AUTOMATIC DATAPATH AND CONTROLLER GENERATION FOR RECONFIGURABLE ASIP

by

Ender Çulha

B.S., in Electrical and Electronics Engineering, Dokuz Eylül University, 2009

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Electrical and Electronics Engineering
Boğaziçi University
2013

ACKNOWLEDGEMENTS

I would like to thank to my supervisor Prof. Ömer Cerid for his support and guidance during this thesis work. I would like to thank my co-supervisor Assoc. Prof. Arda Yurdakul for her guidance, her motivational support and for sparing a huge amount of her time and energy for this thesis.

I would like to thank Asst. Prof. Faik Başkaya for his attendance to the weekly meetings and for his visionary contributions. I would also like to thank Prof. Necati Aras and Assoc. Prof. Şenol Mutlu for their patience and feedback during the thesis examination.

I would like to thank Alp Malazgirt for his help, his support and spending his weekends with me in the laboratory to help me. I would also like to thank Salih Bayar and other CASLAB team members for their help, support and motivation for this thesis.

I would like to thank my family for their support in this thesis work as well as all steps of my life. I would like to thank my father, Erdoğan Çulha, for his insistence on me for applying to master program. I thank to my mother, Adile Çulha, for her endless support and being with me anytime I need. I would like to thank my sister, Pelin Çulha, for her motivational support and being such a nice housemate.

I would also like to thank my friends Feyyaz Melih Akçakaya and Engin Afacan in BETA Lab for their help and friendship, Tuğba Kınaoğlu, Samet Saygılı, Ali Dinçer Dere, KAPSCH team and all my other friends who supported and motivated me during this thesis and listen to me whenever I need.

ABSTRACT

AUTOMATIC DATAPATH AND CONTROLLER GENERATION FOR RECONFIGURABLE ASIP

The need for complex designs that meet the desired application specific criteria and time-to-market pressure increase the importance of High Level Synthesis (HLS) tools, which take high level behavioral representation of the desired functionality as the input and generate HDL description of hardware at RTL level for FPGA or ASIC targets. FPGAs are getting more popular than ASICs and microprocessors due to their architectural flexibility, on-site upgradability and computing power. In this thesis, an HLS tool for FPGAs is proposed. This tool has the following capabilities: (i) generation of optimized RTL which consists of datapath and its controller. To achieve this, the tool extracts the clock period of the optimized RTL by using the optimization results and the delay models of the arithmetic operators. (ii) generation of Golden RTL where there is no optimization and resource sharing on the datapath. (iii) estimation of delay and area of the generated RTL specifications by using the estimation models. This tool is integrated in RH(+) Design Automation Framework. The generated RTLs are tested in Xilinx Spartan-3 FPGA. The estimated delay and area of both the Golden RTL and Optimized RTL generated by the tool are compared with the results of Xilinx ISE tool set for different input applications.

ÖZET

YENİDEN BETİMLENEBİLİR UYGULAMAYA ÖZGÜ İŞLEMCİLER İÇİN VERİYOLU VE KONTROL BİRİMİ TASARIM OTOMASYONU

Uygulamaya özgü kriterleri karşılayan karmaşık tasarımların gerekliliği ve pazara giriş baskısı, Yüksek Seviyeden Üretim (YSÜ) araçlarına olan ihtiyacı arttırmaktadır. YSÜ araçları istenilen işlevin yüksek seviyede davranış tanımını girdi olarak alıp, donanımın Yazmaç Almacı Seviyesinde (YAS) Donanım Tanımlama Dili(DTD) tanımlarını çıktı olarak verir. YSÜ araçları, Alanda Programlamalı Kapı Dizilerini (APKD) veya Uygulamaya Özgü Tümdevreleri (UÖT) hedeflemektedir. APKD'ler, mimari esnekliği, alanda güncellenmeye uygunluğu ve hesaplama güçlerinden dolayı UÖT ve mikroişlemcilerden daha çok ilgi görmektedir. Bu tez çalışmasında, APKD'ler için YSÜ aracı önerilmektedir. Bu aracın yetenekleri şu şekildedir: (i) Veriyolu ve kontrol birimini içinde barındıran eniyilenmiş YAS üretimi. Bunu yapabilmek için, araç eniyileme sonuçlarını ve aritmetik işleçlerin gecikme modellerini kullanarak saat periyodu süresi çıkarılmaktadır. (ii) Veriyolu üzerinde kaynak paylaşımı ve eniyileme yapılmamış Altın YAS üretimi (iii) Üretilen YAS tanımlamalarının, kestirim modelleri kullanılarak, gecikme ve alan kestirimi. Geliştirilen araç RH(+) Tasarım Otomasyonu çerçevesine eklenmiştir. Üretilen YAS'lar Xilinx Spartan-3 APKD'leri kullanılarak test edilmiştir. Araç tarafından üretilen YAS'ların gecikme ve alan kestirimleri, farklı girdiler için Xilinx ISE aracının kestirim sonuçlarıyla karşılaştırılmıştır.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

a_i	ith Bit of the Signal 'a'
$a_{i.z}$	ith Bit of the Group z of the Signal 'a'
$a_{k-1:0}$	k-1 to 0 bits of the Signal 'a'
b_i	ith Bit of the Signal 'b'
$b_{i.z}$	ith Bit of the Group z of the Signal 'b'
$b_{k-1:0}$	k-1 to 0 bits of the Signal 'b'
c_{in}	Input Carry
$c_{i.z}$	Input Carry of the Group z
$cc_{i.z}$	Output Carry of the Group z
c_{out}	Output Carry
mW	miliWatt
ns	nanosecond
P_i	ith bit of Propagate
$P_{i.z}$	ith Bit of the Group z of the Propagate
$P_{i.z+z-1:iz}$	Propagate of the Group z
$P_{k-1:0}$	k-1 to 0 bits of the Propagate
S_i	ith bit of Sum
$S_{k-1:0}$	k-1 to 0 bits of the Sum
t_{D,RCA_8}	Delay of a 8-bit RCA adder
$t_{fulladder}$	latency of full adder
t_{mux}	latency of multiplexer

LIST OF ACRONYMS/ABBREVIATIONS

ALAP	As Late As Possible
AR	Auto Regression
ASAP	As Soon As Possible
ASIP	Application Specific Instruction Set Processor
ASIC	Application Specific Integrated Circuit
CDFG	Control Data Flow Graph
CLA	Carry Lookahead Adder
CLB	Configurable Logic Block
CPU	Central Processing Unit
CSKA	Carry Skip Adder
CSLA	Carry Select Adder
DAG	Directed Acyclic Graph
DCM	Digital Clock Manager
DCT	Discrete Cosine Transform
EDA	Electronic Design Automation
EW	Elliptic Wave
FA	Full Adder
FF	Flip Flop
FPGA	Field Programmable Gate Array
FRH(+)	Framework for RH(+) Model
HBD	Hardware Behavior Description
HDL	Hardware Description Language
HLS	High Level Synthesis
IC	Integrated Circuit
ILP	Integer Linear Programming
IOB	Input Output Block
IIR	Infinite Impulse Response
LRH(+)	Language for RH(+) Model

LSB	Least Significant Bit
LUT	Look Up Table
MIMOLA	Machine Independent Microprogramming Language
MSB	Most Significant Bit
MUX	Multiplexer
NuSMV	Extension of SMV
PRESS	Predicted Error Sum of Squares
RAM	Random Access Memory
RCA	Ripple Carry Adder
RH(+)	Hardware Software Co-design on Reconfigurable Hardware
RTL	Register Transfer Level
SMV	Symbolic Model Verifier
TCL	Tool Command Language
UCF	User Constraint File
VLSI	Very Large Scale Integrated Circuit
XCO	Xilinx Core Generator Parameter File

1. INTRODUCTION

Increasing complexity of the Digital Signal Processing algorithms, the need for complex designs that meet the desired application specific criteria and time to market pressure increase the importance of High Level Synthesis tools. High Level Synthesis (HLS) is the automated generation of the hardware of a digital system after taking the behavioural description of the required system, a set of constraints, and the goals to be satisfied. HLS tools have major importance in both design and verification processes of the digital systems. In terms of design process, it provides shorter design time and better performance by searching design space and making optimizations. In terms of verification process, it provides the functional verification of the digital system in early stages of the design so as to ensure that design is with fewer errors.

HLS tools are firstly developed for ASICs since the history for HLS tools is longer than the FPGAs. However, FPGAs are becoming more and more popular due to their reconfigurable architecture compared to ASIC and their computational density advantage over general microprocessors for certain applications. On the other hand, application developers are more familiar with high level languages such as C, C++ than Hardware Description Languages (HDLs) such as Verilog and VHDL. The increasing popularity of the FPGAs and inexperience of the application developers on HDLs create the necessity of the FPGA-specific HLS tools.

The purpose of this thesis is to propose and design an HLS tool for the RH(+) Design Automation Framework [2]. RH(+) takes an application in a high level language and converts it to Control Data Flow Graphs (CDFGs). These graphs are used as the input to the proposed synthesis tool. The capabilities of the tool are listed below;

- It generates the optimized RTL which consists of datapath and its controller based on the output of the optimization tool proposed in [3]. The optimization tool uses a mathematical model to solve the multiplexer and resource-scheduling binding problem. This mathematical model is solved under minimum latency and minimum area constraints to obtain optimized CDFG. Optimization tool

gives which operators in the CDFG share the same functional datapath unit, the instance number and starting times of the operators in the CDFG as the output. The implementation of the optimization tool is out of the scope of this thesis.

- It generates Golden RTL where there is no optimization and resource sharing on the datapath. In Golden RTL, no two operators share the same functional unit. For that reason, generated datapath is a fully combinational circuit where there are not any multiplexers and a controller. Golden RTL is used for the verification of the optimized RTL by comparing the simulation results of the generated Optimized RTL with the generated Golden RTL.
- It extracts clock period of the Optimized RTL by using the optimization results and the delay models of the arithmetic operators. The optimization tool schedules the operations by only using the delay costs of the operations. For datapath generation, a clock period is extracted and the operations are rescheduled based on this clock period. This is done so as to achieve the synchronous behaviour of the Optimized RTL.
- It estimates delay and area of the generated RTL specifications by using the estimation models of the operators.

1.1. Related Work

1.1.1. High Level Synthesis

Until late 1960s, ICs were designed and optimized manually. Early researches about HLS started in 1970s when the only commercial EDA industry products were physical layout machines. In [4], Barbacci noted that one can compile the instruction set processor specification into hardware from a high-level language specification. This study was on design specification, simulation, and synthesis at both RTL and algorithmic level.

Between 1980s and early 1990s many basic concepts of early HLS such as scheduling and resource allocation were researched. In [5] and [6], the basic algorithms for scheduling; As Soon As Possible (ASAP) and As Late As Possible (ALAP) are investigated. These studies were followed by a number of heuristics that used metrics

such as urgency [7] and mobility [8] to schedule operations. Studies in resource allocation aim reducing registers, functional units, wire and interconnection costs [5]. Later, researchers focused on performing scheduling, resource allocation and binding task in parallel by using Integer Linear Programming(ILP) [9] to [11]. There were also studies for developing HLS tools. In [12], an HLS tool that aids the design of a digital processor with a top-down method is presented by Peter Marwedel. This tool uses the MIMOLA as an input design language that may be a high level functional or a structural description of the hardware. The output of this tool is the description of the hardware at the block level. In [13], a silicon compiler that takes an algorithmic description of a circuit, performs logic synthesis, optimization, and physical layout synthesis is presented. The difficulty of the input languages for the tools and ineffective results because of simple architectures, expensive allocation and primitive scheduling led HLS tools to fail in commercialization during these years.

The time between middle 1990s and early 2000s is the period which major EDA companies have offered their commercial HLS tools. In 1996, Synopsys announced Behavioral compiler which generates RTL implementations from behavioral hardware description language and connects to the downstream tools [14]. Moreover, Cadence offered the Visual Architect tool [15] and Mentor Graphics offered Monet tool [16]. These tools received considerable interest but they were not successful commercially because behavioral HDLs were the inputs for that tools. Moreover, they have not attracted algorithm developers who use high level languages nor RTL designers because of the lack of improvement in quality of area, performance results.

Since 2000, a new generation of HLS tools has been developed by academia and the industry. Unlike the previous tools, high level languages such as C/C++ and Matlab have become the input languages. So, HLS tools are accepted by the algorithm designers who are not familiar with the HDLs. Mentor Catapult C Synthesis tool [17], Forte Synthesize [18], AutoESL's AutoPilot [19], Cadence's C-to-Silicon Compiler [20], NEC's Cyber Workbench [21] are the examples of HLS tools that use C as input language. Moreover, there are HLS tools that use other languages for instance, Esterel Technologies [22] uses graphical state machine capture and Esterel language, and some of the tools use Matlab and Simulink [23] as input language. With the increasing use

of the FPGAs, HLS tools that are targeted to FPGAs are developed. [24] In 2012, Xilinx announced Vivado HLS tool that use a C-based language as input language. Modern C-based HLS tools make some language extensions and restrictions to provide C input more suitable with hardware. HardwareC [25], Handel-C [26], SystemC [27] are the examples for those languages. HLS has been more successful with the use of FPGAs, high level languages as input languages, and more sophisticated algorithms yielding better designs that meet user specifications.

1.1.2. Delay, Area, Power Estimation Models

Shrinking time-to-market and short life-time of the products increase the necessity of early estimation of the design properties of the applications. Common design properties are area, performance and power consumption. HLS tools that estimate these metrics from a high level algorithmic behavior are efficient for ASIC targets. With increasing use of the FPGAs, early estimation models for delay, area, and power consumption for FPGAs have emerged.

Commercial tools such as Xilinx [1] and Altera [28] can estimate these metrics by going through the steps of synthesis, placement, and routing, which can take minutes to hours depending on the application. So, it is clear that early estimation tools, which can be be integrated into HLS tools targeting to FPGAs are highly necessary. For that reason, current studies are focused on HLS tools targeting FPGAs. Hence, accurate modeling of delay, area, and power performances of the resources is necessary.

There exist estimation models for HLS Tools targeting FPGAs. A method of area estimation model for Look-Up-Table (LUT) based FPGAs is proposed in [29]. This paper presents accurate area estimation results but does not present any delay estimation model. The method in [30] develops delay and area model at the Data flow graph (DFG) level. Unfortunately, the number of operations that are supported are limited and estimation error is high for some applications. In [31], area, time, and power models that are integrated to FANTOM design automation tool are given for Xilinx IP core.

1.1.3. Verification

HLS tools are efficient in decreasing design effort and time while satisfying the requirements in latency, area, and power consumption. After the design phase of a digital system, its verification phase begins. Verification consists of acquiring a reasonable confidence that the designed system will function correctly without major timing errors.

Given a high-level algorithm, an HLS tool goes through several interdependent processes, such as specification, scheduling, allocation, optimization etc. For that reason, an HLS tool can be implemented with hundreds of thousands code writing. As new techniques are developed for HLS and the application gets more complex, verification becomes a must. FPGAs are widely used for designing and prototyping of DSP algorithms and it is important to catch bugs in early design phase because they may cause expensive costs after fabrication.

For modern circuits, the verification process requires between 60% and 80% of the design cycle time and up to 80% of the overall design costs [32]. For that reason, verification methodologies for HLS are highly crucial. HLS verification methodologies can be investigated in four categories. These are high-level property checking, translation validation, RTL verification and RTL property checking.

High level property checking allows different kind of properties to be verified at the high level. It can be done either on the high level input description or on the intermediate representation before RTL generation (such as CDFG) of the application design. It verifies design by controlling whether that the design satisfies a given property such as functional behaviour, absence of deadlocks, safety properties. There are two kinds of high level model checking: explicit and symbolic model checking. In explicit model checking, the reachable states of a design are generated using an exhaustive search algorithm. This technique explicitly stores the entire state space in memory and checks if certain error states are reachable. However, this technique suffers as the finite state spaces grow larger [33]. Symbolic model checking techniques store sets of the explored states symbolically by using characteristic functions with canonical structures (such as Binary Decision Diagrams) and traverse the state space

symbolically by exploring a set of states in a single operation [34]. In [35], a verification platform based on symbolic model checking which is applied on NEC's CyberWorkBench is presented.

High level property checking verifies the design by checking the certain properties are satisfied or not. Translation validation verifies that whether these properties are preserved during the synthesis stage. Translation validation is the verification method whether transformation performed by the HLS tool is equivalent to its initial source versions. In [36], a translation validation tool that validates the SPARK HLS tool is presented. This tool takes the Intermediate Representation (IR) produced by the SPARK HLS tool as the input before resource binding and uses bisimulation relation approach to prove equivalence. Bisimulation is a binary relation between state transition system and it relates the points in the input specification to the points in the implementation program in order to show that implementation is equivalent to the specification.

RTL verification is the verification of the generated RTL whether it fits the behavioral description at the high level. In literature, several methodologies for the RTL verification are proposed. In [37], an algorithm for the verification of two sequential circuit descriptions at the same or different levels of abstraction based on equivalence checking is presented. Equivalence checking process is a part of electronic design automation (EDA), used to prove that two representations of a circuit design exhibit exactly the same behavior.

RTL verification is one of the most challenging activities in digital system development: as of today, it is still carried on mostly with ad-hoc tests, scripts and tools developed by the design and verification teams specifically for the current design. Moreover, verification methodology still lacks any standard or even a commonly accepted approach, with the consequence that each hardware engineering team has its own distinct verification practices which even change with subsequent designs by the same team. For that reason, verification with RTL property checking has received an increasing interest. The common trait of these techniques is the attempt to provide some type of mathematical proof that a design is correct, thus guaranteeing that some aspect or property of the circuit behavior holds under every circumstance, and thus its

validity is not limited only to the set of test patterns that have been checked. In [38] and [39], verification is done by using symbolic model checking for both the generated RTL and high-level representation, and comparing the output results of both representation.

1.2. Motivation

The aim of this thesis is to design and implement an RTL generator tool for the RH(+) HLS tool [2] with early estimation of delay and area capabilities. Moreover, the verification is provided by a methodology based on generating the golden RTL reference model for the synthesis tool. The block diagram of the proposed system is given in Figure 1.1.

In Figure 1.1 white blocks represent the modules which are implemented in the scope of this thesis and gray blocks represent the modules that have already been implemented. The gray modules either receive their inputs from or feed their outputs to the remaining modules. These are explained in as follows;

- The *Application* described with High Level Languages such as C or LRH(+) is the input of the system. LRH(+) is the input language of the RH(+) electronic design automation tool [2]. These algorithms are converted to Control Data Flow Graphs (CDFGs) with the RH(+) compiler. CDFG is the intermediate form that represents the algorithm. CDFG is used as input for the remaining blocks of the system.
- *Node Analysis* is the block where arithmetic operators for FPGAs are investigated and selected for RH(+) library. Delay and area models of these operators are also created in this block. In this block, the vertices in the CDFG are mapped to the arithmetic operators selected from the RH(+) library. After node analysis block, the tool has two paths to go as as depicted in Figure 1.1.
- *Performance and Area Estimation of the Golden RTL* is the block which estimation of delay and area of the application graph is done.
- *Golden RTL Generation* is the block where RTL files without any optimization

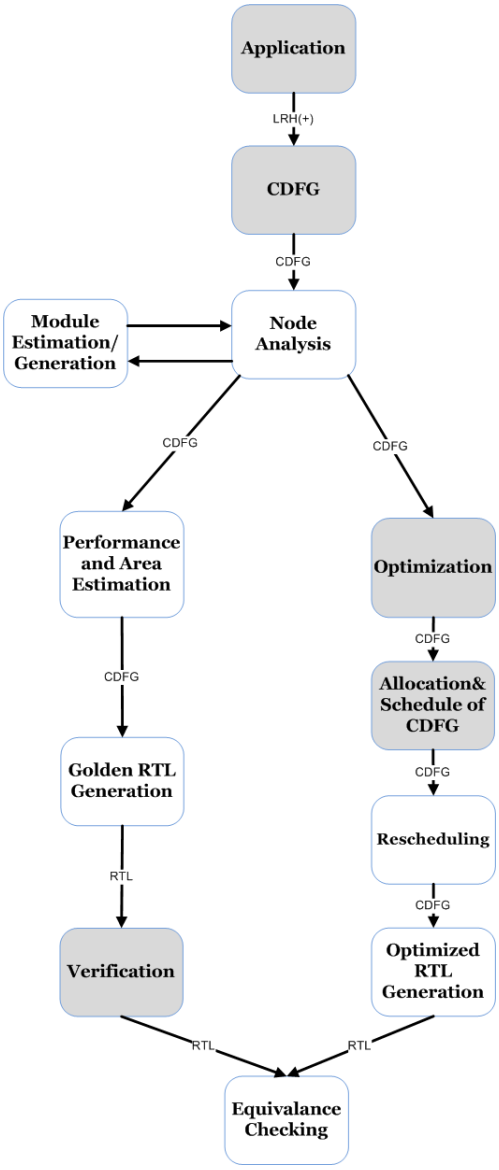

Figure 1.1. Block Diagram of the system.

and resource sharing are generated. Necessary VHDL files are generated and synthesized on Xilinx FPGAs.

