UiO : Department of Informatics
University of Oslo

Hardware acceleration of an evolutionary algorithm on Xilinx Zynq-7000

Investigating the acceleration potential of a Cartesian Genetic Programming algorithm using Xilinx Design Tools and implemented on Zynq-7000 AP SoC EHW platform

Hussain Javaid Butt

Master's Thesis Autumn 2015

Hardware acceleration of an evolutionary algorithm on Xilinx Zynq-7000

Hussain Javaid Butt

10th August 2015

Abstract

The primary goals of this thesis was to design and implement a hardware friendly Zynq-based CGP algorithm and investigate the acceleration potential. It was made several attempts to find out if it was possible to increase the speed of the CGP algorithm by implementing single part of algorithm as hardware component. The Zynq-platform is a unique blend of two technologies, which includes a Dual ARM® Coretex-A9 Processer System and 7-series Programmable Logic. This means that Zynq is able to take advantage of software programming and in addition configure programmable hardware both at the same time.

Contents

List of Figures

List of Tables

List of Acronyms

AMBA Advanced Microcontroller Bus Architecture

AP All Programmable

APU Application Processor Unit

ARM Advanced RISC Machines

ASIC Application Specific Integrated Circuit

AXI Advanced eXtensible Interface

CGP Cartesian Genetic Programming

CLG Configurable Logic Block

EA Evolutionary Algorithms

EP Evolutionary Programing

ES Evolutionary Strategies

EHW Evolvable Hardware

FSM Finite-State Machine

FPGA Field-Programmable Gate Array

GA Genetic Algorithms

GP Genetic Programming

HDL Hardware Description Language

HLS High-Level Synthesis

IP Intellectual property

IDE Integrated Design Environment

MIO Multiplexed I/O

PS Processing System

PL Programmable Logic

RTL Register-Transfer Level

SoC System on a Chip

SDK Software Development Kit

TCL Tool Command Language

VHSIC Very-High-Speed Integrated Circuit

VHDL VHSIC Hardware Description Language

VRC Virtual Reconfigurable Circuit

Acknowledgement

I would like to sincerely thank my supervisor, Associate professor Kyrre Harald Glette, for the continuous guidance and support throughout the thesis work. I especially appreciate your honest and motivating feedback. I would also like to thank Professor Jim Tørresen for his support and motivating words. Special thanks to senior engineer Yngve Hafting for always being there to help with equipment and software tools. This thesis would have been very difficult without his technical support.

I also want to thank my fellow students at the lab for creating a very good study and social environment.

I am very grateful to have such supportive family. I appreciate their patience and endless support. Without my family I would never completed this thesis.

Chapter 1

Introduction

This chapter introduces the thesis and starts with why research and work inspired by Natural Evolution is important and interesting. Next up is the inspiration and motivation behind my thesis, which includes successful examples of earlier work within this field. Specific objectives of this thesis and a chapter overview is presented at the end of this chapter.

1.1 Natural evolution to hardware development

In the last decades the complexity of electronic system have increased drastically along with circuit density[31]. Generally we can observe that the probability of failure and errors in electronic systems increases with higher density and complexity. An important fact to consider is that majority of biological systems are more complex compared to existing electronic systems. In spite of high complexity, most biological systems seems to be more tolerant to errors and failure. This thesis starts off with some questions that have emerged out of curiosity for this field:

What makes biological systems more tolerant to errors and failure?

The answer lies in the way biological systems are developed and designed. Every product of natural evolution has evolved through adaptation to its specific environment. This evolutionary cycle has been repeated over millions of generations and resulted in candidates who are best suited to their environment. Some biological systems are also able to rapidly change their behaviour and function according to changes in the environment, a good example is the human immune system.

Is it possible to mimic the concept of natural evolution to solve computer problems?

This question was answered by some researchers as early as the 1940s and 1950s[5], when the first ideas of *Evolutionary Computing* was created. It was not until the 1960s that ideas of evolutionary computing had its real breakthrough. Different *Evolutionary Algorithms (EAs)* was defined. Since then, the development have proceeded fast forward in this field. New technology have been created, providing new opportunities and challenges.

Is it possible to use inspiration form natural evolution to design or evolve digital hardware?

The answer to this question is yes! By using *Evolvable Hardware(EHW)* as a design method, new and original solutions can be developed[7]. We may find new and better solutions that no one ever thought about. The most important lead is the ability to adapt to changes in environment by improving behaviour in real time.

This thesis is an attempt to bring in recent technologies to evolve nature-inspired architecture and investigate the potential for improvement.

1.2 Inspiration and motivation

The field of EHW combined with evolutionary methods is close to a great breakthrough. This statement is justified by some examples mentioned later in this section. As a result, we can expect higher circuit performance and better functionality. We have already several successful examples of work that has been done within this field with remarkable results. The following sections address some notable and relevant work that have been inspiration for this thesis.

A god example to start with is ***An evolvable combinational unit for FPGAs***[32] created by Sekanina and Friedl at Brno University of Technology. The combinational unit consists of a *Virtual Reconfigurable Circuit (VRC)* and EA, both described in VHDL (Figure 1.1). The VRC is a hardware implementation of a significantly efficient and successful EA approach called **Cartesian Genetic Programming (CGP)**. The most surprising fact is that the unit was able to evolve the required function automatically and autonomously only by interacting with the environment. They manage to successfully evolve circuits, such as multiplexers, adders and encoders directly into the FPGA.

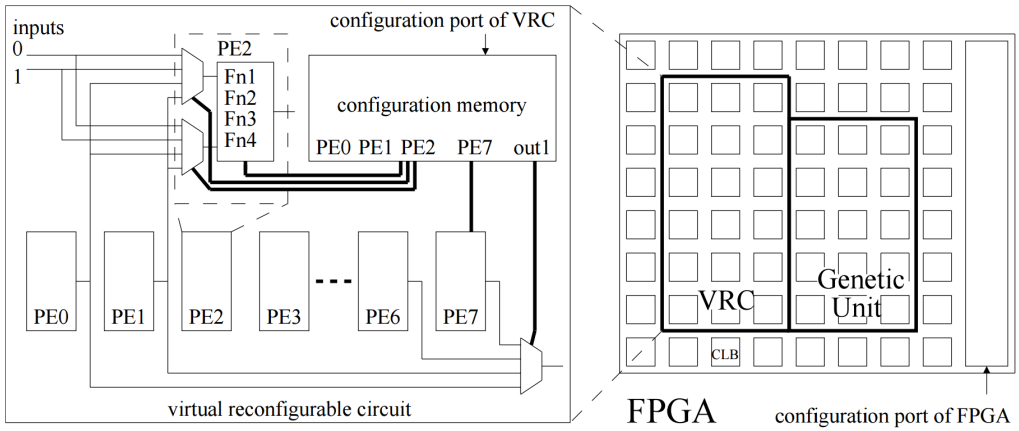


Figure 1.1: Hardware implementation of an evolvable combinational unit for FPGAs[32]

Another similar example from Sekanina is ***An Evolvable Image Filter***[25] which was an experimental evaluation of a complete hardware implementation in FPGA. This system managed to evolve image filters in a few second according to given input images. Also here a CGP approach and VRC was used to solve the problem. The combinational unit[32] and image filter[25] proves that hardware is able to find solutions with inspiration form evolutionary methods.

Ph.D thesis ***Design and Implementation of Scalable Online Evolvable Hardware Pattern Recognition System***[22] by Kyrre Glette focuses on autonomous run-time adaptive digital EHW systems for solving large real-world problems. Glette emphasize that the main challenges that prevents achieving such a system is a combination of two things. The first challenge is the lack in EAs scalability and the second challenge is designing adaptive hardware architecture for evolution. To address the scalability challenge, data-buses and high-level building blocks has been tested and combined with incremental evolution. Based on these features, specialized high-speed classifier architecture has been developed for online evolution. This architecture is achieved by use of an on-chip processor. His work has resulted in a flexible EHW based on-chip system capable of classifying more advanced problems with higher accuracy and speed.

New and interesting development takes place in FPGA architecture field. FPGA is regarded as a good platform for digital EHW systems. According to Dobai and Sekanina the Xilinx Zynq-7000 All Programmable System on Chip (Figure 1.2) platform has the potential to become the next revolutionary step in evolvable hardware design[4]. The paper analyses this platform from an evolvable hardware designers perspective and gives useful results for developing real-world evolvable systems with the Zynq-7000 platform. Details about the Zynq-7000 platform and why it is suitable for EHW are discussed in Section 3.1.1.

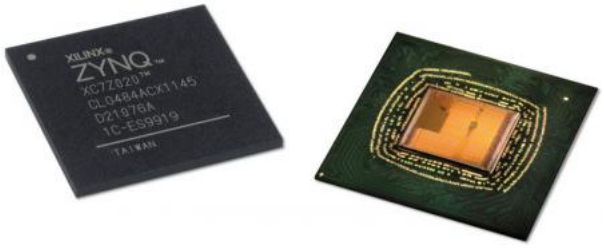


Figure 1.2: Zynq-7000 All Programmable System on Chip[23]

1.3 Objectives of this thesis

The primary goal of this thesis it to design and implement a Zynq-based evolutionary algorith and investigate the acceleration potential by experimenting on different implementation levels. The Zynq-platform is a unique blend of two technologies, which includes a *Dual ARM® Coretex-A9 Processer System* and *7-series Programmable Logic*. This means that Zynq is able to take advantage of software programming and in addition configure programmable hardware both at the same time. I want to find out if it is possible to increase the speed of the evolutionary algorithm by implementing single part or parts of the evolutionary operations as hardware components and by using different implementation of EAs.

To achieve the main goal, the following tasks must be completed:

- Consider the different types of EAs to find which one best suit Evolvable Hardware.
- Implement the selected EA solution in software. Verify functionality and measure performance of the implemented software solution.
- Profile the software solution and consider potential bottlenecks in the algorithm. Evaluate whether the bottlenecks can be eliminated by implementing them as hardware modules.
- Design and implement hardware modules. Verify functionality and

measure performance of the implemented partial hardware/software solution.

- Design hardware-friendly solution and verify functionality and measure performance.
- Compare the result measured on different implementation levels.

1.4 Chapter overview

Chapter 2 Background This presents an overview of the basic theory and hardware relevant to this thesis. It begins with a brief introduction to Evolutionary Computing and its origin. Followed by a detailed overview of Evolutionary Algorithms(EA), which covers characteristics, expressions and methodology. The Cartesian Genetic Programming(CGP) approach is used throughout the thesis and presented in detail. The selection of this approach is justified. Hardware is needed to realize the theoretical background in the real world. FPGAs og microprocessors are presented. These technologies are fundamental for creating Evolvable Hardware(EHW), which is presented in the very end of this chapter.

Chapter 3 Development tools: Hardware and Software This chapter covers the development tools that was used in this thesis. The tools was mainly supplied by a well known technology company named Xilinx Inc. Xilinx is known for programmable logic devices. Over the past few years Xilinx have invested heavily in specialized software for hardware development as well. Both hardware and software tools are briefly introduced. The purpose of this chapter is to give the reader a basic understanding of the tools.

Chapter 4 Implementation This chapter gives an overview of the work done in this thesis. It start with introducing the chosen EA approach followed by a detailed overview. The overview includes all significant information the reader needs to understand the structure, parameters and the purpose of the algorithm. Next up is the development of the embedded Zynq design, which includes both software and hardware implementation. The evolutionary algorithm was implemented, verified and profiled.

Chapter 5 Experiments and Result

Chapter 6 Conclusion and proposals for further work

Chapter 2

Background

The objective of this chapter is to give the reader a foundation for understanding the field of Evolvable Hardware (EHW). This chapter presents an overview of the basic theory and hardware relevant for this thesis. It begins with a brief introduction to Evolutionary Computing and its origin. Followed by a detailed overview of Evolutionary Algorithms(EA). The EA overview covers characteristics, expressions and methodology. After the theoretical part the focus moves on hardware technologies. FPGAs and microprocessors basic structure and usage are presented. These technologies are fundamental for creating Evolvable Hardware(EHW), which is finally presented in the very end of this chapter.

2.1 Evolutionary Computing

As mentioned in the introduction the first concepts about evolutionary computing occurred in late 1940's and 1950's. Evolutionary computing mimics the concept of natural evolution to solve computer problems. In the 1950's different implementations of the evolutionary computing were developed at the same time but in different places[5, 6, 8]. In the United States the researchers Fogel, Owens and Walsh developed a version called *Evolutionary Programming*, while Holland called his version *Genetic Algorithm*. In Germany *Evolution Strategies* was defined by Rechenberg and Schwefel. The last and most important variant, at least for this thesis, was defined in the early 1990s by Koza[21] and is known as *Genetic Programming*. These four variants differ slightly from each other and have defined unique steps, characteristics and procedure. Collective term for these different implementations is *Evolutionary Algorithms*.

2.2 Evolutionary Algorithm (EA)

Evolutionary Algorithms (EAs) are inspired by natural evolution and follows Darwin's theory of evolution by natural selection[3]. Darwin discovered that evolutionary process make spices adapt to their environment. The basic idea is that a given population of individuals are placed in a closed environment, with limited resources. The competition for the resources creates the basis for natural selection where the fittest individuals survives. The variation operators, mutation and recombination, creates diversity within the population.

EAs are a set of computer algorithms mainly used in search and optimization problems. It is important to emphasize that EAs are based on a simplified model of the biological evolution. A problem (environment) is defined and solutions (individuals) are introduced to evolve. The individuals are population-based and have the ability to adapt to environment. Each individual in a population represents one solution to a problem. The population is consequently a set of possible solutions. A fitness function measures the quality of each solution. Evolutionary operators like *selection*, *reproduction*, *recombination* and *mutation* are applied to evolve new solutions and create diversity. EA turns out to find good or satisfying solutions to almost all types of optimization problems. The main reason for this is that EA evolves solutions without making any presumption about the actual environmental fitness.

2.2.1 Expressions used in EAs

Before we move forward to EAs in details it is appropriate to explain some expressions, which is frequently used in EA context. It is common to use expressions from biology. The most commonly used expressions and what they stand for is presented in Table 2.1.

Expression:	**Explanation:**
Environment	The user-defined problem
Individual	A(possible) solution to the problem
Population	Set of possible solutions to the problem
Genotype	Set of parameters, which defines the detailed(decoded) version of the individual.
Phenotype	An individuals genotype encodes its phenotype, which is the expressive version of the individual.
Gene	Set of parameters, which defines a specific part of the of the genotype
Fitness	Real value indicating the quality of an individual as a solution to the problem.
Selection	Policy for selecting one individual(parent) from the population, often the individual with highest fitness.
Recombination	Operation that merges the genotypes of two selected parents, resulting in creation of new offspring
Mutation	Operation that makes small random changes to the genotype.

Table 2.1: Overview of expressions used in Evolutionary Algorithms

2.2.2 The evolutionary cycle

It is many different variants of evolutionary algorithms, but the underlying idea is same. Figure 2.1 shows a enumerated cycle for a generalized EA and the corresponding enumerated explanation to each step is described below.

1. **Initialize population:** The cycle start with initialization of the first population. The first population is often initialized randomly, but smart initialization with problem-specific knowledge is common and may be beneficial.

2. **Evaluation of population:** Step **2** evaluates the population and is the first step in the main loop. All the individuals(solutions) in the population is evaluated and assigned a fitness score. The fitness

score indicates the quality of an individual as a solution to the problem(environment).

3. **Parent selection:** The individual(s) with highest fitness score is chosen as parent(s) in step **3**. There are also used other criteria for parent selection, but fitness based selection is the most common.

4. **Recombination and(or) mutation:** Various recombination and/or mutation operators are applied to the selected parent(s) in step **4**. This step results in offspring (new individuals) originating from the selected parent(s) from step **3**.

5. **Survivor selection:** The new individuals are evaluated in a process called survivor selection. The individuals with highest score have the opportunity to be part of the next generation. It is also common to let the selected parent(s) from step **3** and new individuals from step **4** to compete in survivor selection.

6. **Termination:** The EA loop will run until a stop criterion is met. The most common stop criterion is numbers of generation others are time and age based. Some EA terminates when no improvements are detected within a number of generations.

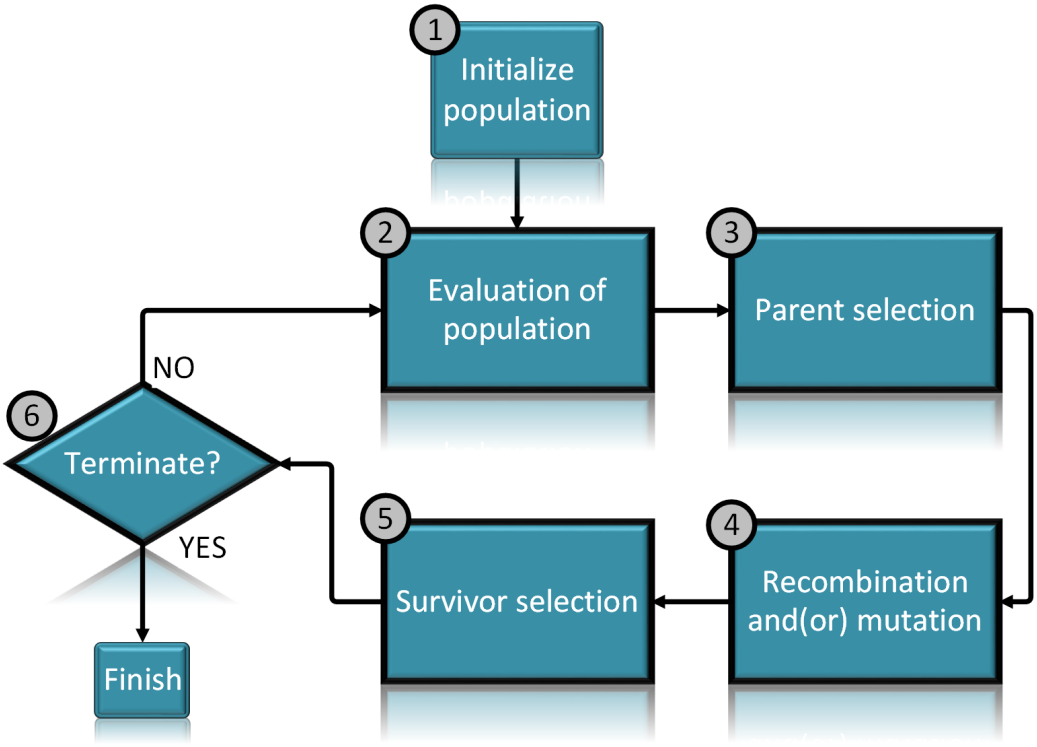


Figure 2.1: Generalized EA cycle

2.2.3 Different types of Evolutionary Algorithms

EA is divided into several subtypes. The most known are **Genetic Programming (GP) , Genetic Algorithms (GA), Evolutionary Programing (EP)** and **Evolutionary Strategies (ES)**. The differences between them are not clear and well-defined because they are based on the same idea. Table 2.2 from *Introduction to Evolutionary Computing*[5] is an excellent attempt to divide up and summarize the different types of EAs. This overview shows typical problems each of the types are suitable for. Search and optimization problems are most common. There is great variation in how genotype is represented. Choice of representation is often done based on what best suits the problem. Mutation and recombination are evolutionary operators and can be implemented in many different ways.

