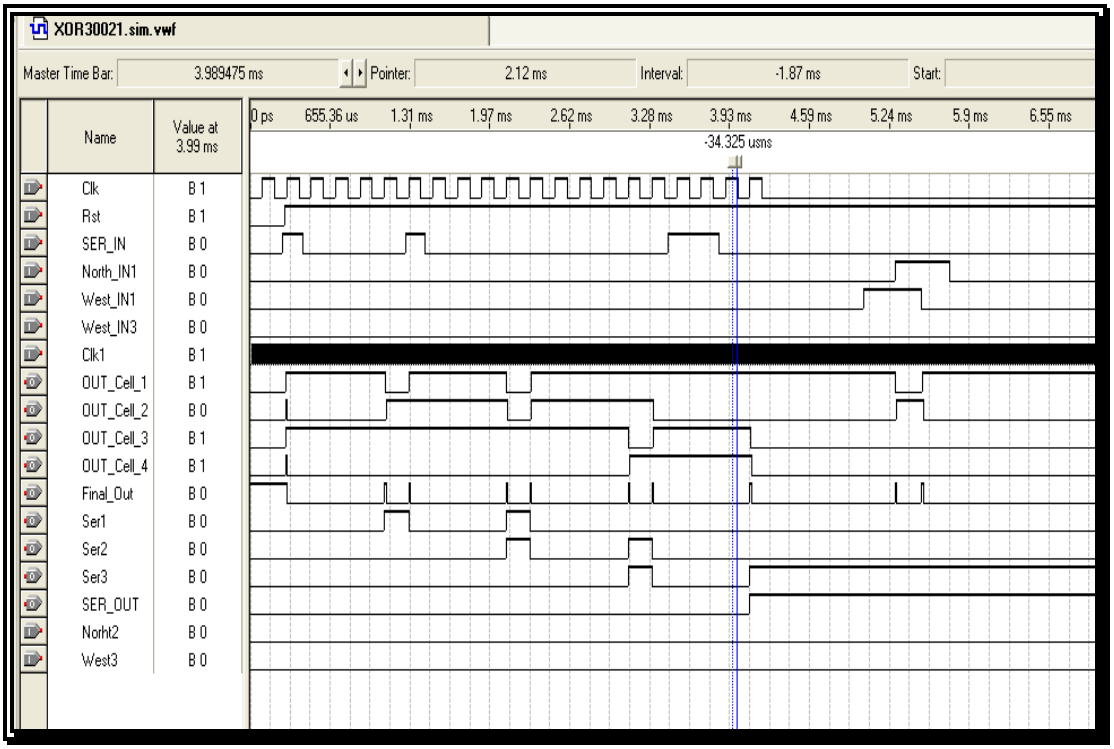


Figure 57 Manual analysis with expected working and simulation result Quartus II

The results obtained after manual analysis of the chromosome configuration are as follows:

- **Cell 1 behaves as NAND gate**
- **Cell 2 behaves as NOT gate**
- **Cell 3 and cell 4 will always have ‘0’ as output as the selected pins by configuration are grounded**

This is a correct solution for ‘AND’ gate.

II. Configuration value, 0x35339:

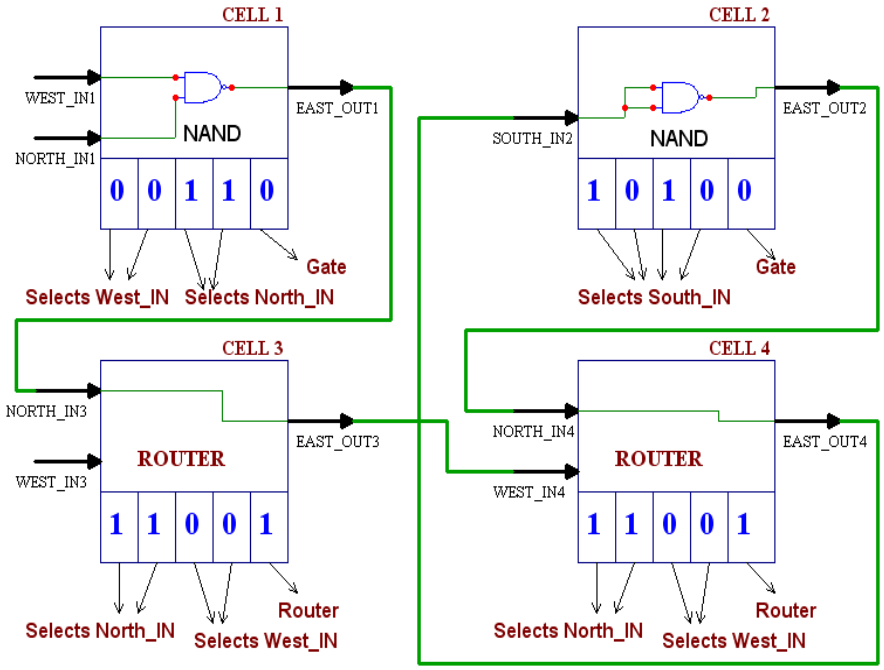


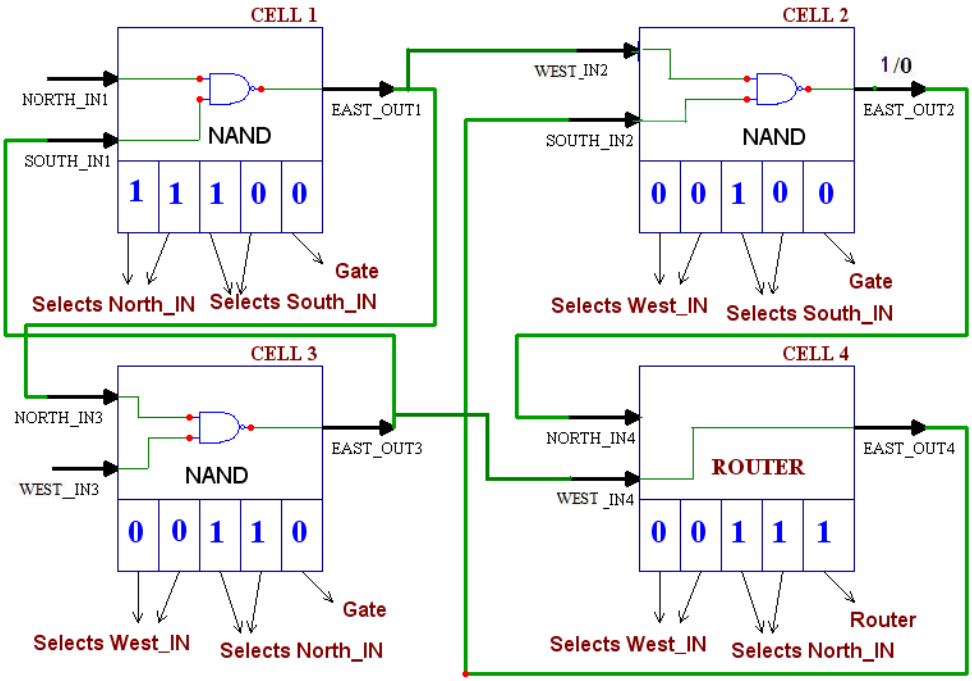
Figure 58 Manual analysis with expected working and simulation result for 0x35339

The results obtained after manual analysis of the chromosome configuration are as follows:

- **Cell 1 is a NAND gate**
- **Cell 3 is a Router**
- **Cell 2 and Cell 4 may oscillate**

This is not a correct solution for ‘AND’ gate.

III. Configuration value, 0xE10C7:



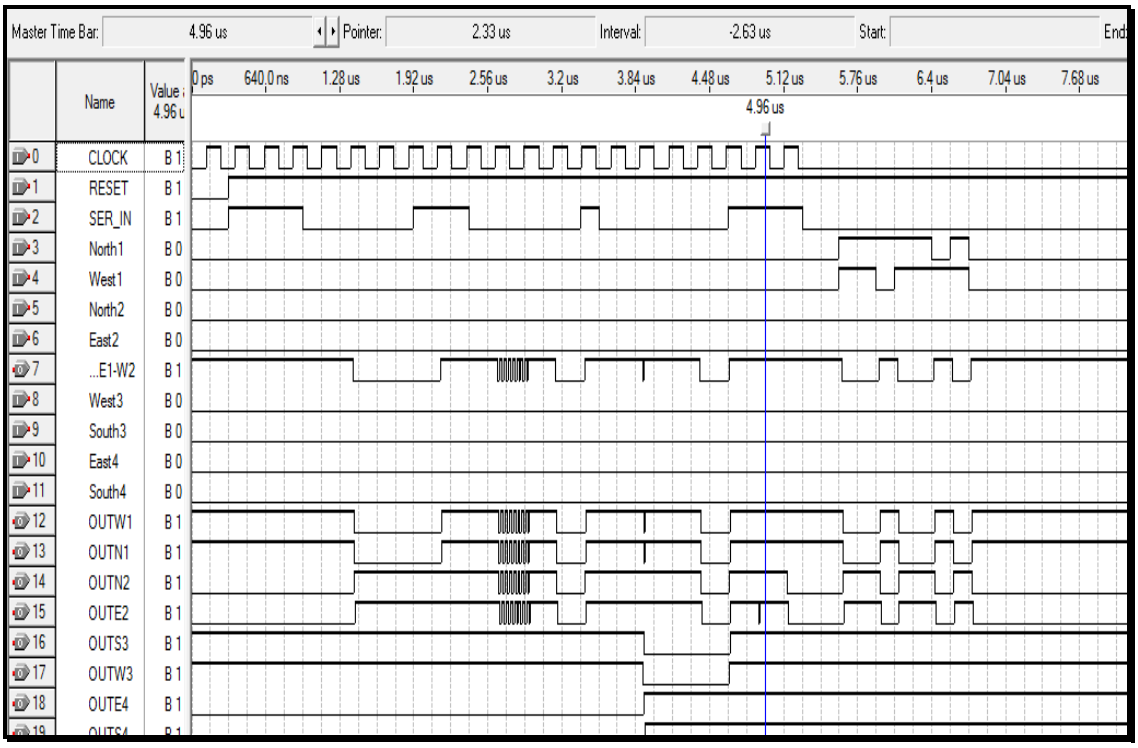


Figure 59 Manual analysis with expected working and simulation result in Quartus II

The results obtained after manual analysis of the chromosome configuration are as follows:

- **Cell 1 and cell 3 may oscillate due to the feedback**
- **Cell 2 is a NAND gate**
- **Cell 4 is Router**

This is not a correct solution for ‘OR’ gate.

7.2 Analysis of the Initial Results

The manual analysis of the evolved results showed that some solutions produced by the GA were not actually correct solutions. The results like 0x35339, that were actually oscillating were identified by the GA as correct solutions. To test the operation of the GP, it was continually loaded with the same chromosome and its fitness evaluated during this run, the chromosome e.g. 0x35339 was constantly loaded with the output pin at Cell 2 being observed on an oscilloscope. The output pin was oscillating with values ‘0’ and ‘1’, in agreement with the manual analysis.

The microcontroller only sampled the circuit output once for each input combination. When the output oscillated the value measured by the microcontroller was random, and sometimes identified the chromosome as a correct solution.

7.3 The Solution for Oscillation Detection

It was not possible to detect oscillation reliably in the GA so a hardware oscillation detector was designed. The oscillation detection circuit was added to each Cell. As shown below, this circuit consisted of a flip-flop and a XOR gate.

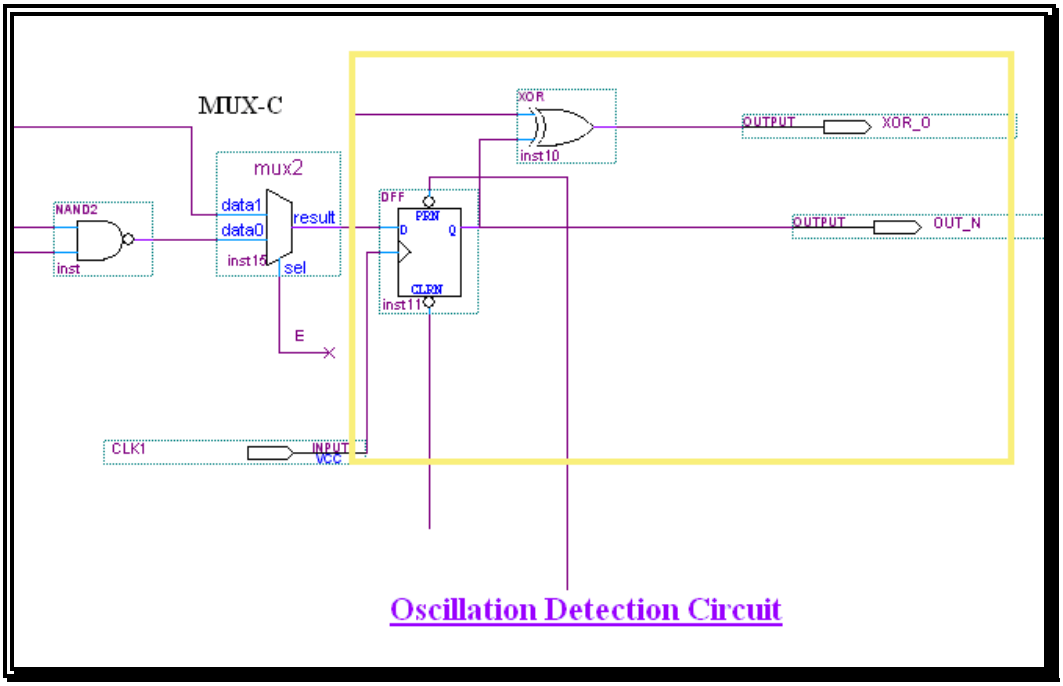


Figure 60 Oscillation Detection Circuit on a Cell

One reason for not detecting the oscillation was the uncontrolled changing of the output of each cell. An edge-triggered flip-flop was added on the output pin of every cell to control the changing of the output pin. The clock for the flip-flop was provided by the microcontroller, so the changes in the cell output could be co-ordinated with the GA. As there were four flip-flops, 5 clock pulses were chosen to check the consistency of the circuit for one combination of inputs on the north and west inputs. Four clock pulses for the flip flops and one extra pulse to check the consistency.