- *Verification* consists of the verification of the golden RTL with symbolic model checking. As symbolic model checker NuSMV [40], which takes a text consisting of program description of a model and some specification is used. The models created from application CDFG and generated golden RTL are given as input to the symbolic model checker and same properties are checked for both of the inputs. The verification is done by comparing the responses of the symbolic model checker for both inputs.
- *Optimization* solves the HLS problem by solving a mathematical model which is formed by taking delay and area models of operations into account [3]. In this block, a mathematical model is used to solve the multiplexer and resource scheduling binding problem. Model is solved with minimum latency under minimum area constraint. The outputs of this block is used in Rescheduling and Datapath Generation Blocks.
- *Rescheduling* extracts the clock period of the optimized graph and reschedules the operations for that clock period. Operator sharing information, selected operator types, start time of the vertices, input port size of the multiplexers and vertices are taken as input from the optimization block. Rescheduling Block uses that information to extract the best clock period for the application graph.
- *Optimized RTL Generation* is the generation of the Datapath, Controller and Register files of the optimized and rescheduled graph. The output of this block is the VHDL files which correspond to the datapath, controller and register files of the optimized RTL. This block also produces a TCL file [41] which synthesizes the VHDL files for Xilinx FPGAs.
- In *Equivalence Checking*, generated datapath is verified by comparing the simulation results of the generated Golden RTL with generated Datapath.

2. BACKGROUND

This chapter firstly provides an overview of the RH(+) design automation model, because the proposed system in this thesis is based on this model. Secondly, it gives information about FPGA architecture, because the proposed automation system targets FPGAs.

2.1. RH(+) Design Automation System

Reconfigurable devices have a significant place in embedded design domain. Reconfigurable devices such as FPGAs, offer ten times benefit in computational density over microprocessors, and often another potential ten times improvement in yielded functional density on low granularity operations [42]. RH(+) is a design environment that is capable of handling the run-time reconfigurable processors as well as other types customizable soft processors and hard processors. It consist of LRH(+) which is a language that allows embedded system to be designed with a single language, FRH(+) which is a framework that satisfies the basics set of the RH(+) model and a compiler which retargets itself automatically by using the templates generated by the instruction selection tool [2].

2.1.1. RH(+) Model

RH(+) model is proposed for embedded system design on run-time reconfigurable architectures like FPGAs that support dynamic reconfiguration. This model has the following properties;

2.1.1.1. Abstraction of Low-Level Details for Instruction Set. In RH(+), user should not deal with the details of the instruction due to the flexibility of the target hardware. Smallest instruction corresponds to an operator, operators can be combined to generate complex instructions.

2.1.1.2. Flexible Operator Definition. The user has the freedom to define the operators. They may define the operators by selecting an operator from an already-designed operators library or writing the operator with HDL or by writing the behavior of the operator with LRH(+). Flexibility on operators strengths the flexibility offered by reconfigurable hardware.

2.1.1.3. Flexible Data Types. For reconfigurable systems, having custom variables for each variable causes low utilization of the underlying architecture. In RH(+) Model, user has the freedom to enter the size of each variable separately, in bits.

2.1.1.4. Constraint Settings. An embedded design has to meet several constraints like area, time and energy etc. Therefore, RH(+) model support entry of global and local constraints.

2.1.2. FRH(+)

FRH(+) is the framework implementation of the RH(+) model. It consists of three following levels;

- In *Design Entry Level* embedded system designer defines the application-specific environment, i.e Board Support Package, and Operator Definitions. Based on these specifications, configuration data is generated automatically. Then the user can develop the application using the configuration data and LRH(+) language.
- In *Design Automation Level* the application and the user defined constraints are handled so as to map the designed system to the reconfigurable architecture requirements. The optimized CDFG is generated at this level. The CDFG is used for instruction selection, instruction set generation and module selection. The CDFG is compressed after the instruction set generation and nodes are replaced with instruction selections.
- In *Outputs Level* The processor architecture, instruction and data memory files are given as output.

2.1.3. LRH(+)

LRH(+) is the input language of the RH(+) EDA tool, which facilitites sequential and concurring programing in the same environment. It includes traditional programming constructs and constructs special to Custom Hardware at the same time. In LRH(+), each statement in the code can be operated concurrently. Data dependency dictates the sequential behavior between the statements. However, "?" operator can be used to force sequential behavior between two data-independent statements.

LRH(+) has a syntax different than the traditional languages used for the embedded system programming which is depicted in Table 2.1. This syntax provides that each operator can have more than one operands. RH(+) framework also permits the designer to enter input using C language in addition to LRH(+). The system proposed

Table 2.1. LRH(+) syntax vs traditional programming language syntax.

Traditional Syntax	LRH(+) Syntax
c = a+b;	.=(c,.+(a,b));
e = d/c*4	.=(e,.*(./d,c),4));

in this thesis is layered on the RH(+) framework and takes the application CDFG after the application design which may be entered either LRH(+) or C is converted to the CDFG as shown in the Figure 1.1. CDFG is the intermediate representation of the applications for the RH(+) HLS tool. A sample LRH(+) code is shown in Table 2.2 and its corresponding CDFG is shown in Figure 2.1.

Table 2.2. LRH(+) Code sample.

.=(e,.*(a,b));
.=(f,.*(c,d));
.=(out,.+(e,f));

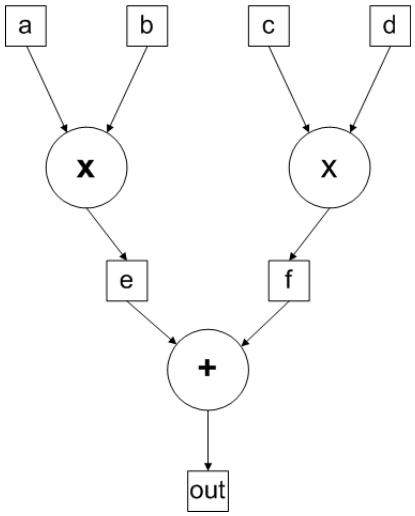

Figure 2.1. Corresponding CDFG of the LRH(+) code in Table 2.2.

2.2. Overview of FPGAs

FPGAs are integrated circuits that can be programmed to implement any digital circuit. FPGAs contain programmable logic components which emulate the functionality of basic logic gates and more complex arithmetic functions. A hierarchy of programmable interconnects allows the logic blocks of an FPGA to be connected as desired by the system designer. With these blocks and interconnects, FPGAs can be programmed to perform any logical function that an application-specific integrated circuit (ASIC) can perform. FPGAs are desirable for implementation of digital systems due to their flexibility and programmability.

When compared to ASIC, FPGAs offer the advantages of rapid prototyping, shorter time to market, reconfigurability, the ability to re-program in the field for debugging. When compared to microprocessors, FPGAs have the advantages of higher performance for especially applications with high computing density and lower power consumption, flexibility and upgradability. In FPGAs, the processing paths are in parallel so different operations do not have to compete for the same processing units as in microprocessors, moreover dedicated structures in FPGAs to implement arithmetic units such as adders, multipliers increases the performance of speed.

FPGAs are used for wide variety of applications including digital signal processing, ASIC prototyping, medical imaging, computer vision, high performance comput-

ing, cryptography, aerospace applications, consumer electronics, telecommunications etc.

2.2.1. FPGA Architecture

As shown in Figure 2.2, an FPGA is a two dimensional array of programmable logic blocks that are connected through a configurable interconnection fabric. The programmable logic resources can be configured to implement any logic function. FPGAs use Look-up tables (LUTs) for implementing combinational logic functions and flip-flops for implementing sequential logic. The structure of a general logic block is shown in Figure 2.3. Altera [28] and Xilinx [1] are the two popular FPGA vendors that currently dominates the market.

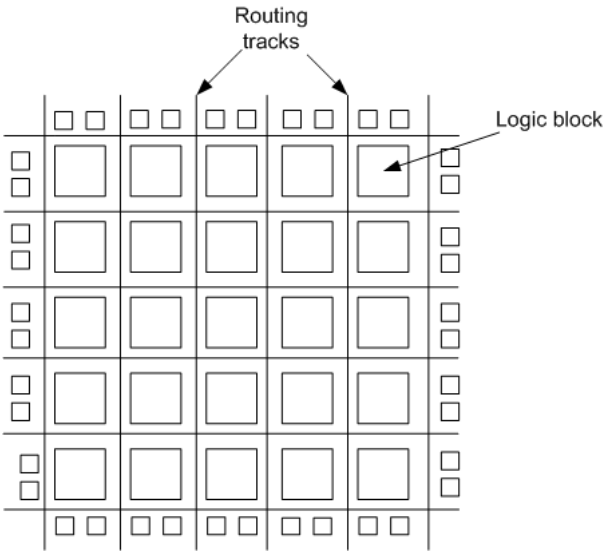

Figure 2.2. Abstract FPGA Architecture.

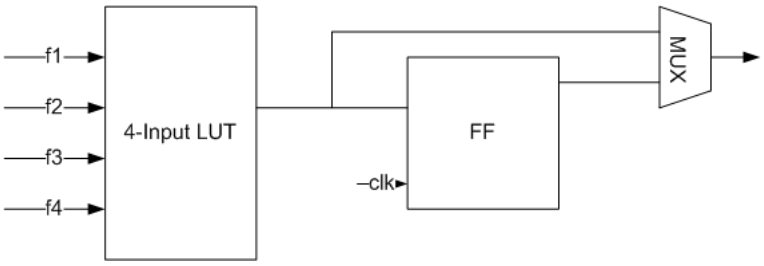

Figure 2.3. Logic Block.

In this thesis, Xilinx FPGAs are used as the target platform, so architecture of Xilinx FPGAs is explained in the text in more detail. Xilinx FPGAs consist of following blocks below;

- Configuration Logic Blocks(CLBs) which consist of LUTs, flip-flops and multiplexer to implement logical functions
- Input/Output Blocks (IOB) which control data flow between I/O pins and internal logic
- Block and Distributed RAMs to store data
- DSP blocks which are used for the arithmetic functions
- Digital Clock Manager (DCM) which is used for the distribution, division, multiplication of clocks

These functional blocks are shown in the Figure 2.4.

Configurable Logic Blocks (CLB) are the fundamental programmable functional elements of the FPGAs. CLBs constitute the main logic resource for implementing synchronous and combinational circuits. Each CLB element is connected to a switch matrix to access to the general routing matrix.

Figure 2.5 shows the CLB arrangement of Xilinx Spartan-3 and Virtex-4 FPGA families. A CLB element contains four interconnected slices. These slices are grouped in pairs. Each pair is organized as a column. SLICEM indicates the pair of slices in the left column, and SLICEL designates the pair of slices in the right column.

Each slice in a CLB contains two LUTs to implement logic, two dedicated storage

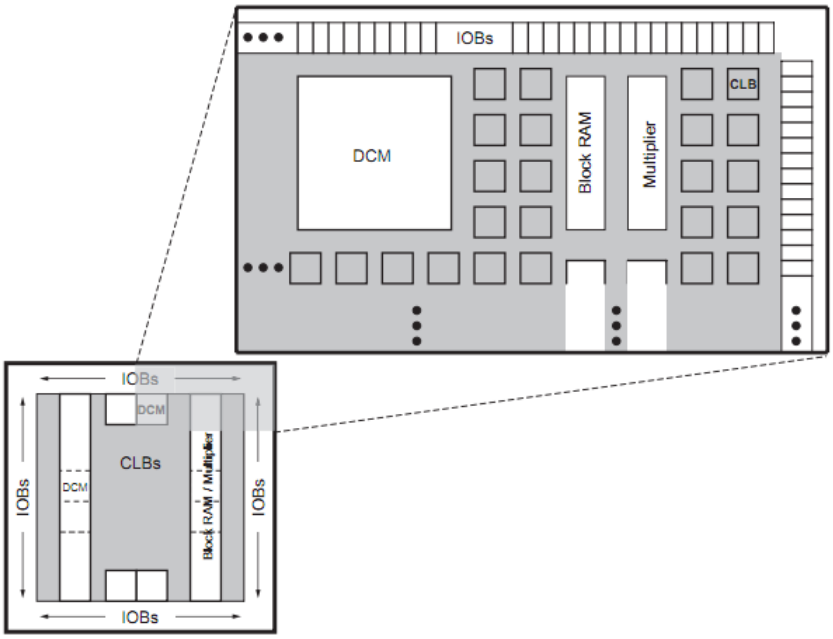

Figure 2.4. Architecture of Xilinx FPGA [1].

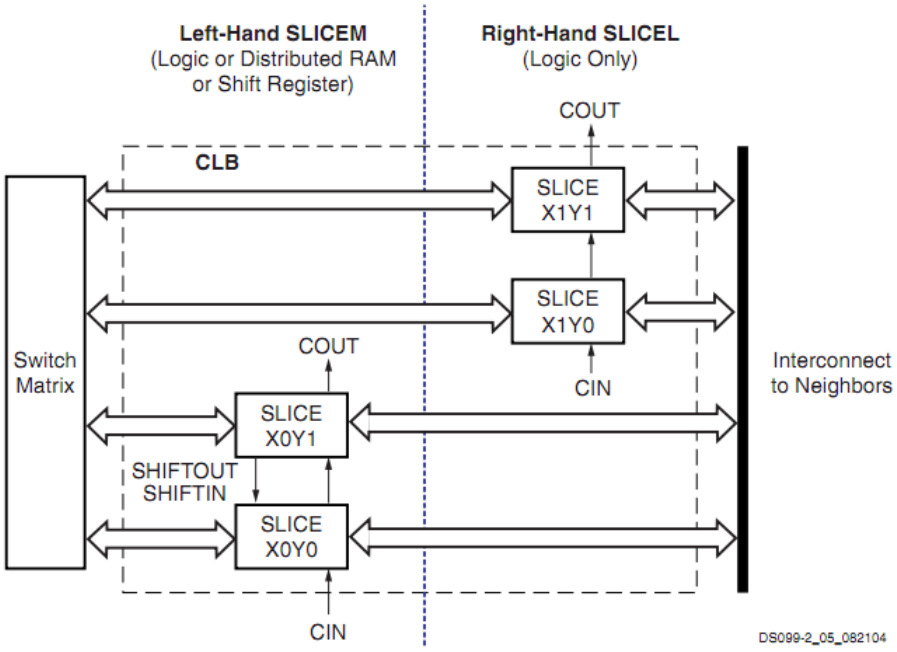

Figure 2.5. CLB configuration of Spartan-3 [1].

elements that can be used as flip-flops or latches. Additional multiplexers and carry logic simplify wide logic and arithmetic functions. Moreover, SliceM supports two additional functions, for storing data using distributed RAM and shifting data with Shift Registers (SRLs). Slice structure for Xilinx Spartan-3 FPGA is shown in Figure 2.6 as an example. These slices are grouped in a pair and each pair is organized as a column with an independent carry chain.

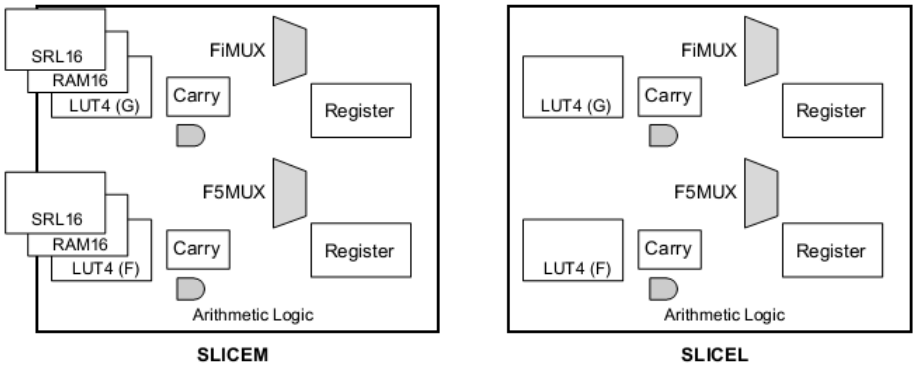

Figure 2.6. Slice Structure of Spartan-3 [1].

LUTs in Xilinx Spartan-3 and Virtex-4 FPGA families have 4 inputs. Multiplexers in the slice (MUXF5 and MUXFX) can be combined up to 8 function generators to provide any function of five to eight inputs in a CLB. Moreover, these LUTs provide the implementation of 4-to-1 MUX in one slice and 16-to-1 MUX in one CLB.

Figure 2.7 shows the CLB arrangement of Xilinx Virtex-5, Virtex-6 and Spartan-6 FPGA families. For each CLB, slices in the bottom of the CLB are labeled as SLICE(0), and slices in the top of the CLB are labeled as SLICE(1). Every slice contains four LUTs, four storage elements, wide-function multiplexers, and carry logic. These elements are used by all slices to provide logic, arithmetic, and ROM functions.In addition to this, some slices support two additional functions: storing data using distributed RAM and shifting data with registers.

In Virtex-5, Virtex-6 and Spartan-6, LUTs have 6 input so one slice can implement any 6 bit input boolean function. Some Slices have two outputs so one LUT can implement one 6 bit input function and two 5 bit input function. In addition to basic

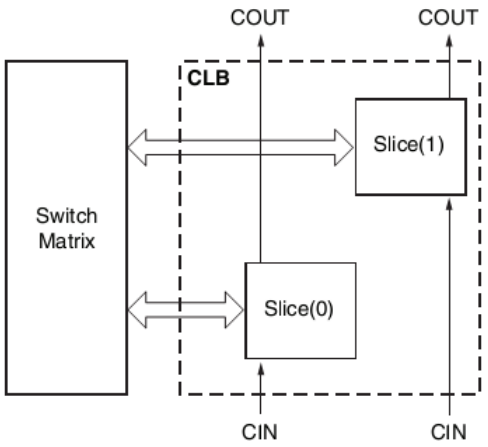

Figure 2.7. CLB configuration of Virtex-5 [1].

LUTs, slices contain 3 multiplexers. These multiplexers are used to combine up to four function generators to provide any function of seven or eight inputs in a slice. Functions with more than eight inputs can be implemented using multiple slices. Three multiplexers in the slice structure provides the implementation of 16-to-1 multiplexer in one slice.

2.2.1.1. Dedicated Structures for Arithmetic Operators. Xilinx FPGAs have dedicated logic to implement arithmetic operators efficiently. In slice structure, there is a carry multiplexer which is used for the implementation of adders. In Figure 2.8, the implementation of one bit adder on the Xilinx FPGA is shown. These adders are cascaded for implementing N bit adders.

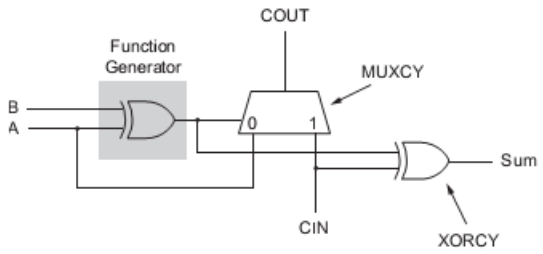

Figure 2.8. Dedicated Logic for Adders [1].

One-bit full adder can be implemented with one LUT, one carry multiplexer (MUXCY) and 1 XOR gate. LUT is programmed to implement XOR gate. The

output of the LUT corresponds to the *propagate* signal.

$$Propagate : P = A \oplus B \tag{2.1}$$

Sum is calculated by XORing the *propagate* and input carry, i.e, *Cin*, signals.

$$Sum : S = P \oplus C_{in} \tag{2.2}$$

Output Carry, *Cout* is calculated by using the carry chain multiplexers (MUXCY). The truth table of the carry chain multiplexer is shown in Table 2.3. According to that, if the *Propagate* signal is set, then the input carry is propagated to the next adder as input carry.

Table 2.3. Truth Table of the Carry Multiplexer For one-bit full adder in Figure 2.8.

A	B	Cout
0	0	0
0	0	0
0	1	Cin
0	1	Cin
1	0	Cin
1	0	Cin
1	1	1
1	1	1

The structure described above allows the carry bit ripple fast by using the dedicated multiplexer carry chain. The latency in RCAs is caused by the slow carry chain in VLSI, however in FPGAs, the dedicated carry chain provides fast adder in small area. This structure provides the carry bit ripples fast by making the adders work faster and consume small area.

There are also structures to implement multipliers efficiently in FPGAs. In

Spartan-3 family, there are 18x18 dedicated multipliers which can implement multipliers fast and efficient up to 18 bits. Multipliers can be cascaded with each other or CLB logic for larger or more complex functions. Moreover, in Xilinx FPGAs there are DSP48 slices which have a different slice architecture to implement arithmetic operations faster and more efficiently. With DSP48 architecture, 18x18 dedicated multiplier can be implemented in Virtex-4 and 25x18 dedicated multipliers can be implemented in Virtex-5 and Spartan-6 FPGA families efficiently.

3. NODE ANALYSIS

3.1. Introduction

RH(+) HLS tool takes the input language with a high level language such as C or LRH(+) [2] from the user and converts it to Control Data Flow Graphs (CDFGs). LRH(+) is the input language specific to RH(+) tool. CDFG is the intermediate representation of the input algorithm which is used as the input for RTL generation, performance estimation and optimization tools.

Node analysis block in Figure 1.1 is the software module where necessary arithmetic operators that correspond to vertices in the CDFG are selected, delay and area metrics for these operators are modelled, necessary files for RTL generation and estimation are created. Node analysis block consists of four sub-blocks including Arithmetic Component Selection, Extraction of Operators' Character, Hardware Behavior Description (HBD) file creation, Parameter and Template file creation as depicted in Figure 3.1.