Component or feature	EA Dialect			
	GA	**ES**	**EP**	**GP**
Typical problems	Combinational optimization	Continuous optimization	Optimization	Modelling
Typical representation	Strings over a finite alphabet	Strings(vectors) of real numbers	Application specific often as in ES	Trees
Role of recombination	Primary variation operator	Important, but secondary	Never applied	Primary/only variation operator
Role of mutation	Secondary variation operator	Important, sometimes the only operator	The only variation operator	Secondary, sometimes not used at all
Parent selection	Random, biased by fitness	Random, uniform	Each individual creates one child	Random, biased by fitness
Survivor selection	Generational: all individuals replaced Steady-state: deterministic biased by fitness	Deterministic, biased. by fitness	Random, biased by fitness	Random, biased by fitness

Table 2.2: Summary of different types of Evolutionary Algorithms[5]

The goal of investigating the different types EA was to find a type that can be combined and integrated with hardware. The book *Introduction to Evolutionary Computing* by Eiben and Smith[5] describes and explains all EA types in detail. As mentioned before, all EA types are based on the same idea of natural evolution and share several features. The four main types were formed over time and the contents of the Table 2.2 is a simplified and generalized projection. There are many versions of the main types that are designed for specific purposes and applications. Selection of EA type best suited for this task are introduced and explained in section **4.1 EA type suited for this task**.

2.3 Field-Programmable Gate Array (FPGA)

Field-Programmable Gate Array (FPGA) is an integrated circuit organized as an array of logic blocks. Logical blocks are hardware resources, which can be configured to perform different logical functionality. Similar to how logical ports can be weird in many ways to perform different logical functionality. The configuration is usually defined by Hardware Description Language (HDL). Configuration includes the functional logic, routing between the logic blocks and definition of input/output blocks.

2.3.1 Structure

Figure 2.2 shows a overview of a typical FPGA structure.
1. Top-level view of FPGA: A two-dimensional grid (array) of Configurable Logic Blocks.(CLBs)
2. Interconnection: Around the CLBs are the routing tracks, which makes interconnection between other tracks, I/O cells and other CLBs.
3. Configurable Logic Block(CLB): The CLBs is the smallest unit of programmable logic. The content of configurable logic is defined (or achieved) through static Look-Up Tables (LUTs) and multiplexers.
4. Look-Up Tables (LUTs): A LUT contains storage cells. A storage cell can store a single logic value 1 or 0. Multiple cells combined are used to implement logic function.

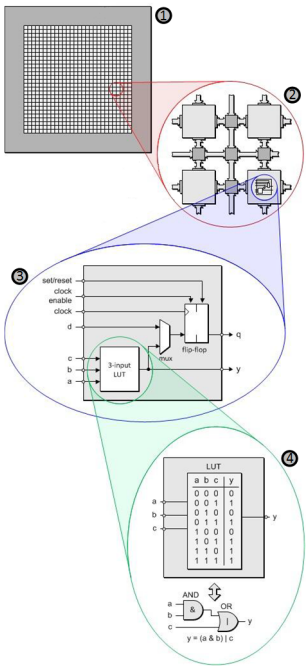


Figure 2.2: FPGA structure[20]

Fuse technology was used to configure the logical blocks in the early stages. This type of FPGAs is circuit-size friendly and fast, but the drawback is that they can only be programmed one time. Today, however, it is more common to use RAM or flash memory to store the desired configuration. The main advantages of RAM/flash based design are that FPGA can be reprogramed (reconfigured) and perform more complex configurations.

The use and number of FPGAs have increased dramatically last 20 years. The reason is FPGAs flexibility to implement circuits with different functionality and allows developers to configure it according to their requirements after manufacturing(In the "field"). The overall performance of FPGAs has increased too. Many FPGAs today have millions of logic gates, several megabit of RAM, I/O ports which can operate on 10 GHz[35]. Many advanced FPGAs also have embedded microprocessor cores onboard. Aforementioned improvements entail many advantages, but new challenges as well. Timing verification problem is one challenge in newer FPGAs, due to fast I/O and data bus speed.

2.3.2 FPGA advantages

- **Flexible:** Implement circuits with different functionality according to our requirements after manufacturing. (In the "field")
- **Reliable:** Errors in the configuration can be corrected in real-time and new configuration updates implemented, while the rest of the FGPA can continue to function.
- **Performance:** Embedded microprocessor cores onboard, millions of logic gates, several megabit of RAM and I/O speed.
- **Price/Cost:** Can use less logic because the configuration can be changed depending on the task.
- **Power:** Smaller size resulting in reduced power consumption.

2.3.3 Reconfiguration

FPGAs are, like mentioned earlier, reconfigurable. This is a great advantage compared to its competitor, Application-Specific Integrated Circuit (ASIC), which is unchangeable. ASICs are consequently far more expensive to produce in small quantity than FPGAs. Many FPGAs are capable to perform partially reconfiguration. Partial reconfiguration allows us to reconfigure certain part of the FPGA. This feature reduces the reconfiguration time dramatically. Some FPGAs are also able to execute partially run-time reconfiguration. This feature allows us to

reconfigure certain part of the FPGA in its operational state. Partially run-time reconfigurable FPGAs can be made with less logic because the configuration can be changed depending on the task. Less logic means smaller boards which reduces the manufacturing cost. Another advantage of smaller size is reduced power consumption. With partially run-time reconfiguration errors in the configuration can be corrected in real-time and new configuration updates implemented, while the rest of the FGPA can continue to function.

2.4 Microprocessor

A microprocessor, also known as the Central Processing Unit (CPU), is the brain and workhorse in all computers and numerous electronic devices[1]. A single microprocessor works sequentially and executes instructions. An instruction is an order to perform an arithmetic or a logical operation. This operation uses storage areas called registers. The instructions are usually quite simple and primitive, such as add, subtract, copy and compare two numbers. Based on this operation, microprocessor manage to execute complex operation very efficiently.

The biggest advantage of processors is flexibility, as it can be used for many different purposes. A microprocessors properties and performance vary according to which instructions that can be executed, numbers of bits in a single instruction(bandwidth) and the clock speed which determines how many instructions the processor can execute. Over the past few years, processors have become significantly faster, but in spite of this is has become much smaller and cheaper.

2.5 Evolvable Hardware(EHW)

Evolvable hardware uses evolutionary principles and refers to hardware that can change its architecture and behaviour dynamically and independently by adapting to the environment[7]. The hardware has been created according to an EA. A basic EA produces a population where each individual represents one circuit solution. These solutions are evaluated based on a fitness score. This score indicates how well a circuit solution satisfies the environmental conditions and specification. New generations are made by applying some random changes to existing circuit solutions. The new generations are, like earlier generation, evaluated. After repeating this algorithm over many generations, we will get a circuit solution with good or satisfying behaviour.

Computers can run a given algorithm to evolve circuits and represent

them by simulation. Computer simulation is the most efficient and fast way to evaluate circuits. Another option is to implement circuits physically in hardware. The last option is often implemented with the help of reconfigurable devices, such as FPGAs (Field-Programmable Gate Arrays).

EHW is an important part of this task. A specific EA have been used to create specialized circuits without manual involvement. FPGA is an important and essential tool in EHW, but in recent years there have been introduced new technology that provide new opportunities: The Zynq platform combines the flexibility of a microprocessor and performance of a FPGA. This technology is introduced in the next chapter, where it is explained why and how it can be useful in EHW development.

Chapter 3

Development tools: Hardware and Software

This chapter covers the development tools that was used in this thesis. The tools was mainly supplied by a well known technology company named Xilinx Inc. Xilinx is known for programmable logic devices. Over the past few years Xilinx have invested heavily in specialized software for hardware development as well. Both hardware and software tools are briefly introduced. The purpose of this chapter is to give the reader a basic understanding of the tools. Table 3.1 shows an overview of hardware tools and Table 3.2 shows an overview of software tools that has been used.

Software - Vivado System Edition	
Name	**Version**
Vivado High-Level Synthesis (HLS)	2015.1
Integrated Design Environment (IDE)	2015.1
Software Development Kit (SDK)	2015.1
ModelSim	Student Edition 10.3c

Table 3.1: Software - Vivado System Edition

Hardware	
Name	**Unit description**
ZedBoard	Development Board
Dual ARM® Cortex™-A9 MPCore™	Processor
Xilinx 7 Series 28nm programmable logic	FPGA

Table 3.2: Hardware

3.1 Hardware: The ZedBoard

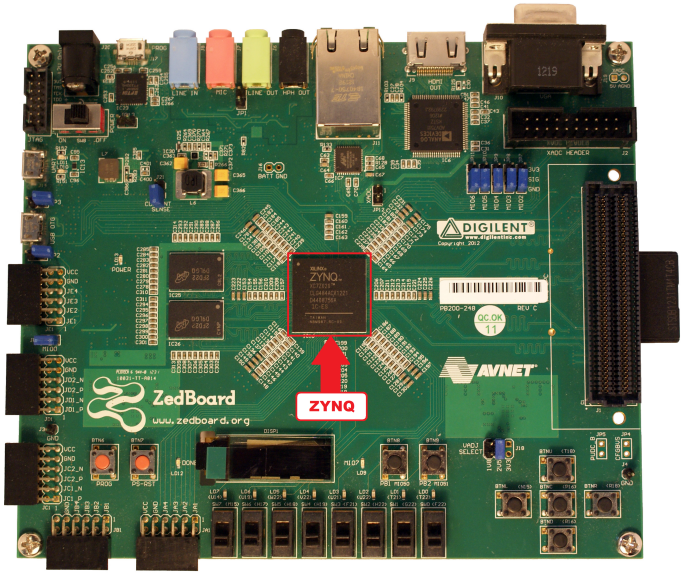


Figure 3.1: The ZedBoard

The ZedBoard(Figure 3.1) was chosen as the development platform for this thesis. The reason for this choice was mainly the Zynq device on board and its architecture. The Zynq architecture is described in the next section. It was also considered that this board offers excellent connectivity and great support online. The name **ZEDBoard** stands for **Z**ynq **E**valuation and **D**evelopment **Board** [2]. The ZedBorad is one of the most popular development and evaluation board for working with Zynq. Reasons for its popularity are low-cost, large number of peripheral interfaces and wide community-based support online. ZedBoard is made in collaboration between the three parties consisting of: *Xilinx* -the producer of Zynq device, *Avnet* -the distributer and *Digilent* -the board manufacturer. The main core of this board consists of a **Zynq(XC7Z020)** device. The board offers 256Mbit flash memory and 512MB DDR3 memory. There are two oscillator clock sources, one at 100MHz, and the other at 33.3333MHz[2].

All peripheral interfaces on the ZedBoard:

- **GPIO:** in total, 9 x LEDs, 8 x switches, 7 x push buttons
- **Audio codec and headphone**
- **Video (HDMI)**
- **Video (VGA)**
- **Organic Light Emitting Diode (OLED) display**

- **Pmod interfaces (x 5)**
- **Ethernet**
- **USB-OTG (peripherals)**
- **USB-JTAG (programming)**
- **USB-UART (communication)**
- **SD card slot**
- **FMC interface**
- **XADC header**
- **Xilinx JTAG header**

3.1.1 Zynq-7000 All Programmable SoC

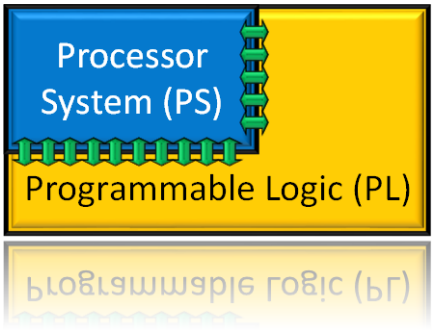


Figure 3.2: Zynq-7000 AP SoC - Simple block diagram

In 2011 Xilinx introduced *Zynq-7000 All Programmable System on Chip*. The name Zynq originates from chemical element zinc(Zn)[2]. The chemical element Zn can be mixed with other metals to form alloy with different desirable properties. Like the chemical element Zn, the Zynq hardware "element" can be applied to many different applications. Zynq is based on the Artix-7 logic fabric, with a capacity of 13,300 logic slices, 220 DSP48E1s, and 140 BlockRAMs.

Zynq is *All Programmable (AP)*, which means that Zynq is able to take advantage of software programming and in addition configure programmable hardware[9]. *System on Chip(SoC)* means that this device have all the common components of a computer(processing system) integrated into a single chip[2]. Consequently, Zynq-7000 AP SoC is not an ordinary FPGA, nor an ordinary microprocessor. It is a unique blend of two technologies, which includes a *Dual ARM® Coretex-A9 Processer System (PS)* and *7-series Programmable Logic (PL)*[19]. The PS and PL, simply

illustrated in Figure 3.2, have connections between which is known as *Advanced eXtensible Interface (AXI)*[10]. AXI is industry standard interfaces, which provides high bandwidth connection with low latency between the PS and PL[2]. The compact architecture(Figure 3.3) of PS and PL with the AXI interface allow Zynq users to combine software programmability of a processor with hardware advantages of an FPGA. Other advantages of the integrated design is lower power, lower cost and smaller form factor.

In the following paragraphs both PS and PL are described in detail. The purpose of doing so is to prepare the reader for Chapter 4 Implementation where the main solution are explained. The goal is to combine strengths of the PS and PL to create an architecture that draws the best of both world.

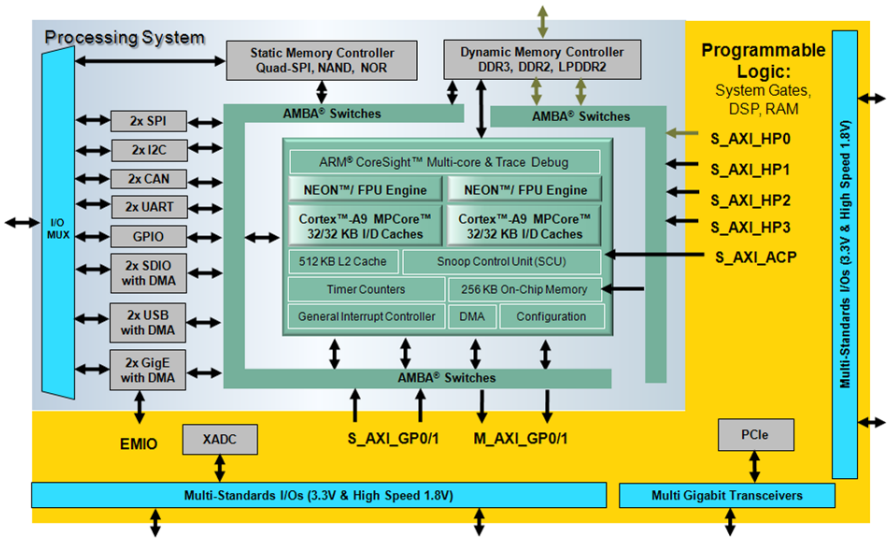


Figure 3.3: Zynq-7000 AP SoC - Detailed block diagram[18]

3.1.2 Processor System (PS) Dual-core ARM® Coretex-A9

The PS can roughly be divided into four major functional units[17]: **Application Processor Unit (APU)**, **I/O peripherals(IOP)**, **Datapath (interconnection and interfaces)** and **Memory resources**.

Application Processor Unit (APU)

APU is the core of the PS (Figure 3.4). Main component in APU is the Dual-core ARM® Coretex-A9 processors with dedicated cache and cache-controllers. The Dual-core ARM Coretex-A9 processor is a so-called "hard" processor meaning that it exists as a dedicated and optimized silicon element on the card.

The letters in **ARM** stands for **A**dvanced **R**ISC **M**achines[24]. ARM processor is based on RISC architecture which is designed to execute computer instructions at high speed. The ARM Coretex-A9 processor can operate up to 1GHz and each of the cores have dedicated 32KB level 1 cache and cache-controllers. The cores share level 2 cache of 512KB. Both cores include the NEON and VFP extensions for single instruction multiple data and double-precision floating point operations respectively[30]. The Dual-core ARM Coretex-A9 processor plays a key role in the main solution in this thesis. The processor boots up first and controls the loading of the PL. More details about the application and structure can be found in chapter 4 Implementation.