The output of the flip-flop was the current output of the cell and the D input to the flip-flop was the next output of the cell.

An XOR gate was also added to compare the current state and next state of the cell. The XOR output is ‘0’ if the cell output is going to change on the next clock pulse or ‘0’ if the cell output will not change.

Hence, the output of the XOR gate only for the output Cell was checked by the GA, as oscillation in other cells did not matter. If the GA detected a '1' on the XOR pin, it discarded the chromosome as an oscillating chromosome and provided it a score of decimal 0. Thus putting the chromosome at the end of the selected population and giving it a high chance to be discarded by the GA selection task. There were four tests run on the inputs for four different inputs. Code for the first test has been shown below:

```
                        // test 1

RD2=0;                          // input 1
RD3=0;                          // input 2

for (l=0;l<5;l++)               // Clock 5 Pulses for the Flip Flop
clock1();               //
for (l=0;l<5;l++)               // Check Oscillation
{       clock1();       // For Five pulses
        if(RD5==1)      // Check pin for Oscillation
        {  score = 0;   // Change Score to 0
                        //-- Used for Oscillation
          return;       // Return from the function
        }
}
if (RD4==0)             // Check circuit Output

score=score+1;          // Increment Score
```

Figure 61 Modified GA code for oscillation detection

Once the oscillation detection circuit was added on to the Cell, a new Generic Platform design was created (shown in Figure 62). This design was again loaded onto the FPGA and the GA was tested by loading it with known chromosome configurations.

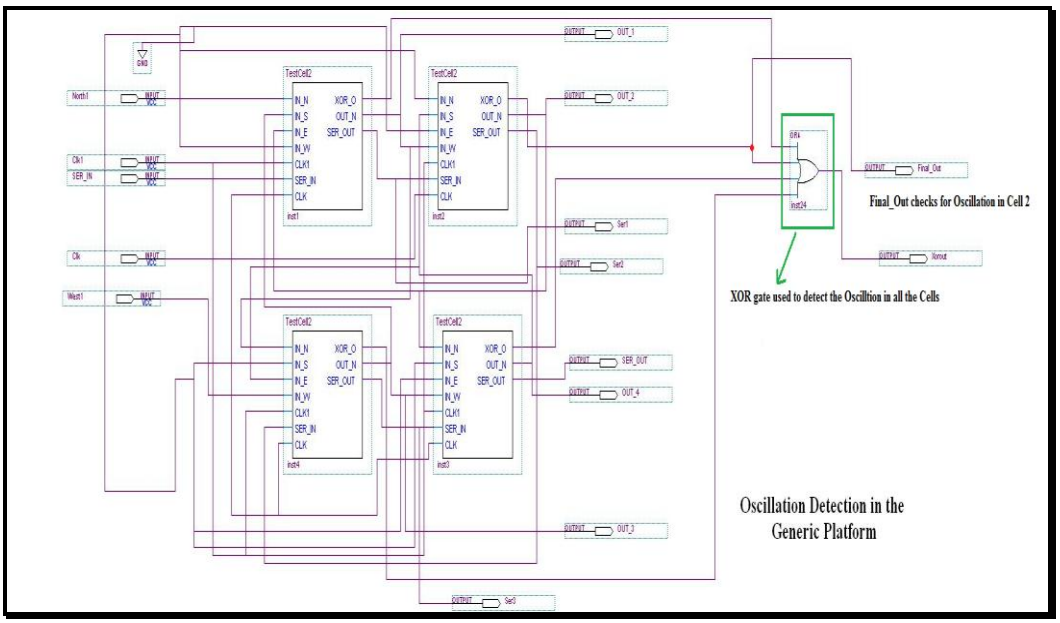


Figure 62 GP design with oscillation detection

The fitness function was modified to produce a fitness of 0 if any oscillation was detected. Otherwise, the fitness was 1 plus the number of the correct outputs. The fitness could now range from 0 to 5.

7.3.1 Observations of the Oscillation Testing

The GA was run using a single 20-bit configuration stream to check the functioning of the oscillation system.

A few runs of the GA yielded the following results:

- The circuit did not always detect oscillation i.e. the error was always different for the same chromosome. – It showed the inconsistency for the same chromosome.
- The oscillating circuit did not detect oscillations in other three cells. It only picked up oscillation in the output cell. – It behaved in the way expected, as oscillation in other cells was not being detected by the design.
- Higher generations/ runs of the GA program gave higher errors as compared to fewer generations. – It again proved to be inconsistent.

These results clearly showed the inconsistency in the oscillation detection and hence were problematic for a design. It was found that introducing a delay in the checking of the oscillations reduced number of errors, but it did not eliminate all oscillation errors.

7.4 Evolution of OR Gate

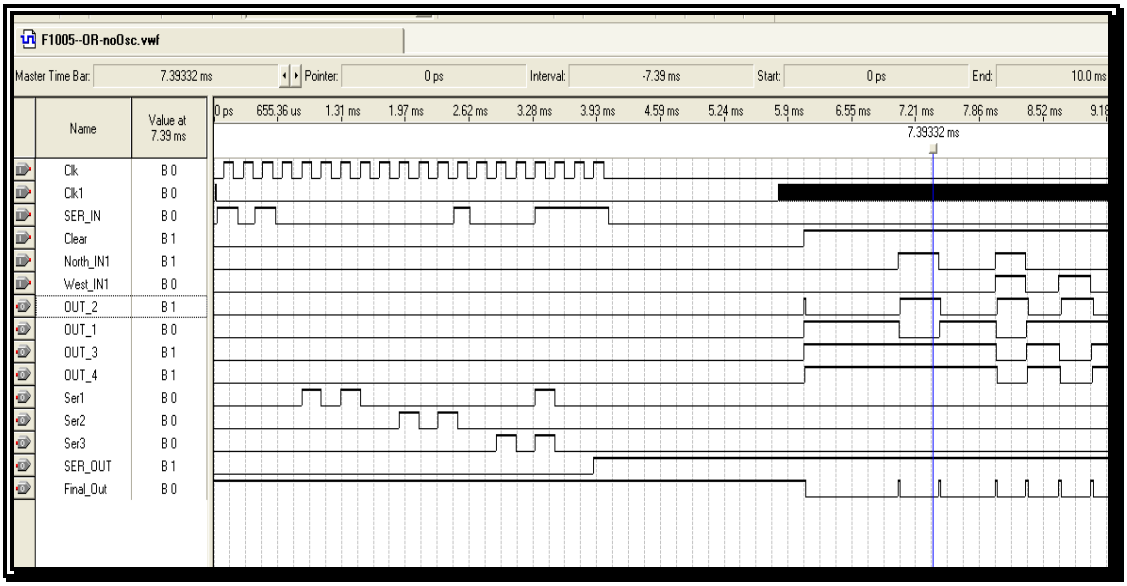
After the introduction of the oscillation circuit in the GP design, the EHW experiment was run for the second time using a complete GA with a 20-bit configuration stream. The EHW system was run to evolve an OR gate. The only difference in the GP was the input pins, this time they were assigned to two different cells i.e. Cell 1 and Cell 3, whereas the output was still at Cell 2. These inputs were named as North1 and West3 and the output was still East2. Some of the solutions evolved by the GA were:

- **0xE10C3,**
- **0xF1005,**
- **0xB1007 and**
- **0x71020**

Figure 63 Results evolved after the introduction of Oscillation Checker circuit

In this run of the GA, there were 99.6 percent incorrect solutions for OR gate in a run of 5024 iterations, leading to a mere 0.4 percent success rate. This time the results were realistic, as the 'OR' gate was expected to have much lower result than the previous run for 'AND' and 'OR' gate. To confirm the results found by the GA, a manual analysis of the derived solution was performed.

IV. Configuration value, 0xF1005:



Final_Out is the XOR output for Oscillation checking.

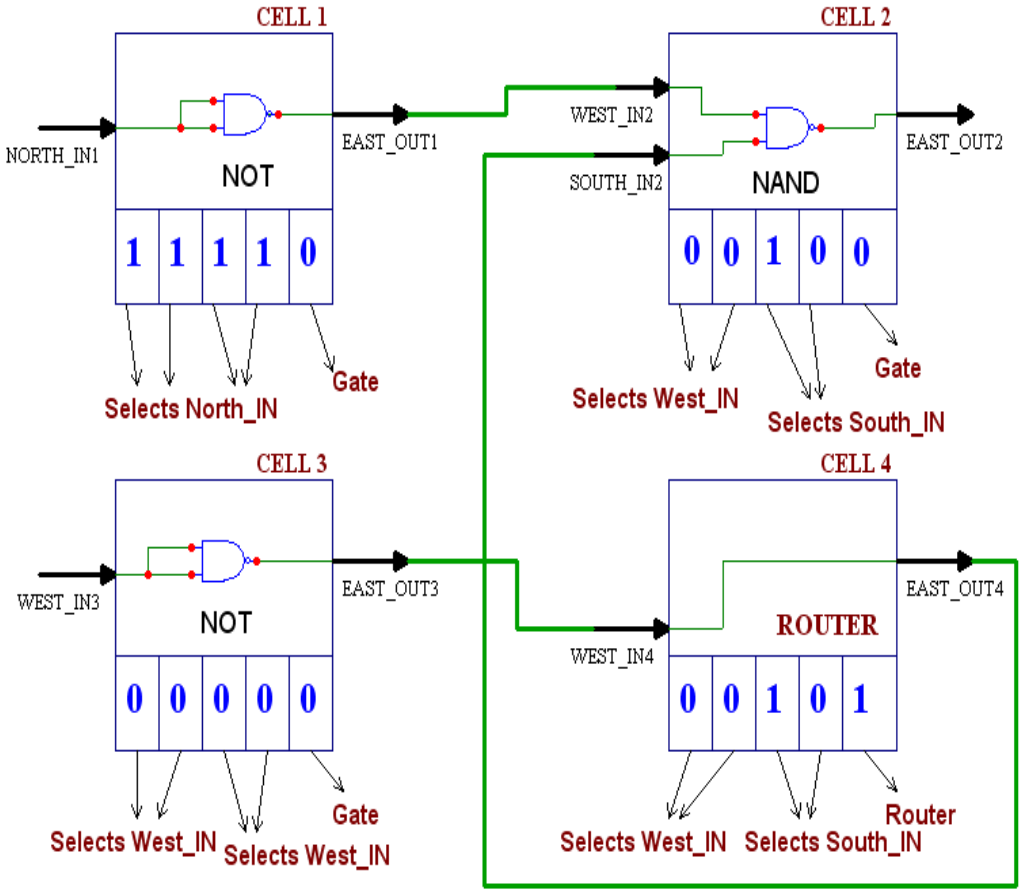


Figure 64 Manual analysis as with expected working and simulation result

The results obtained after manual analysis are as follows:

- **Cell 1 and cell 3 act as NOT gates.**
- **Cell 2 is a NAND gate.**
- **Cell 4 is a Router, routing the West input to the North output.**

This is a correct solution for an ‘OR’ gate as two NOT gates are followed by a NAND gate as shown above.