- *Arithmetic Component Selection* (Section 3.2) Arithmetic operators and subtype of the arithmetic operators are investigated in terms of their propagation delays, areas and power consumption. These metrics are used during design exploration. In this thesis, adders and subtractors are studied, multipliers and dividers (gray blocks in Figure 3.1) are out of the scope of this thesis. Carry Lookahead Adder (CLA), Carry Select Adder (CSLA), Carry Skip Adder (CSKA), Ripple Carry Adder (RCA) are investigated as the subtypes of adders. In contrast with ASIC, RCAs are found to be the most efficient adder/subtractor type due to dedicated carry chain in Xilinx FPGAs. Carry Select Adder is found to have better delay characteristic in very long wordlengths (such as 128 and 256 bits) where its area characteristic is worse in all wordlengths. CLA and CSKA adders have no advantage over RCA in terms of delay, area and power consumption. Hence, CSLA and RCA are added to the RH(+) component library as arithmetic operator for adders and subtractors.

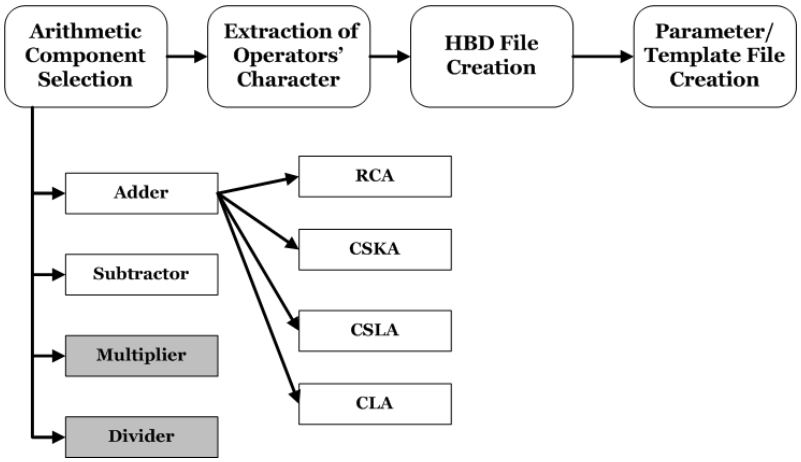

Figure 3.1. Block Diagram of the System.

- *Extraction of Operators' Character* (Section 3.3) Adders and Subtractors are synthesized with changing its parameters on Xilinx FPGAs. By using linear regression analysis, area, delay and power values are used in obtaining the characteristics function.
- *HBD File Creation* (Section 3.4) Files with a special format are prepared to describe the model functions. These files are used in the RH(+) wherever necessary. HBD files provide extendibility of the component library by allowing the users to insert the functions of new components defined by them.
- *Parameter/Template File Creation* (Section 3.5) Template VHDL files are prepared for the arithmetic components. These files consist of fixed and editable fields, which are filled with the values in parameter files during RTL generation. This methodology provides the extendibility on the component library in the way that a user can introduce his arithmetic component by introducing its template and parameter files for RH(+).

3.2. Arithmetic Component Selection

CDFG datastructure represents the high level application given to RH(+) design tool of the intermediate level. In terms of hardware, nodes in CDFG corresponds to arithmetic circuits and edges corresponds to signals between these operators. In Arithmetic Component Selection Block, adder and subtractor operators for FPGAs

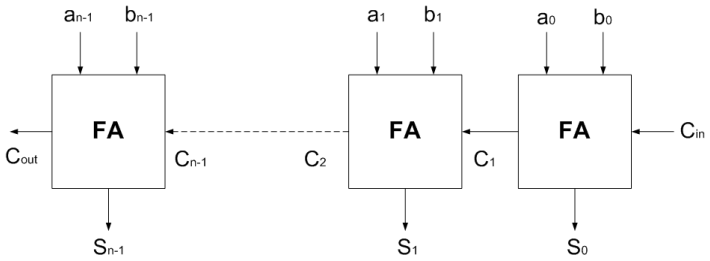

Figure 3.2. The structure of ripple carry adder.

are investigated in terms of delay performance, area and power consumption. Suitable operators are placed in the RH(+) library and their estimated values are generated. In this section, the information about investigated arithmetic operators and the comparison of the investigated arithmetic operators are given.

3.2.1. Adders

Datapath components are designed and implemented with VHDL on Xilinx FPGAs for datapath generation. The power, latency and area metrics are measured for optimal generation of the datapath. Firstly, Ripple Carry Adder (RCA) , Carry Skip Adder (CSKA), Carry Select Adder (CSLA) and Carry Lookahead Adder (CLA) are investigated as a datapath unit for adder types.

In VLSI implementation, CLAs improve the delay of RCA by using parallel prefix algorithms. However, in FPGAs parallel prefix algorithms increase delay, area and power of RCA. Because of the fact that FPGA implementation of CLA has no advantage over RCA, CLA adder is eliminated.

In the following text, an overall information about the adder types and the information of how RCA, CSLA, CSKA addera are implemented in FPGA are given.

3.2.1.1. Ripple Carry Adder. In RCAs, 1-bit full adders are cascaded for creating N-bit adders and carry is cascaded from the least significant bit to most significant. Figure 3.2, below shows the structure of a ripple carry adder.

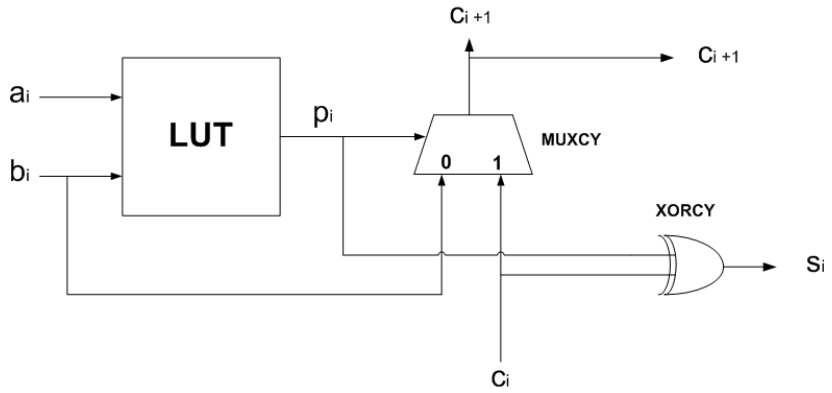

Figure 3.3. Basic Adder Cell.

In RCA, every full adder unit waits for the previous carry to be calculated. The critical path is shown with the arrow in the Figure 3.2. For N-bit RCA, the delay of the critical path is

$$t_{d,RCA} = (N-1) * t_{carry} + t_{sum} \tag{3.1}$$

for VLSI implementation.

RCA on Xilinx FPGAs. The FPGAs have dedicated architecture for RCAs as explained in Section 2.2.1.1. This architecture makes carry propagation faster. This provides fast RCAs on FPGAs. The RCA is implemented by using low level FPGA components so as to use dedicated carry chain. The basic adder cell in Figure 3.3, allows implementation of Ripple Carry Adder. As shown in the Figure 3.3, a LUT, MUXCY and XORCY elements in a slice are used for implementing one-bit adder. These adders are cascaded for implementing N-bit adders. These primitives are instantiated in VHDL code and they are connected with structural coding style. LUT is programmed to implement XOR function. The output of the LUT corresponds to propagate. XORCY XORs the input carry, (c_i), and propagate so as to calculate sum (i.e s_i). If there exists a propagate, MUXCY selects the input carry, (c_i), and give to output carry, (c_{i+1}). If there is no propagate, MUXCY selects the input which

```
                          Gate     Net
Cell:in->out      fanout  Delay    Delay  Logical Name (Net Name)
----------------------------------------  ------------
 LUT2_L:I0->LO         1   0.479    0.000  generate_adder[0].inst_LUT2 (Lut_lo<0>)
 MUXCY_L:S->LO         1   0.435    0.000  generate_adder[0].inst_MUXCY (Muxcy_lo<1>)
 MUXCY_L:CI->LO        1   0.056    0.000  generate_adder[1].inst_MUXCY (Muxcy_lo<2>)
 MUXCY_L:CI->LO        1   0.056    0.000  generate_adder[2].inst_MUXCY (Muxcy_lo<3>)
 MUXCY_L:CI->LO        1   0.056    0.000  generate_adder[3].inst_MUXCY (Muxcy_lo<4>)
 MUXCY_L:CI->LO        1   0.056    0.000  generate_adder[4].inst_MUXCY (Muxcy_lo<5>)
 MUXCY_L:CI->LO        1   0.056    0.000  generate_adder[5].inst_MUXCY (Muxcy_lo<6>)
 MUXCY_L:CI->LO        1   0.056    0.000  generate_adder[6].inst_MUXCY (Muxcy_lo<7>)
 XORCY_L:CI->LO        0   0.786    0.000  generate_adder[7].inst_XORCY (out_d_0<7>)
----------------------------------------
Total                     2.328ns (2.328ns logic, 0.000ns route)
                                  (100.0% logic, 0.0% route)
```

Figure 3.4. Critical Path of 16-bit RCA Adder.

can be either 1 or 0.

The wordlength of the RCA adder in VHDL is given by generics so the RCA adders can be generated with desired wordlengths. The critical path of the 16-bit RCA adder which is taken from the synthesis report is given in the Figure 3.4. It can be seen that the critical path includes I/O buffers of FPGA, LUT (XOR function) and carry multiplexer (MUXCY) of first bit addition, and Carry Multiplexers (MUXCY) of the other bit additions. As carry chain is implemented with dedicated carry multiplexers, RCA is fast and efficient for addition.

The synthesized floorplan of the 8-bit RCA in Xilinx Spartan-3 is shown in Figure 3.5. It is implemented by using one column of slices of the CLBs and these slices are cascaded so that carry output of one slice will be the carry input of the another slice.

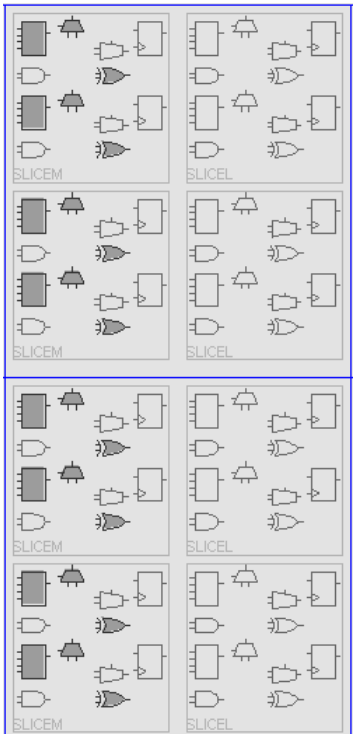

Figure 3.5. Floorplan of the 16-bit RCA adder in Xilinx Spartan-3.

3.2.1.2. Carry Skip Adder. In Ripple Carry Adder, Carry is propagated through the least significant bit to the most significant bit one-by-one. This chain has a significant latency for VLSI implementation that slows down the adder. In CSKAs, RCAs in an adder are grouped and every group generates the signal of propagate. These groups are called "Compressor" blocks or Carry Propagate Adder(CPA) groups . If propagate signal, p_i, is equal to 1, input carry, c_i of the block is equal to the output carry, c_{out} of the block. If the propagate signal, p_i, is equal to 0, c_{out} of the module depends on the inputs and c_i of the module. The equations below show how the carry skip adder works;

$$P_i = a_i \oplus b_i \tag{3.2}$$

$$S_i = P_i \oplus C_i \tag{3.3}$$

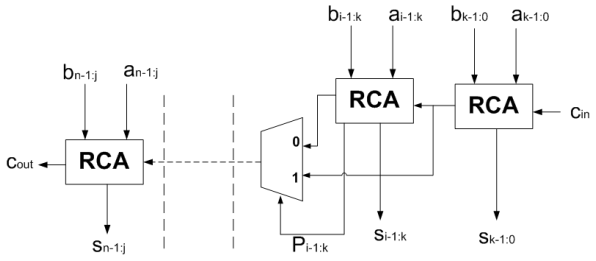

Figure 3.6. The Structure of Carry Skip Adder.

$$C_{out} = a_i b_i + P_i C_i \tag{3.4}$$

If $a_i = b_i$ then C_{i+1} depends only on a_i and b_i where $C_{i+1} = a_i b_i$. If $a_i \neq b_i$ then $P_i = 1$ so $C_{i+1} = C_i$.

With that method, the carry does not have to travel all the way from LSB to MSB if there are groups which propagate the carry. This decreases the delay of an adder in VLSI implementation.

Figure 3.6, shows the structure of Carry Skip Adder. The Multiplexer after each block selects the right carry according to the propagate signal. If the P_i is equal to 1, C_{in} of the block is propagated as C_{out} of the block. This decreases the critical path which corresponds the decrease in the delay as well. However, the area increases because of the multiplexers used for selecting the right carry out. In VLSI implementation, the delay of the critical path for CSKA adder is given in Equation 3.5 where m corresponds to number of RCA groups(full adder) and n corresponds to number of RCA groups which carry bypasses the RCA group. The critical path for CSKA changes according to the input carry of the RCA block, if the input carry of the block is equal to 1, critical path changes where the carry bypasses the RCA block.

$$t_{d,CSKA} = (n) * t_{fulladder} + (m-2) * t_{mux} \tag{3.5}$$

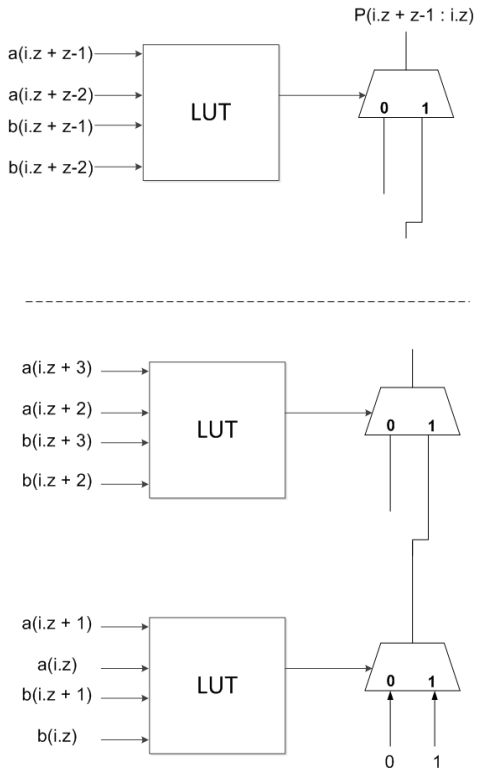

Figure 3.7. Computation of Propagate Signal of the Group.

CSKA Implementation on FPGA. Firstly, CSKA is implemented by using RCA as CPA blocks and multiplexers which are coded in behavioral VHDL style are connected to these blocks. This implementation results worse in delay, area and power consumption compared to Ripple Carry Adder. This implementation is inefficient because multiplexers are synthesized without using dedicated carry multiplexer chains, and although the RCAs are fast, the multiplexers degrades the efficiency substantially. So, the CSKA is implemented in the method which is shown in [43]. In this method, in order to get fast CSKA, dedicated carry logic multiplexers are used in the critical path of CSKA.

In this method of implementation, for calculating the propagate signals of each group , the circuit in Figure 3.7 is used.

As shown in the Figure 3.7, the propagate signal of the group of $i.z$ to $i.z+z-1$ is calculated by using LUTs and dedicated carry Logic. LUTs are programmed as defined in Equation 3.6;

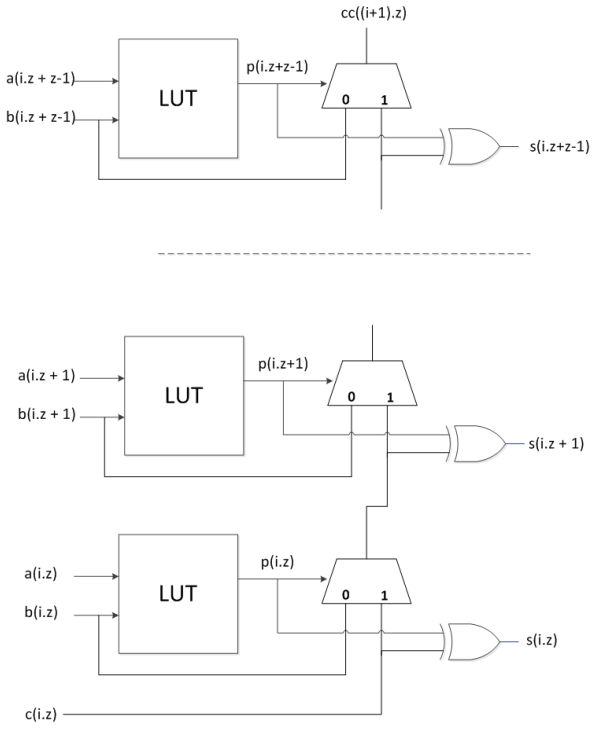

Figure 3.8. An s bit RCA group.

$$p(i.z).p(i.z+1) = (x(i.z) \oplus y(i.z)).(x(i.z+1) \oplus y(i.z+1)) \tag{3.6}$$

If the propagate signals of the all LUTs in a group are 1, the propagate signal of the group equals to 1.

The propagate signals are calculated by the circuit explained in Figure 3.7. The circuit in Figure 3.8 shows the RCA block. In each RCA group the input variables (*a(i.z)*, *b(i.z)*) and input carry (*c(i.z)*) are used in calculating the sum and output carry of the group.

The multiplexer circuit is shown in the Figure 3.9. In the circuit, *cc(z)* are the output carry signals of the groups of adders. The previous carry and the carry of the group are multiplexed in MUXCY (dedicated multiplexer in carry chain) by the

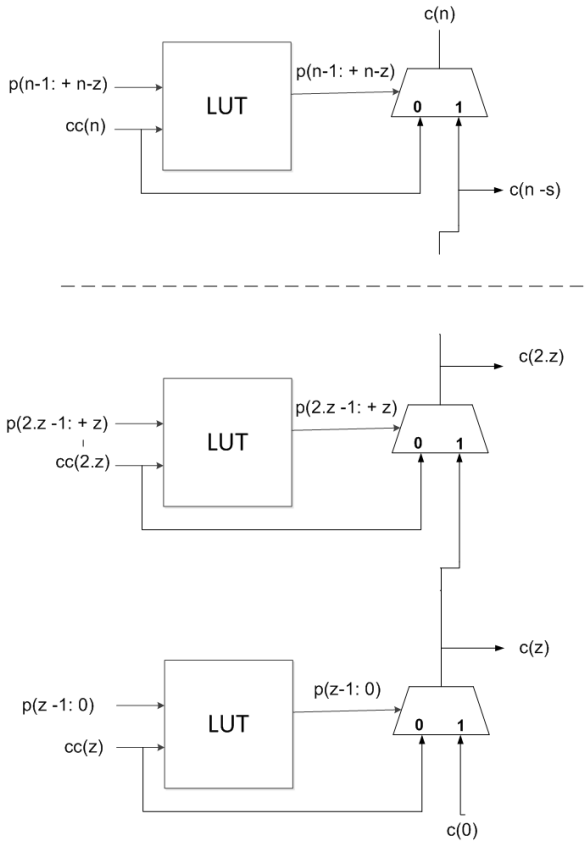

Figure 3.9. Carry Skip Multiplexers.

propagate signals.

The complete circuit of the Carry Skip adder is shown in the Figure 3.10, when RCA group, Propagate Group and Carry Skip Multiplexer groups are combined. The critical path is given as the shaded area [43].

The critical path from the synthesis report of a 8-bit CSKA adder with having compressor size 4-bit is shown in the Figure 3.11. Compressor refers to the RCA groups in the Carry Skip Adder. As it can be seen from the report, the critical path includes LUTs and dedicated Multiplexers.

In Figure 3.12, the floorplan of the 8-bit CSKA adder with compressor size of 4-bits is given.

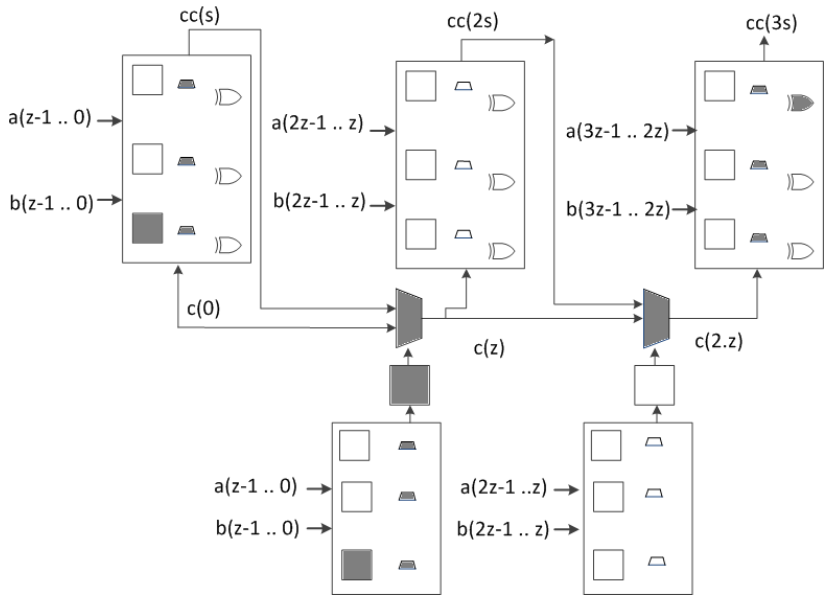

Figure 3.10. Carry Skip Adder with 4 RCA groups.

```
Data Path: a<0> to carry_out
                                Gate     Net
  Cell:in->out      fanout    Delay    Delay  Logical Name (Net Name)
  ----------------------------------------  ------------
   LUT2_L:I0->LO         1    0.479    0.000  GenCSKA[0].n_if0.oAdderPrim0/generate_adder[0]
   MUXCY_L:S->LO         1    0.435    0.000  GenCSKA[0].n_if0.oAdderPrim0/generate_adder[0]
   MUXCY_L:CI->LO        1    0.056    0.000  GenCSKA[0].n_if0.oAdderPrim0/generate_adder[1]
   MUXCY_L:CI->LO        1    0.056    0.000  GenCSKA[0].n_if0.oAdderPrim0/generate_adder[2]
   MUXCY_L:CI->LO        1    0.265    0.976  GenCSKA[0].n_if0.oAdderPrim0/generate_adder[3]
   LUT2_L:I0->LO         1    0.479    0.000  GenCSKA[0].n_if0.gen_LUT2 (int<0>)
   MUXCY_L:S->LO         2    0.435    0.000  GenCSKA[0].n_if0.gen_MUXCY (imux<1>)
   MUXCY_L:CI->LO        1    0.056    0.000  GenCSKA[1].n_if1.oAdderPrim0/generate_adder[0]
   MUXCY_L:CI->LO        1    0.056    0.000  GenCSKA[1].n_if1.oAdderPrim0/generate_adder[1]
   MUXCY_L:CI->LO        1    0.056    0.000  GenCSKA[1].n_if1.oAdderPrim0/generate_adder[2]
   MUXCY_L:CI->LO        1    0.264    0.976  GenCSKA[1].n_if1.oAdderPrim0/generate_adder[3]
   LUT2_L:I0->LO         1    0.479    0.000  GenCSKA[1].n_if1.gen_LUT2 (int<1>)
   MUXCY_L:S->LO         0    0.644    0.000  GenCSKA[1].n_if1.gen_MUXCY (carry_out)
  ----------------------------------------
  Total                      6.004ns (4.053ns logic, 1.952ns route)
                                     (67.5% logic, 32.5% route)
```

Figure 3.11. Critical path of an 8-bit CSKA adder with compressor size of 4.