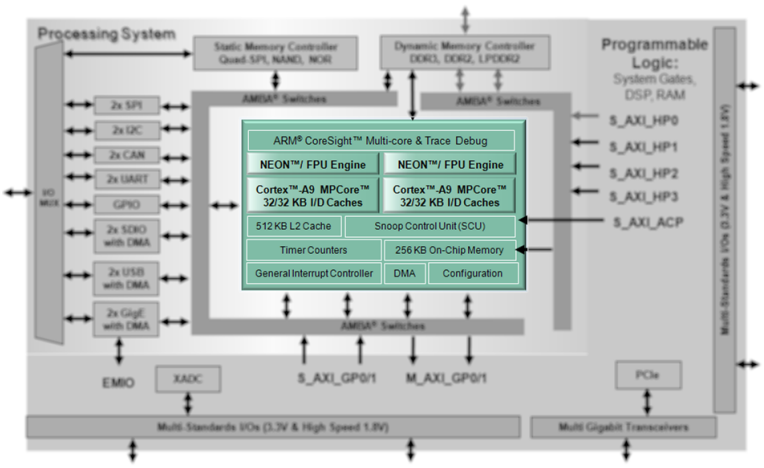


Figure 3.4: Application Processor Unit (APU)

I/O peripherals(IOP)

The most common I/O peripherals is provided by Zynq with built-in support (Figure 3.5). The supported peripherals are USB, Ethernet, SD/SDIO interface, CAN, SPI, I2C, UART and GPIO[17]. There are tow instance of each peripherals and four instance of GPIO. The Multiplexed I/O (MIO) is used to map the peripheral ports to the pins out of the device. The I/O peripherals are selected through an easy and straightforward GUI provided by tools. Xilinx Design Tools are described in section 3.4. Which peripherals and how they are used in this thesis is described in Chapter 4 Implementation.

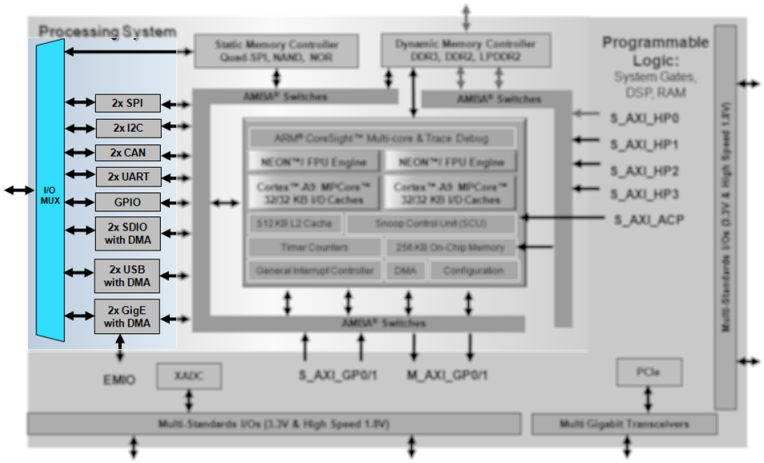


Figure 3.5: I/O peripherals(IOP)

Datapath (interconnection and interfaces)

The AMBA(Advanced Microcontroller Bus Architecture) is responsible for moving data between endpoints (Figure 3.6). The AMBA infrastructure is the basis for the unique PS-PL solution offered by of Zynq. Without the AMBA infrastructure the Zynq would be a twofold chip with an isolated processor and FPGA without any interconnection between them or the peripherals. AMBA can be divided in to two types of interconnections:

- **Internal PS data interconnections:**
    1. **Central interconnect** enables other interconnectors to communicate.
    2. **IOP master/slave interconnects** enables transfers of data to or from specific I/O peripherals
    3. **OCM interconnect** provides access to on-chip memory for APU.

- **AXI (PS-PL) data interconnections:** The AXI(Advanced eXtensible Interface) handles data between PS-PL and consists of three type. Table 3.3 shows a summary of AXI interfaces between PS and PL.

    1. **Memory interconnect** which enables data transfer between PL and PS memory resources(DDR). Uses Accelerator coherency port(**S_AXI_ACP**) for cache and High Performance ports(**S_AXI_HP**) for On Chip memory.
    2. **Master interconnect(PS master)** which enables data transfer between PS in master mode and PL in slave mode and uses General Purpose master ports(**M_AXI_GP0**)
    3. **Slave interconnect(PS slave)** which enables data transfer between PS in slave mode and PL in master mode and uses General Purpose slave ports(**S_AXI_GP0**)

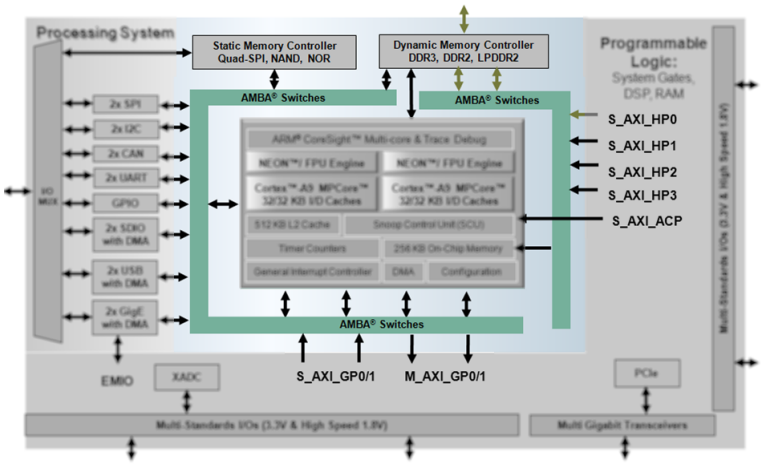


Figure 3.6: Datapath (interconnection and interfaces) and Memory resources

AXI interfaces between PS and PL			
Interface Name	**Interface Description**	**Master**	**Slave**
M_AXI_GP0-1	General Purpose	PS	PL
S_AXI_GP0-1	General Purpose	PL	PS
S_AXI_ACP	Accelerator Coherency Port	PL	PS
S_AXI_HP0-3	High Performance Ports	PL	PS

Table 3.3: AXI interfaces between PS and PL[17]

The datapath (interconnection and interfaces) are defined through GUI provided by tools, this is described in the section 3.2 Xilinx Design Tools. It should be revealed that the AXI data interconnections are a part of the main solution of the thesis.

Memory resources

It is On-Chip memory in PS which constitutes of 256KB SRAM, ROM for boot code and BRAM. PS also have access to external memory through tow integrated memory interfaces. DDRx dynamics memory controller are used for DDR memory and Flash/static controller for SRAM, QSPI and NAND/NOR Flash.

3.1.3 Programmable Logic (PL) 7-series FPGA

PL stands independently from PS and has separate on-chip power plane, memory, clock and reset management (Figure 3.7). A set of JTAG ports are provided in the PL for independent programming and debugging. PL is based on Xilinx Artix device and consists primarily of general purpose FPGA logic fabric(28 nm). Structure and content of general purpose FPGA logic fabric is explained earlier in Chapter 2 Background, section Field-Programmable Gate Array (FPGA). What distinguishes PL from ordinary FPGAs is special communication interfaces. GTX transceivers are a high-speed communication interface block which are embedded into the PL. They are able to support a number of standard interfaces including PCI Express, Serial RepidIO, SCSI and SATA[2].

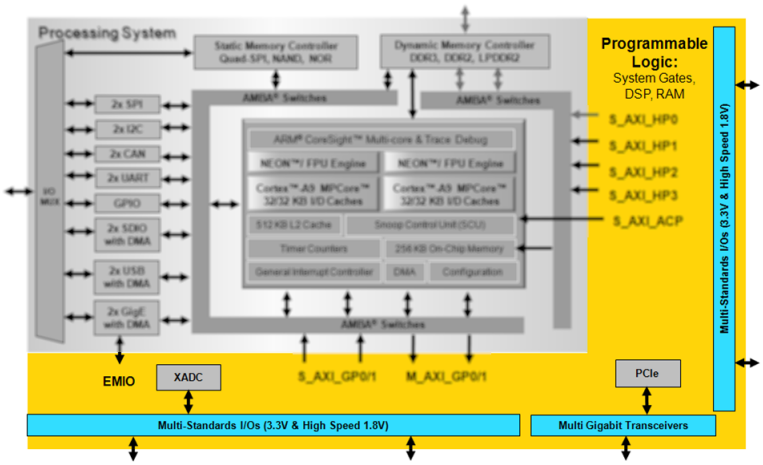


Figure 3.7: Programmable Logic (PL) 7-series FPGA

3.2 Software: Xilinx Design Tools

Designing hardware and implementing software are both challenging tasks. However connecting the two parts together so they function as one system is even more challenging and complicated. Xilinx has developed tools (Figure 3.8) to simplify and streamline the hole process. This section addresses the various software tools that are used in this thesis and explains their purpose and function. Chapter 4 Implementation addresses the detail about what was done in each of the tools.

Figure 3.8: Xilinx Design Tools

3.2.1 Vivado Design Suite System Edition

Vivado Design Suite is a collection of development tools provided by Xilinx. The Vivado collection consists of several specialized tools and are designed to address various tasks at many stages of embedded development. The *System Edition* version was used in this thesis and an overview of the functions are listed in Table 3.4.

Vivado Design Suite System Edition	
Pillars of Productivity	**Features**
Implementation	Synthesis and Place and Route Partial Reconfiguration
Verification	Vivado Simulator Vivado Device Programmer Vivado Logic Analyzer Vivado Serial I/O Analyzer Debug IP (ILA/VIO/IBERT)
Integration	Vivado High-Level Synthesis System Generator for DSP

Table 3.4: Vivado Design Suite System Edition[14]

3.2.2 Vivado Integrated Design Environment (IDE)

Vivado Integrated Design Environment provides a range of powerful features and have an intuitive Graphical User Interface (GUI). Everything that is done by using the GUI, can also be done using Tool Command Language (TCL) format where commands are entered directly into a TCL terminal or by running TCL script from file. In this thesis Vivado IDE was used to synthesize/compile the design, examine RTL diagrams, timing analysis and design simulation. In other words: all the HW structure for ZYNQ were defined and built through the Vivado IDE.

Most important features supported by Vivado IDE[15]:

- **Register-transfer level (RTL):** Design in VHDL, Verilog, and System-Verilog. Used to describe hardware design.
- **Intellectual property (IP) Integrator:** Integration for predefined Xilinx Targeted Design Platforms building blocks[12].
- **Vivado Simulator:** Behavioral simulation.
- **Vivado implementation for place and route:** Pin- and device-planning and visualisation facilities.
- **Vivado Synthesis:** Synthesis tool for 7 series and subsequent devices.
- **Vivado Power Analyser:** For evaluating the power consumption of designs operating on a target device.
- **Vivado serial I/O and logic analyser** For debugging.
- **Static timing analysis and Bitstream generation**

3.2.3 Xilinx Software Development Kit (SDK)

Xilinx Software Development Kit (SDK) is a complete embedded software development environment. Xilinx SDK is based on standard Eclipse IDE which provides a well known intuitive GUI. Unlike ordinary Eclipse IDE, Xilinx SDK is highly specialized embedded software development environment for Xilinx embedded processors including Zynq.

Most important features supported by Xilinx SDK[13]:

- C/C++ code editor and compilation environment with error navigation
- Project management and archiving

- Source-level debugging and profiling of embedded targets
- Reference software applications
- Xilinx Microprocessor Debugger (XMD), used to communicate with Xilinx embedded processors using JTAG.
- SoC programmer, used to program the Xilinx SoC with bitstream.
- Flash programmer and First Stage Boot Loader(FPBL) generator, used for automatically bootloading embedded software applications from Flash.
- Linker script generator for mapping application image across the hardware memory space.

Xilinx SDK supports bare-metal and operating system-based applications. GCC compilers and GNU GDB Debugger are included and preconfigured for bare-metal application. Xilinx SDK provides directly interface to Vivado embedded hardware design environment. Other advantages of using Xilinx SDK are embedded development plug-ins, Xilinx-specific tools and source code libraries. Xilinx SDK is a complete package making embedded software development easier and more seamless than previous development tools.

Another key feature of the Xilinx SDK is profiling. Profiling is a form of program analysis that is used to aid the optimisation of a software application[2]. Profiling gathers information about memory usage, execution time of function calls, frequency of function calls and instruction usage. Profiling results are used to identify bottlenecks in the code execution and is an important tool to accelerate an algorithm on Zynq.

3.2.4 Vivado High-Level Synthesis (HLS)

In previous paragraphs, it was emphasized that Vivado IDE focuses on hardware creation and Xilinx SDK focuses on software development. Vivado High-Level can be placed between the Vivado IDE and Xilinx SDK. Vivado HLS transforms C, C++ or System C designs into RTL implementations. Vivado HLS design flow is displayed in Figure 3.9. The RTL implantation can be synthesized and implemented directly into Xilinx AP devices without the need to manually create RTL[16]. In other words, HLS accelerates IP creation process. HLS performs interface level and functionality level design analysis. The interface analysis addresses top-level connections of the design. The functionality analysis addresses the algorithms that is implemented.

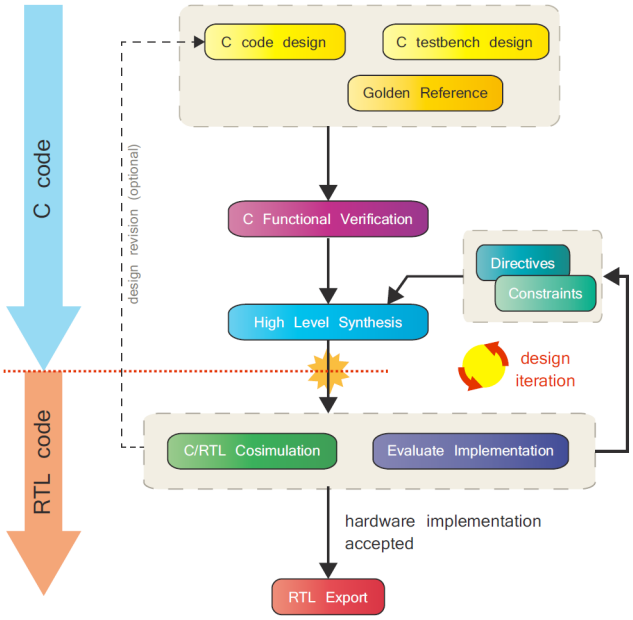


Figure 3.9: Vivado HLS Design Flow[2]

3.2.5 ModelSim

ModelSim is a powerful simulation tool provided by Mentor Graphics. The hardware languages VHDL, Verilog and SystemC are compatible languages. MoselSim is highly flexible and can be used to simulate virtually any FPGA design. The user have many different options to analyse and interpret simulation results. Simulation can either be run through Graphical User Interface(GUI) or run automatically using commands/scripts.

Chapter 4

Implementation

This chapter gives an overview of the work done in this thesis. It start with introducing the chosen EA approach followed by a detailed overview. The overview includes all significant information the reader needs to understand the structure, parameters and the purpose of the algorithm. Next up is the development of the embedded Zynq design, which includes both software and hardware implementation. The evolutionary algorithm was implemented, verified and profiled.

4.1 EA type suited for this task

The goal of investigating the different EA types in the Background chapter was to find a EA that can be combined and integrated with hardware. A specialized version of GP called Cartesian Genetic Programming(CGP), was the result of this investigation. It should be mentioned that CGP also was recommended by supervisor. CGP is a highly efficient and flexible form of GP. CGP was originally developed for evolving digital circuits[28] and earlier work[25, 32] could confirm that this approach was the right choice.

The software implementation og the CGP algorithm is added to the Appendix. The different part of CGP algorithm presented is the main focus in this section. The CGP approach is used throughout the thesis because it provides some special benefits in the development of hardware architectures. The last statement will be justified and presented clearly in the following sections.

4.2 Cartesian Genetic Programming(CGP)

Cartesian Genetic Programming (CGP) is a form of Genetic Programming (GP) proposed in 2000 by Julian F. Miller[28]. Miller originally invented CGP as a method for the purpose of evolving digital circuits. The candidate solutions are represented in the form of directed acyclic graphs[29]. These graphs are two-dimensional grid of computational nodes. The simple structure of two-dimensional graphs is versatile and can represent many computational structures including mathematical equations, computer programs, natural networks and digital circuits[33]. Since CGP was originally developed to evolve digital circuits, one can assume that the two-dimensional graphs was inspired by classic FPGA block structure. Figure 4.1 is an attempt to illustrate the similarities between (1) a two-dimensional CGP graph, (2) a digital circuit and (3) FPGA block structure. The basic idea is that a given instance of CGP graph can be implemented as an reconfigurable circuit in the FPGA.

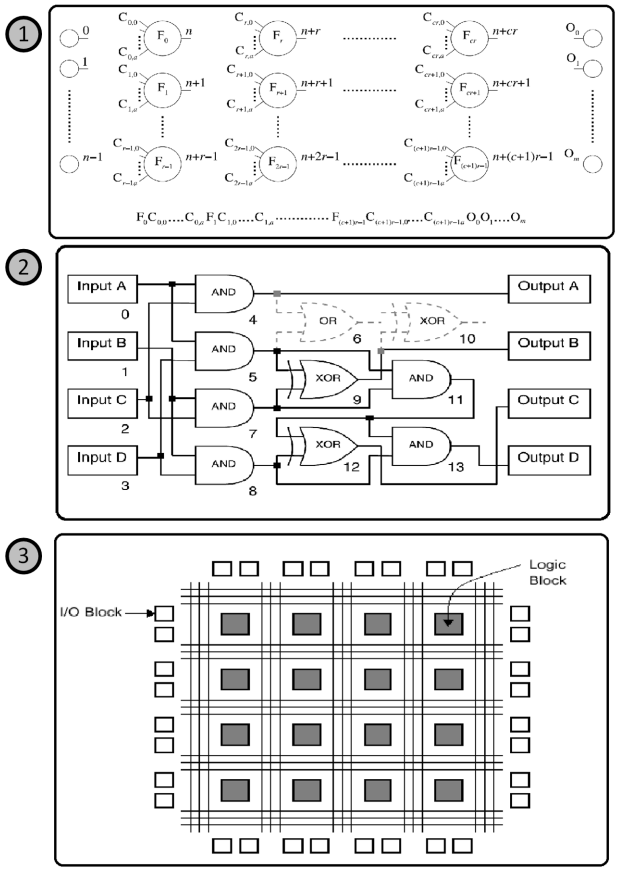


Figure 4.1: 1) A two-dimensional CGP graph. 2) Phenotype for a digital circuit evolved by CGP. 3) Generalized FPGA block structure.