V. Configuration value, 0x71020:

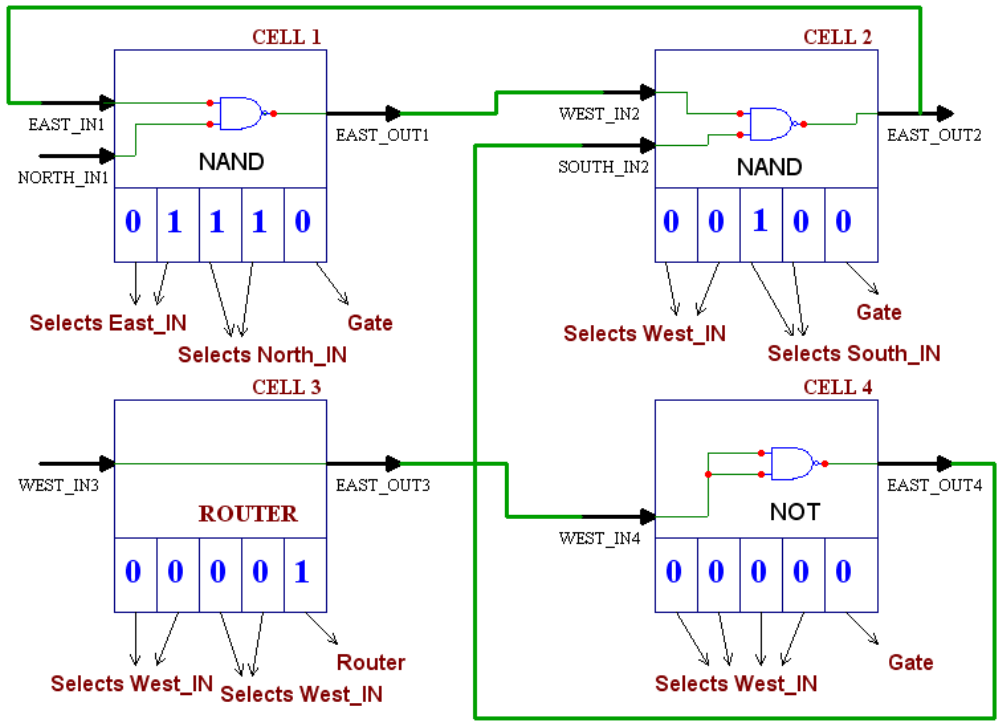


Figure 65 Manual analysis with expected working and simulation result

The results obtained after manual analysis are as follows:

- **Cell 1 is a NAND gate with East input as feedback from cell 2.**
- **Cell 2 is a NAND gate of cell 1 and cell 4 outputs**
- **Cell 3 is a Router, routing the West input to the West input of cell 4.**
- **Cell 4 is a NOT of West input**.

This is not a correct solution for ‘OR’ gate as per the manual analysis

VI. Configuration value, 0xB1007:

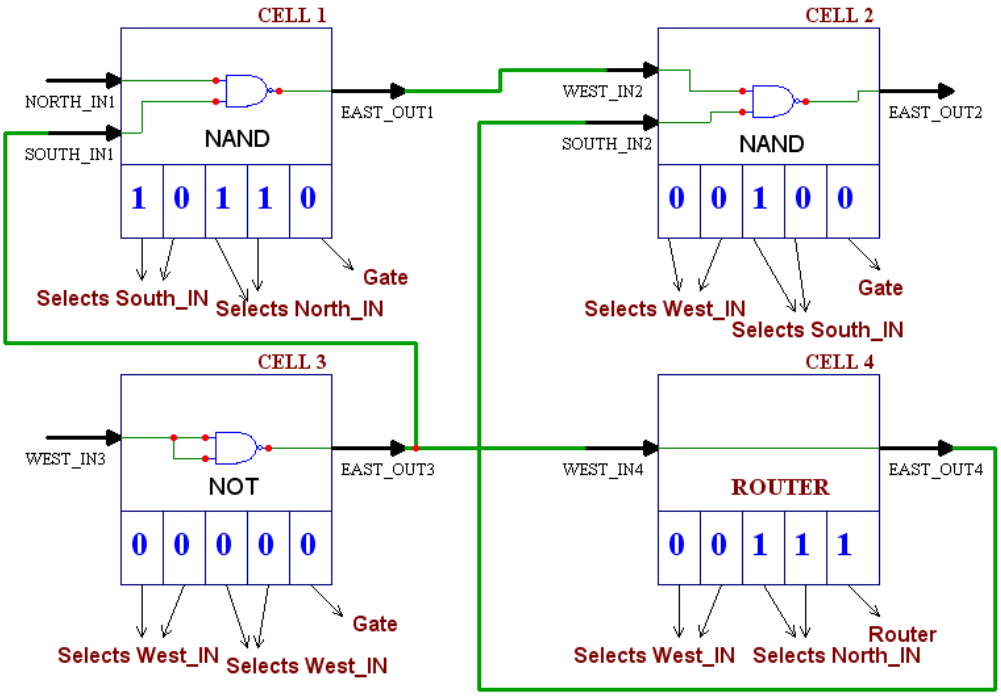


Figure 66 Manual analysis with expected working

The results obtained after manual analysis are as follows:

- **Cell 1 is a NAND gate with South input coming from cell 3.**
- **Cell 2 is a NAND gate with inputs from cell 1 and cell 4.**
- **Cell 3 is a NOT of West input.**
- **Cell 4 is a Router, routing the West input to the North output**

This is a correct solution for ‘OR’ gate.

7.5 Analysis of the OR Gate Results

From the manual analysis of the evolved results, it was found that some of the experiment-evolved solutions did not behave as expected. The chromosomes 0xB1198 and 0x71020 that were not correct solutions were identified as correct solutions. To find the reason for these results the incorrect chromosomes (e.g.0x71020) were continuously loaded into the GP and the fitness evaluated.

It was observed that the output pin had different values on different tests run for the same chromosome value as compared to the manual analysis results.

To check the difference in the manual results and the results found by the GA, we analysed the effect of using input combinations in a different order for 0x71020. It was found that this analysis agreed with evolved behaviour of the 0x71020 circuit i.e. a different sequence of input values produced different outputs. When observed as per the previous section, many chromosomes had outputs depending on the sequence in which input combination was applied. Hence, the reason for different solutions was discovered and a detailed explanation is given in the next section.

To explain the causes of the problem consider sequence of inputs for 0x71020 below:

'X' =Undefined Value

Table 7. SEQUENCE OF INPUTS FOR 0X71020

North1	West3	Out 1	*Out 2*	Out 3	Out 4
0	0	1	0	0	1
0	1	1	1	1	0
1	0	0	1	0	1
1	1	X	X	1	0

Table 7 above shows the sequence of inputs for 0x71020 chromosome with inputs North1 and West 3 and the output of 'Cell 2' as the final output. When 0x71020 had North 1 and West 3 second and third input combinations as '10' followed by '11' instead of '01' and '10' the truth table was very different as shown below

Table 8. ALTERNATIVE SEQUENCE OF INPUTS FOR 0X71020

:

North1	West3	Out 1	*Out 2*	Out 3	Out 4
0	0	1	0	0	1
1	*0*	*1*	*0*	*0*	*1*
1	1	X	X	1	0
0	1	1	1	1	0

This analysis shows that the sequence in which input combinations are tested can affect the final output obtained (Out 2). These incorrect chromosomes (e.g. 0x71020, 0x74301) have feedback between cells, leading to a situation in which the GP can give different output values for the same combinational input depending on the sequence in which the combinational inputs are tested.

We use the term 'bistable' to refer to the condition in which outputs can have different values due to feedback.

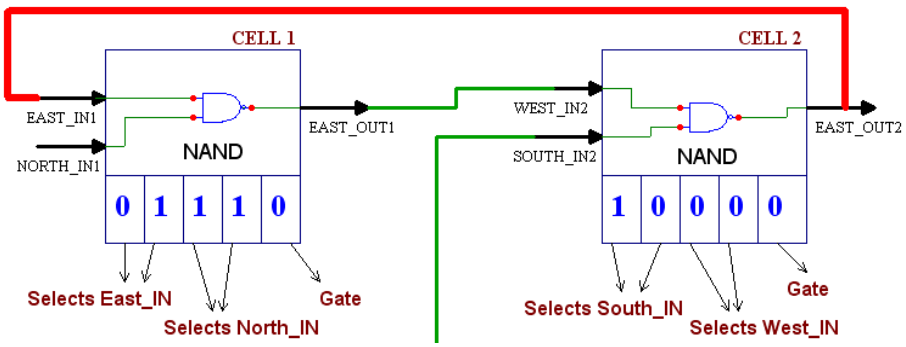


Figure 67 Gate structure emphasising feedback for 0x 71020

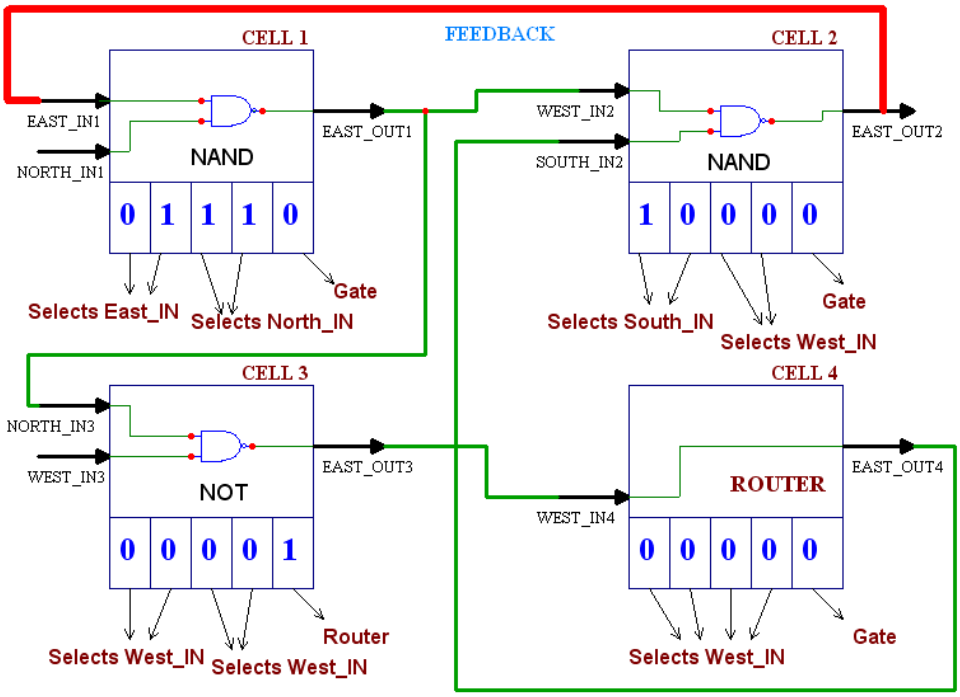
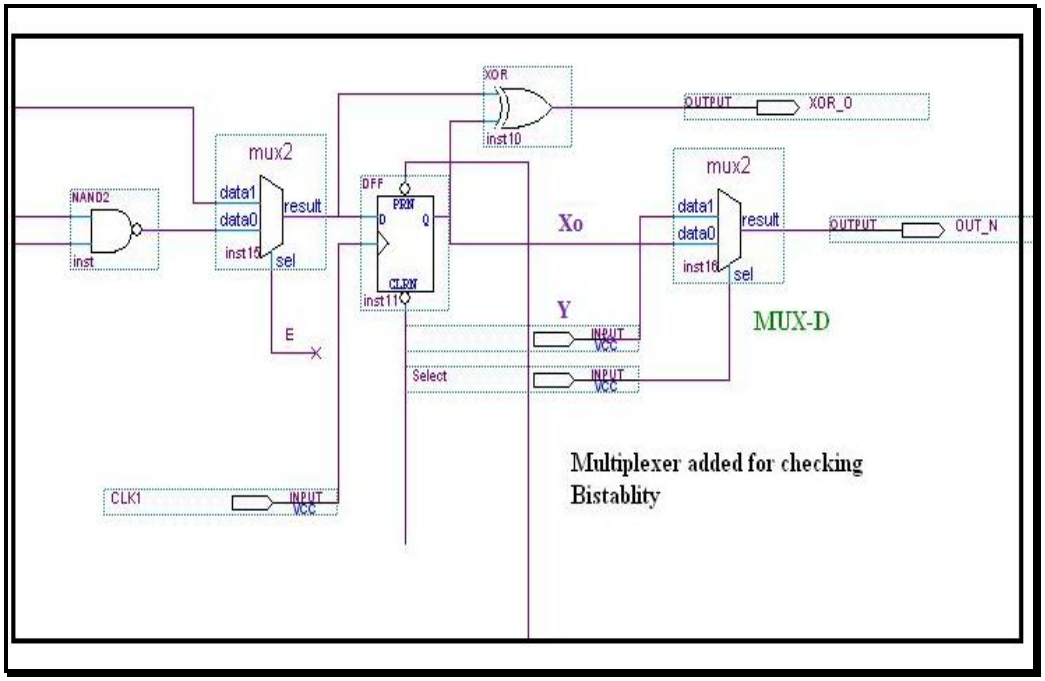


Figure 68 Gate structure emphasising on feedback in 0x74301

For some designs, this bistable state could be beneficial. However, in this research, the bistable state was undesirable because we were attempting to evolve combinational circuits.