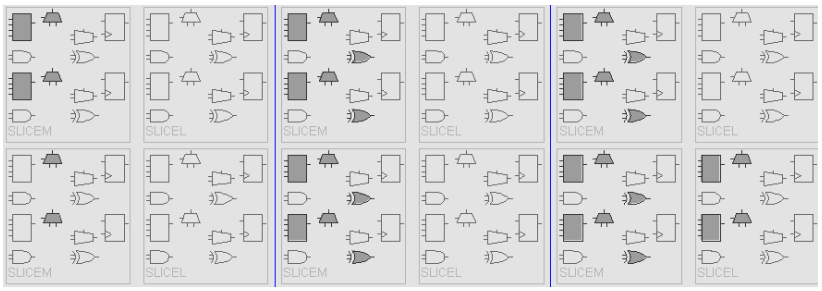

Figure 3.12. The Floorplan of the 8-bit CSKA with 4 bitsize of compressors.

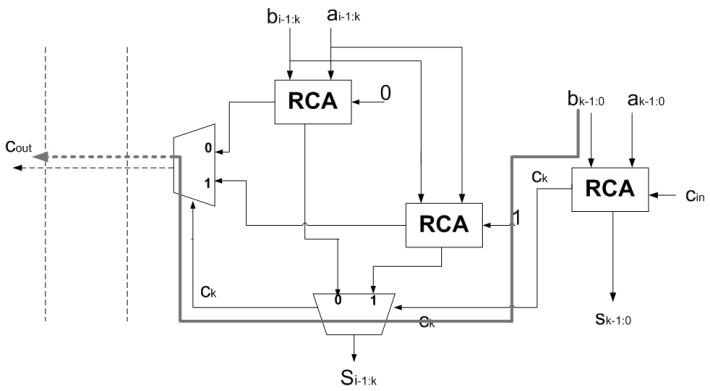

Figure 3.13. Structure of CSLA.

3.2.1.3. Carry Select Adder. CSLA, divides the adder into blocks with smaller wordlengths. These blocks are called 'compressor' blocks because they compress n-bit input to one sum output and one carry output. Each block in the CSLA is duplicated to calculate sum and output carry for the both conditions where input carry of the block is 1 and 0. The sum and output carry, C_{out} of the blocks become ready before the input carry arrives. When the input carry arrives, multiplexer chooses the right sum and the right output carry according to the input carry. This structure provides CSLA to function faster than the RCA in VLSI. Figure 3.13 shows the structure of a Carry Select Adder.

With this implementation, critical path is reduced because full adders do not have to wait the input carry to be calculated. The sum and C_{out} have already been calculated and multiplexers select the right one just after the previous carry arrives. The critical path is shown with the gray arrow in Figure 3.13. In the carry path, there exist multiplexers instead of full adders. However, the area consumed is higher than RCAs. The critical path delay for an n-bit adder with m compressor size is given in Equation 3.7 for VLSI.

$$t_{d,CSLA} = m * t_{fulladder(carry)} + (n/m - 1) * t_{mux} \tag{3.7}$$

Carry Select Adder Implementation on FPGA. CSLA is the same with VLSI implementation except RCA blocks are implemented as in Section 3.2.1.1. The outputs of these blocks are selected with multiplexers in FPGA.

CSLA is implemented using VHDL by giving generics of both input wordlength of the adder and wordlength of the compressor, so that desired CSLA can be synthesized by just giving generics. For example, when wordlength of the adder is chosen 16 and wordlength of the compressor is 4, it is synthesized as first 4-bit block is a RCA adder which sums LSBs of the operands and takes 0 as the input carry. The next three blocks, each of which has 4-bit wordlength, consist of one RCA adder with input carry is equal to 0, and one RCA adder with input carry is equal to 1 and two multiplexers for sum and output carry selection. The C_{out} of the 16-bit adder is concatenated as the MSB of the adder which in this case corresponds to 17th bit. When the wordlength of CSLA is not a multiple of the compressor size, the compressor block with MSB has a lower wordlength than the block wordlength. For example, if the adder wordlength is 18-bit and compressor wordlength is 8-bit, the last block's wordlength will be 2-bit as shown in Figure 3.14.

The critical path of a 8-bit adder with compressor wordlength of 2-bit is shown in Figure 3.15. The critical path consists of the carry multiplexers of the first compressor block and the multiplexer, which is synthesized as LUT here.

The Floorplan of the synthesized CSLA of 8-bit with 2-bit compressor size is given in Figure 3.16 for the Xilinx Spartan-3 FPGA. As it can be seen, compressors are synthesized as RCA in Section 3.2.1.1 and they are connected with multiplexers.

3.2.2. Subtractors

Subtractors are implemented in the same way as adders. Because RCA and CSLA are selected to place RH(+) component library as datapath unit, Ripple Borrow Subtractor and Borrow Select Subtractor are implemented as the same way that RCA and CSLA are implemented.

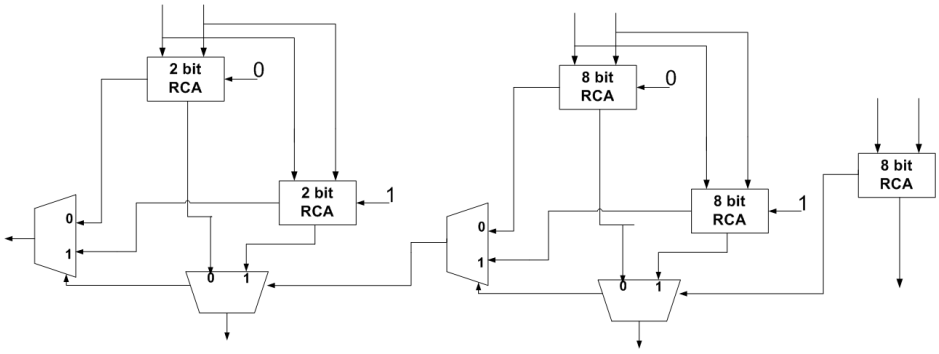

Figure 3.14. Block Diagram shows how 18-bit adder is synthesized with 8-bit block size.

```
Data Path: a<0> to co
                                   Gate     Net
  Cell:in->out      fanout   Delay    Delay  Logical Name (Net Name)
  ----------------------------------------  ------------
   LUT2_L:I0->LO         1   0.479    0.000  n_if.oGen[1].oRCAprim1/generate_adder[0].gen_LUT2
   MUXCY_L:S->LO         1   0.435    0.000  n_if.oGen[1].oRCAprim1/generate_adder[0].gen_MUXCY
   MUXCY_L:CI->LO        1   0.265    0.851  n_if.oGen[1].oRCAprim1/generate_adder[1].gen_MUXCY
   LUT3:I1->O            3   0.479    1.066  n_if.oGen[1].oMux2to1/o1 (c<1>)
   LUT3:I0->O            3   0.479    1.066  n_if.oGen[2].oMux2to1/o1 (c<2>)
   LUT3:I0->O            3   0.479    1.066  n_if.oGen[3].oMux2to1/o1 (c<3>)
   LUT3:I0->O            0   0.479    0.000  n_if.oGen[4].oMuxn2to1/oGen[1].oMux2to1/o1 (s<7>)
  ----------------------------------------
  Total                      7.438ns (3.390ns logic, 4.049ns route)
                                     (45.6% logic, 54.4% route)
```

Figure 3.15. Critical path of an 8-bit adder with compressor bit size of 2.

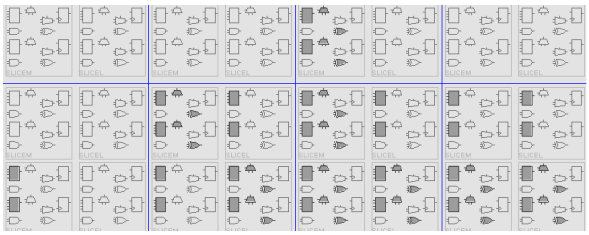

Figure 3.16. Floorplan of 8-bit CSLA with 2-bit compressor size for Xilinx Spartan-3 FPGA.

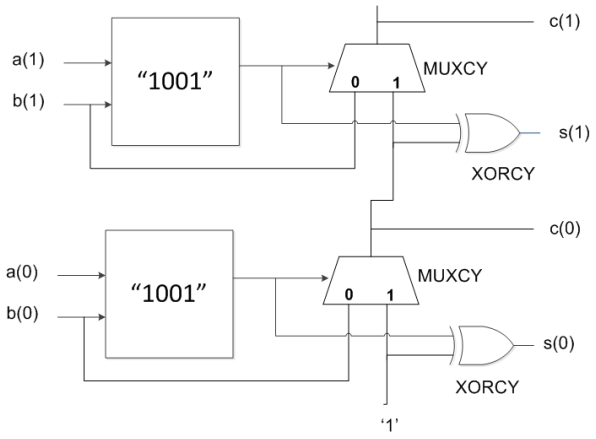

Figure 3.17. Implementation of 2-bit ripple borrow subtractor.

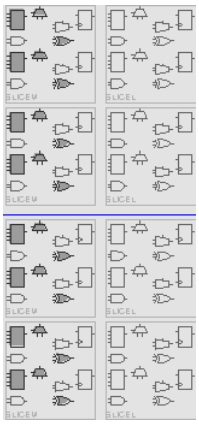

Figure 3.18. FloorPlan of the Ripple Borrow Subtractor.

Ripple Borrow Subtractor. Ripple Borrow Subtractor is the same with ripple carry adder structurally except the LUT in slices are programmed as "1001" where in RCA it is "011" and the carry-in of the first bit of the subtractor is '1' where carry-in of the first bit of the RCA is '0'. The implementation of 2-bit ripple borrow subtractor in Xilinx FPGA is given in Figure 3.17.

In Figure 3.18, the floorplan of the ripple borrow subtractor is shown.

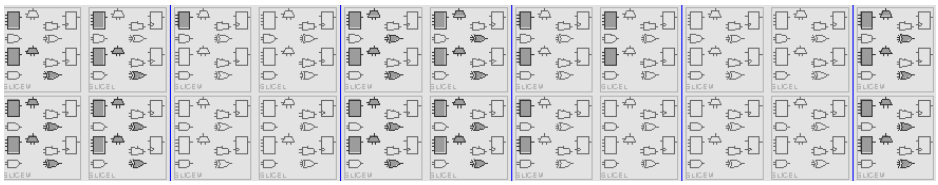

Figure 3.19. Floorplan of a 8-bit Borrow Select Subtractor with compressor bit size is 2.

Borrow Select Subtractor. The Borrow Select Subtractor is implemented as the same way of Carry select adder but here ripple borrow subtractors are used as blocks. The floorplan of a 8-bit Borrow Select Subtractor with compressor bitsize of 2 is shown in the Figure 3.19 below;

3.2.2.1. Comparison of the Adders. The delay, area, and power consumption of the adders are compared so as to determine which type of adders will exist in the RH(+) library. Ripple Carry Adders have the best performance on the delay, area and power metrics because FPGAs have special carry multiplexers that implements Ripple Carry Adder fast and in small area. Carry Skip Adder and Carry Select Adder are compared with Ripple Carry Adder. The delay is given as critical path delay in nanoseconds and area is the number of used slices. These values are extracted from the synthesis report of the Xilinx ISE 12.4.

Ripple Carry Adder and Carry Skip Adder Comparison. The delay and area of the CSKA are given in Table 3.1. Power consumption is not compared since as we can see from area values, CSKA consumes more hardware that we can conclude CSKA consumes more power than RCA.

Table 3.1. Comparison of CSKA and RCA on Xilinx Spartan-3 FPGA.

Adder Type	**Wordlength**	**Compressor Wordlength**	**Delay (ns)**	**Area (n of slices)**	**Delay with I/O buffers (ns)**
RCA	16	-	2.477	8	9.758
CSKA	16	2	9.820	13	17.165
	16	4	10.351	17	17.969
	16	8	6.154	15	13.499
	16	16	4.055	14	11.4
RCA	32	-	3.365	16	10.646
CSKA	32	2	18.572	26	25.917
	32	4	19.635	35	26.98
	32	8	11.239	30	18.584
	32	16	7.042	27	14.387
	32	32	4.943	26	12.288
RCA	64	-	5.141	32	12.422
CSKA	64	2	36.076	53	43.421
	64	4	38.201	70	45.506
	64	8	21.411	61	28.756
	64	16	13.015	56	20.360
	64	32	8.818	53	16.163
	64	64	6.719	50	14.064
RCA	128	-	8.693	64	15.974
CSKA	128	4	75.335	140	82.680
	128	8	41.753	122	49.098
	128	16	24.963	113	32.308
	128	32	16.567	108	23.912
	128	64	12.37	103	19.715
RCA	256	-	15.797	128	23.078
CSKA	256	4	149.602	280	156.947
	256	8	82.439	245	89.784
	256	16	48.857	227	56.202
	256	32	32.067	217	39.412
	256	64	23.671	211	31.106
	256	128	19.474	204	26.819

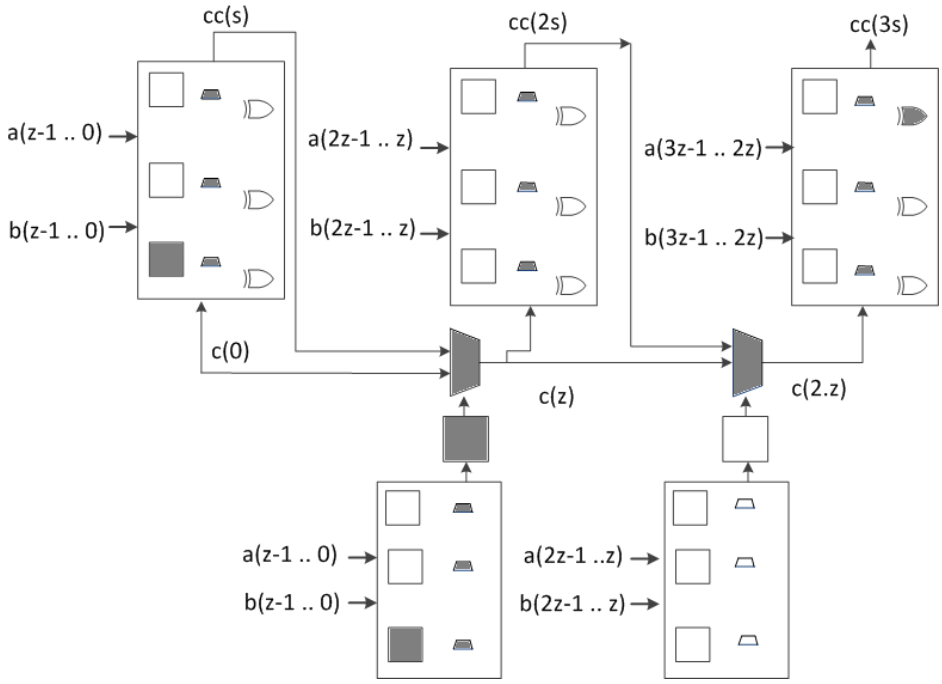

Figure 3.20. Critical path of the CSKA adder (shaded area).

In Table 3.1, delay and area metrics of the CSKA according to adder wordlength and compressor wordlength is given. CSKA has no advantage in consumed area or delay performance over RCA in all wordlengths. The reason why there is no any improvement in delay is that critical path shown in Figure 3.10 which is given in [43] is not true. The correct critical path is found as in the the shaded area of the Figure 3.20.

Ripple Carry Adder and Carry Select Adder Comparison. The delay and area of the CSKA are given in Table 3.2, where it can be seen that RCA is more efficient in terms of delay and area due to FPGAs dedicated carry chain multiplexers which implements the RCA fast and small in size. However, CSLA is better in delay when the bit-size of the adder is larger than 64. For example in Table 3.2, RCA adder has 8.693 ns delay value when bit-size of the adder is 128 and CSLA adder has 7.129 ns when bit-size of the adder is 128 and compressor bit-size is 64. In terms of area, CSLA consumes more area than RCA at all wordlengths.

Table 3.2. Comparison of CSLA and RCA on Xilinx Spartan-3 FPGA.

Adder Type	**Bit-Size**	**Compressor Bit-Size**	**Delay (ns)**	**Area (n of slices)**	**Delay with I/O buffers(ns)**
RCA	16	-	2.477	8	9.758
CSLA	16	2	11.993	27	19.274
	16	4	5.96	23	13.241
	16	8	3.363	17	10.708
	16	16	2.477	8	9.758
RCA	32	-	3.365	16	10.646
CSLA	32	2	24.352	57	31.663
	32	4	12.187	50	19.468
	32	8	6.699	44	13.98
	32	16	3.872	34	11.153
	32	32	3.365	16	10.646
RCA	64	-	5.141	32	12.422
CSLA	64	2	49.072	116	56.353
	64	4	24.692	105	31.973
	64	8	13.617	96	20.898
	64	16	7.707	85	14.988
	64	32	5.197	67	12.478
	64	64	5.141	32	12.422
RCA	128	-	8.693	64	15.974
CSLA	128	4	49.552	215	56.833
	128	8	27.452	202	34.733
	128	16	15.376	188	22.657
	128	32	9.905	169	17.186
	128	48	**8.517**	151	15.798
	128	64	**7.129**	133	14.41
	128	128	8.693	64	15.974
RCA	256	-	15.797	128	23.078
CSLA	256	8	55.123	412	62.404
	256	16	30.714	395	37.995
	256	32	19.321	373	26.602
	256	48	**15.376**	354	22.657
	256	64	**12.149**	336	19.43
	256	128	**10.993**	266	18.274
	256	192	**14.233**	197	21.514
	256	256	15.797	128	23.078

Power is not compared in Table 3.2, since as we can see from area value, CSLA consumes more hardware that we can conclude CSLA consumes more power than RCA.

3.2.2.2. Results of the Comparisons. Comparisons show that RCA is an efficient adder in both area and delay because of the dedicated carry chain multiplexers of the FPGA architecture. Carry Skip Adder has no advantage in terms of area and delay over RCA in any wordlength value. So, CSKA is eliminated and it is not placed in RH(+) library. Although CSLA adder consumes more area, its delay performance is slightly better for the cases where wordlength is higher than 64. This may be useful for applications that require high wordlength calculations such as image processing. So CSLA is placed in the RH(+) library.

3.2.3. Multiplexers

Multiplexers are widely used in the datapath of the processor. Multiplexers are implemented with VHDL coding style which is shown in the Xilinx Synthesis Manual [1]. Multiplexers are coded with CASE statements that synthesize optimized multiplexers in FPGAs.

Slices in Xilinx FPGAs have wide-function multiplexers that effectively combine LUTs in order to permit more complex logic operations. In Spartan-3, each slice has two wide-function multiplexers which one is at the bottom of the slice (F5MUX) and the other one (FIMUX) is at the top portion of the slice. These multiplexers are shown as gray in Figure 3.21. F5MUX multiplexes the two LUTs in a slice. The FiMUX multiplexes two CLB inputs which connect directly to the F5MUX and FiMUX results from the same slice or from other slices. With the use of wide-function multiplexers, any Spartan-3 generation device easily implements:

- a 2:1 mux in one LUT
- a 4:1 mux in one slice
- a 16:1 mux in one CLB

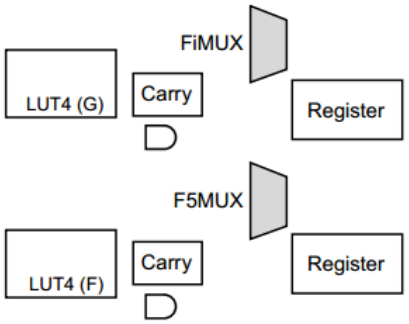

Figure 3.21. F5MUX and FIMUX in a Spartan-3 Slice [1].

- a 32:1 mux in two CLBs

The connection from LUTs to muxes and between the muxes are dedicated and have zero connection delay. The combination of LUTs and dedicated multiplexers allows very efficient implementation of large multiplexers.