All the parameters in the implemented CGP algorithm was selected to fit the specific task/problem and was continuously adjusted during the development phase. These parameters form the basis for the two-dimensional graph and varying the different parameters results in different kinds of graph topologies (Figure 4.2).

These parameters are *number of columns*(N_c), the *number of rows*(N_r) and *level-back*(l). The two-dimensional grid of computational nodes consists of $N_c * N_r$ nodes(L_n). Node inputs are connected to either the output of a node in a previous column or the top-level program(graph) inputs. These restrictions in node connection do not allow feedback and results in an acyclic graph form. The program inputs are given the addresses form 0 to N minus 1, where N is the total number of graph inputs. Outputs from nodes are given addresses in increasing order along with the columns, starting from n_i to $n_i + L_n - 1$, where L_n is total number of nodes in the graph. Each node can perform one predefined primitive function F. The level-back parameter l controls the connectivity of the nodes. Level-back regulates which columns a node can get its inputs from. In the scenario where $l = 1$, a node can only get its inputs from a node in the first left column. With $l = 2$ a node can get inputs from nodes in the first left and second left column. l can be set to number of columns(N_c) if the user wants to allow nodes to connect any nodes on their left. Varying the different parameters; number of columns(N_c), the number of rows(N_r) and level-back(l) results in different kinds of graph topologies.

Parameters in the implemented CGP algorithm:

```
// CGP parameters:
  int inputs;
  int outputs;
  int cols;
  int rows;
  int lback;
  int nodeinputs;
  int nodeoutputs;
  int nodefuncs;
```

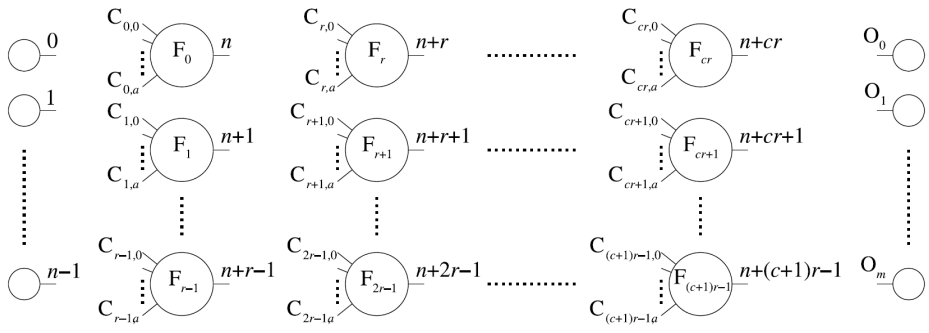


Figure 4.2: Two-dimensional grid of computational nodes

4.2.1 Genotype and Phenotype

The genotype consists of all information about CGP solutions. The genes that make up the genotype are bit-values (Figure 4.3). We encode the genotype to acquire the phenotype, which is the actual representation of the CGP solution. During the decoding process, of genotype to phenotype, some genes and their corresponding nodes are ignored. They are ignored because the node outputs are not used in any calculation of output data. These nodes and their genes are referred as non-coding and will not appear in the phenotype. This means that CGP solutions are carrier of unknown information (properties) and characteristics. This phenomena is often referred as *neutrality* because the non-coding nodes have a natural effect on the fitness[28]. The influence of neutrality in CPG have been investigated in detail by Julian F. Miller, the inventor of CGP[27, 28, 34]. The results of these experiments have confirmed that neutrality is extremely beneficial to the efficiency of the evolutionary process. In one experiment Miller showed that percentage of inactive nodes could be as high as 95%[27]. In an other experiment Miller is able to clearly illustrate the importance of neutrality[34]. The results from this experiment is shown in Figure 4.4. In the first set of runs(◇) neutral drift was allowed and in the other set of runs(+) neutral drift was not allowed. The result of the fist case was 27 successfully evolved solutions. Without neutral drift no solutions was able to meet the minimum condition of a successful solution.

```
1 00000000000000000010000000100000001000000010000000100000001000000010000000100000001000000010000000
2 00000100000000000001010000011010000000100000101100001000011101010001100000001000000011100001
3 00010100001000000010000000100100000000010010100000110000101000010111000011000000101000100 0
4 00100100011000010011000000100000011000100010110010100000000000100000000000100001000011100 0
5 01000010001000001011001000001001010100000010000000110100001101000110001000101000001010000 0
```

Figure 4.3: Example of a population with genome(encoded). The first(1) solution is a parent and four(2-5) children.

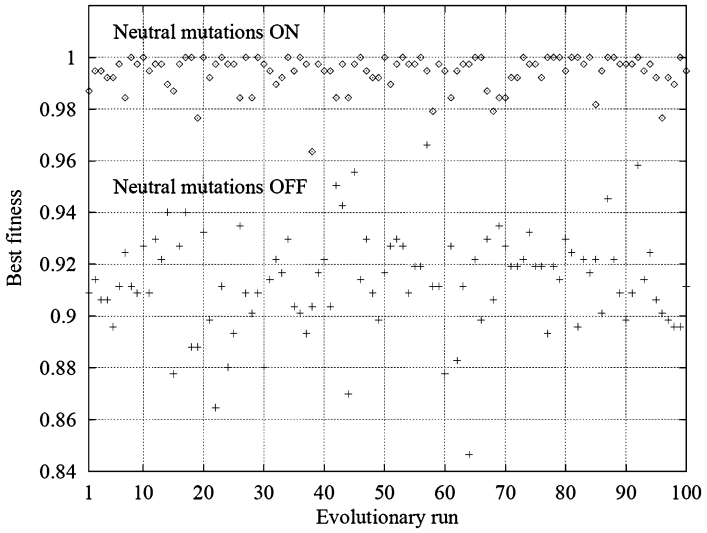


Figure 4.4: Experiment that illustrates the importance of neutrality[34]

4.2.2 The genome: Structure and constraints

The genes that make up the genotype are integers of fixed structure and length. The genotype is made up by ***connection genes***, ***function genes*** and ***output genes***[26].

- ***The function genes(F)*** represent integer addresses of functions(F) in a look up table. Programmable nodes perform the corresponding function(F) on its input data.
- ***The connection genes(C)*** represent integer addresses which are used to describe the interconnection between all the programmable nodes.
- ***The output genes(O)*** are addresses of nodes where the program output is taken from. Output genes are added to the end of the genotype.

$$\mathrm{F}_0\mathrm{C}_{0,0}\ldots\mathrm{C}_{0,a}\,\mathrm{F}_1\mathrm{C}_{1,0}\ldots\mathrm{C}_{1,a}\ldots\ldots\ldots\ldots\,\mathrm{F}_{(c+1)r-1}\mathrm{C}_{(c+1)r-1,0}\ldots\mathrm{C}_{(c+1)r-1a}\,\mathrm{O}_0\mathrm{O}_1\ldots\mathrm{O}_m$$

Figure 4.5: General CGP genome structure

The genotype representation is highly constrained and these constraints must be obeyed under genome initialization and mutation[26]. The function genes(F) must represent valid look-up table addresses. Function look-up table contains n_f numbers of primitive functions defined by the user. Consequently F must obey:

$$0 \leq F \leq n_r \tag{4.1}$$

Like function genes(F), connection gene(C) must also follow some restrictions. If we have a node in a column j and j is grater or equal to the user defined level-back parameter l, then C values must obey: (If $j \geq l$)

$$n_i + (j-1)n_r \leq C \leq n_i + jn_r, \tag{4.2}$$

where n_i is the number of program input and n_r is number of rows.

If column number, j, is smaller than level-back parameter l, the C values must obey: (If $j < l$)

$$0 \leq C \leq n_i + jn_r \tag{4.3}$$

Finally, the output genes O which can connect to any node or input. O can therefore have a value from 0 to number of inputs(n_i) plus number of nodes in the graph(L_n):

$$0 \leq O \leq n_i + L_n \tag{4.4}$$

A single Functional Unit with function gene, connection gene and output gene:

```
//  FunctionalUnit:
//  Genom:['F1',   'F2', 'C']
//  Bit:['[0-5' ,'6' , '7-14']
/*
  ____________________________________________________________
  |  _________________          _________________________    |
X-|-| P = X if F1 = 0 \___P___| if F2 = 0 then (P >= C) |__|__t(1)
Y-|-|_P_=_Y if F1_=_1_/        | if F2_=_1 then_(P_<=_C)_|  |  f(0)
  |       |                        |              |       |
  |      F1                       F2             C       |
  |_______|____________________________|_____________|______|
          F1                       F2          C = C1C2C3
*//////////////////////////////////////////////////////////
 struct FU {
    int F1[F1_LENGTH];
    int F2;
    int C[CONSTANT_LENGTH];
  };
```

4.2.3 The phenotype: Representation of a digital circuit

As mentioned in the introduction CGP can represent many different kinds of computational structures. CGP was originally developed to evolve digital circuits. The genome, introduced in the previous section, can easily be encoded to a digital circuit. The most distinct way to illustrate the encoding from genotype to phenotype is by an example[26]. In Figure 4.7 we have the evolved genome of a two-bit multiplier and Figure 4.8 represents the corresponding phenotype. Figure 4.6 is a help figure to explain and illustrate the order of genomic elements: *connection genes(C)*, *function genes(F)* and *output genes(O)*.

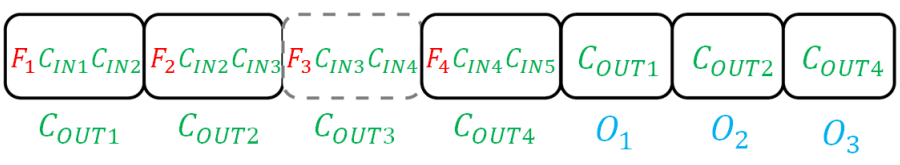


Figure 4.6: CGP genome with the different genes

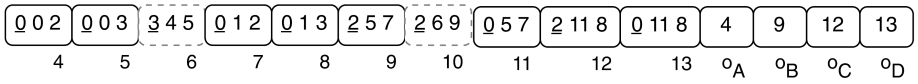


Figure 4.7: CGP genotype represents two-bit multiplier

The user-defined parameter in this example is set to ten columns ($n_c = 10$), one row ($n_r = 1$) and the level-back parameter is set to ten ($l = 10$). The latter allows nodes to connect to any other node on their left. It is defined four logical functions and their corresponding addresses(F) in a look-up Table 4.1. A two-bit multiplier multiplies two two-bit numbers together and results in a four-bit number, which means that the CGP graph requires four inputs and four outputs. In the evolved genome we have some nodes that are marked with gray dotted line (Figure 4.7). These nodes are referred as non-coding nodes because the node outputs are not used in any calculation of circuit output. During the encoding process these nodes are ignored, but in this example included with gray dotted lines.

Function gene(F):	Logical function:
0	AND
1	NAND
2	XOR
3	OR

Table 4.1: Function lookup table for CGP example

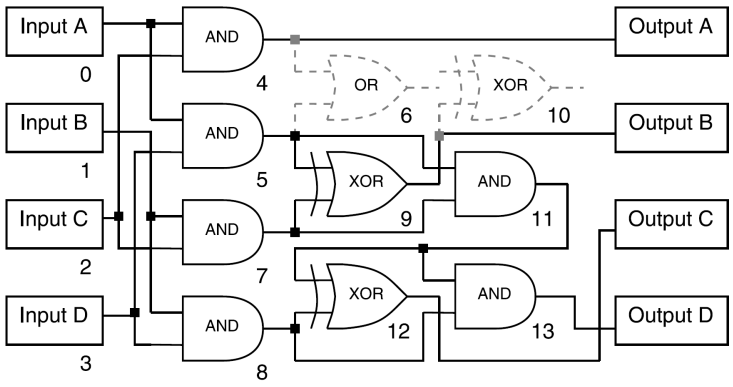


Figure 4.8: CGP phenotype represents two-bit multiplier

The algorithm for genotype-encoding works recursively from the outputs throughout the the graph to the inputs. The process begins with analysing the output genes. The output genes contains addresses of active node. The respective (active) nodes are evaluated to find other active nodes their genome require. This step is repeated until all the active nodes are found in the graph. The process is complete when addresses of input genes are found. This decoding algorithm does not evaluate non-coding (inactive) nodes, thus they will not be translated to the phenotype.

4.2.4 Mutation

Point mutation was used in CGP algorithm. This type of mutation implies that different part of the genotype(connection-, function-, and output-genes) can randomly be chosen and changed to another valid random value. The various parts are treated slightly differently when they are chosen to be mutated:

- ***Mutation of the function genes(F):*** If a function gene is chosen for mutation, then a valid address from the function lookup table is chosen as a replacement. The new address is selected randomly.
- ***Mutation of the connection genes(C):*** If a connection gene is chosen for mutation, then there are two possibilities. Connection gene can either be replaced by a valid address of a randomly chosen node output or a randomly chosen address of a program input.
- ***Mutation of the output genes(O):*** Mutation of output gene is managed at the same way as the connection gene.

The mutation rate (μ_r) is a percentage of total number of genes. There is no definitive answer to what the ideal mutation rate should be. It often requires some experimentation and trailing to find the optimal mutation rate for a specific problem. As a rule of thumb, one should use about 1% mutation if a maximum of 100 nodes are used[26].

4.2.5 Recombination

Recombination or crossover operators was not used in CGP algorithm. The reason for this was the strict genome constraints and structure. The genome consists mostly of addresses. Combining two addresses will most likely not result in a new valid address. In order to have a valid and functional genome, all the addresses must be valid. Switching two complete connection genes would be complicated because swapping can easily lead to violations of conditions set by level-back parameter(l). Taken this outcome into consideration, recombination has not been implemented in this thesis.

4.2.6 Fitness

The quality of a solution or individual is measured with a fitness score. Fitness score is calculated by at fitness function that estimates how close the a solution is to a given goal or an optimal solution. Calculation of

fitness score is a process which is divided in two. The first part consists of decoding CGP chromosome in terms of Function genes(F), Connection gene(C) and Output gene(O). The second part tests the quality of the solution with respect to the training vectors. The fitness function returns a fitness score for each solution. Fitness function can be studied in the attached code.

4.2.7 Population(1+ λ) and parent selection

1+ λ population is commonly used in the CGP evolutionary algorithm. In this thesis λ was set to 4 which means that each population consists of five individuals(1+4). This is a reasonable size in terms of computational power needed to evaluate the population and adds sufficient variation in a population. The parent is derived from the previous generation and four children originating from the same parent. The offspring with higher fitness scores than the parent is chosen as the parent of the next generation. To create more redundancy in CGP genotypes and increase the influence of neutrality a new criterion was added for selecting the next parent (see line 10). An offspring with same fitness score can replace the current parent. The effect of neutrality is described in section 4.2.1.

The main loop CGP algorithm:

```
1  int highest = 0;
2  makeRandomStartIndividuals();
3  evalRandomStartIndividuals();
4  makeFirstGeneration();
5  unsigned long long gen = 0;
6  for (gen = 0; gen < 10000; gen++) {
7      gen = gen + 1;
8      eval();
9      makeNewGeneration();
10     if (globScore > highest || globScore == highest) {
11         highest = globScore;
12         replaceParent(globScore);
13     }
14     printf("Score: %d (%d) GEN: %d\n", globScore, highest, gen);
15     printf("Score: %d (%d)\n", globScore, highest);
16 }
```

4.3 Zynq design

The implemented CGP algorithm was tested, integrated and accelerated on Zynq. This section addresses the the work that has been done in order to realize the Zynq embedded design. Most of the work has been done within the Xilinx environment that follows a distinctive development flow. Zynq development flow is quite different from the traditional way to develop embedded systems. The following two subsections will briefly describe the main differences. The purpose is to give the reader an overview of the development method that has been used in this thesis.

4.3.1 Traditional embedded design flow

In traditional or standard embedded systems, the development flow of software and hardware design is usually isolated from each other (Figure 4.9). After the software and hardware designs are successfully developed they are merged together in to a two-chip solution. Software is usually implemented on a standalone processor and hardware design on costume Application-Specific Integrated Circuit (ASIC) or FPGA. In a typical two-chip solution the main drawback is limited connectivity between hardware and software processor. Isolated hardware and software development-flow may also result in longer development time due to time spent on integration and optimization between two separate entities.

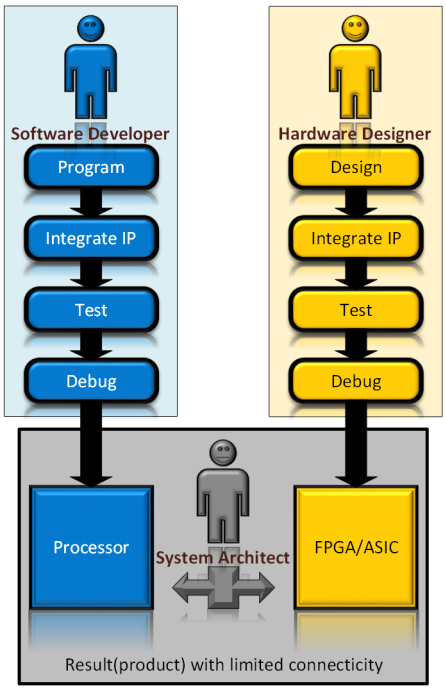


Figure 4.9: Traditional/standard embedded design flow

4.3.2 Zynq embedded design flow

The integration of processing system with programmable hardware on one single-chip presents unique advantages over standard embedded two-chip systems. Unlike a traditional processor, the Zynq PS has a configurable set of built-in peripherals and direct access to the the PL. This is in particular important during the development phase. Xilinx provides powerful software tools to streamline the development flow (Figure 4.10). Xilinx Design Tools introduced in section 3.2 is used in this thesis. The main advantages of these tools is that software and hardware can be coordinated early in development phase and time spent on integration and optimization is reduced significantly.