7.6 Bistable Detection

It was found that when the evolved circuit included feedback, the behaviour of the circuit depended on the sequence in which input combinations were tested. To detect the bistability of a circuit, the GP was modified by adding a 2-to-1 multiplexer on the output pin of the output cell, cell 2. The modified design of the GP with a multiplexer added has been shown in the figure below:



Pin connections:

Xo –Input to the multiplexer, connected to the output from the cell2 i.e. East2

Y – Input 2 to the multiplexer, connected to the microcontroller

Sel – Select input, connected to the microcontroller

Figure 69 Bistable Detection Circuit on a Cell with a MUX

Based on the observations of the evolved bistable chromosomes (0x74301 and 0x71020), MUX was used for detecting the bistability in the circuits by manipulating the feedback to Cell 1. To explain the functioning of MUX please refer to the figure below:

Table 9. TABLE FOR MUX BEHAVIOUR

Y=0	Y=1	State
Xo =0	Xo=0	Stable with Xo =0
Xo =0	Xo =1	Oscillation
Xo =1	Xo =0	Bistable
Xo =1	Xo =1	Stable with Xo =1

X0 is the output from the cell.

To check the exact behaviour of the evolved circuit depending on the Xo input to the MUX, 'Y' was changed and fed through the circuit. Depending on the initial value of Xo , 'Y' was fed to the circuit using the 'Sel' input, so as to check the state of the circuit after the feedback to the circuit was changed. This was done to test the effect of feedback on the final output 'OUT_N', as shown in table 9. Hence, the behaviour of the circuit was concluded depending on the table above and constantly reading the final output 'OUT_N'.

The GP was loaded with the same 20-bit configuration stream to check the functioning of the bistable system. After a few runs of the modified GA, while trying to make the program run faster and efficiently by changing the clock cycles, it was found, introduction of a MUX did find the correct evolved solutions but if the clock cycles in the testing bit of the GA were altered, the results came different from expected. For example when the evolved solution 0x71320 was observed, it was detected as bistable on 4-clock cycles and oscillating on 2-clock cycles. Similarly chromosome 0x61741 was detected oscillating on 5-clock cycles and stable on 8-clock cycles.

Hence, these results clearly showed the inconsistency in the bistable detection and were challenging for a design.

Again, a careful manual analysis of the design was done where it was decided that based on the number of flip-flops present in the design of the GP, the maximum clock cycles to check

circuit stability would be defined. In addition, the results were only confirmed once the result was stable for at least half the maximum clock pulses required and an extra clock pulse. Therefore, the following algorithm for the GA was created:

Read Mux = X0;
Clock (2^n+2^{n-1}) pulses or until stable for consecutive (2^{n-1}) clock cycles.
Read Xout;
If stable
Mux =! X;
Read Mux = Xout;
If stable
Test the conditions for the 'OR' gate

Figure 70 Algorithm for determining clock cycles

7.6.1 Explanation of the Algorithm:

The GP contained four cells (n=4) so the number of combination states of the cell flip-flops is $2^n = 16$.

Hence, maximum clock pulses required to force the system to pass from these sixteen states would be sixteen. Also, to confirm the functionality of the GP another $2^{n-1} = 8$ clock pulses plus one extra clock pulse would be required (as decided to check the stability of the circuit) hence a total of 25 clock pulses are required to test the complete functionality of the GP system

Due to memory and pin out limitations in the microcontroller, it was decided to test for bistability in the GP design itself. Hence, a new modified version of the Generic Platform (GP) was designed without the testing multiplexer.

In this new design, a four-bit counter was introduced to implement the logic of the clock pulses and initial states of the four cell flip-flops as explained earlier. In addition, to remove the inconsistency of the starting value of the flip-flops, they were attached to a control circuit that set their starting value equal to the value of the counter. The XOR output for cell2 was read individually as well as it was XORed with other cell outputs as shown in figure 71.

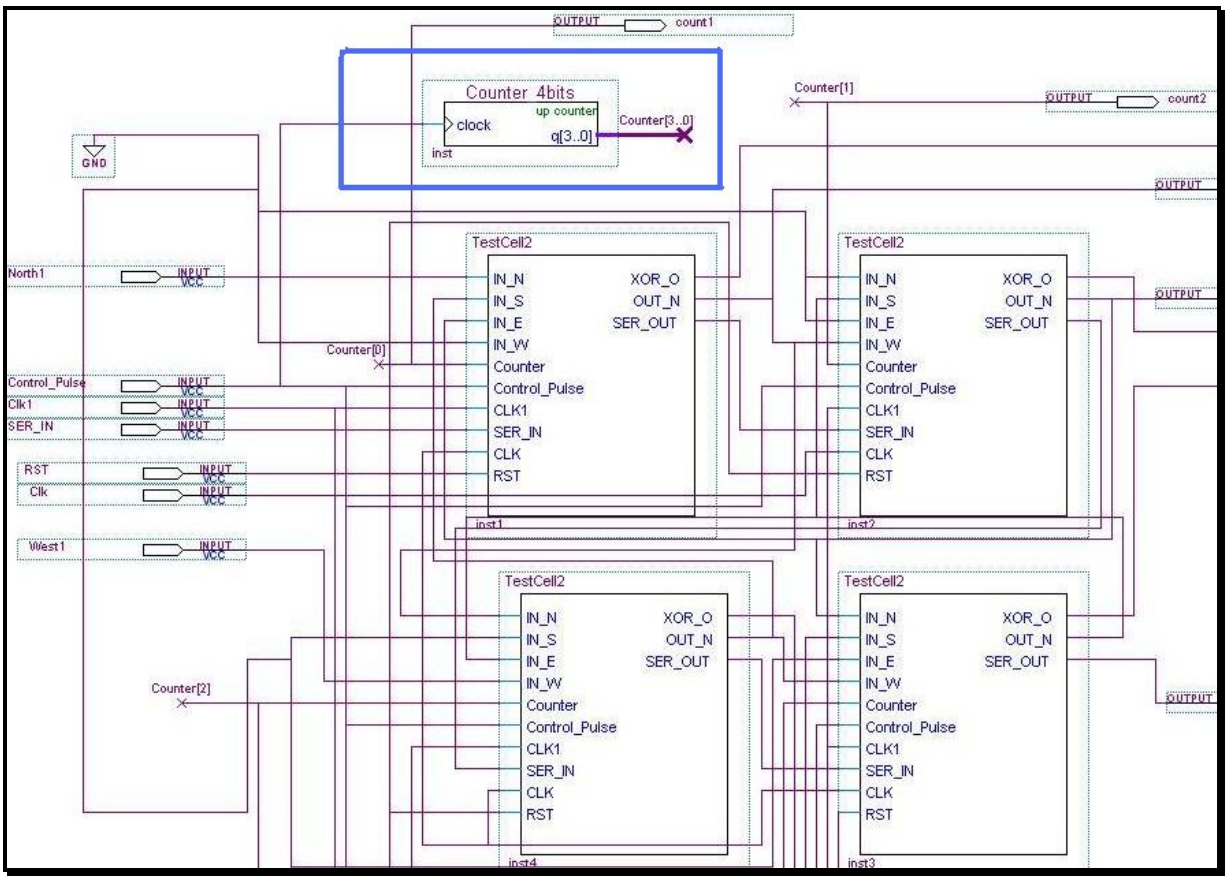


Figure 71 GP with Counter

Figure 71 shows the entire generic platform while figure 72 shows the control circuit of a single cell.

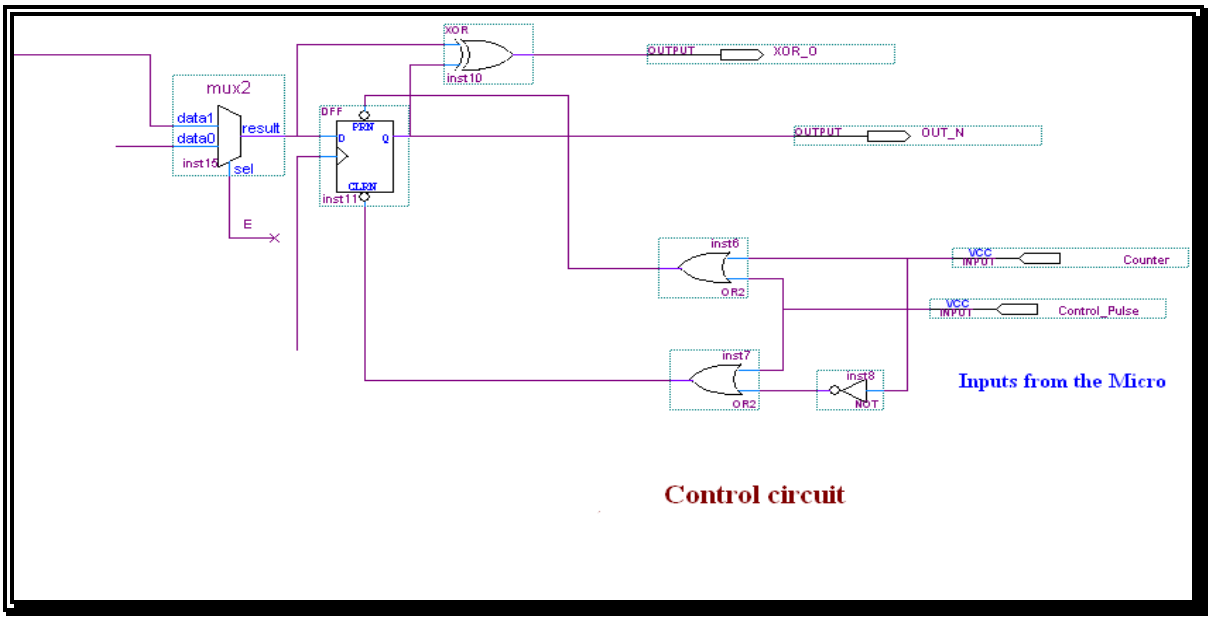


Figure 72 Control circuit

The four bits of the counter are used to set the initial values of the four flip-flops. The control (clock) pulse to the counter is fed from the microcontroller. The same control pulse is also fed to the control circuit as shown in figure 72. To set the initial state of each flip-flop, a control circuit consisting of a NOT gate and two OR gates is used. This circuit is connected to the 'Clear' and 'Preset' pins of the flip-flop.

When the control pulse is provided from the microcontroller, it increments the counter and sets or clears the four flip-flops according to the four counter outputs. This is done using the clear and preset pins on the flip-flops.

When the clear is '1' it has no effect on the flip-flop and when it is '0' it clears the flip-flop to '0', whereas when preset is '1' it has no effect but when it is set to '0' it sets the flip-flop to '1'. Figure 73 shows the introduction of the counter in the GP and explains the change of 'Final_out' to Q' instead of Q from cell2.

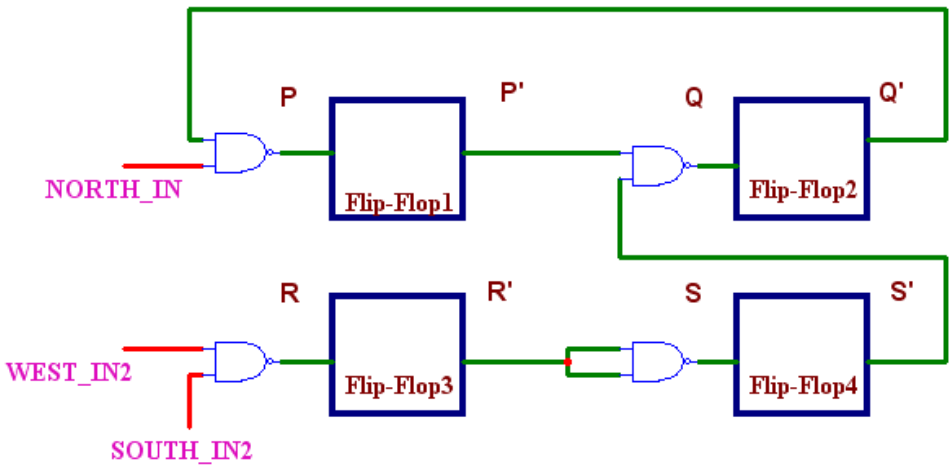


Figure 73 Working of Control Circuit in the GP with the introduction of flip-flops e.g 0x71020.

Table 10. TRUTH TABLE SHOWING INTRODUCTION OF COUNTER FOR 0X71020.

N	W	P	Q	R	S	P'	Q'	R'	S'
1	0	1	0	0	0	1	1	0	1
1	0	1	1	0	1	0	0	0	1
1	0	0	0	0	1	1	1	0	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
Continues for all the possible states of North and West									

Here P, Q, R and S are the outputs of the cells and the P', Q', R' and S' are the outputs of the cells after the flip-flops.

Based on the truth table (Table 10), the steps to be implemented in the genetic algorithm running on the microcontroller were as follows:

STEPS:

I. Set the states for the two external inputs, e.g. North1 = '0' and West3= '0'.
II. Force the flip-flops to an initial state using the counter.
III. Clock the circuit once and read the XOR output for any oscillations in the circuit.
IV. Clock the circuit up to 24 times, reading the XOR output each time, comparing the output from the first clock pulse (step 3). If the XOR output is same for all the consecutive pulses the circuit is stable, otherwise it is oscillating.
V. Repeat the steps from 2 to 4 for all sixteen counter values, to test all possible flip-flops starting value combinations. If the final output is different for any of the sixteen states, the circuit is bistable
VI. Repeat steps 1 to 5 for each of the 4 external input combinations.