It is important to note that while writing Multiplexer Code, the Case Statements should include all possibilities; otherwise Synthesizer creates latches connected to multiplexers.

3.3. Extraction of Operators' Characteristics

In this module, delay, area and power consumption of the adders/subtractors and multiplexers are obtained at selected wordlengths. Then, linear regression methods are applied on these data points to obtain the characteristic functions of these operators.

3.3.1. Obtaining Data points

The VHDL files of the arithmetic components are synthesized by Xilinx 12.4 for Spartan-3 and Virtex-4 families. In Figure 3.22, synthesis delay report of an 8-bit RCA is given as example. As it can be seen, total delay is 9.314 ns but it includes

input and output buffer delays so these values are extracted as shown in Equation 3.8.

$$t_{D,RCA_8} = 9.314 - 0.715 - 0.976 - 4.909 - 0.681 = 2.033ns \tag{3.8}$$

The area is measured as the number of slices used which is also extracted form the

```
=========================================================================
Timing constraint: Default path analysis
  Total number of paths / destination ports: 124 / 9
-------------------------------------------------------------------------
Delay:               9.314ns (Levels of Logic = 11)
  Source:            a<0> (PAD)
  Destination:       sum<7> (PAD)

  Data Path: a<0> to sum<7>
                              Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     IBUF:I->O             1   0.715   0.976  a_0_IBUF (a_0_IBUF)
     LUT2_L:I0->LO         1   0.479   0.000  generate_adder[0].gen_LUT2 (im1<0>)
     MUXCY_L:S->LO         1   0.435   0.000  generate_adder[0].gen_MUXCY (ci_temp<1>)
     MUXCY_L:CI->LO        1   0.056   0.000  generate_adder[1].gen_MUXCY (ci_temp<2>)
     MUXCY_L:CI->LO        1   0.056   0.000  generate_adder[2].gen_MUXCY (ci_temp<3>)
     MUXCY_L:CI->LO        1   0.056   0.000  generate_adder[3].gen_MUXCY (ci_temp<4>)
     MUXCY_L:CI->LO        1   0.056   0.000  generate_adder[4].gen_MUXCY (ci_temp<5>)
     MUXCY_L:CI->LO        1   0.056   0.000  generate_adder[5].gen_MUXCY (ci_temp<6>)
     MUXCY_L:CI->LO        1   0.056   0.000  generate_adder[6].gen_MUXCY (ci_temp<7>)
     XORCY_L:CI->LO        1   0.786   0.681  generate_adder[7].gen_XORCY (sum_7_OBUF)
     OBUF:I->O                 4.909          sum_7_OBUF (sum<7>)
    ----------------------------------------
    Total                      9.314ns (7.657ns logic, 1.657ns route)
                                       (82.2% logic, 17.8% route)

=========================================================================
```

Figure 3.22. Synthesis Delay Report of an 8-bit RCA for Spartan-3 FPGA.

synthesis report. The area synthesis report for an 8-bit RCA adder is given as example in Figure 3.23. As it can be seen in the figure, the number of used slice for 8-bit RCA is 4.

```
Selected Device : 3s1000fg676-5

 Number of Slices:                        4  out of   7680     0%
 Number of 4 input LUTs:                  8  out of  15360     0%
 Number of IOs:                          25
 Number of bonded IOBs:                  25  out of    391     6%
```

Figure 3.23. Synthesis Area Report of an 8-bit RCA for Spartan-3 FPGA.

For power measurement, Xilinx Power Analysis tool is used. Xilinx power analysis tool can be used only when the circuit can be synthesizable and fully implementable. The power is measured after the adder is fully placed & routed and post place & route simulation is done with random inputs. In the post place & route simulation, the inputs are toggled as the size of the input bits. For example, for an 8-bit adder 8

different input is given and for an 64-bit adder 64 different input is given. The signal power and logic power of dynamic power is gathered from the Power Analysis tool since the static power has a fixed value which changes from FPGA to FPGA.

3.3.2. Curve Fitting

Polynomial linear regression is applied to the values of the delay, area, and power of the synthesized component.

3.3.2.1. PRESS Analysis. Predicted Sum of Square (PRESS) analysis is used in curve fitting to determine best polynomial order that the delay, area and power functions can be fitted. As the polynomial order of the fitted polynomial function increases Sum of Squared Error(SSE) decreases. However, complexity of the fitted function increases and also the fitted function start to memorize points rather than learning the interpolation.

A PRESS analysis function is coded in MATLAB and this program takes the datapoints as inputs and looks at the results of the fitted functions for all orders and choose the fitted function with the lowest PRESS value. The program flow is listed below;

(i) Takes the datapoints which may be the area, delay or power with respect to wordlength value.
(ii) Makes a function fitting with all datapoints and calculates SSE.
(iii) Removed one datapoint from the input data and makes function fitting for the remained datapoints. Calculates SSE of that function and subtracts that SSE from the SSE found in step ii.
(iv) Applies step iii for all datapoints in input data and calculates the PRESS value by averaging the value calculated in step iii.
(v) Applies the steps of i,ii,iii and iv for all possible polynomial order that the function can be fitted. For example, if there are 8 datapoints in the input data, curve fitting is done for all polynomial order from 1 to 7.

(vi) Chooses the function with lowest PRESS value.

3.3.2.2. Ripple Carry Adder. The area, delay and power cost of the RCA depends only on the input wordlength. When the wordlength of the ports of the adder are not equal, the delay, area costs are depend on the wordlength of the bigger port. For example, when one port of an adder has 8 bits and the other port has 6 bits, this adder has the same delay, area value with an adder which has the both ports with 8 bits length. So, the functions of ripple carry adder are one dimensional.

Polynomial curve fitting is applied to obtain delay, area and power functions of the Ripple Carry Adder. As the order of the function increases, Sum Squared Error (SSE) decreases but function becomes complexer. Predicted Error Sum of Square (PRESS) analysis is also used to determine which order of the polynomial to choose.

RCA Delay for Spartan-3 FPGA. Figure 3.24 shows the datapoints and fitted function. The order of the function is 1 where the PRESS value is lowest (3.630732e-

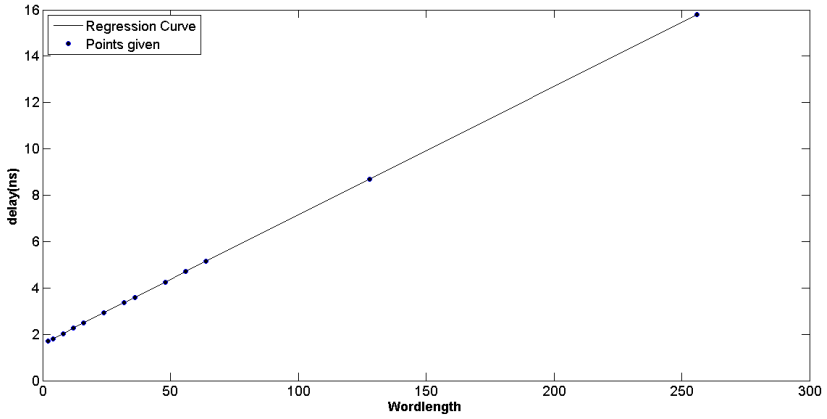

Figure 3.24. Delay Function of the RCA for Spartan-3 FPGA.

028). The RCA delay function for Spartan-3 FPGA is given in Equation 3.9, where x corresponds to wordlength and y corresponds to delay in nanoseconds.

$$Y = 0.0559x + 1.5316 \tag{3.9}$$

RCA Area for Spartan-3 FPGA. Figure 3.25 shows the datapoints and related function.

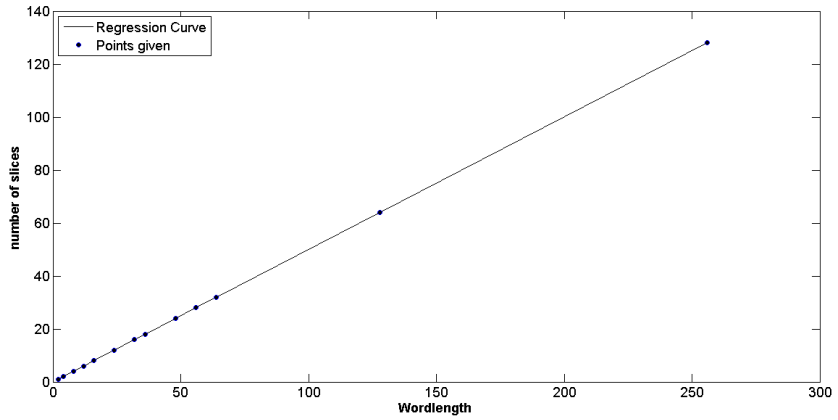

Figure 3.25. Area Function of the RCA for Spartan-3 FPGA.

The lowest PRESS value is 4.088e-1 where the order of the function is 1. The RCA Area function for Spartan-3 FPGA is given in Equation 3.10 where x corresponds to number of bits and y corresponds to area in number of slices.

$$Y = 0.4996x + 0.055 \tag{3.10}$$

RCA Power for Spartan-3 FPGA. Figure 3.26 shows the datapoints and the function.

The PRESS value is 1.923e-1 which is the lowest when the order of the polynomial is 1. The RCA power function for Spartan-3 FPGA is given in Equation 3.11 where x corresponds to number of bits and y corresponds to power in miliwatts.

$$Y = 0.0119x - 0.0619 \tag{3.11}$$

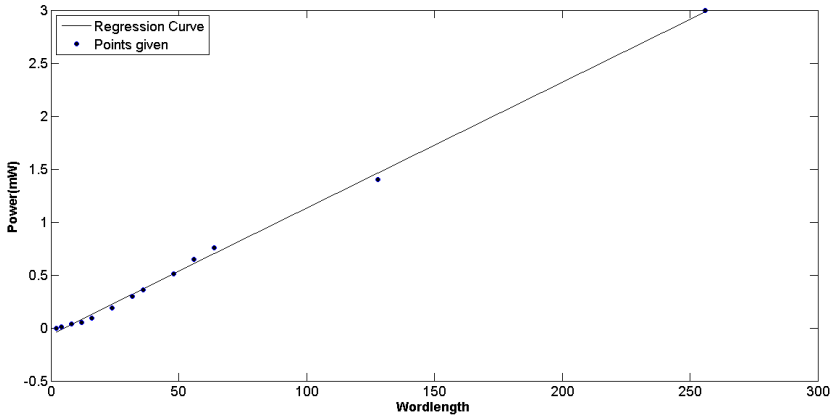

Figure 3.26. Power Function of the RCA for Spartan-3 FPGA.

RCA Delay for Virtex-4 FPGA. Figure 3.27 shows the datapoints and obtained function. The order of the function is 1 where the PRESS value is lowest (8.906e-1). The RCA delay function for Virtex-4 FPGA is given in Equation 3.9, where x corresponds to wordlength and y corresponds to delay in nanoseconds.

$$Y = 0.0339x + 0.6190 \tag{3.12}$$

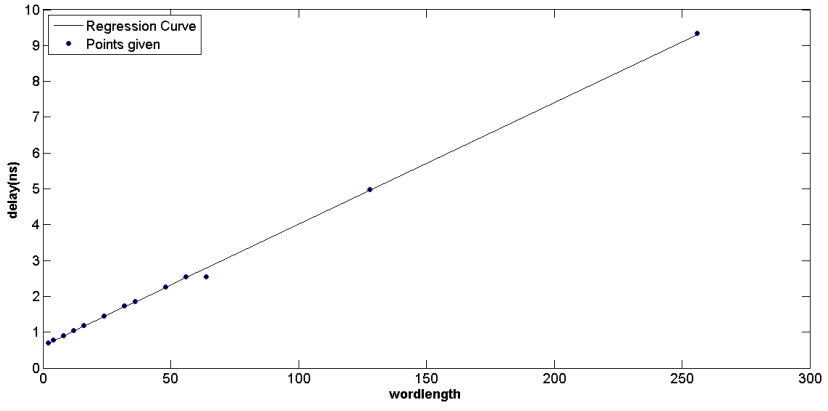

Figure 3.27. Delay Function of the RCA for Virtex-4 FPGA.

RCA Area for Virtex-4 FPGA. The area values and function is the same with Spartan-3. The obtained function is shown in Figure 3.10 and is given in Equation 3.10.

RCA Power for Virtex-4 FPGA. Figure 3.28 shows the datapoints and the function. The PRESS value is 4.4741 which is the lowest when the order of the polynomial is 1. The RCA power function for Virtex-4 is given in Equation 3.13 where x corresponds to number of bits and y corresponds to power in miliwatts.

$$Y = 0.0391x + 0.3603 \tag{3.13}$$

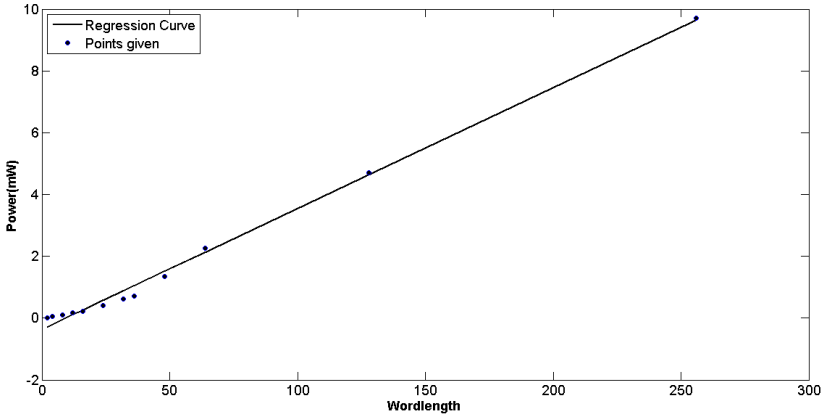

Figure 3.28. Power Function of the RCA for Spartan-3 FPGA.

3.3.2.3. Carry Select Adder. The delay, area and power consumption of the CSLA depends on the both the wordlength of the adder and the wordlength of the compressor blocks in the CSLA. So, delay, area and power functions of the CSLA are two-dimensional polynomial functions. As given in Table 3.2, CSLA is worse than RCA in both delay and area for the wordlengths lower than 64 bits. Hence, CSLA functions are obtained for the wordlength values higher than 64 bits. Piece-wise polynomial functions are used for better curve fitting.

For CSLA, wordlength of the adder and blocksize of the adder have a distinct

relation on the delay and area of the CSLA. For the small compressor wordlengths, the delay and area of the CSLA are very high and as the compressor wordlength increases, the delay and area decrease. Hence, more efficient adders can be obtained. Generally, as the wordlength of the compressor circuits increases the delay and consumed area decreases. Figure 3.29 and Figure 3.30 shows the plots of 48-bit CSLA with respect to compressor block wordlength. The area of the CSLA decreases as the wordlength of the compressor increases. However, for delay, there are exceptions for CSLA with wordlengths higher than 64-bit. For example, in Figure 3.31, plot of the 128-bit CSLA with respect to the wordlength of the compressor is given. The delay of 128-bit CSLA with 64-bits compressor size is lower than the delay with 128-bit compressor size. 128-bit CSLA adder with compressor wordlength of 128-bit corresponds to RCA. So, delay of 128-bit adder when compressor wordlength is 64-bit is better in delay than RCA. This trend can be seen in Figure 3.31, i.e delay decreases from compressor wordlength 0 to 64 bits and then it increases again from 64 to 128 bits. This trend exists for the CSLA adder with wordlength higher than 64-bits. When the compressor wordlength is half of the wordlength of the adder, the delay becomes minimal. The delay values also can be seen in Table 3.2. That situation does not exist for the area of the CSLA, area always decreases as the wordlength of the compressor increases even for CSLA adders with high wordlengths as shown in Figure 3.32.

The area and delay of CSLA adder when the wordlength of the compressor is much lower than the wordlength of the adder, the delay and area cause a peak in the function. For better function fitting, the CSLA values are fitted for the compressor wordlength starting from the adder wordlength divided by three. While using the CSLA functions for delay and area in Equations 3.14, 3.15 and 3.16 this should be considered. For example, for 72-bit CSLA adder, the lowest compressor wordlength can be $72/3 = 24$ in in the functions. For example, for 128-bit CSLA adder, compressor wordlength shall be bigger than $128/3 = 42.66$. That means, the lowest value can be 43 while using these Equations of 3.14, 3.15, 3.16.

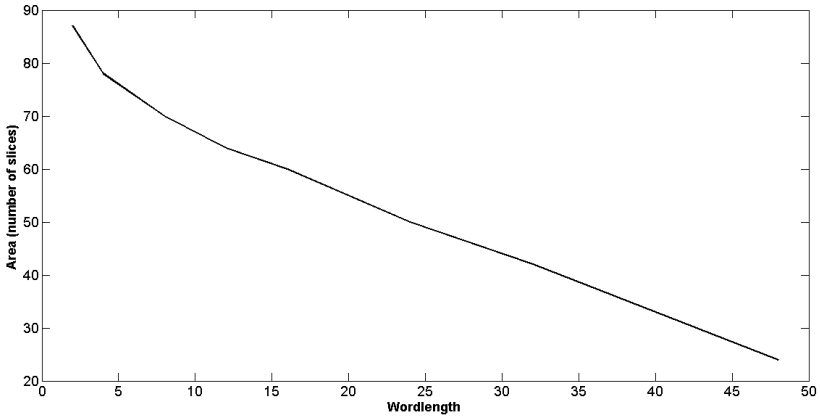

Figure 3.29. Compressor Wordlength versus Delay of 48-bit CSLA for Spartan-3 FPGA.

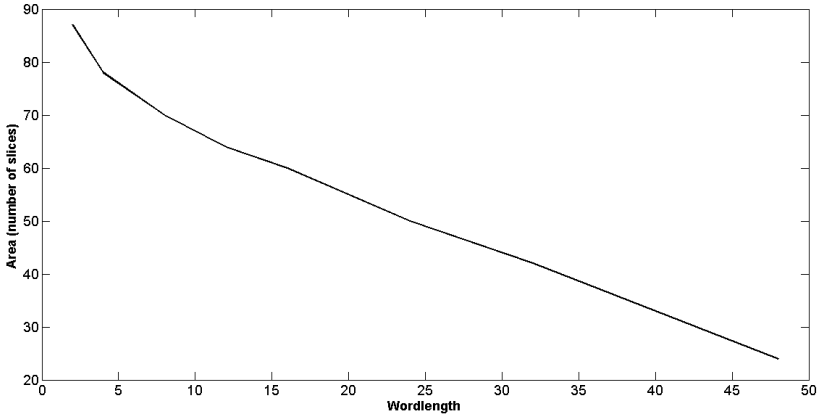

Figure 3.30. Compressor Wordlength versus Area of 48-bit CSLA for Spartan-3 FPGA.

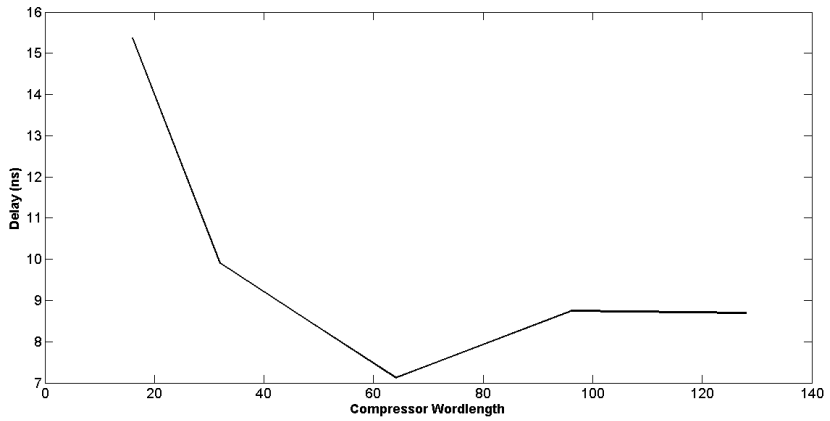

Figure 3.31. Compressor Wordlength versus Delay of 48-bit CSLA for Spartan-3 FPGA.

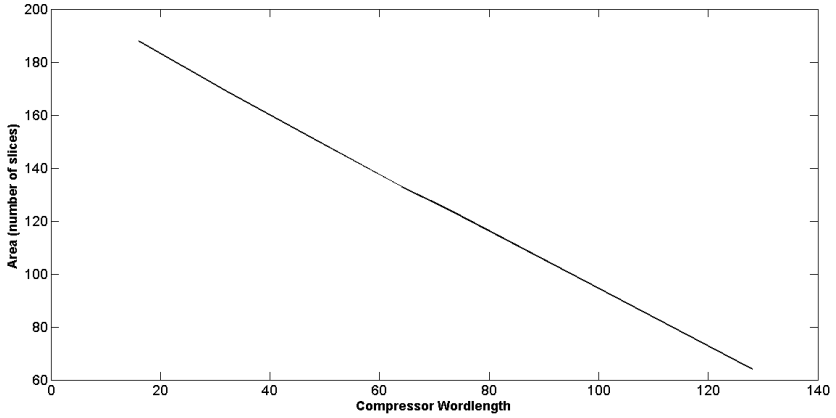

Figure 3.32. Compressor Wordlength versus Area of 48-bit CSLA for Spartan-3 FPGA.