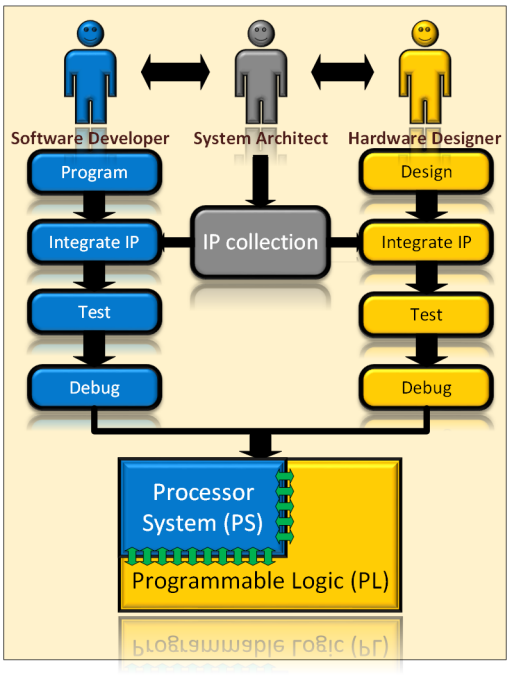


Figure 4.10: Design flow using Zynq-7000 AP SoC

4.4 Zynq Hardware

Vivado IDE was used to configure, design and build the Zynq *hardware platform*. This hardware platform defines how the ARM PS is configured and provides customize hardware design for the PL. The hardware platform was exported out of Vivado for use in Xilinx SDK when successfully build.

4.4.1 ARM Processing System

The first step was to create a new *RTL-project* in Vivado and target the Zynq device, which in this case was the *ZedBoard XC7Z020CLG484*. The *IP Integrator* was used to create a new block design. An embedded ARM processor core with IP-name *ZYNQ7 Processing System* (Figure 4.11) was added to the project. In the following paragraphs, PS will be configured so it meets the requirements for this thesis.

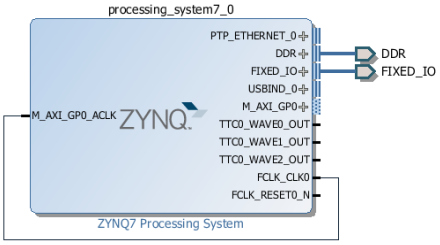


Figure 4.11: ZYNQ7 Processing System

4.4.2 I/O Peripherals

Zynq PS have configurable set of built-in peripherals. The required peripherals was enabled through GUI(Figure 4.12) that appears when one double-click the PS block. UART peripheral was enabled and mapped. UART was added for communication with PS and baud rate was set to 115200. Bank voltages was set to LVCMOS 3.3V for Bank 0 and LVCMOS 1.8V for Bank 1. Other required peripherals that were enabled was USB and GPIO. SD card and Ethernet was also enabled, but was not used in this thesis.

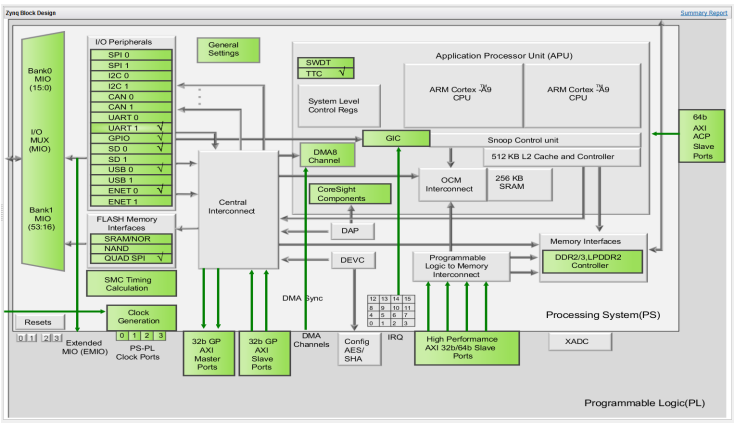


Figure 4.12: Zynq Block Design.

4.4.3 Memory and System clocks

System clocks and memory must be configured correctly before the Zynq PS can run software applications. System clocks was defined in the *Clock Configuration* menu. Input frequency was set to 33.33 MHz, CPU frequency to 666.66 MHz and DDR frequency to 533.33 MHz. The PL fabric clock was enabled and set to 100 MHz. These clock frequencies were set upon recommendation from Zynq data sheets and claims to give the Zynq design the most optimal speed.

DDR(3) memory was enabled and configured in the *DDR Configuration* menu. Memory settings and parameters must be specified for ZedBoard that was used in this experiment. The memory part on ZedBoard was MT41J128M16HA-15E. Other parameter was automatically updated after memory part was selected. The DRAM Bus With was set to 32-bit and read/write functions was enabled.

4.4.4 AXI interconnect

AXI Interconnect blocks was added for interaction and communication between PS and PL. Two instances of AXI Interconnect blocks were selected from IP directory. The PS and AXI Interconnect blocks must be customized in order to be connected together. AXI blocks were configured to have one master interface and one slave interface. For PS, general purpose AXI master interface(M_AXI_GP0 and M_AXI_GP1) was enabled in *PS-PL Configuration* menu. The missing ports of the PS-block appeared after the customization and then all blocks was successfully wired together (Figure 4.13).

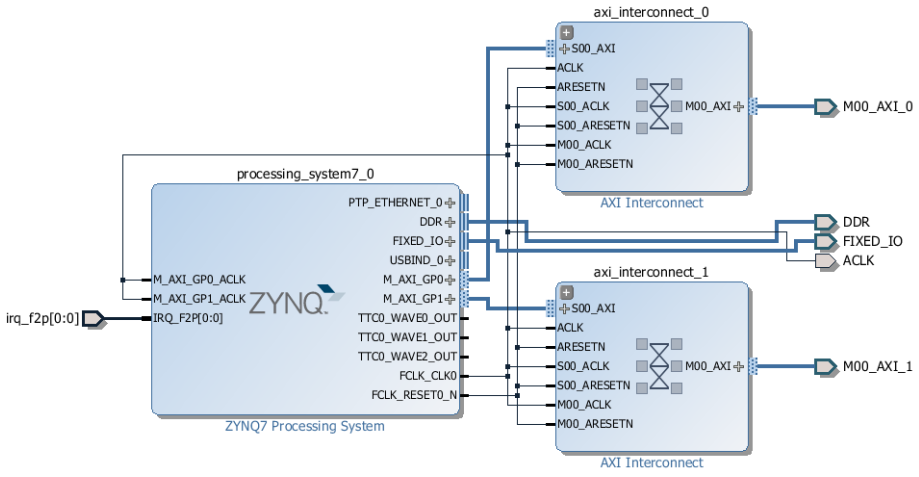


Figure 4.13: Zynq Block Design with AXI Interconnect blocks

4.4.5 MicroBlaze Soft Processor

MicroBlaze is a soft-core processor which means that it is entirely implemented in the general-purpose memory and logic fabric (PL) of FPGAs. It is based on a RISC architecture and support a variety of embedded applications. A MicroBlaze processor was added to an new block design. AXI interconnect was added to allow PS-PL connectivity. Other vital blocks were automatically added and connected using auto connection function in Vivado.

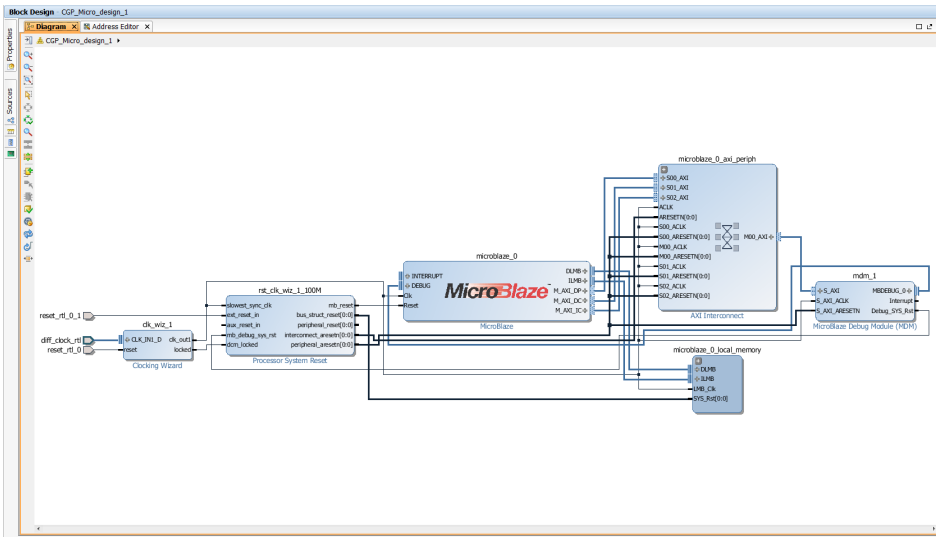


Figure 4.14: MicroBlaze Block Design

4.4.6 Building and exporting the hardware platform

The Zynq hardware platform was successfully validated with integrated auto validation function. Zynq hardware platform was ready to be exported to Xilinx SDK for software development. A top-level HDL Wrapper was created in Vivado. HDL Wrapper is the Zynq hardware platform defined in a Hardware Description Language, which in this case was VHDL. The design was synthesized and bitstream was successfully generated. Hardware was exported to Xilinx SDK and bitstream was included. The directory structure of Zynq Hardware platform contains some specific files. Overview of file names and what they contain are found in Table 4.2.

File name	Description:
ps7_init.c	Defines all the register settings required for initialize the ARM processing system (C-format)
ps7_init.h	Header file used by ps7_init.c
ps7_init.html	Readerfriendly information file that describes the ARM processing system
ps7_init.tcl	Defines all the register settings required for initialize the ARM processing system (TCL-format)
design_1.xml	Customized hardware specification file
design_1_wrapper.bit	PL configuration bitstream

Table 4.2: Zynq hardware platform file overview

4.5 Zynq Software

The complete Zynq hardware platform was exported directly to Xilinx SDK and appeared in the Porject Explorer(Figure 4.14). SDK uses the concept of workspaces to hold information about the software development work. Workspace hold on to SDK settings, software project files, and logs. The next sections addresses further development and implementation of Zynq software in Xilinx SDK.

4.5.1 Standalone Board Support Package(BSP)

First step in Xilinx SDK is to create a *Standalone Board Support Package (BSP)*. Standalone environment provides basic features such as standard input/output functions and access to processor hardware. BSP is a collection of libraries and drivers that will form the lowest layer of the application software [11]. BSP must be crated first because the software application runs on top of it.

4.5.2 Application Project

Next step was to crate a new *Application Project* and link it to the previously made BSP. The CPG algorithm, described in section 4.2, was imported as source to the Application Project. Figure 4.15 shows the Project Explorer window in Xilinx SDK. Here we can see the (1)Zynq embedded hardware design, (2)C/C++ software application and the (3)BSP. Figure 4.16 is a graphical illustration of the SDK application development flow. CGP

algorithm was built and tested through many iterations. Several changes in the algorithm were made to achieve optimal results.

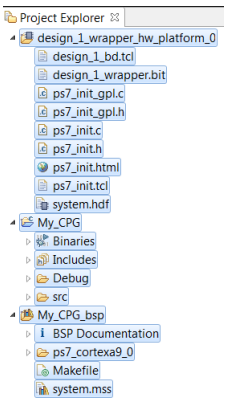


Figure 4.15: Xilinx SDK Project Explorer

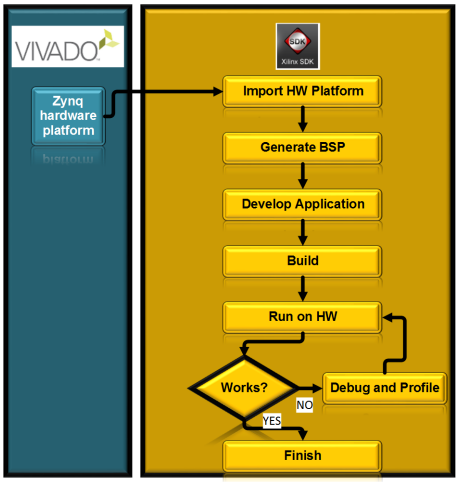


Figure 4.16: SDK Application Development Flow

4.5.3 Programming the Zedboard

Zedboard was connected to the host computer in order to enable the embedded ARM processor and run software applications. Two micro-USB cables was connected to the JTAG(pin:J17) and UART(pin:J14) on the ZedBoard. The jumpers was set to JTAG mode and the board was powered up. Baud rate was set to 115200bps and 8 bits data. Serial connection was established via COM-port for the USB-UART. The purpose of the serial connection was to track the processor activity. After the set up, the programming was done through Xilinx SDK. The Zynq design (bitstream) was programmed into the PL and the ARM processor was configured. The

ELF-file was created and software application was successfully executed by the ARM processor.

4.6 Measuring Performance

The CGP algorithm was implemented as software-only solution. The aim was to accelerate this solution, but before that could be done it was necessary to measure system performance. Vivado Analyser tools was used to confirm that the algorithm was properly executing and determine if the software was meeting the performance specification.

4.7 Profiling

In order to accelerate the solution an analysis was carried out. The analysis consisted of identifying software bottlenecks using profiling. A bottleneck is the same as *the critical path*. This is defined as the path in which the longest amount of time is spent.

Profiling was perform in Xilinx SDK (Figure 4.17). The *Build Settings* for CGP application and BSP was modified to enable profiling. The CGP algorithm needed several modifications in order to get the best profiling results. The purpose of profiling was to find which part(function) or parts(functions) of the code was taking the longest amount of time to execute. Xilinx SDK saves the profiling results in a file called *gmon.out*. The result of profiling was evaluated and a candidate was selected for moving to a hardware accelerator (PL). Hardware accelerator was realized by making a custom IP in Vivado HLS, this is described in the next section.

Name (location)	% Time
Summary	100,0%
CGP.c	92,34%
calc_fitness	75,85%
makeNewGeneration	14,35%
eval	2,1%
main	0,05%
makeRandomStartIndividuals	0,0%
makeFirstGeneration	0,0%
evalRandomStartIndividuals	0,0%
??	6,92%
xuartps_hw.c	0,44%
profile_cg.c	0,22%
xil_io.c	0,04%
profile_mcount_arm.S	0,04%
xscugic_hw.c	0,0%
write.c	0,0%
vfprintf.c	0,0%
_profile_timer_hw.c	0,0%

Figure 4.17: Profiling result

4.8 Building the Hardware Accelerator

This was an attempt to build a hardware accelerator to improve the performance of CGP (high-level) algorithm. This process aims to take out a part of the CGP algorithm and replicate it into a custom IP. This section explains how the hardware accelerator was built in Vivado HLS.

4.8.1 Accelerator Construction in Vivado HLS

It is not possible to realize all type of code in HW and for that reason Vivado HLS have impose some restrictions to the high-level function. Various changes had to done with the selected function in order to be moved out to HW as a custom IP. For example, memory allocation and system calls was not allowed. Recursive calls in the high-level function had to be replaced. These are typical examples that can not be implemented in HW. Vivado HLS also require a test program(testbench) that tested the functionality of the selected function.

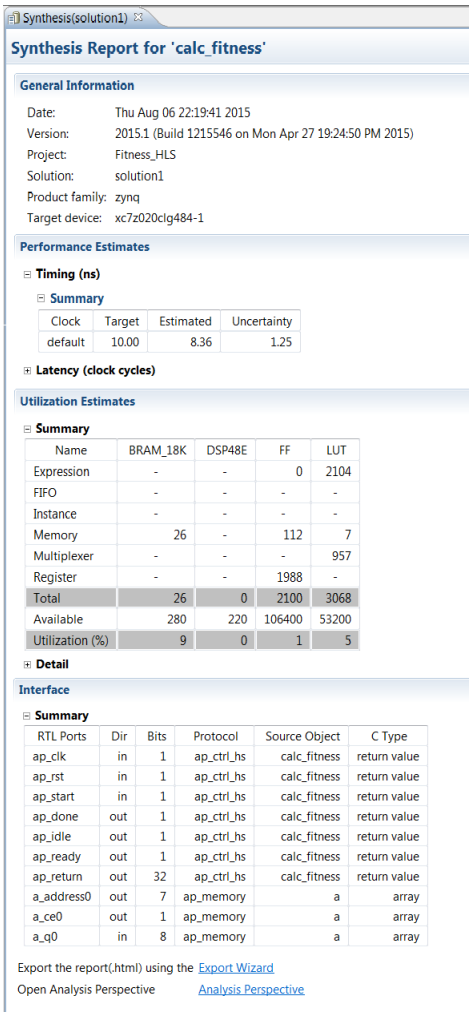
A new project was created in Vivado HLS. Source files, the function and test program, was imported. Function name and board information was provided in the creation. The next step was to run the test program to verify that the expected results appeared. The purpose of testing functionality, before custom IP generation, was to ensure which results was to be expected after hardware acceleration.

The high-level function was synthesized and translated into auto generated RTL source files. Synthesis report was generated and gave useful hint about the final accelerated design. The report contained information about performance, area estimates and latency (Figure 4.18).

4.8.2 Co-simulation and export of the IP core

After synthesis, Co-simulation (Figure 4.19) was performed to check whether the synthesis was proceeded correctly and to check that the generated RTL code have the expected functionality. The same test program was used and expected results was returned by the generated RTL code.

The last thing that was done in Vivado HLS was exporting the RLT as an IP core (Figure 4.20). Programming languages was set to VHDL and the rest of the IP core exporting was straight forward and easy task since Vivado HLS handled the rest.