This is shown in the flow chart in Figure 74.

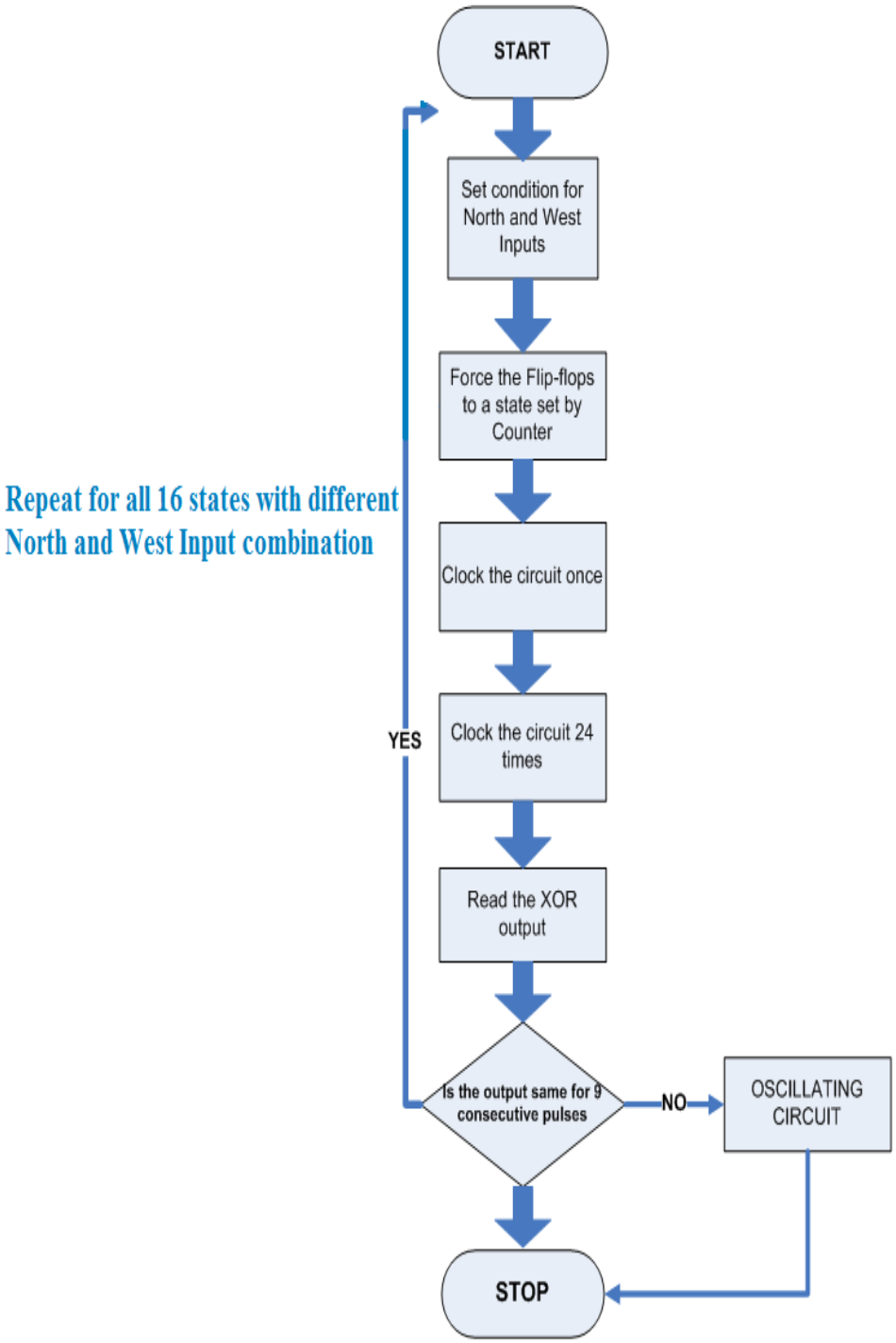


Figure 74 Flow Chart depicting the steps above

8 Conclusion and Future Work

8.1 Conclusion

The literature review for evolvable hardware found that there have been many experiments carried out in this field. Mostly these experiments are specific to an individual field or, are confined to particular hardware only. The experimental evolvable hardware system described herein has attempted to generalise this limitation of current and past experiments. Firstly, the designed generic platform is limited to a particular, hardware or platform. Secondly, it can be used for evolvable hardware even in the absence of evolvable friendly features. Thirdly, it uses a very small bit stream of 5 bits per cell, much less than other designs using a cell structure. Finally, the generic platform has eliminated FPGA intellectual property limitations, such as the non-availability of information bit stream used to control the FPGA hardware.

Though an intrinsic evolution was done on this evolvable hardware system, but due to constrains of the microcontroller the bonus objective of expansion of the generic platform to a larger structure could not be achieved at this stage. The experimental results do show the intrinsic evolution on the generic platform implemented on different hardware platforms as opposed to the confinement of evolutionary experiments on single hardware, described in the reviewed literature.

The oscillation and bistable problems of the generic platform were handled successfully, thus providing a stable error free base for the development of such systems. However, the clocking of the generic platform depends on the number of flip-flops, with the number of clock pulses increasing exponentially, thus limiting the scalability of this platform.

A generic platform for the intrinsic evolution on any available hardware was developed, but due to the available hardware constrains, the system could not be elaborated to a large scale. In addition, the solution developed for bistability was confined to this particular generic platform and could not be extended to higher array GP, as clocking large GP structures could be cumbersome.

It could be possible that all of the above problems are resolvable, and the work involved would be subject to future research.

8.2 Future Work

The generic platform can be extended to a greater array structure to facilitate complex electronic designs for evolvable hardware. A larger structure would thus require a better hardware as compared to the one used, hence the hitch of microcontroller can be resolved by introducing an FPGA instead for the running of GA.

Another option would be the incorporation of the microcontroller into the FPGA as a soft-core processor, to increase the speed of the system.

A faster method can be developed for the clocking of individual flip-flops, if both the GP and GA are run in the same hardware by the same clock.

Provided the microcontroller used for the GA has enough memory and speed, the generic platform can be modified so that it uses fewer resources on the FPGA hardware. This can be done by running the counter for bistability checking in the GA rather than in the GP .

It is hoped that the introduction of a better hardware for the platform would solve these glitches efficiently, thus leading to a resource friendly and superior generic platform.

APPENDICES

APPENDIX I : Procedure for Running Quartus Experiment

Basic Cell Structure, Programming and Testing Procedure

1. Make a folder for your work e.g. 'Testcell'.
2. Invoke Quartus II.
3. Go to File | New Project Wizard. On the introduction, screen click- Next.
4. Enter the working directory e.g. C:\Testcell.
5. The Wizard will automatically select the working directory name as the default for project name i.e. Testcell. Also ensure the Top-level design entity name is also same as project name i.e. Testcell – click Next.
6. The wizard will ask to add files in the project, leave the space blank and click Next. Note: Do not add any files yet.
7. In the 'Family and Device Settings' choose 'MAX7000S' in the Family and select 'EPM7128SLC84-15' in the Target device category. Leave other options to default and click Next.
8. On the EDA Tool setting leave everything to default – Next.
9. Click Finish on the Summary page.
10. As we have already created the schematic design for the project – Copy and rename the '.bdf' and 'lpm' max files into the working folder i.e. C:\Testcell. Note: The 'bdf' file should be named exactly as the working folder.
11. *Checking of the Code*: Open the 'Testcell.bdf' and go to Select – Processing – Start – Start Analysis & Synthesis.
12. *Assigning Pins*: Pin assignment in the design can be done by selecting Assignments – Pins. Make sure the Category (on the Right Hand Side) is 'Pins'.
13. Under the 'To' column or <<New>> double left click and highlight the pin name e.g. Clk.
14. Then under the 'location' column or <<New>>double left click and highlight the Pin number e.g. Pin_76. Complete the list of pins as shown in the Appendix.
    Hint: Already allocated pins would show in *Italics*.
15. Close the window and save when prompted.

16. Go to Assignments – Settings – Simulator – choose ‘Functional’ in the Simulation mode on the Right Hand *Side an*d leave rest to default.
17. *Compilation*: To compile the design go to Processing – Start Compilation. Compilation should complete, showing usage as 10 % of the total macrocells.
18. *Simulation*: Now to simulate the design, do the following:
    i. Processing – Generate Functional Simulation Net list.
    ii. File – New – Other Files – Vector Waveform File – Ok.
    iii. Edit – End Time – 300ms.
    iv. View – Fit in Window.
    v. Edit – Insert Node or Bus.
    vi. Click on Node Finder
    vii. In Filter select Pins: all – List.
    viii. Transfer all the signals on LHS to RHS – Ok – Ok.
    ix. In the waveform editor move any inputs to the top of the sequence by selecting (Highlight + Hold +Move) the desired ones.
    x. *Place clock signal*: Highlight clock and right click – value – clock – accept default 10 ns – Ok.
    xi. Make sure ‘RST’ is low for the 1st clock period, after that is should be ‘1’ throughout. As shown :

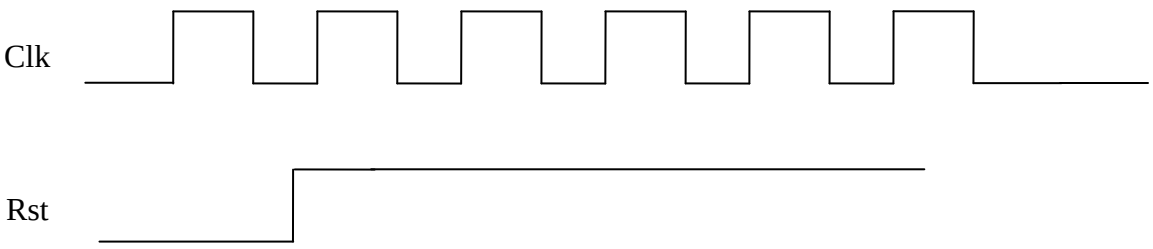
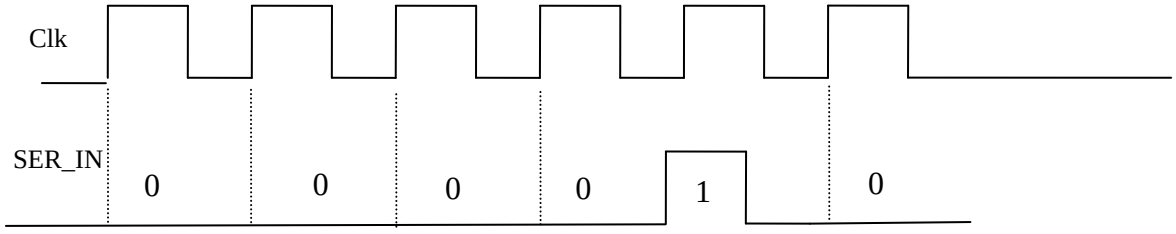


Figure 75 Selection of clock cycles

    xii. We wish to simulate the configuration of the following function :
    B (1) =0 B (0) =0 A (1) =0 A (0) =1 E=0
    xiii. Set the ‘SER_IN’ as shown, after the ‘RST’ is set to 1 :



Make sure clock signal is only 6 clock periods.

Figure 76 Setting up of SER_IN

The desired function will be:

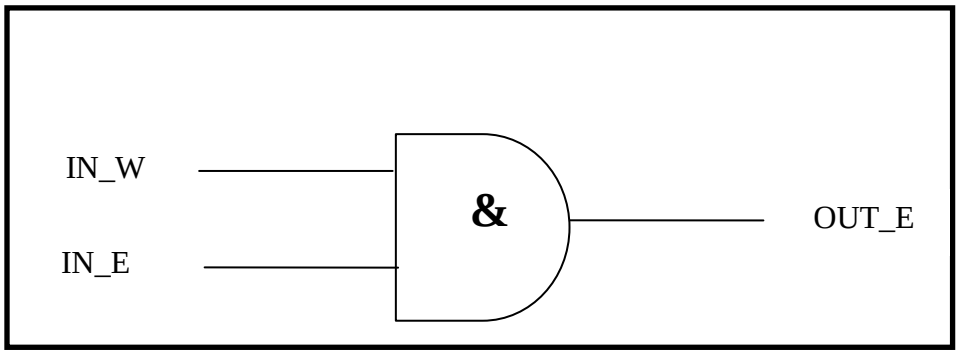


Figure 77 AND gate function

xiv. After the configuration is loaded, set 'IN_E' to '1' for more than one clock period and 'IN_W' to '1' for the same time. This will set the desired inputs 'East' and 'West' to '1'.

19. Save the design as 'Testcell.vwf'.
20. Processing – Start Simulation.
    You should see 'OUT_N' go to '0' when 'IN_E' =1 and 'IN_W'= '1'.

Programming the Device

1. Connect Power Supply for MAX7000S Target Board to the Power point.
2. Connect 'Byte Blaster' cable to the parallel port of the PC and to the Target Board.
3. In Quartus, go to Tools – Programmer – Select Hardware Setup (RHS).
4. Add Hardware—Hardware Setup.