CSLA Delay for Spartan-3 FPGA. Equation 3.14 gives the delay function of CSLA for Spartan-3 FPGA. Here, x corresponds to adder wordlength, y corresponds to wordlength of compressor blocks, z corresponds to delay in nanoseconds.

$$\begin{aligned}
Z &= 9.865 - 0.07657 * x - 0.1425 * y + 15.53 * 10^{-4} * x^2 \\
&+8.949 * 10^{-4} * x * y + 2.489 * 10^{-3} * y^2 - 2.708 * 10^{-5} * x^2 * y \\
&+4.823 * 10^{-5} * x * y^2 - 3.3 * 10^{-5} * y^3 \quad when \quad 64 < x \leq 128 \\
Z &= 148.5 - 2.298 * x - 0.5797 * y + 1.023 * 10^{-2} * x^2 \\
&+1.191 * 10^{-2} * x * y + 2.958 * 10^{-3} * y^2 - 9.354 * 10^{-5} * x^2 * y \\
&+8.293 * 10^{-5} * x * y^2 - 3.055 * 10^{-5} * y^3 \quad when \quad 128 < x \leq 192 \\
Z &= 111 - 0.525 * x - 1.029 * y + 8.363 * 10^{-4} * x^2 \\
&+2.512 * 10^{-3} * x * y + 4.592 * 10^{-3} * y^2 - 1.535 * 10^{-6} * x^2 * y \\
&+4.404 * 10^{-6} * x * y^2 - 7.091 * 10^{-6} * y^3 \quad when \quad 192 < x \leq 256
\end{aligned} \tag{3.14}$$

CSLA Area for Spartan-3 FPGA. Equation 3.15 gives the area function of CSLA for Spartan-3 FPGA. Here, x corresponds to adder wordlength, y corresponds to wordlength of compressor blocks, z corresponds area in number of slices.

$$\begin{aligned}
Z &= 2.491 + 1.583 * x - 1.139 * y + 7.304 * 10^{-5} * x^2 \\
&-1.785 * 10^{-4} * x * y + 4.382 * 10^{-4} * y^2 \quad when \quad 64 < x \leq 128 \\
Z &= 0.6903 + 1.593 * x - 1.098 * y + 2.386 * 10^{-5} * x^2 \\
&-5.975 * 10^{-5} * x * y + 9.672 * 10^{-5} * y^2 \quad when \quad 128 < x \leq 192 \\
Z &= 502.2 - 3.208 * x - 0.8307 * y + 1.058 * 10^{-2} * x^2 \\
&+4.258 * 10^{-4} * x * y - 8.546 * 10^{-4} * y^2 \quad when \quad 192 < x \leq 256
\end{aligned} \tag{3.15}$$

CSLA Delay for Virtex-4 FPGA. Equation 3.16 gives the delay function of CSLA for Virtex-4 FPGA. Here, x corresponds to adder wordlength, y corresponds to

wordlength of compressor blocks, z corresponds to delay in nanoseconds.

$$\begin{aligned} Z &= 4.653 - 0.05059 * x - 0.0502 * y + 6.962 * 10^{-4} * x^2 \\ &+3.68 * 10^{-4} * x * y + 5.38 * 10^{-3} * y^2 - 1.324 * 10^{-5} * x^2 * y \\ &+1.782 * 10^{-5} * x * y^2 - 1.077 * 10^{-5} * y^3 \quad when \quad 64 < x \leq 128 \\ Z &= 10.79 - 0.1151 * x - 0.06393 * y + 6.663 * 10^{-4} * x^2 \\ &+6.245 * 10^{-4} * x * y + 2.021 * 10^{-4} * y^2 - 7.807 * 10^{-6} * x^2 * y \\ &+9.657 * 10^{-6} * x * y^2 - 5.272 * 10^{-6} * y^3 \quad when \quad 128 < x \leq 192 \\ Z &= 0.6835 - 1.646 * 10^{-2} * x - 1.396 * 10^{-2} * y + 1.026 * 10^{-4} * x^2 \\ &-1.96 * 10^{-4} * x * y + 1.539 * 10^{-4} * y^2 - 1.515 * 10^{-6} * x^2 * y \\ &+3.275 * 10^{-6} * x * y^2 - 1.939 * 10^{-6} * y^3 \quad when \quad 192 < x \leq 256 \end{aligned} \tag{3.16}$$

CSLA Area for Virtex-4 FPGA. Area function for Virtex-4 FPGA is the same with Spartan-3 FPGA which is given in Equation 3.15. In this equation, x corresponds to adder wordlength, y corresponds to wordlength of compressor blocks, z corresponds area in number of slices.

3.3.2.4. Multiplexer. As described in Section 3.2.3, Xilinx FPGAs have a special architecture which can synthesize multiplexers with 2, 4, 8, 16 and 32 ports perfectly. Because of that reason, in this tool only multiplexers with port size 2, 4, 8, 16 and 32 are used. For other values, closest higher input portsize value is used. For example, when a multiplexer with 15 ports is required, a multiplexer with 16 ports is used where one of the port is left unconnected. Piece-wise polynomial functions are used for curve-fitting of the multiplexer. In the following paragraphs, the model functions and the plot of these functions are given for Xilinx Spartan-3 family FPGA.

Multiplexer Delay for Spartan-3 FPGA. Delay function of the multiplexer is a piece-wise linear function. The delay function of multiplexer for Spartan-3 FPGA is

given in Equation 3.17.

$$Y = 0.479X_{(0,2]} + 0.793X_{(2,4]} + 1.091X_{(4,8]} + 1.389X_{(8,16]} + 1.687X_{(16,32]} + 3.017X_{(32,64]} \tag{3.17}$$

Here, x defines the port size of the multiplexer and y corresponds to delay in nanoseconds. Delay of the multiplexer is independent from the wordlength of the multiplexer. The explicit definition of Equation 3.17 can be seen in Equation 3.18.

$$\begin{aligned} Y &= 0.479 \quad when \quad 0 < x \leq 2 \\ Y &= 0.793 \quad when \quad 2 < x \leq 4 \\ Y &= 1.091 \quad when \quad 4 < x \leq 8 \\ Y &= 1.389 \quad when \; 8 < x \leq 16 \\ Y &= 1.687 \; when \; 16 < x \leq 32 \\ Y &= 3.017 \; when \; 32 < x \leq 64 \end{aligned} \tag{3.18}$$

Figure 3.33 shows the delay function of the multiplexer for Spartan-3 FPGA.

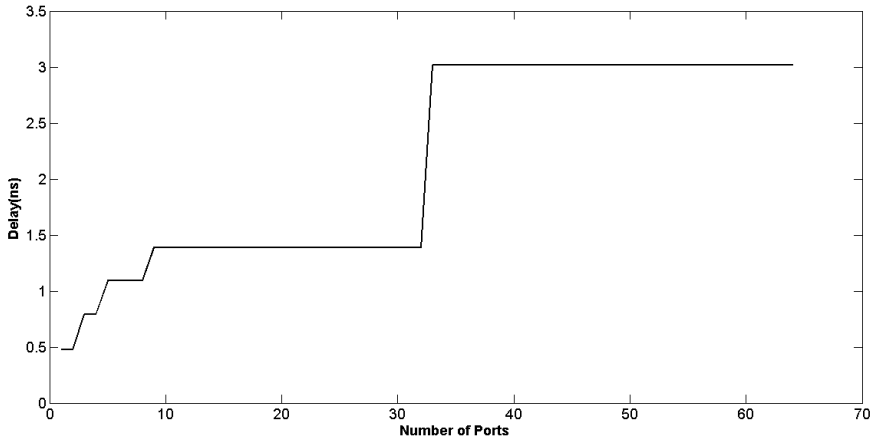

Figure 3.33. Delay Function of the Multiplexer for Spartan-3 FPGA.

Multiplexer Area for Spartan-3 FPGA. Area function of the multiplexer is a piece-wise polynomial function that depends on both the port size of the multiplexer and the wordlength. The area function of the multiplexer for Spartan-3 FPGA is given in Equation 3.19. In that equation, x corresponds to port size, y corresponds to wordlength of the multiplexer in bits, z corresponds to area in number of slices.

$$\begin{aligned} Z &= 0.5 * y \;\; when \;\; 0 < x \leq 2 \\ Z &= 1 * y \;\; when \;\; 2 < x \leq 4 \\ Z &= 2 * y \;\; when \;\; 4 < x \leq 8 \\ Z &= 4 * y \;\; when \; 8 < x \leq 16 \\ Z &= 8 * y \;\; when \; 16 < x \leq 32 \\ Z &= 17 * y \; when \; 32 < x \leq 64 \end{aligned} \tag{3.19}$$

Figure 3.34 shows the plot of multiplexer area functions with respect to the wordlength for Spartan-3 FPGA.

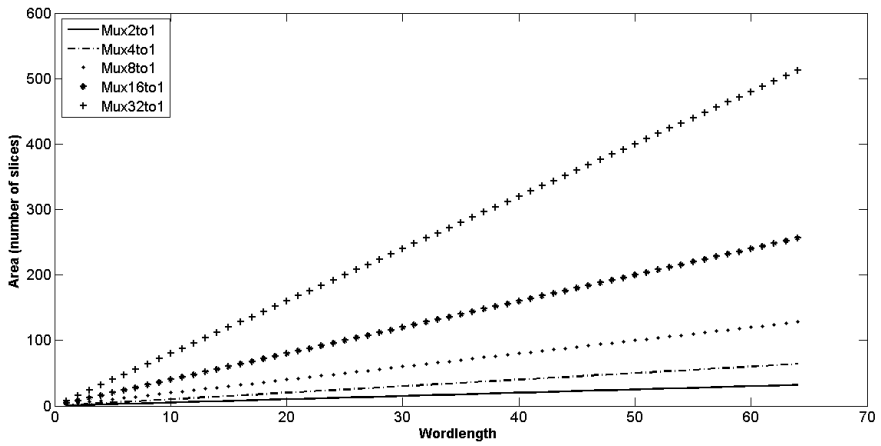

Figure 3.34. Area Function of the Multiplexer for Spartan-3 FPGA.

Multiplexer Power for Spartan-3 FPGA. Power of the multiplexer is a piece-wise polynomial function that depends on both the port size of the multiplexer and the wordlength. The power function of multiplexer for Spartan-3 FPGA is given

in Equation 3.20. In that equation, x corresponds to port size, y corresponds to wordlength of the multiplexer in bits, P corresponds to power in miliwatts.

$$
\begin{aligned}
P = {} & -1.119941 * 10^{-7} * y^4 + 1.429113 * 10^{-5} * y^3 - 3.841262 * 10^{-4} * y^2 \\
& + 1.283894 * 10^{-2} * y - 1.553344 * 10^{-2} \quad when \quad 0 < x \leq 2 \\
P = {} & 1.903192 * 10^{-7} * y^4 - 3.391324 * 10^{-5} * y^3 + 2.121420 * 10^{-3} * y^2 \\
& - 1.397037 * 10^{-2} * y + 5.057577 * 10^{-2} \quad when \quad 2 < x \leq 4 \\
P = {} & 9.0433559 * 10^{-7} * y^4 - 1.253094 * 10^{-4} * y^3 + 5.819870 * 10^{-3} * y^2 \\
& - 3.372168 * 10^{-2} * y + 1.037093 * 10^{-1} \quad when \quad 4 < x \leq 8 \\
P = {} & 3.562281 * 10^{-4} * y^2 + 1.010322 * 10^{-1} * y + 2.052144 * 10^{-1} \\
& when \ 8 < x \leq 16 \\
P = {} & 3.765742 * 10^{-5} * y^3 - 1.969882 * 10^{-3} * y^2 - 2.081174 * 10^{-1} * y \\
& + 1.762812 * 10^{-1} \ when \ 16 < x \leq 32 \\
P = {} & 6.230542 * 10^{-3} * y^2 + 1.512600 * 10^{-1} * y + 3.574 * 10^{-1} \\
& when \ 32 < x \leq 64
\end{aligned}
\tag{3.20}
$$

Figure 3.35 shows the plot of multiplexer power functions with respect to the wordlength for Spartan-3 FPGA.

Multiplexer Delay for Virtex-4 FPGA. The delay function of multiplexer for Spartan-3 FPGA is given in Equation 3.21. Here, x defines the port size of the multiplexer and y corresponds to delay in nanoseconds. The Figure 3.36 shows the delay

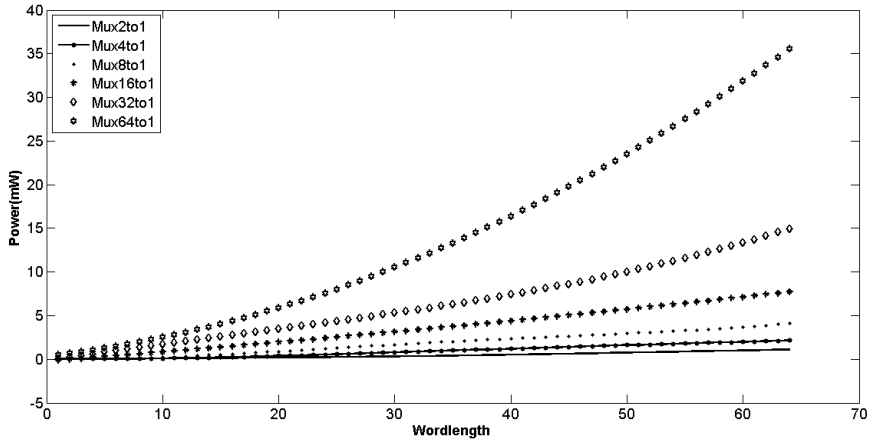

Figure 3.35. Power Function of the Multiplexer for Spartan-3 FPGA.

function of the multiplexer for Virtex-4 FPGA.

$$Y = 0.147X_{(0,2]} + 0.438X_{(2,4]} + 0.738X_{(4,8]} + 1.038X_{(8,16]} + 1.338X_{(16,32]} + 1.921X_{(32,64]} \tag{3.21}$$

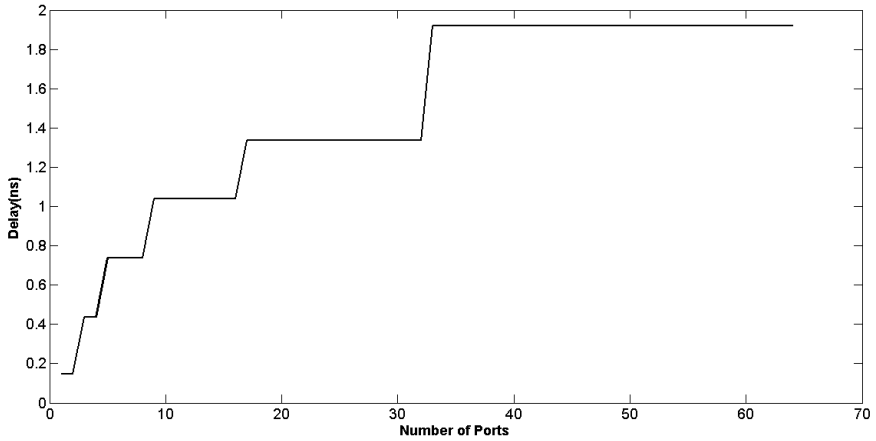

Figure 3.36. Delay Function of the Multiplexer for Virtex-4 FPGA.

Multiplexer Area for Virtex-4 FPGA. The area values and function is the same with Spartan-3 FPGA. The obtained function is given in Equation 3.19 and is shown in Figure 3.34.

Multiplexer Power for Virtex-4 FPGA. The power function of multiplexer for Virtex-4 FPGA is given in Equation 3.22. In that equation, x corresponds to port size, y corresponds to wordlength of the multiplexer in bits, P corresponds to power in miliwatts. The Figure 3.37 shows the plot of multiplexer power functions with respond to the wordlength for Virtex-4 FPGA.

$$
\begin{aligned}
P &= 8.232878 * 10^{-5} * y^2 + 7.14039 * 10^{-3} * y - 2.2282 * 10^{-4} \quad when \quad 0 < x \leq 2 \\
P &= 2.094261 * 10^{-4} * y^2 - 1.201691 * 10^{-2} * y - 1.901691 * 10^{-2} \quad when \quad 2 < x \leq 4 \\
P &= 6.32192 * 10^{-8} * y^4 - 1.171733 * 10^{-5} * y^3 + 9.543956 * 10^{-4} * y^2 - 8.756838 * 10^{-3} * y \\
&\quad + 1.174720 * 10^{-2} \quad when \quad 4 < x \leq 8 \\
P &= 2.583598 * 10^{-7} * y^4 - 2.69511 * 10^{-5} * y^3 + 1.352495 * 10^{-3} * y^2 + 2.483099 * 10^{-2} * y \\
&\quad + 3.543867 * 10^{-3} \quad when \ 8 < x \leq 16 \\
P &= 2.304150 * 10^{-9} * y^6 - 4.071611 * 10^{-7} * y^5 + 2.650932 * 10^{-5} * y^4 - 7.556543 * 10^{-4} * y^3 \\
&\quad + 9.74488 * 10^{-3} * y^2 + 1.941828 * 10^{-2} * y + 2.60367 * 10^{-2} \ when \ 16 < x \leq 32 \\
P &= 5.958643 * 10^{-5} * y^3 - 1.158905 * 10^{-3} * y^2 + 1.252257 * 10^{-1} * y^1 + 1.458445 * 10^{-1} \\
&\quad when \ 32 < x \leq 64
\end{aligned}
\tag{3.22}
$$

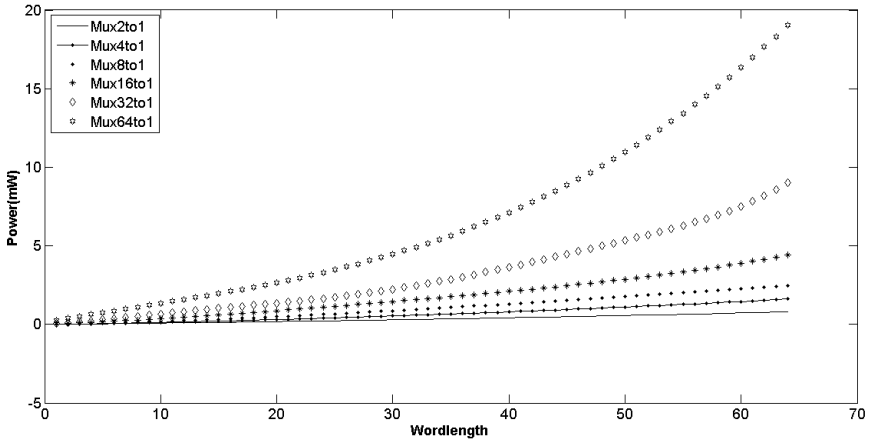

Figure 3.37. Power Function of the Multiplexer for Virtex-4 FPGA.

3.4. Hardware Behavior Description File (HBD) Generation

Hardware Behavior Description (HBD) files describes the property behavior of the template or component in a structured way. This property may be delay, area and power consumption. Every arithmetic operator has an HBD file, and these HBD files are used for the calculation of the delay, area and power of an operator in the application graph. These costs are used in Performance and Area Estimation module (see Section 4), which estimates the delay and area of the graph, and Rescheduling module (see Section 6), where the clock period of the optimized graph is extracted and the optimized graph is rescheduled according to that clock period. The use of HBD file provides the extendibility of the RH(+) tool by allowing the user to insert their new operators and describe their model property to the tool. User can describe the delay, area and power model of the newly introduced operator by preparing the HBD file in its format. HBD file has a specific format which can describe polynomial or a piecewise polynomial functions.

3.4.1. Hardware Behavior Description File

This description is stored in a file and used by RH(+) in order to generate behavior description functions. It has the file extension of “hbd”. The properties of

HBD file are;

- The HBD file is not case sensitive
- The HBD file starts with @synth and ends with @endsynth
- *variable, *calculationvariable and *portselectvariable are used for defining the variable names. These fields and function description after these fields starts with '*', they can be written in the same line if these fields separated with '*'. It is better to write line by line for readability.
- The lines between every @hbd and @endhbd define one function
- *Property defines the property that function calculates. For example *delay*, *area* and *power*.
- *FunctionType defines the type of the function of that property. It can be *poly* for polynomial function, *piecewise* for piecewise polynomial function.
- The lines between *FunctionType and @endhbd describes the function to be calculated

Table 3.3 summarizes the keywords, corresponding value, explanation of the keywords for the HBD file format.

The syntax and the properties of the HBD file is explained with the following examples;

3.4.1.1. HBD File Example 1. The first example HBD file is given in Figure 3.4.1.1.

- As described, the hbd file starts with @synth and ends with @endsynth.
- *variable defines that the variable names are x and y. This also shows that the function is 2 dimensional.
- The first @hbd and @endhbd part defines first function. "*Property = delay" defines that this function is used for calculation of delay. "FunctionType =poly" defines that this is a polynomial function. *(2_3:10.2)+(3_2:4)+(1_2:3)+(2_1:7.2)

Table 3.3. HBD File Keywords.