Synthesis(solution1)

Synthesis Report for 'calc_fitness'

General Information

Date: Thu Aug 06 22:19:41 2015
Version: 2015.1 (Build 1215546 on Mon Apr 27 19:24:50 PM 2015)
Project: Fitness_HLS
Solution: solution1
Product family: zynq
Target device: xc7z020clg484-1

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
default	10.00	8.36	1.25

Latency (clock cycles)

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	2104
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	26	-	112	7
Multiplexer	-	-	-	957
Register	-	-	1988	-
Total	26	0	2100	3068
Available	280	220	106400	53200
Utilization (%)	9	0	1	5

Detail

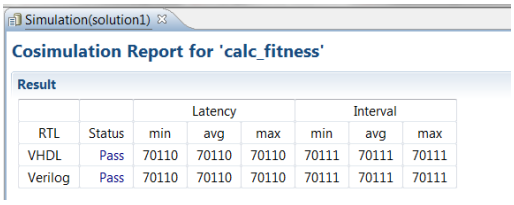
Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	calc_fitness	return value
ap_rst	in	1	ap_ctrl_hs	calc_fitness	return value
ap_start	in	1	ap_ctrl_hs	calc_fitness	return value
ap_done	out	1	ap_ctrl_hs	calc_fitness	return value
ap_idle	out	1	ap_ctrl_hs	calc_fitness	return value
ap_ready	out	1	ap_ctrl_hs	calc_fitness	return value
ap_return	out	32	ap_ctrl_hs	calc_fitness	return value
a_address0	out	7	ap_memory	a	array
a_ce0	out	1	ap_memory	a	array
a_q0	in	8	ap_memory	a	array

Export the report(.html) using the Export Wizard
Open Analysis Perspective Analysis Perspective

Figure 4.18: Synthesis report



Simulation(solution1)

Cosimulation Report for 'calc_fitness'

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	Pass	70110	70110	70110	70111	70111	70111
Verilog	Pass	70110	70110	70110	70111	70111	70111

Figure 4.19: Cosimulation report

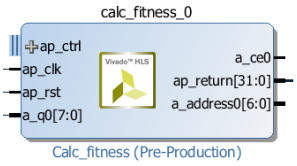


Figure 4.20: Exported IP core Vivado IP catalogue

4.9 Integrate the Custom IP with Embedded System

Work continues in Vivado IDE. The purpose was to integrate the custom IP with rest of the embedded system. HLS project from previous section was targeted and the custom IP was imported to the IP-catalogue by using the *Project Manager*. The custom IP was added to the block design. Zynq PS block and AXI Interconnect was added and customized. The Zynq embedded system was connected and functionality verified(Figure 4.21) Performance was measured and system was profiled in the same way as described in Section 4.7.

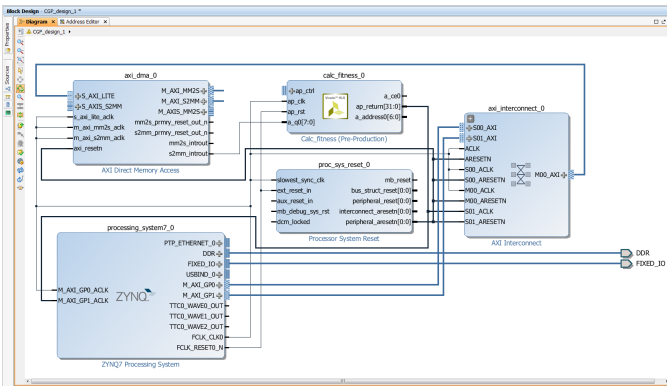


Figure 4.21: Custom IP with Zynq Embedded System

New block design was created for the Microblaze embedded system. Figure 4.22 shows the first attempt to build the Microblaze design in the same way as the accelerated Zynq design. Unfortunately, MicroBlaze design could not synthesize because of some connectivity errors. It was made several attempts to get a working system, but due to shortage of time, it was not completed. It should be noted that it may not be possible to combine/accelerate a MicroBlaze system with a custom IP.

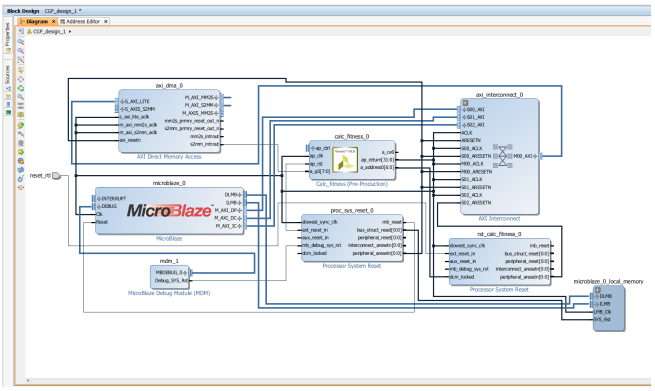


Figure 4.22: Custom IP with Microblaze Embedded System (Unsuccessful attempt)

Chapter 5

Experiments and Result

5.1 Experiments

A CGP algorithm was created to investigate the acceleration potential by implementing and running it on different hardware levels. The algorithm was exactly the same through all experiments. It is also important to emphasize that the number of generations, EA parameters and final result of the algorithm was same for all experiments.

These experiments were conducted:

1. **MicroBlaze Soft Processor:** Soft-core processor implemented on the PL part of the Zynq device.

2. **ARM Cortex-A9 (PS):** Hard-core processor on the PS part of the Zynq device. One ARM core was used to execute CPG algorithm (singe thread).

3. **i7-Q740 CPU:** 64-bit Quad-Core i7 CPU on Sony laptop. One single core was used to execute CPG algorithm (singe thread).

4. **MicroBlaze and Custom-IP (PL):** Soft-core processor implemented on the PL part and a (accelerated) custom-IP. The algorithm was not execute on this design because it was not successfully implemented. Se section 4.9.

5. **ARM Cortex-A9 (PS) and Custom-IP (PL):** Hard-core processor on the PS part and a (accelerated) custom-IP on the PL.

5.1.1 Comparison

EXPERIMENTAL RESULTS		
Experiments:	**Time(s):**	**Acceleration (1)**
MicroBlaze Soft Processor	4,236	1
ARM Cortex-A9 (PS)	2,396	1,8
i7-Q740 CPU (laptop)	1,253	3,4
MicroBlaze and Custom-IP (PL)	XX	XX
ARM Cortex-A9 (PS) and Custom-IP (PL)	0,515	8,2

Table 5.1: Experimental results

5.1.2 Profiling results

MicroBlaze Soft Processor:

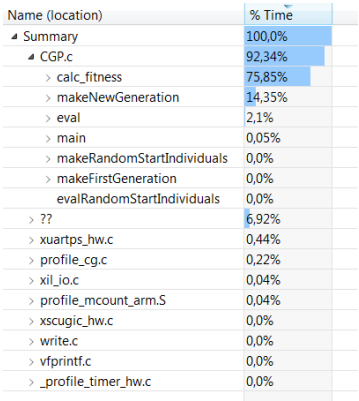


Figure 5.1: Profiling results: MicroBlaze Soft Processor

ARM Cortex-A9 (PS)

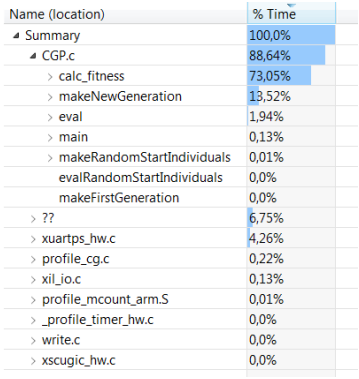
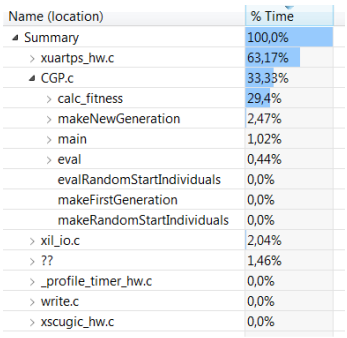


Figure 5.2: Profiling results: ARM Cortex-A9 (PS)

ARM Cortex-A9 (PS) and Custom-IP (PL)



Name (location)	% Time
Summary	100,0%
xuartps_hw.c	63,17%
CGP.c	33,33%
calc_fitness	29,4%
makeNewGeneration	2,47%
main	1,02%
eval	0,44%
evalRandomStartIndividuals	0,0%
makeFirstGeneration	0,0%
makeRandomStartIndividuals	0,0%
xil_io.c	2,04%
??	1,46%
_profile_timer_hw.c	0,0%
write.c	0,0%
xscugic_hw.c	0,0%

Figure 5.3: Profiling results: ARM Cortex-A9 (PS) and Custom-IP (PL)

5.2 Discussion

The results of the profiling (Figures 5.1 and Figures 5.2) shows that calc_fitness function was a major bottleneck in the CGP algorithm. Calc_fitness function uses up to 75% of execution time. This bottleneck was resolved by moving the calc_fitness function into hardware as a custom IP core. Figure 5.3 shows the profiling results of the accelerated solution. Here the calc_fitness function uses (only) 33% of execution time. This means 42% reduction.

The results in Table 5.1 clearly shows that hardware acceleration gave good results. The accelerated solution was 8 time faster than the other Zynq implemented solutions.

The execution time of the algorithm on the laptop(i7 CPU) was measured differently than others. *Gettimeofday* function was used to measure the execution time on i7 CPU whereas the other measurements were measured by Xilinx tools. Due to this difference measurement I assert that the i7 CPU should have higher acceleration than the value(3,4) in the table.

The unfinished and untested MicroBlaze accelerated solution was placed between i7 CPU and ARM accelerated solution in Table 5.1. My assumption is that this solution should be faster than i7 CPU implementation, but somewhat slower than ARM accelerated solution.

Chapter 6

Conclusion and proposals for further work

6.1 Conclusion

The primary goals of this thesis was to design and implement a hardware friendly Zynq-based CGP algorithm and investigate the acceleration potential. It was made several attempts to find out if it was possible to increase the speed of the CGP algorithm by implementing single part of algorithm as hardware component.

The Zynq-platform is a unique blend of two technologies, which includes a Dual ARM® Coretex-A9 Processer System and 7-series Programmable Logic. This means that Zynq is able to take advantage of software programming and in addition configure programmable hardware both at the same time. PL runs heavy data path algorithms and take advantage of HW parallelism. PS controls and updates elements of the CGP algorithm. This thesis shows that it is possible to increase the speed of the CGP algorithm by implementing single part or parts of the evolutionary operations as hardware components.

6.2 Further Work

Xilinx Zynq-7000 AP SoC platform has the potential to become the next revolutionary step in evolvable hardware design.

The most important future work for this thesis is to complete MicroBlaze accelerated solution and confirm my assumptions. It is also

Specific tasks for further work:

- Use multiple cores on ARM processor.

- Implement a own IP core and compare it up to automatically generate the IP core

Appendices

Appendix A

CGP algorithm code

```
#include "h_file.h"
#include "fitness.h"

//Global variable declarations
int BIT[] = {0, 1};
int globScore;
char genom[5];
char genom2[15];
char genom3[90];
char population[SIZE_OF_POP][sizeof (genom3)];
char first_pop[SIZE_OF_POP][sizeof (genom3)];
unsigned char sonarDataOriginal[NUM_VECTORS][NUM_POINTS];
unsigned int vectorsOutOriginal[NUM_VECTORS];

//unsigned char sonarData[NUM_VECTORS][NUM_POINTS-1];
//int klasse[NUM_VECTORS];

//Function declaration
void makeRandomStartIndividuals();
void evalRandomStartIndividuals();
int calc_fitness(const char *a);
int decodeBinary8ToDecimal(int a[]);
int decodeBinary6ToDecimal(int a[]);
void makeFirstGeneration();
void eval();
void makeNewGeneration();

/********
 * Generates five random starting individuals.
 * Saves the genome in the array first_pop
 */
void makeRandomStartIndividuals() {

#ifdef PRINT_FUNCTION_CALL
    printf("@@makeRandomStartIndividuals() \n");
#endif
    int i, j;
    for (i = 0; i < SIZE_OF_POP; i++) {
        for (j = 0; j<sizeof (genom3); j++) {
```

```
40              first_pop[i][j] = BIT[rand() % BIT_ARRAY_SIZE];
41          }
42      }
43
44  #ifdef PRINT_GENOM
45      int k = 0;
46      int l = 0;
47      printf("RandomStartIndividuals:");
48      for (k = 0; k < SIZE_OF_POP; k++) {
49          printf("\n %d ", k + 1);
50          for (l = 0; l < sizeof (genom3); l++) {
51              printf("%d", first_pop[k][l]);
52          }
53      }
54      printf("\n");
55  #endif
56  }
57
58  /*********
59   * Evaluates the first randomly generated individuals(first_pop).
60   * Sets the individual with the highest score as the initial
       parent.
61   */
62  void evalRandomStartIndividuals() {
63  #ifdef PRINT_FUNCTION_CALL
64      printf("@@evalRandomStartIndividuals() \n");
65  #endif
66      //Computes fitness score and save the scores in the array
       score[]
67      int i, j;
68      int t = 0;
69      int score_row[6];
70      char* temp_pop;
71
72      //Splits up individuals in a genome
73      //and sends each of them to the fitness function
74      for (i = 0; i < SIZE_OF_POP; i++) {
75          j = 0;
76          t = 90 * j; //next genom
77          temp_pop = first_pop[i];
78          temp_pop = temp_pop + t;
79          score_row[i] = calc_fitness(temp_pop);
80          j++;
81          //score[i] = fitness(first_pop[i]);
82  #ifdef DEBUG
83          printf(" #Pointer_addr:%x ", temp_pop);
84          printf("#NR:%d #Score:%d \n", i + 1, score_row[i]);
85  #endif
86      }
87
88      //Find the individual with the highest score
89      int pos = 0;
90      int k, max_score;
91      int temp_score[SIZE_OF_POP] = {0, 0, 0, 0, 0};
92      max_score = 0;
93  #ifdef PRINT_SCORE
94      printf(" ScoreBoard:\n");
95  #endif
96      for (k = 0; k < 6; k++) {
```

```
97  #ifdef PRINT_SCORE
98          printf(" %d ", score_row[k]);
99  #endif
100         temp_score[k] = temp_score[k] + score_row[k];
101
102         if (temp_score[k] > max_score) {
103             max_score = temp_score[k];
104             pos = k;
105         }
106     }
107 #ifdef PRINT_RESULT
108     printf("\nRESULT: Max score: %d on pos: %d \n", max_score, pos
    );
109 #endif
110
111
112     //Sets the individual with the highest score as the initial
    parent.
113     int m = 0;
114     for (m = 0; m < sizeof (genom3); m++) {
115         population[0][m] = first_pop[pos][m];
116     }
117 }
118
119 /********************
120  * Decode 6-bit Binary To Decimal.
121  */
122 int decodeBinary6ToDecimal(int a[]) {
123     int C_sum = 0;
124     C_sum = (a[0]*32 + a[1]*16 + a[2]*8 + a[3]*4 + a[4]*2 + a[5]);
125     return C_sum;
126 }
127
128 /******************
129  * Decode 8-bit Binary To Decimal
130  */
131 int decodeBinary8ToDecimal(int a[]) {
132     int C_sum = 0;
133     C_sum = (a[0]*128 + a[1]*64 + a[2]*32 + a[3]*16 + a[4]*8 +
134             a[5]*4 + a[6]*2 + a[7]);
135     return C_sum;
136 }
137
138
139 void makeFirstGeneration() {
140 #ifdef PRINT_FUNCTION_CALL
141     printf("@@makeFirstGeneration()\n");
142 #endif
143     int i, j;
144     for (i = 1; i < SIZE_OF_POP; i++) {
145         for (j = 0; j < sizeof (genom3); j++) {
146             population[i][j] = population[0][j];
147             if ((rand() % MUT_RATE) == 1) {
148                 population[i][j] = !population[i][j];
149             }
150         }
151     }
152 #ifdef PRINT_GENOM
153     int k = 0;
```

```
154     int l = 0;
155     printf("First generation:");
156     for (k = 0; k < SIZE_OF_POP; k++) {
157         printf("\n %d ", k + 1);
158         for (l = 0; l < sizeof (genom3); l++) {
159             printf("%d", population[k][l]);
160         }
161     }
162     printf("\n");
163 #endif
164 }
165 
166 void makeNewGeneration() {
167 #ifdef PRINT_FUNCTION_CALL
168     printf("@@makeNewGeneration() \n");
169 #endif
170     int m, n;
171     for (m = 1; m < SIZE_OF_POP; m++) {
172         for (n = 0; n < sizeof (genom3); n++) {
173             population[m][n] = population[0][n];
174             if ((rand() % MUT_RATE) == 1) {
175                 population[m][n] = !population[m][n];
176             }
177         }
178     }
179 #ifdef PRINT_GENOM
180     int i = 0;
181     int j = 0;
182     printf("NEW generation:");
183     for (i = 0; i < SIZE_OF_POP; i++) {
184         printf("\n %d ", i + 1);
185         for (j = 0; j < sizeof (genom3); j++) {
186             printf("%d", population[i][j]);
187         }
188     }
189     printf("\n");
190 #endif
191 }
192 
193 
194 void eval() {
195 #ifdef PRINT_FUNCTION_CALL
196     printf("@@eval() \n");
197 #endif
198 
199     int i, j;
200     int t = 0;
201     int score [SIZE_OF_POP][6];
202     int score_row[6];
203     char* temp_pop;
204     for (i = 0; i < SIZE_OF_POP; i++) {
205         j = 0;
206         t = 90 * j;
207         temp_pop = population[i];
208         temp_pop = temp_pop + t;
209         score_row[i] = calc_fitness(temp_pop);
210         j++;
211 #ifdef DEBUG
212         printf(" #Pointer_addr:%x", temp_pop);
```

```
213         printf(" #NR %d #Score: %d \n", i + 1, score[i][j]);
214 #endif
215     }
216
217     //Find the individual with the highest score
218     int pos = 0;
219     int k, l, max_score;
220     int temp_score[SIZE_OF_POP] = {0, 0, 0, 0, 0};
221     max_score = 0;
222 #ifdef PRINT_SCORE
223     printf(" ScoreBoard:\n");
224 #endif
225     for (k = 0; k < 6; k++) {
226 #ifdef PRINT_SCORE
227         printf(" %d ", score_row[k]);
228 #endif
229         temp_score[k] = temp_score[k] + score_row[k];
230
231         if (temp_score[k] > max_score) {
232             max_score = temp_score[k];
233             pos = k;
234         }
235     }
236     globScore = max_score;
237
238 #ifdef PRINT_RESULT
239     printf("\nRESULT: Max score is: %d on pos: %d \n", max_score,
    pos);
240 #endif
241     int m = 0;
242     for (m = 0; m < sizeof (genom3); m++) {
243         population[0][m] = population[pos][m];
244     }
245 }
246
247 /*****************************
248  *
249  ************************
250 int main(void) {
251     // Genom:['F1','F2', 'C1','C2','C3']
252     //        ['0' ,'1' , '2' ,'3','4']
253     //        ['0-1' ,'0-1' , '0-1' ,'0-1','0-1']
254     /*FU:
    ------------------------------------------------------
255         |  ----------------          -------------------------
     |
256         X-|-| P = X if F1 = 0 \___P___| if F2 = 0 then (P >= C) |
    __|__true(1)
257         Y-|-|_P_=_Y_if_F1_=_1_/        |_if_F2_=_1_then_(P_<=_C)_|
     |  false(0)
258           |       |                        |             |
     |
259           |       F1                      F2             C
     |
260           |_______|_____________________|_____________|
    ______|
261                   F1                      F2           C =
    C1C2C3
262      */
```

```
263     //printf("%c\n", genom[0]);
264
265     // Genom:['F1',    'F2', 'C']
266     //    Bit:['[0-5' ,'6' , '7-14']
267     /*FU:
        ____________________________________________________________
268           |  _________________          ____________________________
         |
269         X-|-| P = X if F1 = 0 \___P___| if F2 = 0 then (P >= C) |
        __|__true(1)
270         Y-|-|_P_=_Y_if_F1_=_1_/       |_if_F2_=_1_then_(P_<=_C)_|
         |  false(0)
271           |       |                          |             |
         |
272           |       F1                         F2            C
         |
273           |_______|__________________________|_____________|
        ______|
274                   F1                         F2          C =
        C1C2C3
         */
276     ////////////////////////////////////////////////////////
277     int i, j;
278     //sonarData
279     for (i = 0; i < NUM_VECTORS; i++) {
280         for (j = 0; j < NUM_POINTS - 1; j++) {
281             sonarDataOriginal[i][j] = sonarData[i][j];
282             //printf("%d", sonarDataOriginal[i][j]);
283         }
284         //printf("\n");
285     }
286     //sonarData class
287     for (i = 0; i < NUM_VECTORS; i++) {
288         for (j = 0; j < NUM_POINTS; j++) {
289             if (j == 60) {
290                 //printf(" %c ",sonarDataOriginal[i][j]);
291                 vectorsOutOriginal[i] = sonarData[i][j];
292             }
293
294         }
295         //printf("%d\n", vectorsOutOriginal[i]);
296     }
297
298     for (i = 0; i < SIZE_OF_POP; i++) {
299         for (j = 0; j<sizeof (genom3); j++) {
300             population[i][j] = 0;
301             first_pop[i][j] = 0;
302         }
303     }
304     //Makes a new seed for the random function
305     srand(time(NULL));
306     int highest = 0;
307     makeRandomStartIndividuals();
308     evalRandomStartIndividuals();
309     makeFirstGeneration();
310     unsigned long long gen = 0;
311     for (gen = 0; gen < 10000; gen++) {
312         gen = gen + 1;
313         eval();
```

```
314         makeNewGeneration();
315         if (globScore > highest || globScore == highest) {
316             highest = globScore;
317         }
318         //printf("Score: %d (%d) GEN: %d\n", globScore, highest,
    gen);
319         //printf("Score: %d (%d)\n", globScore, highest);
320     }
321 
322     printf("LAST generation:");
323  printf("Score: %d (%d) GEN: %d\n", globScore, highest, gen);
324     printf("Score: %d (%d)\n", globScore, highest);
325     int k = 0;
326     int l = 0;
327 
328     for (k = 0; k < SIZE_OF_POP; k++) {
329         printf("\n %d ", k + 1);
330         for (l = 0; l < sizeof (genom3); l++) {
331             printf("%d", population[k][l]);
332         }
333     }
334 
335     printf("\n");
336     printf("Score: %d \n", globScore);
337 
338 
339     return 0;
340 }
```