Make sure Byte Blaster MV or Byte Blaster II is selected on LPT1—Ok—Close.

5. Make Sure File 'Testcell.pof' is selected.
6. Also make sure that the Programming /Configuration box is checked.
7. Select Start, Observe Progress --- 100%.
   It should show that the CPLD has been programmed.

Make a simple microcontroller program and connect the micro controller to the CPLD. Select the Baud Rate of 4800.

A test program was designed for 16F877 called 'cell_test.c', which was connected to the CPLD as follows:

Table 11. PIN SELECTION

PORT D	PIN Name	PIN Number
RD0	Clk	76
RD1	SER_IN	64
RD2	IN_E	75
RD3	IN_W	73
RD4	OUT_E	70

Testing

The simulation of the design showed the following results:

Table 12. TRUTH TABLE FOR DESIRED FUNCTION.

IN_E	IN_W	OUT_E
0	0	1
0	1	1
1	0	1
1	1	0

The Pin setup to be used while configuring the device:

Table 13. PIN SETUP TABLE

PIN Name	PIN Number
Clk	76
IN_E	75
IN_N	74
IN_W	73
OUT_E	70
OUT_N	69
OUT_S	68
OUT_W	67
RST	1
SER_IN	64
SEROUT	63
IN_S	61

APPENDIX II: Testing done before running experiment

1. *Pin selection for FLEX CPLD and ATMEL Microprocessor*

Table 14. PIN SELECTION TABLE FOR FLEX

Pin name on ATMega128	Pin Number on FLEX	Pin state from Micro
Input1	158	Output
Input2	156	Output
SER_IN	161	Output
Clk	163	Output
Clk1	91	Output
OUT2	149	Intput
Final_Out	144	Output
Ser_Out	153	Output

2. *Pin selection for MAX CPLD and PIC Microprocessor*

Table 15. PIN SELECTION FOR MAX

Pin name on PIC 16F877	Pin Number on MAX	Pin state from Micro
Input1	17	Output
Input2	18	Output
SER_IN	16	Output
Clk	15	Output
Clk1	24	Output
OUT2	20	Intput
Final_Out	21	Output
Ser_Out	22	Output