Keyword	**Value**	**Explanation**
@synth	N/A	defines the start of the hbd file
*variable	(<v1>),(<v1>,<v2>), (<v1>, <v2>, <v3>)	defines the variable names used in polynomial functions. The variable number must be the same with dimension of the function. The order of variables are important because for example the function defined as (2_1:10) is $10 * x^2 * y$ when variable value is (x,y) and $10*y^2*x$ when variable value is (y,x)
*portselectvariable	(<v1>), (<v2>)	defines the name of the variable which is used for port selecting when the function is piecewise polynomial
*calculationvariable	(<v1>), (<v1>,<v2>)	defines the name of the variable which are used in the calculation of the function when the function is piecewise polynomial
@hbd	N/A	implies that function description of one model property is started
@endhbd	N/A	implies that function description of one model property is finished
*property	Delay, Area, Power	defines the property that function describes
*FunctionType	Poly, Piecewise	defines the function type. poly corresponds to polynomial function, piecewise corresponds to piecewise polynomial function
*port	<v1>;<v2>	In piecewise functions, it defines the lower and upper boundaries. For example, 0;2 means $0<variable \leqslant 2$

```
@synth
*variable = (x,y)
@hbd
*Property = Delay
*functionType = poly
*(2_3:10.2)+(3_2:4)+(1_2:3)+(2_1:7.2)
@endhbd
@hbd
*Property = Area
*functionType = poly
*(2_1:8)+(3_2:3)+(1_3:2)+(4_2:7.2)
@endhbd
@hbd
*Property = Power
*functionType = poly
*(2_3:3)+(7_2:3)+(1_2:10)+(3_1:7)
@endhbd
@endsynth
```

Figure 3.38. HBD file example 1.

describes the function of

$$10.2x^2y^3 + 4x^3y^2 + 3xy^2 + 7.2x^2y \tag{3.23}$$

If the variable name was given like "*variable = (y,x)" the function would describe

$$10.2y^2x^3 + 4y^3x^2 + 3yx^2 + 7.2y^2x \tag{3.24}$$

- The second @hbd and @endhbd part defines second function."*Property = area" defines that this function is used for calculation of area. "FunctionType = poly" defines that this is a polynomial function. *(2_1:8)+(3_2:3)+(1_3:2)+(4_2:7) describes the function of

$$8x^2y + 3x^3y^2 + 2xy^3 + 7x^4y^2 \tag{3.25}$$

- The third @hbd and @endhbd part defines third function."*Property = power" defines that this function is used for calculation of power. "FunctionType = poly" defines that this is a polynomial function. *(2_3:3)+(7_2:3.1)+(1_2:10)+(3_1:7) defines the function of

$$3x^2y^3 + 3.1x^7y^2 + 10xy^2 + 7x^3y \tag{3.26}$$

- The result of this hbd file when executed with x=1 and y=2; 123.6 for delay, 72 for area, 90 for power.
- While defining function, the left side of ":" defines the constant, the right side of ":" describe exponents. The exponents are separated by "_"s , for example, 2_3_4_5 describes xˆ2*yˆ3*zˆ4*tˆ5.

3.4.1.2. HBD File Example 2. The second example HBD file is given in Figure 3.4.1.2.

```
@synth
*variable = (x)
@hbd
*Property = Delay
*functionType = poly
*(1:10.2)+(2:4)+(3:3)+(4:7)
@endhbd
@hbd
*Property = Area
*functionType = poly
*(1:8)+(2:3)+(3:2)+(4:7)+(5:1)
@endhbd
@hbd
*Property = Power
*functionType = poly
*(2:3)+(1:2.2)+(3:10.5)+(4:7.5)+(0:4)
@endhbd
@endsynth
```

Figure 3.39. HBD file example 2.

- As described, the hbd file starts with @synth and ends with @endsynth.
- *variable defines that the variable name is x. This also shows that the function is 1 dimensional.
- The first @hbd and @endhbd part defines first function. "*Property = delay" defines that this function is used for calculation of delay. "FunctionType = poly" defines that this is a polynomial function. "*(1:10.2)+(2:4)+(3:3)+(4:7)" defines the function of

$$10.2x + 4x^2 + 3x^3 + 7x^4 \tag{3.27}$$

- The second @hbd and @endhbd part defines second function. "*Property =area" defines that this function is used for calculation of area. "FunctionType = poly" defines that this is a polynomial function. "*(1:8)+(2:3)+(3:2)+(4:7)+(5:1)" defines the function of

$$8x + 3x^2 + 2x^3 + 7x^4 + 1x^5 \tag{3.28}$$

- The third @hbd and @endhbd part defines third function. "*Property =power" defines that this function is used for calculation of power. "FunctionType = poly" defines that this is a polynomial function. "*(2:3)+(1:2.2)+(3:10.5)+(4:7.5)+(0:4)" defines the function of

$$3x^2 + 2.2x + 10.5x^3 + 7.5x^4 + 4 \tag{3.29}$$

- The result of this hbd file when executed with x=2 , 172.4 for delay, 188 for area, 224.4 for power.

3.4.1.3. HBD File Example 3. The third example HBD file is given in Figures 3.4.1.3 and 3.4.1.3.

- As described, the hbd file starts with @synth and ends with @endsynth.

```
@synth
*portselectvariable = (x)
*calculationvariable = (y)

@hbd
*Property=Delay
*functionType = piecewise
*port=0;2
*(0:0.479)
*port = 2;4
*(0:0.793)
*port=4;8
*(0:1.091)
*port=8;16
*(0:1.687)
*port=16;32
*(0:3.017)
*port=32;64
*(0:5.7)
@endhbd

@hbd
*Property=Area
*functionType = piecewise
*port=0;2
*(1:0.5)
*port = 2;4
*(1:1)
*port=4;8
```

Figure 3.40. HBD file example 3.

```
*(1:2)
*port=8;16
*(1:4)
*port=16;32
*(1:8)
*port=32;64
*(1:17)
@endhbd

@hbd
*Property=Power
*functionType = piecewise
*port=0;2
*(2:8)+(1:3)+(0:3.2)
*port = 2;4
*(2:4)+(1:2)
*port=4;8
*(3:0.4)+(1:0.3)+(0:0.23)+(2:5)
*port=8;16
*(3:2)+(2:0.3)+(1:0.23)
*port=16;32
*(3:7)+(1:5)+(2:1)
*port=32;64
*(2:0.4)+(3:5)+(1:0.2)
@endhbd
@endsynth
```

Figure 3.41. HBD file example 3 cont.

- "*portselectvariable = (x)" shows that the variable x is used for selecting the right function to be calculated. "*calculationvariable = (y)" shows that y variable will be used in functions for calculations when the function is piecewise polynomial. The existence of portselectvariable and calculationvariable shows that functions are piecewise functions.
- The first @hbd and @endhbd part describes the first function. "*Property=Delay" shows that this function is used for calculation of delay. "FunctionType = piecewise_linear" defines that this is a piecewise linear function. Between functiontype and @endhbd the function is described as below.
  - "*port=0;2
    *(0:0.479)" shows that x is 0.479 between (0,2].
  - "*port=2;4
    *(0:0.793)" shows that x is 0.793 between (2,4].
    - So the delay function describes;
    - Result = 0.479 when x = (0,2]
    - Result = 0.793 when x = (2,4]
    - Result = 1.091 when x = (4,8]
    - Result = 1.687 when x = (8,16]
    - Result = 3.017 when x = (16,32]
    - Result = 5.7 when x = (32,64]
- The second @hbd and @endhbd part describes the second function. "*Property=Area" shows that this function is used for calculation of area. "FunctionType = piecewise_linear" defines that this is a piecewise linear function. Between functiontype and @endhbd the function is described as below. Because of the fact that, area function is described here, calculation variable is taken into account.(calculation variable accounts for wordlength for multiplexer)
  - "*port=0;2
    *(1:0.5)" shows that the result is 0.5*y when x = (0,2].
  - "*port = 2;4
    *(1:1)" shows that the result is 1*y when x = (2,4].
    - So the area function describes;

  - Result = 0.5 * y when x = (0,2]
  - Result = 1y when x = (2,4]
  - Result = 2y when x = (4,8]
  - Result = 4y when x = (8,16]
  - Result = 8y when x = (16,32]
  - Result = 17y when x = (32,64]

- The third @hbd and @endhbd part describes the third function. "*Property=Power" shows that this function is used for calculation of power. "FunctionType = piecewise" defines that this is a piecewise polynomial function. Between functiontype and @endhbd the function is described as below.
  - "*port=0;2
    *(2:8)+(1:3)+(0:3.2)" shows when x = (0,2] the function is

$$8y^2 + 3y + 3.2 \tag{3.30}$$

  - "*port = 2;4
    *(2:4)+(1:2)" shows when x = (2,4] the function is

$$4y^2 + 2y \tag{3.31}$$

    - So power function describes
    - 8y^2 + 3y +3.2 when x = (0,2]
    - 4y^2 +2y when x = (2,4]
    - 0.4y^3 +0.3y +0.23 +5y^2 when x = (4,8]
    - 2y^3 +0.3y^2 +0.23y when x = (8,16]
    - 7y^3 +5y +1y^2 when x = (16,32]
    - 0.4y^2 +5y^3 +0.2y when x = (32,64]

3.4.1.4. HBD File Example 4. The fourth example HBD file is given in Figure 3.4.1.4.

```
@synth
*portselectvariable = (x)
*calculationvariable = (x,y)
@hbd
*Property=Delay
*functionType = piecewise
*port=0;36
*(1_2:10)+(3_2:4)+(1_1:3)+(0_0:7)
*port = 36;48
*(1_2:5)+(3_1:4)+(1_3:1)+(2_1:3)
*port=48;64
*(3_2:7)+(2_3:2)+(5_3:1)+(4_2:10)
@endhbd
@hbd
*Property=Area
*functionType = piecewise
*port=0;36
*(2_2:3.1)+(2_3:4)+(3_3:7.2)+(1_2:7)
*port = 36;48
*(1_2:10.003)+(3_2:4)+(1_3:3)+(3_1:5)
*port=48;64
*(7_2:2.2)+(3_4:4)+(2_1:2)+(2_2:1)
@endhbd

@endsynth
```

Figure 3.42. HBD file example 4.

- As described, the hbd file starts with @synth and ends with @endsynth.
- "*Portselectvariable = x" defines that x is used for selecting the correct function and "*calculationvariable = (x,y)" defines that x and y are used in functions for calculation. Two calculation variables also show that the functions are 2 dimensional. The difference of this example than the previous one is that x here used both as portselect and calculation variable.
- The first @hbd and @endhbd part describes the first function. "*Property=Delay" shows that this function is used for calculation of delay. "FunctionType = piecewise" defines that this is a piecewise polynomial function. Between functiontype and @endhbd the function is described as below.
  - "*port=0;36
    *(1_2:10)+(3_2:4)+(1_1:3)+(0_0:7)" shows that when x is (0,36] the function is

$$10 * x * y^2 + 4 * x^3 * y^2 + 3 * x * y + 7 \tag{3.32}$$

  - "*port = 36;48
    *(1_2:5)+(3_1:4)+(1_3:1)+(2_1:3)" shows that when x is (36,48] the function is

$$5 * x * y^2 + 4 * x^3 * y + 1 * x * y^3 + 3 * x^2 * y \tag{3.33}$$

- The second @hbd and @endhbd part describes the second function. "*Property=area" shows that this function is used for calculation of area. "FunctionType = piecewise" defines that this is a piecewise polynomial function. Between functiontype and @endhbd the function is described as below.
  - "*port=0;36
    *(2_2:3.1)+(2_3:4)+(3_3:7.2)+(1_2:7)" shows that when x is (0,36] the function is

$$3.1 * x^2 * y^2 + 4 * x^2 * y^3 + 7.2 * x^3 * y^3 + 7 * x * y^2 \tag{3.34}$$

3.5. Parameter and Template File Creation

Template and parameter files are used in the tool for the generation of the VHDL files of the arithmetic components during Golden RTL generation and Optimized RTL generation. Template File is a VHDL component file of the arithmetic operator which has fixed fields that can not be changed and editable fields which can be filled or edited with using parameter files. Parameter file stores the key fields that needs to be changed in the template file and their corresponding values. Template files have the file extension of ".temp" and parameter files have the file extension of ".prm". During VHDL file generation of the components parameter file is used to fill the fields in the template file. Table 3.4 and Table 3.5 shows the fields that a parameter file can contain and the explanation of those fields.

3.5.1. Example Parameter and Template File of RCA

The parameter file in Figure 3.43 is an example parameter file for Ripple Carry Adder. The field of "Subtypename" shows that this is RCAAdder, "p_size" shows that one parameter exists which is "BITSIZE_0". There are two input ports and one output port. "BITSIZE_0" corresponds to bit-size of the input port. The template file of the Ripple Carry Adder is shown in Figure 3.44. The fields in the template file are replaced with the values in parameter file during VHDL component file generation. The generated VHDL file is shown in Figure 3.45.

3.5.2. Example Template and Parameter File of CSLA

The parameter file in Figure 3.46 is an example parameter file for Carry Select Adder. The field of "Subtypename" shows that this is CSLAAdder, "p_size" shows that two parameters exist which are "BITSIZE_0" and "BLOCKSIZE_0". There are 2 input ports and 1 output port. "BITSIZE_0" corresponds to bit-size of the input port and "BLOCKSIZE_0" corresponds to bit-size of the compressor blocks used in the CSLA. The template file of the CSLA is shown in Figure 3.47; <template_id>, <component_id>, <BITSIZE_0>and <BLOCKSIZE_0>fields are replaced with the

Table 3.4. Parameter File Keywords and Their Explanations 1.

Keyword	**Value**	**Explanation**
@param	N/A	defines the start of the param file
*isComponentorTemplate	Template, Component	it is used for determining whether the given VHDL file is template or component
*subtypename	RCAAdder, CSLAAdder, RCASubtractor, CSLA-Subtractor, SignedDivider, SignedMultiplier	defines the component type. As the new components introduced in the RH(+) library, new types can be added.
templateid	a number	The database ID of the template. This field is filled automatically by the database. When a new template is introduced by the user, this field is not filled. When this new template is defined in the database, it gets a template ID and later this template is is used in parameter file.
componentid	a number	The database ID of the component. This field is filled automatically by the database. When a new component is introduced by the user, this field is not filled. When this new component is defined in the database, it gets a component ID and later this component is used in parameter file.
p_size	a number	p_size is the parameters size and defines how many parameters change in template for example, for RCA, only bitsize is the parameter so p_size is 1, for CSLA adder bitsize and compressor size changes so p_size is 2.

Table 3.5. Parameter File Keywords and Their Explanations 2 cont.

Keyword	**Value**	**Explanation**
BITSIZE_0	a number	Bit size of a port. This field is used to fill the places of <BITSIZE_0>in templates with the value here. BITSIZE_0 is used as parameter variable to define a port of the template and it must exist in template as <BITSIZE_0>
BITSIZE_1	a number	Bit size of an another port. This field is used to fill the places of <BITSIZE_1>in templates with the value here. BITSIZE_1 is used as a parameter variable to define another port of the template and it must exist in the template as <BITSIZE_1>. For now, in the tool, <BITSIZE_0>and <BITSIZE_1>are used, these values can be increased in parameter file where this keyword shall be placed in template file.
BLOCKSIZE_0	a number	Blocksize of the CSLA. This field is used to fill the places of the <BLOCKSIZE_0>in the template with the value here. CSLA is composed of blocks so the bitsize of the blocks are named as BLOCKSIZE in param file and template. It is used as parameter variable and it must exist in the template as <BLOCKSIZE_0>
num_of_input	a number	defines how many input ports exist
num_of_output	a number	defines how many output ports exist

```
@param
*isComponentOrTemplate=Template;
*subtypename=RCAAdder
*template_id=212
*component_id=213
*p_size=1
*BITSIZE_0=16
*num_of_input=2
*num_of_output=1
@endparam
```

Figure 3.43. Example Parameter File of RCA.

corresponding values during VHDL file generation of the components. The generated VHDL file is shown in Figure 3.48.

```
--------------------------------------------------------------------------------
-- Author :  Ender Culha Copyright :  CASLAB
--------------------------------------------------------------------------------
--Module Description :
--File RCA_adder.vhd
--- Module Name :  LUT_2, MUXCY, XORCY is used
--------------------------------------------------------------------------------
--Description :  This entity is the RCA Adder which uses the carry chain structure of the FPGA.
--The Code is written in low level structural style so as to use carry chain for every bit size.
--LUT 2s, MUXCYs and XORCYs are connected structurally.
--------------------------------------------------------------------------------
----------libraries-------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
------------entity ----------------------------------------------------------
entity <subtypename>_<template_id>_<component_id>_i1_<BITSIZE_0>_i2_<BITSIZE_0>is
generic (g_BITSIZE: integer := <BITSIZE_0>);
port (d_0 :  in std_logic_vector(g_BITSIZE-1 downto 0);
     d_1 :  in std_logic_vector(g_BITSIZE-1 downto 0);
     out_d_0 :  out std_logic_vector(g_BITSIZE downto 0));
end entity;
--------------------------------------------------------------------------------
architecture Structural of <subtypename>_<template_id>_<component_id>_i1_<BITSIZE_0>_i2_<BIT-
SIZE_0>is
..
..
..

end architecture;
```

Figure 3.44. Example Template File of RCA.

```
----------------------------------------------------------------------------------
-- Author :  Ender Culha Copyright :  CASLAB
----------------------------------------------------------------------------------
--Module Description :
--File RCA_adder.vhd
--- Module Name :  LUT_2, MUXCY, XORCY is used
----------------------------------------------------------------------------------
--Description :  This entity is the RCA Adder which uses the carry chain structure of the FPGA.
--The Code is written in low level structural style so as to use carry chain for every bit size.
--LUT 2s, MUXCYs and XORCYs are connected structurally.
----------------------------------------------------------------------------------------------------
----------libraries-----------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
-----------libraries for low level primitives----------------------------------
library UNISIM;
use UNISIM.VComponents.all;
use IEEE.STD_LOGIC_ARITH.ALL;
------------entity ------------------------------------------------------------
entity RCAAdder_001_5_i1_8_i2_8 is
 generic (g_BITSIZE: integer := 8 );
 port (d_0 :  in std_logic_vector(g_BITSIZE-1 downto 0);
       d_1 :  in std_logic_vector(g_BITSIZE-1 downto 0);
       out_d_0 :  out std_logic_vector(g_BITSIZE downto 0));
 end entity;
--------------------------------------------------------------------------------
 architecture Structural of RCAAdder_001_1_i1_8_i2_8 is
..
..
..

end architecture;
```

Figure 3.45. Generated VHDL file of RCA.

```
@param
*isComponentOrTemplate=Template;
*subtypename=CSLAAdder
*template_id=456
*component_id=321
*BITSIZE_0=16
*BLOCKSIZE_0=4
*p_size=2
*num_of_input=2
*num_of_output=1
@endparam
```

Figure 3.46. Example Parameter File of CSLA.

```
--------------------------------------------------------------------------------
-- Author :  Ender Culha Copyright :  CASLAB
--------------------------------------------------------------------------------
--Module Description :
--File CSLA_ adder.vhd
--- Module Name :  RCA_prim.vhd , mux.vhd ,muxarray.vhd are used
--------------------------------------------------------------------------------
--Description :  This entity is the CSLA adder, which takes the bit size of the adder and block
--size with a generic.  It uses 2to1 Multiplexer as a submodule for selecting right carry and
--- Mux array of 2to1 multiplexers for selecting right sum.  If the bitsize is a multiple of g_
BLOCKSIZE
-- there are blocks with equal sizes which is implemented in generate ifmod 1.  If the bitsize is
not mult
--iple of block size, last block is smaller than selected block size and it is implemeneted in
generate ifmod 2.
--------------------------------------------------------------------------------------------
-----------------------libraries-----------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-------------------------libraries for low level -------------------------------------------
library UNISIM;
use UNISIM.VComponents.all;
--------------------entity CSLA Adder---------------------------------------------------------
entity <subtypename>_<template_id>_<component_id>_i1_<BITSIZE_0>_i2_<BITSIZE_0>is
 generic (g_BITSIZE : integer := <BITSIZE_0>; g_BLOCKSIZE : integer := <BLOCKSIZE_0>);
 port ( d_0 :  in STD_LOGIC_VECTOR (g_BITSIZE-1 downto 0);
     d_1 :  in STD_LOGIC_VECTOR (g_BITSIZE-1 downto 0);
     out_d_0 :  out STD_LOGIC_VECTOR (g_BITSIZE downto 0));
end entity;
-----------------Architecture-------------------------------------------------------------
architecture Structural of <subtypename>_ <template_ id>_<component_id>_i1_<BITSIZE_0>_i2_
<BITSIZE_0>is
..
..
..
end Structural;
```

Figure 3.47. Example Template File of CSLA.

```
-------------------------------------------------------------------------------
-- Author :  Ender Culha Copyright :  CASLAB
-------------------------------------------------------------------------------
--Module Description :
--File CSLA_ adder.vhd
--- Module Name :  RCA_prim.vhd , mux.vhd ,muxarray.vhd are used
-------------------------------------------------------------------------------
--Description :  This entity is the CSLA adder, which takes the bit size of the adder and block
--size with a generic.  It uses 2to1 Multiplexer as a submodule for selecting right carry and
--- Mux array of 2to1 multiplexers for selecting right sum.  If the bitsize is a multiple of g_
BLOCKSIZE
-- there are blocks with equal sizes which is implemented in generate ifmod 1.  If the bitsize is
not mult
--iple of block size, last block is smaller than selected block size and it is implemeneted in
generate ifmod 2.
-------------------------------------------------------------------------------
-------------------------libraries-----------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-------------------------libraries for low level -----------------------------------------
library UNISIM;
use UNISIM.VComponents.all;
--------------------entity CSLA Adder-----------------------------------------------------
entity CSLAAdder_002_2_i1_16_i2_16 is
generic (g_BITSIZE : integer := 16 ; g_BLOCKSIZE : integer := 4);
port ( d_0 :  in STD_LOGIC_VECTOR (g_BITSIZE-1 downto 0);
     d_1 :  in STD_ LOGIC_ VECTOR (g_ BITSIZE-1 downto 0);
     out_d_0 :  out STD_LOGIC_VECTOR (g_BITSIZE downto 0));
end entity;
-----------------Architecture-----------------------------------------------------------
architecture Structural of CSLAAdder_002_2_i1_16_i2_16 is
..
..
..
end Structural;
```

Figure 3.48. Generated VHDL file of CSLA.