Appendix B

Custom IP code

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#include <ap_axi_sdata.h>
#include <ap_int.h>


#define F1_LENGTH 6
#define CONSTANT_LENGTH 8

struct FU2 {
    int F1[F1_LENGTH];
    int F2;
    int C[CONSTANT_LENGTH];
};

int calc_fitness(char a[90]) {
#define NUM_VECTORSX 208
#define NUM_POINTSX 61

    int pos;
    int p;
    int FU_nr = 0;
    //Copy genome into a structure of FU
    struct FU2 FU_0, FU_1, FU_2, FU_3, FU_4, FU_5;
    for (pos = 0; pos < 90;) {
        for (p = 0; p < 6; p++) {
            if (FU_nr == 0 && p < 6) {
                FU_0.F1[p] = a[pos];
                if (p == 5) {
                    FU_0.F2 = a[pos + 1];
                    pos = pos + 9;
                }
            }
            if (FU_nr == 1 && p < 6) {
                FU_1.F1[p] = a[pos];
                if (p == 5) {
                    FU_1.F2 = a[pos + 1];
                    pos = pos + 9;
```

```
                }
            }
            if (FU_nr == 2 && p < 6) {
                FU_2.F1[p] = a[pos];
                if (p == 5) {
                    FU_2.F2 = a[pos + 1];
                    pos = pos + 9;
                }
            }
            if (FU_nr == 3 && p < 6) {
                FU_3.F1[p] = a[pos];
                if (p == 5) {
                    FU_3.F2 = a[pos + 1];
                    pos = pos + 9;
                }
            }
            if (FU_nr == 4 && p < 6) {
                FU_4.F1[p] = a[pos];
                if (p == 5) {
                    FU_4.F2 = a[pos + 1];
                    pos = pos + 9;
                }
            }
            if (FU_nr == 5 && p < 6) {
                FU_5.F1[p] = a[pos];
                if (p == 5) {
                    FU_5.F2 = a[pos + 1];
                    pos = pos + 9;
                }
            }
            pos++;
        }
        FU_nr++;
    }

    FU_nr = 0;
    for (pos = 0; pos < 90;) {
        for (p = 0; p < 8; p++) {
            if (p == 0) {
                pos = pos + 7;
            }
            if (FU_nr == 0 && p < 8) {
                FU_0.C[p] = a[pos];
            }
            if (FU_nr == 1 && p < 8) {
                FU_1.C[p] = a[pos];
            }
            if (FU_nr == 2 && p < 8) {
                FU_2.C[p] = a[pos];
            }
            if (FU_nr == 3 && p < 8) {
                FU_3.C[p] = a[pos];
            }
            if (FU_nr == 4 && p < 8) {
                FU_4.C[p] = a[pos];
            }
            if (FU_nr == 5 && p < 8) {
                FU_5.C[p] = a[pos];
            }
```

```
100             pos++;
101         }
102         FU_nr++;
103     }
104
105
106
107     //Decode 6-bit Binary To Decimal.
108     int C_sum[6];
109     C_sum[0] = (FU_0.C[0]*128 + FU_0.C[1]*64 + FU_0.C[2]*32 + FU_0
        .C[3]*16 + FU_0.C[4]*8 +
110             FU_0.C[5]*4 + FU_0.C[6]*2 + FU_0.C[7]);
111     C_sum[1] = (FU_1.C[0]*128 + FU_1.C[1]*64 + FU_1.C[2]*32 + FU_1
        .C[3]*16 + FU_1.C[4]*8 +
112             FU_1.C[5]*4 + FU_1.C[6]*2 + FU_1.C[7]);
113     C_sum[2] = (FU_2.C[0]*128 + FU_2.C[1]*64 + FU_2.C[2]*32 + FU_2
        .C[3]*16 + FU_2.C[4]*8 +
114             FU_2.C[5]*4 + FU_2.C[6]*2 + FU_2.C[7]);
115     C_sum[3] = (FU_3.C[0]*128 + FU_3.C[1]*64 + FU_3.C[2]*32 + FU_3
        .C[3]*16 + FU_3.C[4]*8 +
116             FU_3.C[5]*4 + FU_3.C[6]*2 + FU_3.C[7]);
117     C_sum[4] = (FU_4.C[0]*128 + FU_4.C[1]*64 + FU_4.C[2]*32 + FU_4
        .C[3]*16 + FU_4.C[4]*8 +
118             FU_4.C[5]*4 + FU_4.C[6]*2 + FU_4.C[7]);
119     C_sum[5] = (FU_5.C[0]*128 + FU_5.C[1]*64 + FU_5.C[2]*32 + FU_5
        .C[3]*16 + FU_5.C[4]*8 +
120             FU_5.C[5]*4 + FU_5.C[6]*2 + FU_5.C[7]);
121
122     //Decode 8-bit Binary To Decimal
123     int addr[6];
124     addr[0] = (FU_0.F1[0]*32 + FU_0.F1[1]*16 + FU_0.F1[2]*8 + FU_0
        .F1[3]*4 + FU_0.F1[4]*2 + FU_0.F1[5]);
125     addr[1] = (FU_1.F1[0]*32 + FU_1.F1[1]*16 + FU_1.F1[2]*8 + FU_1
        .F1[3]*4 + FU_1.F1[4]*2 + FU_1.F1[5]);
126     addr[2] = (FU_2.F1[0]*32 + FU_2.F1[1]*16 + FU_2.F1[2]*8 + FU_2
        .F1[3]*4 + FU_2.F1[4]*2 + FU_2.F1[5]);
127     addr[3] = (FU_3.F1[0]*32 + FU_3.F1[1]*16 + FU_3.F1[2]*8 + FU_3
        .F1[3]*4 + FU_3.F1[4]*2 + FU_3.F1[5]);
128     addr[4] = (FU_4.F1[0]*32 + FU_4.F1[1]*16 + FU_4.F1[2]*8 + FU_4
        .F1[3]*4 + FU_4.F1[4]*2 + FU_4.F1[5]);
129     addr[5] = (FU_5.F1[0]*32 + FU_5.F1[1]*16 + FU_5.F1[2]*8 + FU_5
        .F1[3]*4 + FU_5.F1[4]*2 + FU_5.F1[5]);
130
131 #ifdef DEBUG
132     int Dim_Cons = 0;
133     for (Dim_Cons = 0; Dim_Cons < 6; Dim_Cons++) {
134         printf(" #Dimensjon:%d #Constant:%d \n", addr[Dim_Cons],
        C_sum[Dim_Cons]);
135     }
136 #endif
137
138     //Dataset contains 60 dimensions.
139     //If addr is greater than 60 then 0 (Score) will be returnd.
140     if (addr[0] > NUM_POINTSX) {
141         return 0;
142     }
143
144     unsigned char sonarDataOriginalX[NUM_VECTORSX][NUM_POINTSX];
145     unsigned int vectorsOutOriginalX[NUM_VECTORSX];
```

```
146     unsigned char sonarDataX[NUM_VECTORSX][NUM_POINTSX] = {
147         {12, 7, 15, 32, 45, 48, 56, 19, 31, 51, 24, 63, 56, 81,
        85, 84, 70, 75, 75, 44, 83, 97, 89, 137, 133, 114, 136, 135,
        99, 88, 104, 118, 142, 146, 162, 192, 176, 157, 137, 143, 165,
         122, 95, 71, 50, 62, 45, 43, 19, 6, 6, 2, 3, 4, 1, 3, 4, 3,
        8, 11, 0},
148         {5, 9, 10, 5, 24, 25, 39, 40, 79, 53, 41, 40, 57, 16, 16,
        57, 79, 76, 129, 122, 147, 129, 110, 141, 171, 163, 181, 206,
        173, 98, 33, 66, 130, 192, 217, 216, 170, 155, 126, 69, 13,
        72, 72, 108, 67, 35, 26, 34, 9, 8, 5, 0, 1, 4, 1, 4, 4, 2, 2,
        0, 1},
149         {33, 59, 78, 108, 102, 45, 47, 1, 49, 56, 74, 57, 79, 84,
        93, 100, 111, 119, 134, 95, 57, 50, 110, 166, 129, 71, 106,
        151, 133, 106, 94, 60, 22, 36, 73, 116, 94, 85, 96, 106, 91,
        69, 42, 49, 71, 57, 48, 38, 16, 4, 3, 9, 5, 3, 4, 0, 2, 3, 4,
        5, 0},
150         {11, 13, 21, 17, 30, 65, 54, 88, 85, 73, 125, 167, 176,
        198, 190, 240, 255, 226, 204, 199, 132, 103, 100, 99, 82, 81,
        83, 70, 112, 51, 96, 75, 50, 59, 33, 106, 97, 26, 46, 50, 42,
        14, 35, 41, 15, 5, 13, 18, 10, 1, 3, 2, 2, 1, 2, 4, 3, 1, 1,
        1, 1},
151         {16, 27, 38, 31, 36, 30, 22, 23, 53, 91, 139, 132, 130,
        137, 167, 221, 249, 238, 201, 188, 176, 98, 17, 12, 69, 72,
        56, 48, 10, 64, 50, 48, 62, 49, 68, 34, 27, 51, 45, 5, 33, 45,
         39, 41, 18, 3, 20, 19, 2, 1, 6, 2, 6, 2, 3, 4, 5, 1, 2, 3,
        0},
152         {6, 14, 28, 27, 24, 58, 61, 96, 142, 157, 161, 180, 141,
        135, 165, 176, 172, 192, 227, 219, 203, 171, 109, 93, 135, 61,
         129, 217, 153, 217, 217, 128, 47, 69, 107, 77, 155, 172, 137,
         120, 118, 65, 54, 56, 53, 4, 34, 18, 3, 2, 0, 5, 4, 2, 4, 6,
        8, 4, 2, 1, 1},
153         {14, 30, 32, 38, 36, 14, 21, 17, 34, 59, 81, 108, 132,
        125, 151, 184, 230, 232, 222, 195, 186, 133, 78, 80, 57, 41,
        44, 46, 52, 42, 70, 79, 86, 113, 128, 71, 42, 67, 81, 49, 23,
        11, 19, 18, 10, 14, 27, 29, 19, 6, 5, 8, 2, 2, 4, 3, 0, 2, 5,
        3, 0},
154         {2, 4, 15, 5, 5, 9, 27, 32, 15, 32, 22, 50, 4, 57, 44, 54,
         17, 58, 103, 101, 69, 94, 141, 123, 80, 136, 134, 64, 53, 90,
         159, 187, 156, 89, 100, 76, 137, 224, 251, 233, 156, 127, 81,
         81, 109, 93, 67, 40, 17, 7, 6, 3, 0, 3, 2, 1, 1, 1, 1, 2, 1},
155         {4, 6, 6, 9, 16, 12, 26, 35, 15, 34, 75, 121, 144, 145,
        163, 190, 229, 255, 247, 230, 195, 178, 169, 152, 94, 23, 12,
        22, 26, 43, 83, 117, 100, 128, 123, 89, 59, 102, 93, 38, 18,
        35, 39, 9, 16, 11, 6, 5, 4, 1, 3, 1, 2, 5, 5, 1, 0, 2, 1, 0,
        0},
156         {19, 16, 12, 10, 15, 16, 30, 62, 90, 113, 105, 100, 108,
        105, 115, 135, 186, 157, 51, 118, 105, 109, 146, 137, 80, 58,
        178, 255, 185, 120, 130, 139, 73, 25, 49, 106, 117, 82, 72,
        61, 50, 62, 47, 21, 17, 13, 9, 2, 5, 1, 3, 0, 1, 2, 2, 0, 1,
        1, 2, 2, 1},
157         {10, 14, 18, 15, 30, 38, 39, 9, 15, 2, 32, 63, 76, 95,
        115, 137, 167, 181, 193, 221, 214, 203, 213, 237, 218, 157,
        105, 83, 79, 65, 85, 113, 127, 130, 132, 117, 107, 111, 109,
        113, 94, 84, 65, 64, 54, 45, 25, 13, 11, 3, 0, 3, 4, 2, 1, 3,
        2, 2, 2, 1, 0},
158         {8, 24, 33, 35, 42, 43, 18, 35, 53, 89, 45, 16, 13, 95,
        138, 138, 131, 108, 51, 107, 196, 248, 239, 141, 134, 174,
        145, 138, 55, 54, 148, 161, 75, 47, 75, 131, 156, 109, 139,
        156, 127, 60, 49, 44, 33, 15, 28, 26, 12, 4, 4, 5, 6, 3, 1, 3,
```