3. *Simulation of NOT gate —Input is the IN_W (West input)*

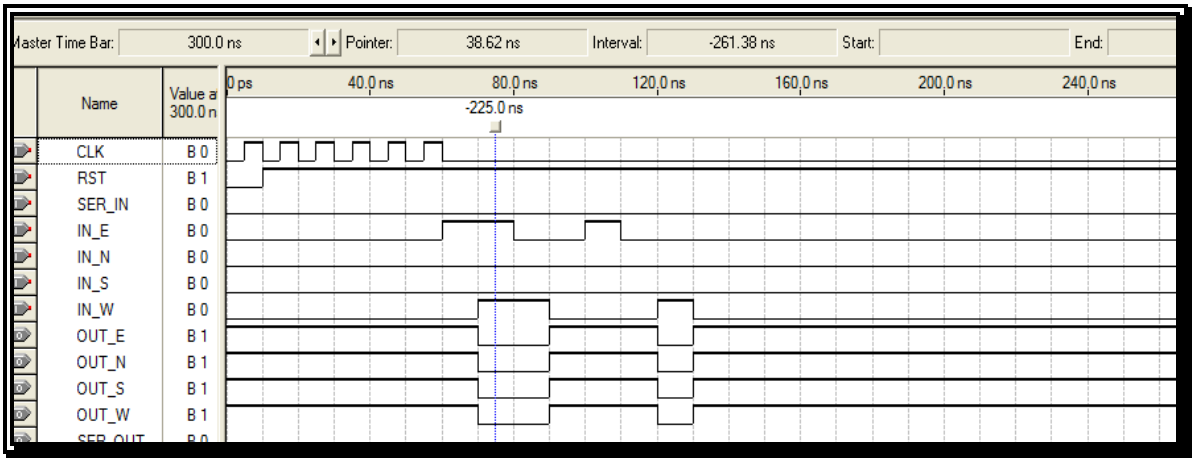


Figure 78 Figure showing NOT gate simulation

4. *Simulation of NAND gate—Inputs are the IN_N(North)and IN_W(West)*

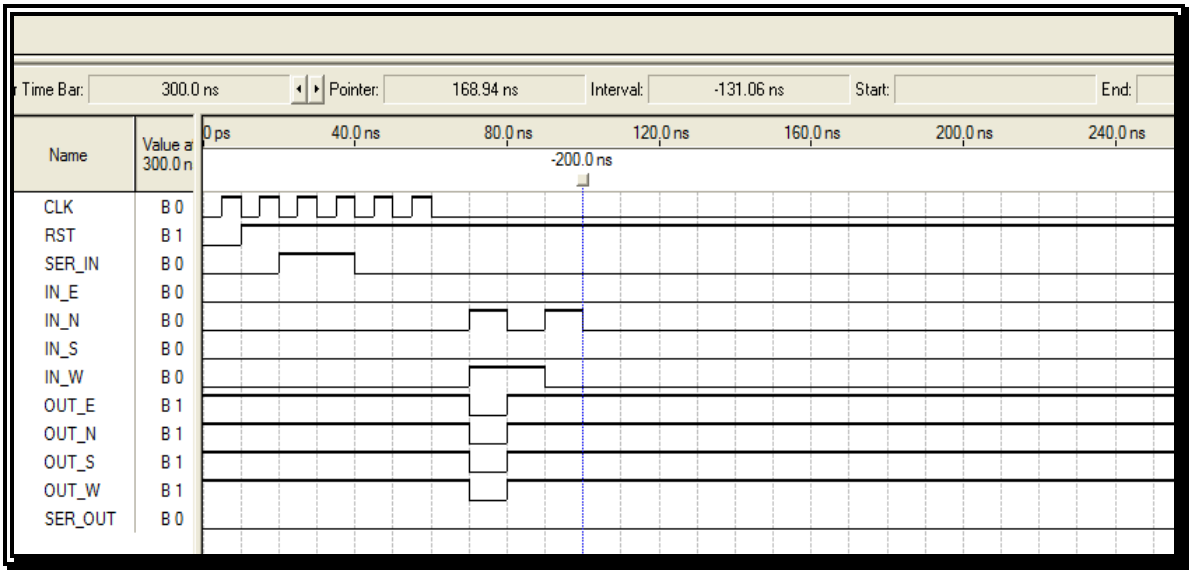


Figure 79 Simulation of NAND gate

5. *Test Point Diagram for 2x2 array structure*

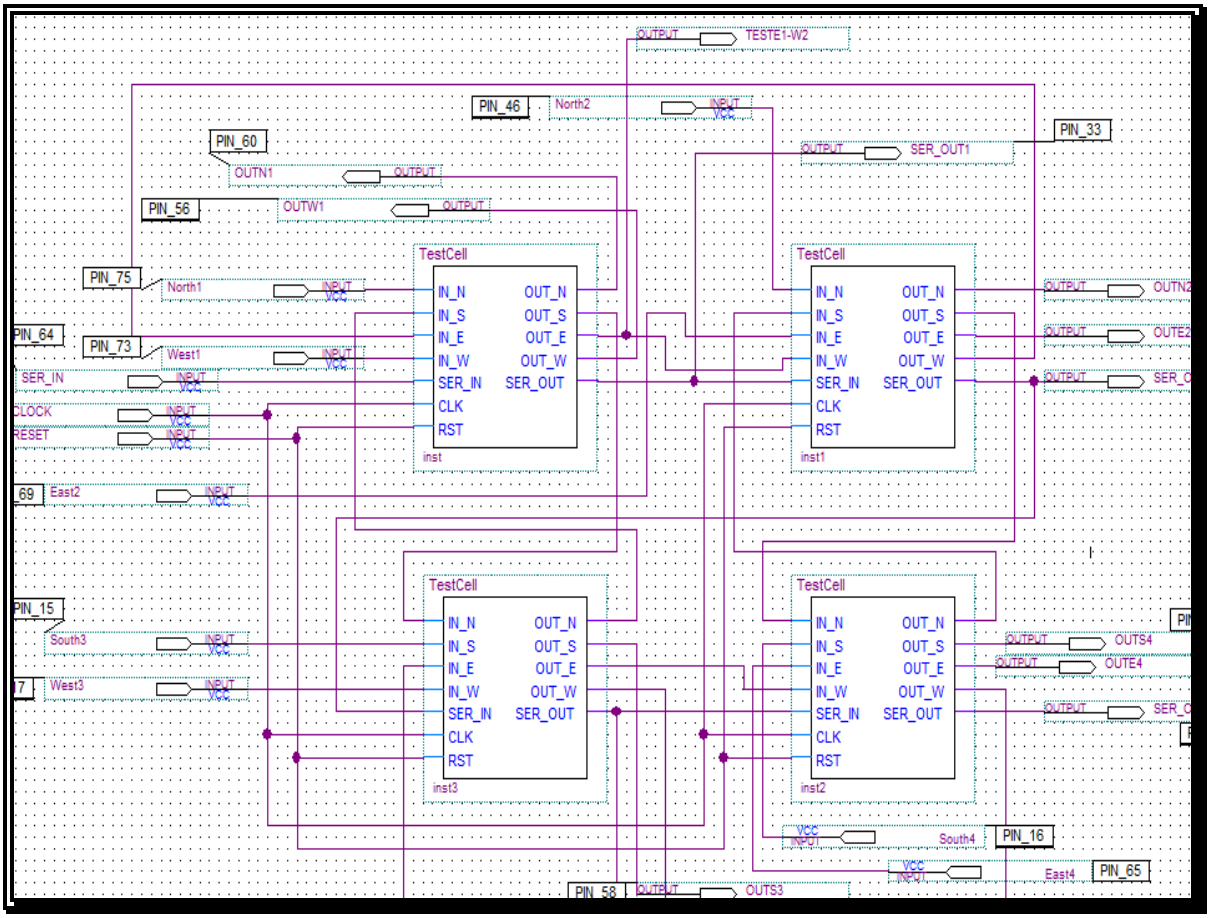


Figure 80 GP showing Test points

6. Simulation of 2x2 array with Test point

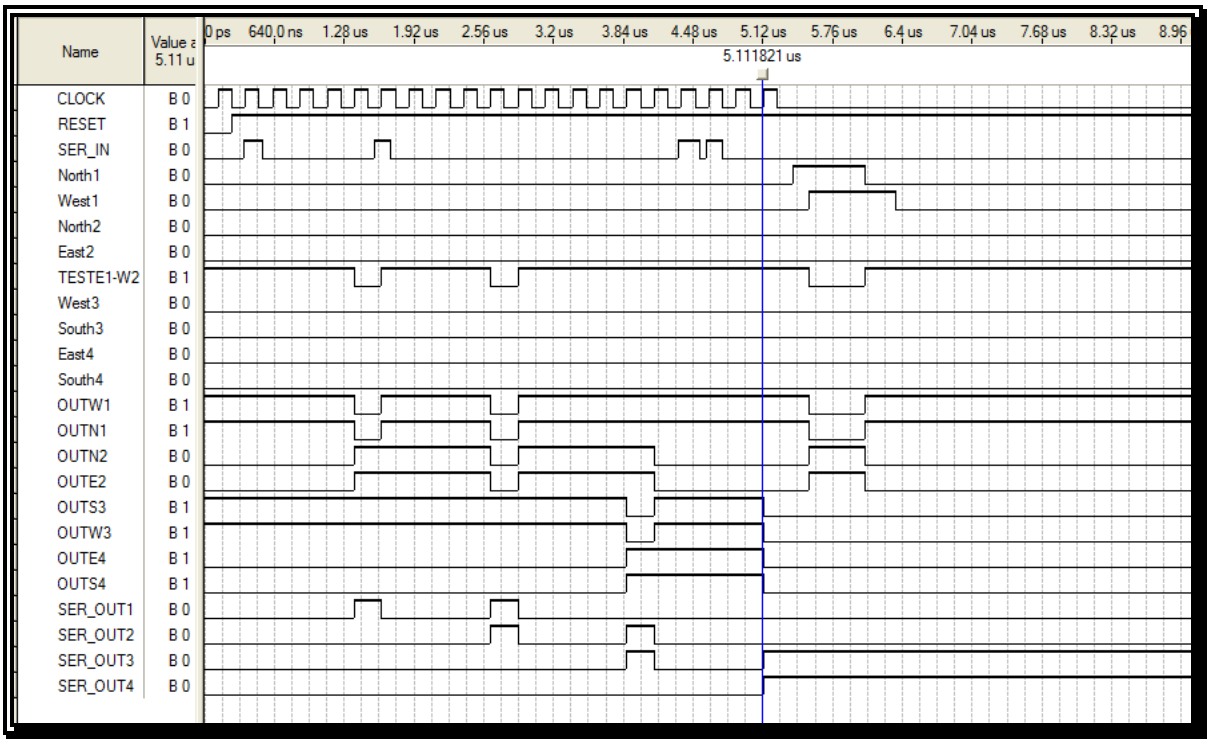


Figure 81 Simulation of 2x2 array

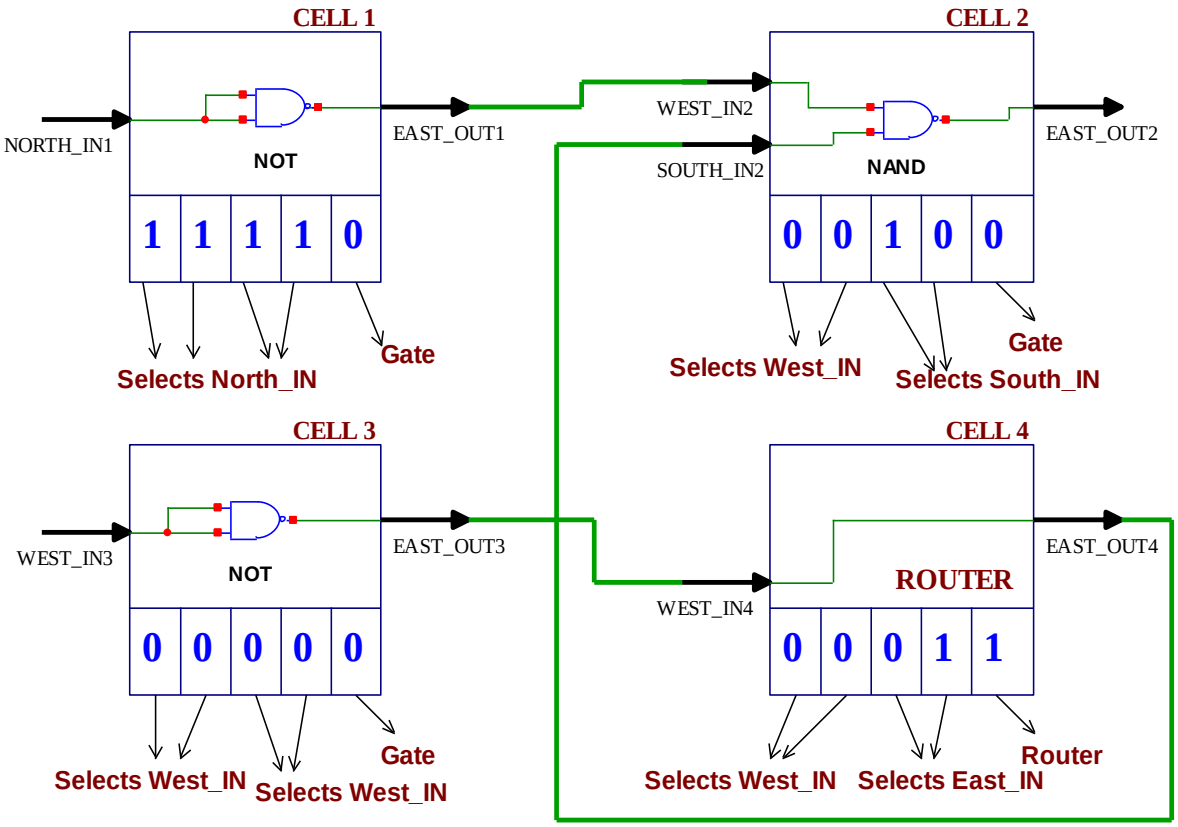
APPENDIX III: Results achieved during experimentation

List of some of the evolved solutions after the first run of experiment (all the values are Hexadecimal digits)

- **F10D9**
- **B1303**
- **B10C5**
- **E4303**
- **F1005**
- **B4307**
- **34130**
- **64DD6**
- **30021**
- **C03F0**
- **C028D**
- **35539**
- **3130A**

MANUAL ANALYSIS OF EVOLVED SOLUTIONS (OR GATE) AFTER THE FIRST RUN OF EXPERIMENT

1. *Manual analysis of 0xF1003 while circuit was behaving as an OR Gate*



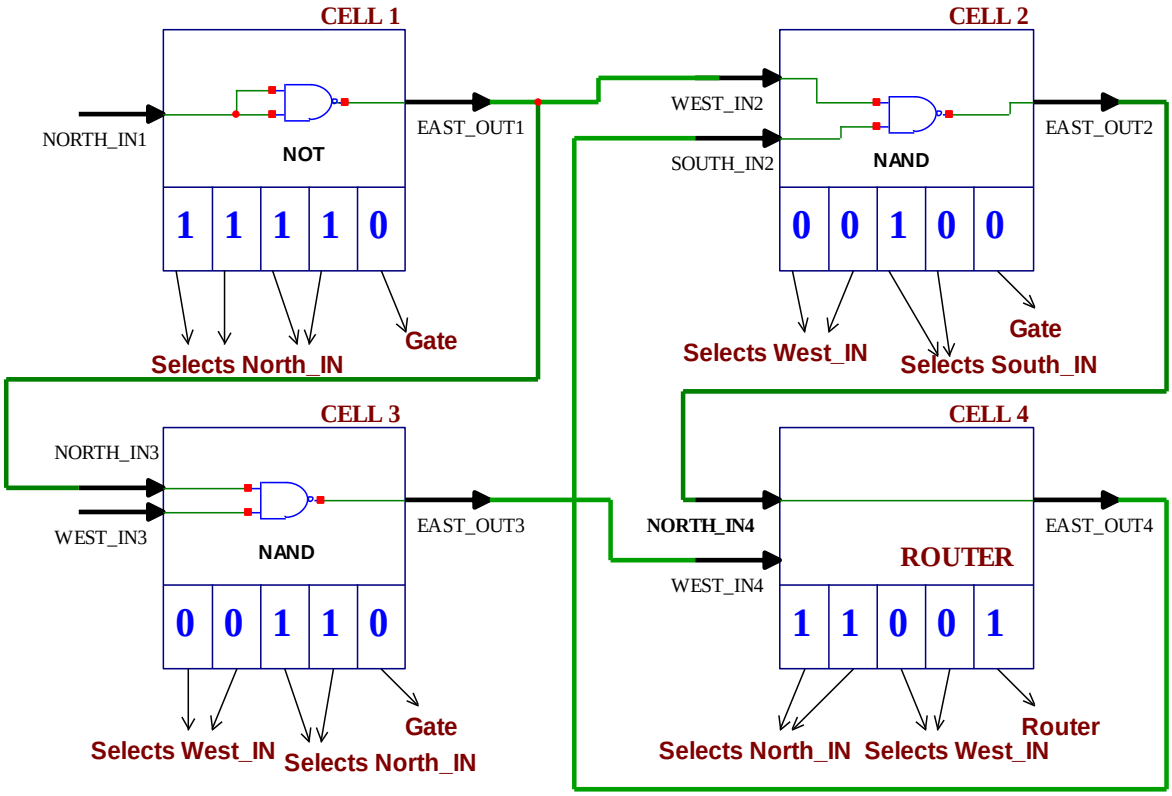
The results obtained after manual analysis are as follows:

- **Cell 1 and Cell 3 act as a NOT gate**
- **Cell 2 is a NAND gate.**
- **Cell 4 is a Router, routing the West input to the North output**.

This is a correct solution for 'OR' gate.

Figure 82 Manual analysis of 0xF1003 while circuit was behaving as an OR Gate

2. *Manual analysis of 0xF10D9 while circuit was behaving as an OR Gate*



The results obtained after manual analysis are as follows:

- **Cell 1 is a NOT gate**
- **Cell 2 and Cell 3 act as NAND gates.**
- **Cell 4 is a Router, routing the North input to the North output.**

This is not a correct solution for ‘OR’ gate.

Figure 83 Manual analysis of 0xF10D9 while circuit was behaving as an OR Gate

- **Simulation of 0xF1003**

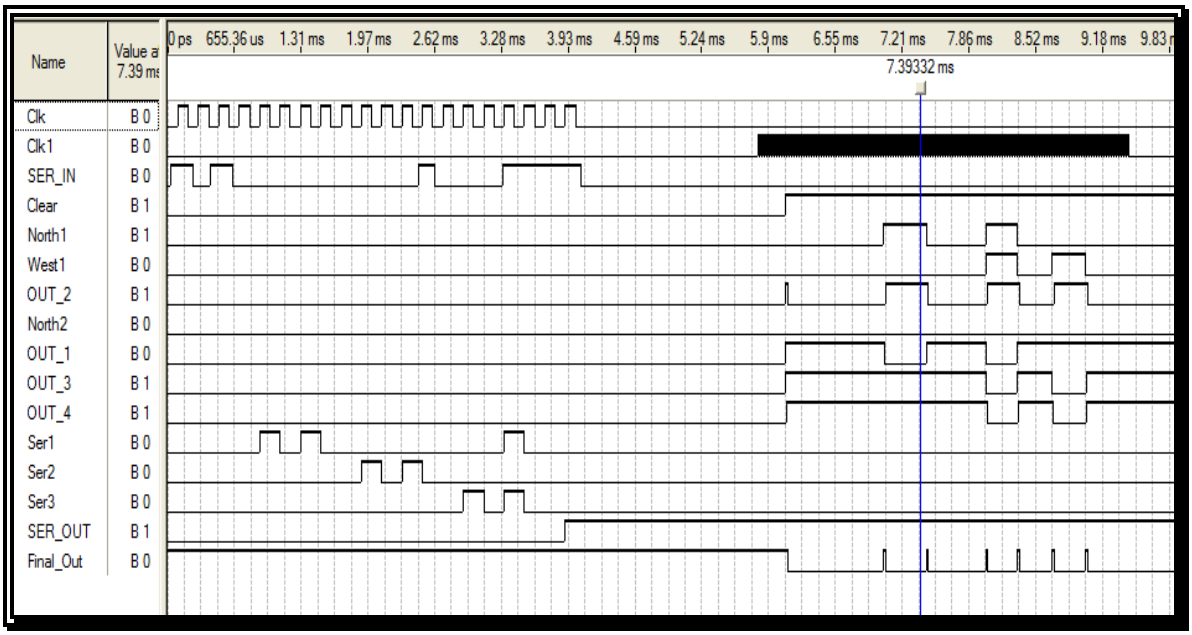


Figure 84 Simulation of 0xF1003

- **Simulation of 0xF10D9**

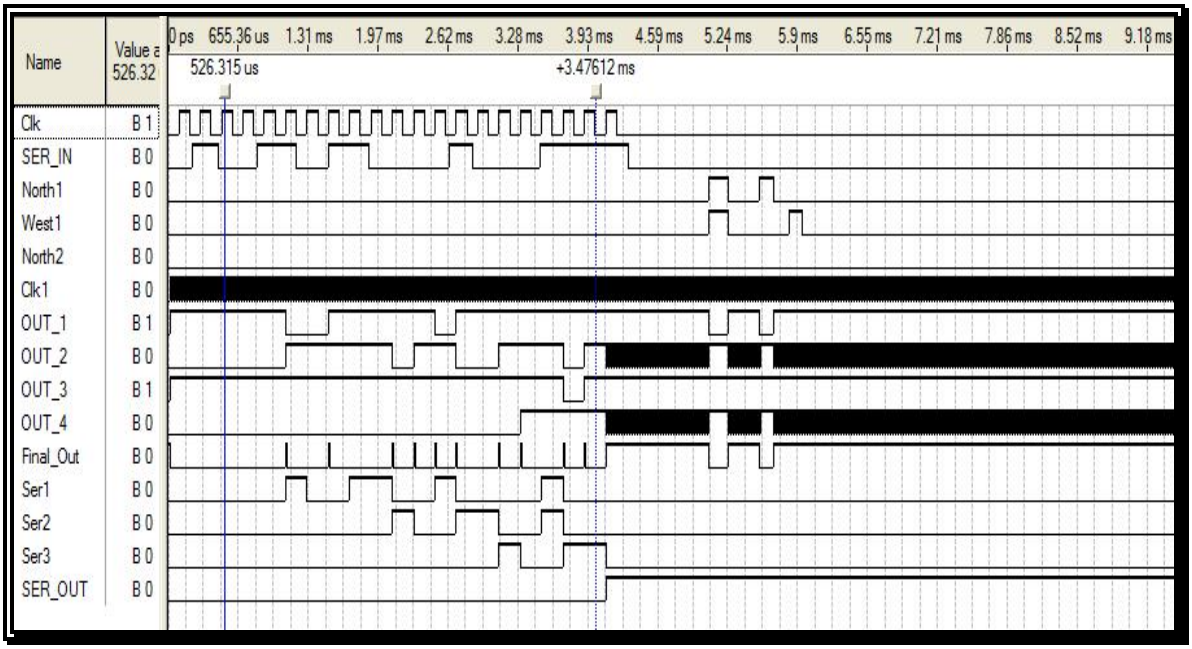


Figure 85 Simulation of 0xF10D9

REFERENCES

[1] J. Torreson, "An Evolvable Hardware Tutorial " in *14th International* Antwerp, Belgium, 2004.