```
 2, 3, 0, 2, 1},
159      {6, 1, 8, 20, 26, 19, 36, 29, 11, 24, 28, 21, 18, 50, 67,
106, 128, 171, 202, 213, 223, 240, 255, 253, 244, 220, 183,
147, 126, 124, 104, 62, 45, 63, 89, 95, 74, 58, 80, 90, 95,
113, 83, 50, 22, 43, 50, 31, 16, 8, 6, 5, 6, 4, 0, 1, 3, 1, 1,
 1, 0},
160      {13, 13, 21, 8, 29, 23, 26, 15, 37, 72, 71, 78, 67, 96,
143, 111, 66, 30, 170, 239, 199, 136, 173, 233, 194, 209, 226,
 155, 75, 28, 33, 15, 25, 102, 93, 26, 48, 100, 109, 64, 29,
55, 47, 36, 54, 60, 28, 6, 7, 4, 1, 2, 3, 1, 3, 2, 2, 1, 1, 1,
 1},
161      {11, 12, 17, 26, 35, 38, 37, 19, 29, 42, 71, 84, 102, 109,
 136, 137, 175, 205, 234, 233, 255, 254, 231, 201, 170, 135,
91, 63, 80, 96, 80, 43, 74, 133, 151, 138, 115, 86, 81, 55,
42, 67, 75, 51, 29, 23, 22, 5, 9, 3, 3, 3, 1, 6, 3, 2, 5, 8,
3, 1, 0},
162      {5, 9, 12, 12, 16, 15, 19, 2, 17, 37, 29, 42, 97, 91, 43,
28, 8, 96, 188, 253, 249, 226, 171, 109, 86, 187, 245, 187,
123, 40, 76, 104, 80, 84, 86, 55, 62, 69, 42, 71, 65, 44, 54,
28, 25, 32, 37, 29, 19, 11, 1, 3, 3, 3, 1, 1, 1, 2, 1, 0, 1},
163      {7, 15, 16, 5, 21, 42, 29, 27, 34, 45, 50, 73, 83, 96,
106, 128, 157, 204, 237, 239, 236, 240, 212, 198, 178, 156,
148, 113, 94, 73, 55, 43, 91, 100, 73, 61, 92, 84, 95, 121,
103, 93, 59, 36, 25, 28, 16, 6, 6, 4, 2, 3, 4, 4, 1, 1, 1, 0,
1, 2, 0},
164      {4, 4, 8, 1, 4, 17, 26, 17, 24, 6, 20, 26, 32, 22, 5, 28,
72, 82, 82, 110, 154, 195, 234, 247, 234, 192, 211, 227, 186,
175, 148, 125, 79, 7, 6, 28, 40, 34, 17, 11, 15, 36, 40, 35,
9, 17, 22, 13, 2, 5, 3, 2, 5, 4, 2, 1, 0, 0, 1, 1, 1},
165      {2, 8, 7, 9, 33, 45, 48, 43, 47, 43, 56, 79, 105, 131,
136, 130, 164, 186, 208, 225, 252, 255, 223, 220, 231, 221,
197, 168, 143, 102, 59, 29, 27, 10, 15, 49, 54, 82, 104, 74,
48, 43, 25, 38, 36, 27, 29, 24, 17, 7, 5, 3, 2, 6, 4, 1, 1, 1,
 2, 1, 0},
166      {0, 1, 3, 8, 7, 7, 10, 6, 8, 11, 12, 25, 36, 30, 16, 23,
30, 36, 31, 28, 47, 94, 111, 145, 169, 190, 195, 218, 247,
235, 191, 183, 197, 154, 128, 114, 100, 109, 112, 95, 50, 13,
20, 32, 39, 17, 10, 13, 5, 1, 1, 1, 3, 1, 1, 2, 1, 0, 1, 0,
1},
167      {9, 12, 7, 5, 35, 48, 34, 41, 43, 36, 50, 54, 73, 91, 101,
 111, 136, 160, 178, 189, 213, 220, 229, 240, 243, 255, 243,
248, 228, 183, 139, 80, 64, 50, 55, 39, 18, 38, 44, 58, 33,
26, 50, 43, 47, 30, 19, 12, 12, 4, 6, 2, 4, 1, 2, 1, 1, 2, 1,
0, 0},
168      {2, 2, 1, 6, 8, 13, 13, 24, 25, 31, 27, 30, 47, 86, 82,
69, 100, 164, 185, 221, 246, 251, 241, 204, 174, 130, 78, 21,
102, 59, 48, 31, 43, 59, 63, 93, 86, 40, 25, 27, 26, 21, 35,
20, 17, 16, 30, 17, 3, 6, 4, 4, 3, 2, 2, 0, 1, 1, 1, 0, 1},
169      {6, 6, 5, 19, 34, 38, 33, 42, 47, 51, 73, 93, 89, 89, 110,
 137, 159, 175, 186, 206, 214, 220, 223, 244, 247, 232, 218,
224, 196, 145, 108, 72, 79, 52, 29, 4, 33, 63, 41, 28, 54, 51,
 48, 34, 35, 7, 9, 15, 10, 4, 3, 5, 4, 3, 0, 3, 1, 0, 0, 0,
0},
170      {2, 1, 6, 12, 30, 40, 35, 25, 24, 48, 48, 64, 103, 76, 73,
 135, 102, 40, 77, 99, 90, 113, 163, 117, 153, 221, 212, 195,
129, 117, 137, 137, 98, 91, 188, 197, 98, 17, 93, 155, 89, 58,
 55, 77, 49, 40, 12, 13, 3, 1, 4, 1, 2, 4, 2, 3, 4, 1, 4, 2,
1},
171  };
```

```
172
173     int i, j;
174     //sonarData
175     for (i = 0; i < NUM_VECTORSX; i++) {
176         for (j = 0; j < NUM_POINTSX - 1; j++) {
177             sonarDataOriginalX[i][j] = sonarDataX[i][j];
178         }
179     }
180     //sonarData class
181     for (i = 0; i < NUM_VECTORSX; i++) {
182         for (j = 0; j < NUM_POINTSX; j++) {
183             if (j == 60) {
184                 vectorsOutOriginalX[i] = sonarDataX[i][j];
185             }
186         }
187     }
188
189
190     /*-Calculates the score.
191      * 1. Checks function (F2) that determines comparison
    operation.
192      * 2. The point is compared with the constant(C).
193      * 3. Find out what class the point belongs to.
194      * 4. Checks against the class that is specified in the data
    set.
195      * 5. If class matches, add +1.
196      */
197     int class0 = 0; // Class 0
198     int class1 = 1; //Class 1
199     int class; //Given class from genom
200     int points = 0; //++ if all score in one row is 1(match)
201     int score[6] = {0, 0, 0, 0, 0, 0};
202     for (i = 0; i < NUM_VECTORSX; i++) {
203         if (FU_0.F2 == 1) {
204             if (sonarDataOriginalX[i][addr[0]] <= C_sum[0]) {
205                 class = 1;
206             } else {
207                 class = 0;
208             }
209             if (class == vectorsOutOriginalX[i] && class == class0
    ) {
210                 score[0] = 1;
211             }
212         } else if (FU_0.F2 == 0) {
213             if (sonarDataOriginalX[i][addr[0]] >= C_sum[0]) {
214                 class = 1;
215             } else {
216                 class = 0;
217             }
218             if (class == vectorsOutOriginalX[i] && class == class0
    ) {
219                 score[0] = 1;
220             }
221         }
222         if (FU_1.F2 == 1) {
223             if (sonarDataOriginalX[i][addr[1]] <= C_sum[0]) {
224                 class = 1;
225             } else {
226                 class = 0;
```

```
227             }
228             if (class == vectorsOutOriginalX[i] && class == class0
    ) {
229                 score[1] = 1;
230             }
231         } else if (FU_1.F2 == 0) {
232             if (sonarDataOriginalX[i][addr[1]] >= C_sum[0]) {
233                 class = 1;
234             } else {
235                 class = 0;
236             }
237             if (class == vectorsOutOriginalX[i && class == class0
    ]) {
238                 score[1] = 1;
239             }
240         }
241         if (FU_2.F2 == 1) {
242             if (sonarDataOriginalX[i][addr[2]] <= C_sum[0]) {
243                 class = 1;
244             } else {
245                 class = 0;
246             }
247             if (class == vectorsOutOriginalX[i] && class == class0
    ) {
248                 score[2] = 1;
249             }
250         } else if (FU_2.F2 == 0) {
251             if (sonarDataOriginalX[i][addr[2]] >= C_sum[0]) {
252                 class = 1;
253             } else {
254                 class = 0;
255             }
256             if (class == vectorsOutOriginalX[i] && class == class0
    ) {
257                 score[2] = 1;
258             }
259         }
260         if (FU_3.F2 == 1) {
261             if (sonarDataOriginalX[i][addr[3]] <= C_sum[0]) {
262                 class = 1;
263             } else {
264                 class = 0;
265             }
266             if (class == vectorsOutOriginalX[i] && class == class0
    ) {
267                 score[3] = 1;
268             }
269         } else if (FU_3.F2 == 0) {
270             if (sonarDataOriginalX[i][addr[3]] >= C_sum[0]) {
271                 class = 1;
272             } else {
273                 class = 0;
274             }
275             if (class == vectorsOutOriginalX[i] && class == class0
    ) {
276                 score[3] = 1;
277             }
278         }
279         if (FU_4.F2 == 1) {
```

```
280             if (sonarDataOriginalX[i][addr[4]] <= C_sum[0]) {
281                 class = 1;
282             } else {
283                 class = 0;
284             }
285             if (class == vectorsOutOriginalX[i] && class == class0
    ) {
286                 score[4] = 1;
287             }
288         } else if (FU_4.F2 == 0) {
289             if (sonarDataOriginalX[i][addr[4]] >= C_sum[0]) {
290                 class = 1;
291             } else {
292                 class = 0;
293             }
294             if (class == vectorsOutOriginalX[i] && class == class0
    ) {
295                 score[4] = 1;
296             }
297         }
298         if (FU_5.F2 == 1) {
299             if (sonarDataOriginalX[i][addr[5]] <= C_sum[0]) {
300                 class = 1;
301             } else {
302                 class = 0;
303             }
304             if (class == vectorsOutOriginalX[i] && class == class0
    ) {
305                 score[5] = 1;
306             }
307         } else if (FU_5.F2 == 0) {
308             if (sonarDataOriginalX[i][addr[5]] >= C_sum[0]) {
309                 class = 1;
310             } else {
311                 class = 0;
312             }
313             if (class == vectorsOutOriginalX[i] && class == class0
    ) {
314                 score[5] = 1;
315             }
316         }
317         if (score[0] == 1 && score[1] == 1 && score[2] == 1 &&
    score[3] == 1 && score[4] == 1 && score[5] == 1) {
318             points = points + 1;
319         }
320         int temp;
321         for (temp = 0; temp < 6; temp++) {
322             score[temp] = 0;
323         }
324     }
325     return points;
326 }
```

Appendix C

h file

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <time.h>

#define NUMBER_OF_POINTS 40
#define X_Y_CLASS 3
#define TEST_POINTS 5
#define SIZE_OF_POP 5
#define MUT_RATE 5
#define MUT_RATE_2 4
#define BIT_ARRAY_SIZE 2
#define CONSTANT_LENGTH 8
#define F1_LENGTH 6
#define NUM_VECTORS 208
#define NUM_VECTORS_HALF 104
#define NUM_POINTS 61


////////New Struct////////
typedef struct {
  //ES parameters
  unsigned long int maxgenerations;
  int popsize;
  int mutations;

  // CGP parameters
  int inputs;
  int outputs;
  int cols;
  int rows;
  int lback;
  int nodeinputs;
  int nodeoutputs;
  int nodefuncs;
}tparams;

/////End NEW Struct////////
struct FU {
```

```
41      int F1;
42      int F2;
43      int C1;
44      int C2;
45      int C3;
46  };
47
48  struct FU2 {
49      int F1[F1_LENGTH];
50      int F2;
51      int C[CONSTANT_LENGTH];
52  };
53
54
55
56  };
```

Bibliography

[1] Intel Corporation. *What is a Microprocessor?* URL: http://download.intel.com/newsroom/kits/40thanniversary/pdfs/What_is_a_Microprocessor.pdf (visited on 11/10/2014).

[2] L.H. Crockett, R.A. Elliot and M.A. Enderwitz. *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Strathclyde Academic Media, 2014.

[3] C Darwin. 'John Murray; London, UK: 1859'. In: *On the origin of species by means of natural selection, or the preservation of favoured races in the struggle for life* ().

[4] Roland Dobai and Lukáš Sekanina. 'Towards Evolvable Systems Based on the Xilinx Zynq Platform'. In: *2013 IEEE International Conference on Evolvable Systems (ICES)*. Proceedings of the 2013 IEEE Symposium Series on Computational Intelligence (SSCI). IEEE Computational Intelligence Society, 2013, pp. 89–95. ISBN: 978-1-4673-5869-9.

[5] Agoston E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. SpringerVerlag, 2003. ISBN: 3540401849.

[6] Lawrence J Fogel, Alvin J Owens and Michael J Walsh. 'Artificial intelligence through simulated evolution'. In: (1966).

[7] Garrison W. Greenwood and Andrew M. Tyrrell. *Introduction to Evolvable Hardware: A Practical Guide for Designing Self-Adaptive Systems (IEEE Press Series on Computational Intelligence)*. Wiley-IEEE Press, 2006. ISBN: 0471719773.

[8] John H Holland. 'Genetic algorithms and the optimal allocation of trials'. In: *SIAM Journal on Computing* 2.2 (1973), pp. 88–105.

[9] Xilinx Inc. *A GENERATION AHEAD FOR SMARTER SYSTEMS: 9 REASONS WHY THE XILINX ZYNQ-7000 ALL PROGRAMMABLE SOC PLATFORM IS THE SMARTEST SOLUTION*. URL: http://www.xilinx.com/publications/prod_mktg/zynq-7000-generation-ahead-backgrounder.pdf (visited on 04/02/2015).

[10] Xilinx Inc. *AXI Reference Guide*. UG761 (v14.3). Nov. 2012. URL: http://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug761_axi_reference_guide.pdf.

[11] Xilinx Inc. *Board support packages (SDK)*. URL: http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_2/SDK_Doc/concepts/sdk_c_bsp_internal.htm (visited on 05/10/2015).

[12] Xilinx Inc. *Intellectual Property*. URL: http://www.xilinx.com/products/intellectual-property.html (visited on 04/11/2014).

[13] Xilinx Inc. *Software Development Kit (SDK)*. URL: http://www.xilinx.com/tools/sdk.htm (visited on 06/08/2014).

[14] Xilinx Inc. *Vivado Design Suite Evaluation and WebPACK*. URL: http://www.xilinx.com/products/design-tools/vivado/vivado-webpack.html (visited on 11/10/2014).

[15] Xilinx Inc. *Vivado Design Suite User Guide: Embedded Processor Hardware Design (UG898)*. (Vivado Design Suite 2014.3-2014.4). Oct. 2014. URL: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_4/ug893-vivado-ide.pdf.

[16] Xilinx Inc. *Vivado High-Level Synthesis*. URL: http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html (visited on 06/08/2014).

[17] Xilinx Inc. *Zynq-7000 All Programmable SoC, Technical Reference Manual*. UG585 (v1.10). Feb. 2015. URL: http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf.

[18] Xilinx Inc. *Zynq-7000 Diagram(1388706340885.jpg)*. URL: http://www.xilinx.com/content/xilinx/en/products/silicon-devices/soc/zynq-7000/_jcr_content/mainParsys/xilinxcolumns_9aa7/column1/xilinximagemodal_331/image.img.jpg/1388706340885.jpg (visited on 04/02/2015).

[19] Xilinx Inc. *Zynq-7000 Silicon Devices*. URL: http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/silicon-devices.html (visited on 10/09/2014).

[20] Andrew Kohlsmith. *fpgafabric.jpg, Modification: number have been added to the original image.* 2012. URL: https://www.mixdown.ca/redmine/attachments/download/84/fpgafabric.jpg (visited on 04/07/2014).

[21] John R Koza. *Genetic programming: on the programming of computers by means of natural selection*. Vol. 1. MIT press, 1992.

[22] Glette Kyrre. 'Design and Implementation of Scalable Online Evolvable Hardware Pattern Recognition Systems'. Ph.D. University of Oslo, 2008.

[23] RTC magazine(PRODUCTS and TECHNOLOGY). *Zynq-7000*. URL: http://rtcmagazine.com/files/images/3339/rtc1204pr_xil_fig01_medium.jpg (visited on 04/02/2015).

[24] Rouse Margaret and Stephen J. Bigelow Matthew Haughn. *ARM processor*. URL: http://whatis.techtarget.com/definition/ARM-processor (visited on 01/01/2015).

[25] Tomas Martinek and Lukas Sekanina. 'An Evolvable Image Filter: Experimental Evaluation of a Complete Hardware Implementation in FPGA'. In: *Lecture Notes in Computer Science* 2005.3637 (2005), pp. 76–85. ISSN: 0302-9743.

[26] Julian F. Miller. 'Cartesian Genetic Programming'. In: *Cartesian Genetic Programming*. Ed. by Julian F. Miller. Natural Computing Series. Springer Berlin Heidelberg, 2011, pp. 17–34. ISBN: 978-3-642-17310-3.

[27] Julian F Miller and Stephen L Smith. 'Redundancy and computational efficiency in cartesian genetic programming'. In: *Evolutionary Computation, IEEE Transactions on* 10.2 (2006), pp. 167–174.

[28] Julian F Miller and Peter Thomson. 'Cartesian genetic programming'. In: *Genetic Programming*. Springer, 2000, pp. 121–132.

[29] Julian F Miller, Peter Thomson and Terence Fogarty. *Designing electronic circuits using evolutionary algorithms. arithmetic circuits: A case study*. 1997.

[30] ARM Official product information page. *Cortex-A9 Processor*. URL: http://www.arm.com/products/processors/cortex-a/cortex-a9.php (visited on 01/09/2014).

[31] Robert R. Schaller. 'Moore's Law: Past, Present, and Future'. In: *IEEE Spectr.* (), pp. 52–59.

[32] Lukás Sekanina and Stepan Friedl. 'An Evolvable Combinational Unit for FPGAs.' In: *Computers and Artificial Intelligence* 23.5 (2004), pp. 461–486.

[33] Zdenek Vasicek and Karel Slany. 'Efficient Phenotype Evaluation in Cartesian Genetic Programming'. In: *Proceedings of the 15th European Conference on Genetic Programming, EuroGP 2012*. Ed. by Alberto Moraglio et al. Vol. 7244. LNCS. Malaga, Spain: Springer Verlag, Nov. 2012, pp. 266–278.

[34] Vesselin K Vassilev and Julian F Miller. 'The advantages of landscape neutrality in digital circuit evolution'. In: *Evolvable systems: from biology to hardware*. Springer, 2000, pp. 252–263.

[35] N.H.E. Weste and D.M. Harris. *Integrated Circuit Design*. Pearson Education, Limited, 2011. ISBN: 9780321696946.