[2] A. Parikh and S. Soltic, "Evolvable Hardware," in *BE mini conference* Manukau Institute of Technology,Auckland,New Zealand, 2005.

[3] P. Haddow and G. Tufte, "Bridging The Genotype-Phenotype Mapping For Digital FPGAs," in *The Third NASA/DoD Workshop on Evolvable Hardware*, Los Alamitos, CA, USA, 2001, p. 0109.

[4] T. Higuchi, T. Niwa, T. Tanaka, H. Iba, H. de Garis, and T. Furuya, "Evolvable Hardware with Genetic Learning," in *Proc. of Simulated Adaptive Behavior*, 1993, pp. 417-424.

[5] T. Higuchi, M. Iwata, I. Kajitani, H. Yamada, B. Manderick, Y. Hirao, M. Murakawa, S. Yoshizawa, and T. Furuya, "Evolvable hardware with genetic learning," *Circuits and Systems, 1996. ISCAS'96.,'Connecting the World'., 1996 IEEE International Symposium on*, vol. 4, 1996.

[6] "NASA/DOD 2005 Conference on Evolvable Hardware." vol. 2007 Washington D.C., 2005.

[7] Y. Sato, "Proposal for a Field-Evolvable Hardware based on a Microprocessor Incorporated Flash Memory," in *Proceedings of the 2001 Congress on Evolutionary Computation*, Seoul, Korea, 2001, pp. 608-615.

[8] I. Kajitani, T. Hoshino, N. Kajihara, M. Iwata, and T. Higuchi, "An evolvable hardware chip and its application as a multi-function prosthetic hand controller," *Proceedings of the sixteenth national conference on artificial intelligence and eleventh innovation applications of AI conference on Artificial intelligence and innovative applications of artificial intelligence table of contents*, pp. 182-187, 1999.

[9] D. Keymeulen, M. Durantez, K. Konaka, Y. Kuniyoshi, and T. Higuchi, "An evolutionary robot navigation system using a gate-level evolvable hardware," in *Evolvable Systems: From Biology to Hardware*, 1997, pp. 193-209.

[10] J. F. Miller, "Digital filter design at gate-level using evolutionary algorithms," *Proceedings of the Genetic and Evolutionary Computation Conference,* vol. 2, pp. 1127-1134, 1999.

[11] T. C. Fogarty, J. F. Miller, and P. Thomson, "Evolving Digital Logic Circuits on Xilinx 6000 Family FPGAs," in *Soft Computing in Engineering Design and Manufacturing*, London, 1998, pp. 299-305.

[12] "Proceedings of the First NASA/DoD Workshop on Evolvable Hardware," in *The First NASA/DoD Workshop on Evolvable Hardware*, Jet Propulsion Laboratory, California Institute of Technology,Pasadena, California, USA., 1999.

[13] U. o. Sussex, "Evolutionary and Adaptive Systems at Sussex," Sussex.

[14] Y. Kamiya, S. Denno, Y. Mizuguchi, M. Katayama, A. Ogawa, and Y. Karasawa, "Development of an adaptive array based on subband signal processing," *Electronics and Communications in Japan (Part III: Fundamental Electronic Science),* vol. 85, pp. 19-30, 2002.

[15] A. Stoica, R. S. Zebulum, D. Keymeulen, M. I. Ferguson, V. Duong, and X. Guo, "Evolvable hardware techniques for on-chip automated reconfiguration of programmable devices " *Soft Comput.*, vol. 8, pp. 354-365, 2004 2004.

[16] P. Bentley, J and T. Gordon, G.W, "On Evolvable Hardware," in *Soft Computing in Industrial Electronics*, S. Ovaska and L. Sztandera, Eds. Heidelberg,Germany: Physica-Verlag, 2002.

[17] C. Darwin, *The Origin of Species by Means of Natural Selection*. New York: Random House Inc., 1859.

[18] D. Rumelhart , E, B. Widrow, and M. A. Lehr, "The basic ideas in neural networks," *Commun. ACM,* vol. 37, pp. 87-92, 1994.

[19] Y. Zhang, S. Smith, L. , and A. Tyrnell, M. , "Digital Circuit Design using Intrinsic Evolvable Hardware," in *6th NASA/DoD Workhsop on Evolution Hardware (EH 2004)*, Seattle,USA, 2004, pp. 55-62.

[20] T. Gordon and P. Bentley, "Evolving Hardware," in *Handbook of Nature-Inspired and Innovative Computing*, 2006, pp. 387-432.

[21] A. J. Hirst, "Notes on the evolution of adaptive hardware," in *Adaptive Computing in Engineering Design and Control 1996* University of Plymouth,UK, 1996.

[22] J. R. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection. . London,England: The MIT Press, 1992.

[23] D. J. Montana, "Strongly Typed Genetic Programming," *Evolutionary Computation*, vol. 3, pp. 199-230, 1995.

[24] S. Systems, "Evolutionary Strategy," P. Small, Ed. Berkshire, 2003.

[25] H. Jing-song, Z. Hao-ran, F. Qian-sheng, and W. Xu-fa, "Evolving the strategy of evolutionary strategy," vol. 24, pp. 1715-17, September 2003 2003.

[26] P. Haddow, G. Tufte, and P. V. Remortel, "Evolvable Hardware: Pumping life into dead silicon," in *On growth, Form and Computers*, S. Kumar and P. J. Bentley, Eds. London: Academic Press, 2003.

[27] G. Borriello, C. Ebeling, S. A. Hauck, and S. Burns, "The Triptych FPGA architecture," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 3, pp. 491-501, 1995.

[28] H. de Garis, "CAM-Brain the evolutionary engineering of a billion neuron artificial brain by 2001 which grows/evolves at electronic speeds inside a cellular automata machine (CAM)," in *Towards Evolvable Hardware*, 1996, pp. 76-98.

[29] N. Macias, "Ring Around the PIG: A Parallel GA with Only Local Interactions Coupled with a Self-Reconfigurable Hardware Platform to Implement an O (1) Evolutionary Cycle for Evolvable Hardware," in *Congress on Evolutionary Computation* Washington, DC, USA, 1999, p. 1075.

[30] G. Tufte and P. C. Haddow, "Prototyping a GA Pipeline for complete hardware evolution," in *Evolvable Hardware, 1999. Proceedings of the First NASA/DoD Workshop on*, Pasadena, CA, United States, 1999, pp. 18-25.

[31] M. Murakawa, S. Yoshizawa, I. Kajitani, X. Yao, N. Kajihara, M. Iwata, and T. Higuchi, "The GRD chip: genetic reconfiguration of DSPs for neural network processing," *IEEE Transactions on Computers*, vol. 48, pp. 628-639, June 1999.

[32] C. Bauer, P. Zipf, and H. Wojtkowiak, "System Design with Genetic Algorithms," in Field-Programmable Logic and Applications. The Roadmap to Reconfigurable Computing: 10th International Conference, FPL 2000, Villach, Austria, August 27-30, 2000. Proceedings, 2000, p. 250.

[33] A. Joglekar and M. Tungare, "Genetic Algorithms and their use in the design of Evolvable Hardware," in *IEEE Region 10 Conference for Student Papers, Mumbai, India.*

[34] J. E. Baker, "Reducing bias and inefficiency in the selection algorithm," Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application table of contents, pp. 14-21, 1987.

[35] H. Pohlheim, "GEATbx: Genetic and Evolutionary Algorithm Toolbox for use with MATLAB Documentation," 2005.

[36] M. Mitchell, *An introduction to genetc algorithms*. Cambdrige: MIT Press, 1998.

[37] A. Thompson, "Silicon evolution," in *Proceedings of First Annual Conference*, Stanford University,CA,USA, 1996, pp. 444-452.

[38] L. Huelsbergen, E. Rietman, and R. Slous, "Evolution of Astable Multivibrators in Silico," in *Proceedings of the Second International Conference on Evolvable Systems: From Biology to Hardware* London,UK, 1998, pp. 66-77.

[39] U. Tangen and J. S. McCaskill, "Hardware Evolution with a Massively Parallel Dynamically Reconfigurable Computer: POLYP," *Proceedings of the Second International Conference on Evolvable Systems: From Biology to Hardware*, pp. 364-371, 1998.

[40] A. Thomson, "On the automatic design of robust electronics through artificial evolution " in *Evolvable Systems: From Biology to Hardware*. vol. 1478/1998 Heidelberg: Springer Berlin 1998, pp. 13-24.

[41] S. D. Brown, "Fundamentals of Digital Logic with VHDL Design (McGraw-Hill Series in Electrical and Computer Engineering)," 2005.

[42] J. A. Lewis, "Architectural innovations for high performance in PLDs," 1991, pp. P2-2/1-4.

[43] S. Brown and J. Rose, "FPGA and CPLD architectures: a tutorial," *Design & Test of Computers, IEEE,* vol. 13, pp. 42-57, 1996.

[44] A. Corporation, "Max 7000 Programmable Logic Device Family," in *Data Sheet*, ver. 6.7 ed: Altera Corporation, 2005, pp. 1-66.

[45] J. Rose, A. El Gamal, and A. Sangiovanni-Vincentelli, "Architecture of field-programmable gate arrays," *Proceedings of the IEEE,* vol. 81, pp. 1013-1029, 1993.

[46] G. Hollingworth, S. Smith, and A. Tyrrell, "Safe intrinsic evolution of Virtex devices," *Evolvable Hardware, 2000. Proceedings. The Second NASA/DoD Workshop on,* pp. 195-202, 2000.

[47] Y. Thoma and E. Sanchez, "A Reconfigurable Chip for Evolvable Hardware," in *Genetic and Evolutionary Computation – GECCO 2004*. vol. Volume 3102/2004: Springer Berlin 2004, pp. 816-827.

[48] X. Corporation, "XC6200 Field Programmable Gate Arrays," in *Data Sheet*, Ver 1.10 ed: Xilinx Corporation, 1997, pp. 1-73.

[49] Channakeshav, K. Zhou, R. Kraft, and J. F. McDonald, "Gigahertz FPGAs with new power saving techniques and decoding logic," in *NASA/DoD Conference on Evolvable Hardware* NY, USA, 2002, pp. 60-62.

[50] P. Layzell, "A new research tool for intrinsic hardware evolution," in *Evolvable Systems: From Biology to Hardware*: Springer, 1998, pp. 47-56.

[51] N. Macias, "The PIG Paradigm: The Design and Use of a Massively Parallel Fine Grained Self-Reconfigurable Infinitely Scalable Architecture," *Proceedings of The First NASA/DOD Workshop on Evolvable Hardware (EH'99),* 1999.

[52] A. R. M. Limited, "AMBA Specification (Rev 2.0)," *ARM Limited,* 1999.

[53] Y. Thoma, E. Sanchez, J. M. M. Arostegui, and G. Tempesti, "A Dynamic Routing Algorithm for a Bio-inspired Reconfigurable Circuit," in *Submitted to the 13th International Conference on Field Programmable Logic (FPL'03)*, 2003, pp. 681-690.

[54] S. A. Guccione, D. Levi, and P. Sundararajan, "JBits: A Java-based Interface for Reconfigurable Computing," *2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD),* vol. 261, 1999.

[55] M. Iwata, I. Kajitani, Y. Liu, N. Kajihara, and T. Higuchi, "Implementation of a Gate-Level Evolvable Hardware Chip," in Evolvable Systems: From Biology to Hardware: 4th International Conference, ICES 2001 Tokyo, Japan, October 3-5, 2001, Proceedings, 2001, pp. 38-49.

[56] I. Kajitani, T. Hoshino, D. Nishikawa, H. Yokoi, S. Nakaya, T. Yamauchi, T. Inuo, N. Kajihara, M. Iwata, D. Keymeulen, and T. Higuchi, "A Gate-Level EHW Chip: Implementing GA Operations and Reconfigurable Hardware on a Single LSI," in *Evolvable Systems: From Biology to Hardware*. vol. 1478/1998: Springer Berlin / Heidelberg, 1998, pp. 1-12.

[57] D. Thierens and D. Goldberg, "Elitist recombination: an integrated selection recombination GA," Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on, pp. 508-512, 1994.

[58] D. Debnath and T. Sasao, "Minimization of AND-OR-EXOR Three-Level Networks with AND Gate Sharing," *IEICE TRANSACTIONS on Information and Systems,* vol. 80, pp. 1001-1008, 1997.

[59] P. D. Hortensius, R. D. McLeod, and H. C. Card, "Parallel random number generation for VLSI systems using cellular automata," *Transactions on Computers,* vol. 38, pp. 1466-1473, 1989.

[60] P. C. Haddow and G. Tufte, "An evolvable hardware FPGA for adaptive hardware," 2000, pp. 553-560 vol.1.

[61] W. Encyclopedia, "Multiplexer," Wikimedia Foundation, 2007.

[62] W. Encyclopedia, "Shift register," Wikimedia Foundation, Inc., 2006.

[63] A. Corporation, "Quartus II Introduction Using VHDL Design." vol. 2006: Altera, 2006, p. OnlineTutorial.

[64] W. Encyclopedia, "Crossover," Answers.com, 2007.