4. PERFORMANCE AND AREA ESTIMATION OF THE GOLDEN RTL

This estimation tool takes the application CDFG as input and make estimation in two levels; component level estimation and graph level estimation. Component level estimation is the estimation of every vertex in the graph and graph level estimation is the estimation of the overall graph. These steps are explained in the following sections.

4.1. Component Level Estimation

Firstly, the delay and area cost of each vertex in the graph is calculated. These costs are calculated according to the estimation model explained in Section 3.3. The tool reaches the estimation model by using Hardware Behavior Description (HBD) files. Each arithmetic operator in the component library has an HBD file and HBD files describe the estimation model of an operator which is explained in Section 3.4.1. During vertex estimation, the corresponding HBD file of the vertex operator is read, estimation model is extracted from that HBD file and according to bit-length information of that operator the delay and area cost of that vertex is calculated.

4.2. Graph Level Estimation

In component level estimation, delay and area of all vertices in the graph are calculated. In graph level estimation these costs are used so as to estimate delay and area of the whole graph. Area estimation of the graph is calculated by the summation of the areas cost of all vertices in the graph. For delay estimation, firstly the critical path is found for the graph. The critical path is found by using the vertex delay costs calculated in component level estimation. Then these costs are given as edge costs to the outgoing edges of the vertices. Afterwards, Directed Acyclic Graph (DAG) shortest path algorithm is applied. Here, the edge costs are negated while applying DAG shortest path algorithm [44]. Then, delays of the vertices in the critical path are

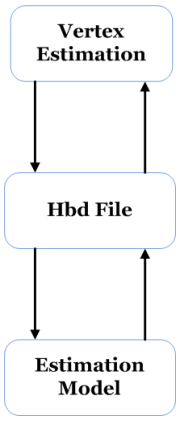

Figure 4.1. Component Level Estimation.

summed to calculate the delay of the entire graph.

In Figure 4.2, an example graph is shown. The number on the edges represent the signal wordlengths between vertices. The delay and areas of the vertices of the graph calculated by using the HBD files are shown in Table 4.1. The vertices in the critical path are shown in gray color. The total delay of the graph is SE1 + SE2 + SE4 + SE6 + SE7 + SE8 = 25.75 ns and the total area is SE1 + SE2 + SE3 + SE4 + SE5 + SE6 + SE7 + SE8 = 128 slices.

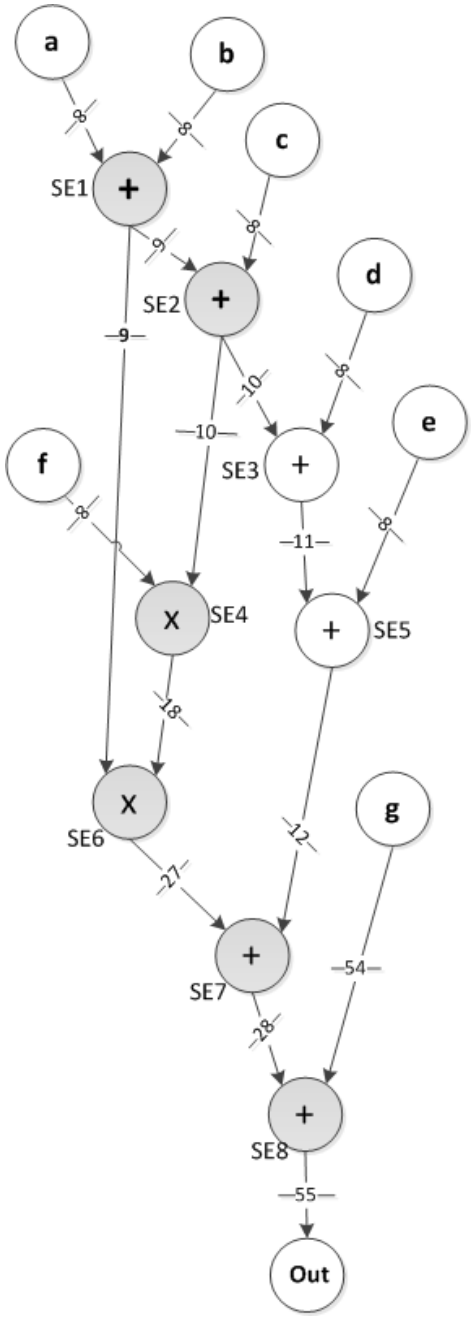

Figure 4.2. Critical Path of Example Graph.

Table 4.1. Calculated Delay and Area costs of the example graph.

Vertex Name	**Delay (ns)**	**Area (number of slices)**
SE1	1.978	4.015
SE2	2.037	4.55
SE3	2.09	5.05
SE4	7.42	40.59
SE5	8.62	82.76
SE6	2.54	9.04
SE7	2.59	9.55
SE8	4.55	27.03

5. GOLDEN RTL GENERATION

In Golden RTL generation, the VHDL equivalent of the application graph is generated without any optimization so in this generated circuit, no resource sharing and scheduling exist. The golden RTL can be used for quick and initial estimation of the area and delay of the design. The golden RTL is also used for the verification of the datapath by comparing the simulation results of the optimized generated datapath with the generated Golden RTL.

In Figure 1.1, it is shown that generation is splitted into two paths, on the left path, Golden RTL generation and on the right path Optimized RTL Generation. After the consumed area of the graph and delay is estimated, Golden RTL generation is done. Golden RTL is the RTL where there is no resource sharing so there is no need of extra processing on the graph before it is given as input to the Golden RTL generation tool.

5.1. Golden RTL Generation Methodology

The flow of the Golden RTL generation is shown in the Figure 5.1. CDFG is the input of the Golden RTL generation tool. Every operator in the CDFG corresponds to an operator and every edge corresponds to a relation between these operators. So, for every operator, a VHDL component file is created and then they are connected at the top level file. VHDL components are generated using parameter files and template files which are explained in Section 3.5. Moreover, the estimated delay is put as a clock constraint while generating user constraint file (UCF) for the Xilinx Synthesizer [1].

During Golden RTL generation, every vertex in the graph corresponds to a VHDL arithmetic component and every edge corresponds to a signal between vertices. In Figures 5.2 and 5.3, an example graph and its generated Golden RTL is given. Each vertex in the graph is mapped to a different arithmetic operator in VHDL and they are connected. There is no multiplexer required because there is no resource sharing between the operators. Since, the circuit is fully combinational, registers are not

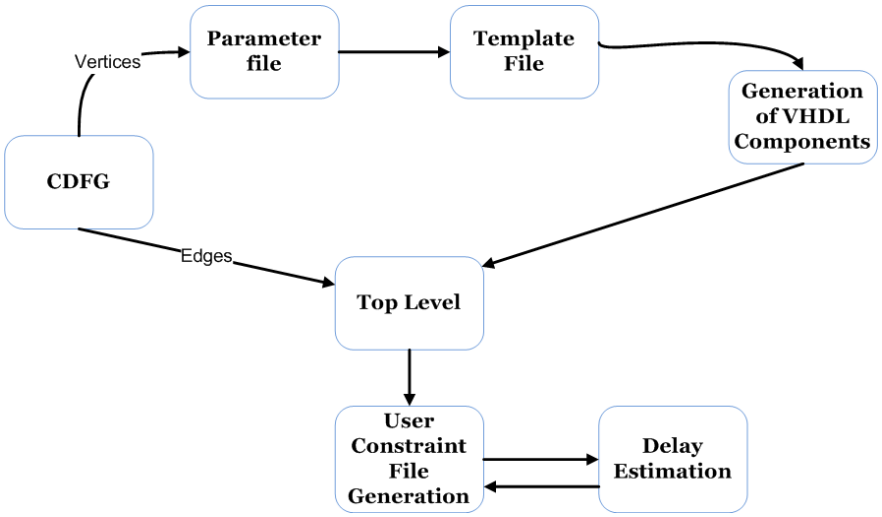

Figure 5.1. Block Diagram of Golden RTL Generation.

mandatory. However, registers are placed for the inputs and outputs of the Golden RTL if desired by the user. By placing input and output registers, Golden RTL becomes suitable to work with clock.

5.1.1. Commenting

In generated VHDL Code, components and signal names are not the same as the vertex and edge names in the data flow graph. There is a different naming convention for VHDL that is explained in Section 5.1.2. Hence, vertex and edge names and their corresponding component and signal names are held at the beginning of the top module VHDL file in the comment part. An example graph and its generated comment section are shown in Figures 5.4 and 5.5. As shown in the figures, it is written that which vertex in the graph corresponds to which component in the VHDL code and which edge in the graph corresponds to which signal in the VHDL code.

5.1.2. VHDL Naming Convention for Golden RTL

5.1.2.1. Components. The name of the components at the top level are the same as the entity names of the generated VHDL files to be connected at the top level. Since,

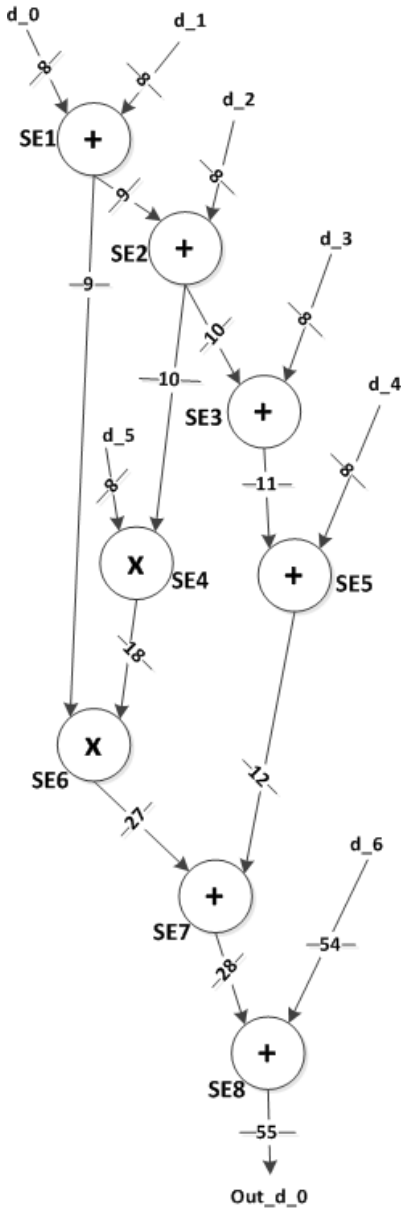

Figure 5.2. An Example Application Graph.

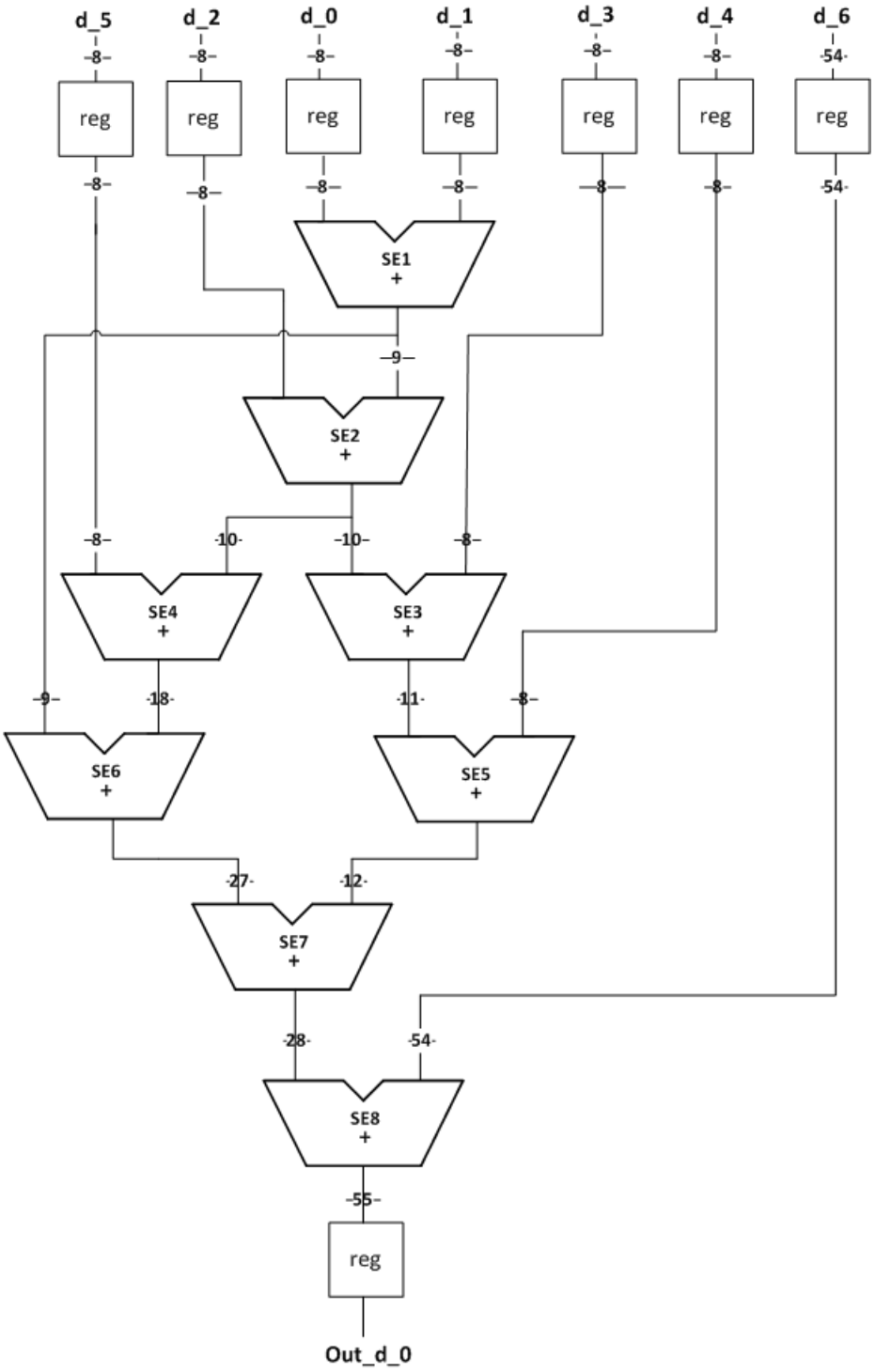

Figure 5.3. Generated Golden RTL of the Graph.

these component VHDL files are generated by using template files, which is given in Figures 3.44 and 3.47 in Section 3.5, the component name convention is the same as entity name convention. This convention is ;
<*subtypename*>_<*template_id*>_<*component_id*>_*i*1_<*BITSIZE_0*>_*i*2_
<*BITSIZE_0*>
where the explanation of the fields are given in Tables 3.4 and 3.5. For example, in RCAAdder_001_1_i1_8_i2_8, RCAAdder corresponds to subtypename, 001 corresponds to template id, 1 corresponds to component id, i1_8 corresponds to wordlength of first port and i2_8 corresponds to wordlength of second port.

5.1.2.2. Signals. Convention for the different kinds of signals are given in the following text.
Port Signals. The convention for input data signals are *d_*<*signal_number*> i.e d_0, d_1 etc. The naming convention for output data signals are *out_d_*<*signal_number*> i.e out_d_0, out_d_1.
Internal Signals. The naming convention for internal signals is the <*subtypename*>_<*instancenumber*>_*d_*<*portnumber*> i.e RCAAdder_0_d0, SignedMultiplier0_d0 etc. If the signal is registered the convention is <*signalname_r_*<*number*> where number corresponds to how many times the signal is registered. If the signal is combinational, this convention is not so as not to cause any complexity, but if it is registered once the numbering starts from 0. For example, d_0_r0 is the d_0 signal after it is registered for one time.
Extended Signals. The signals may required to be extended during port mapping. If the signal is extended the naming convention is <*signal_name*> i.e RCAAdder3_d0_e, d_3_r0_e.

5.1.3. Components

A VHDL component file is generated for every vertex which is an operator and these operators are connected at the top level VHDL file. During Golden RTL generation, all the vertices are traversed in the data flow graph by the software and compo-

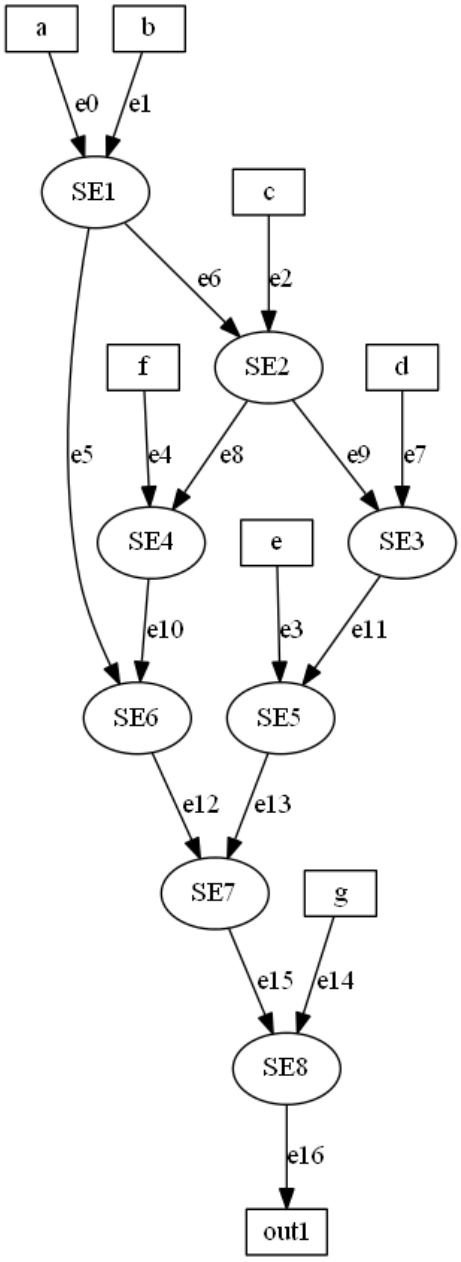

Figure 5.4. An Example Graph with Edge and Vertex Names.

```
---------- CDFG vs VHDL --------------
--- node a = d_0
--- node b = d_1
--- node SE1 = RCAAdder_001_1_i1_8_i2_8
--- node c = d_2
--- node SE2 = RCAAdder_001_1_i1_9_i2_9
--- node d = d_3
--- node e = d_4
--- node SE4 = Multiplier_002_1_i1_8_i2_10
--- node SE3 = RCAAdder_001_1_i1_10_i2_10
--- node f = d_5
--- node SE6 = Multiplier_002_1_i1_9_i2_18
--- node SE5 = RCAAdder_001_1_i1_11_i2_11
--- node SE7 = RCAAdder_001_1_i1_27_i2_27
--- node SE8 = RCAAdder_001_1_i1_28_i2_28
--- node out1 = out_d_0
--- node g = d_6
--- edge e0 = d_0_r0
--- edge e1 = d_1_r0
--- edge e5 = RCAAdder0_d0
--- edge e6 = RCAAdder0_d0
--- edge e2 = d_2_r0_e
--- edge e8 = RCAAdder1_d0
--- edge e9 = RCAAdder1_d0
--- edge e7 = d_3_r0_e
--- edge e3 = d_4_r0_e
--- edge e10 = SignedMultiplier0_d0
--- edge e11 = RCAAdder2_d0
--- edge e4 = d_5_r0
--- edge e12 = SignedMultiplier1_d0
--- edge e13 = RCAAdder3_d0_e
--- edge e15 = RCAAdder4_d0
--- edge e16 = RCAAdder5_d0
--- edge e14 = d_6_r0_e
```

Figure 5.5. Comment Part of the Generated Golden RTL.

nent VHDL files are generated by using the parameter and template files (explained in Section 3.5) of that arithmetic operators.

5.1.4. Top Level File

The generated component VHDL files are connected at the top level VHDL file. Input and output registers are optional. The user may choose to generate the golden RTL with and/or without input output registers. Components are connected with port mapping at the top level.

Signal Extension might be necessary during port mapping of the adders and subtractors. For example, in the example graph in Figure 5.2, SE2 is a 9 bit adder but one of its input (d_2) is 8 bits. So it shall be extended during port mapping. Extension changes according to the graph operation is signed or unsigned. If the operation is signed, it is extended as

d_2_r0_e <= "0" & d_2_r0 ;

Here, zeros are placed to the left of the signal as the MSBs. and if the operation most significant bit is extended as below;

d_2_r0_e <= conv_std_logic_vector(-conv_integer(d_7_r0(7)),1)&d_7_r0;

5.1.5. User Constraint File (UCF) Generation

Total Estimated delay is put as a constraint at the clock signal (when inputs/outputs are registered). The total estimated delay of the graph is calculated by firstly calculating the delay costs of the operators in the application graph and then summing the delay costs for the vertices in the critical path as explained in Section 4. The delay found here is put as a clock period in the UCF file as shown below;

NET "clk" PERIOD = 15.38255804 ns;

5.2. Golden RTL Generation Software

Golden RTL generation software can be divided into two parts as below;

- Golden RTL generation which creates the VHDL files of the components and connect them at the top level
- UCF file generation which estimates the total delay value of the application and put that value as clock period constraint by generating the UCF file.

In the software, for graph processing, Quickgraph library for .NET is used [45]. Datastructure of the CDFG is held by the bidirectional graph type of Quickgraph library in which edges are directed. Vertex structure of the circuit to be synthesized holds the arithmetic operator and variable information and edge structure holds the signal information.

5.2.1. Golden RTL Generation Function

GenerateGoldenRTL function creates the VHDL file of each arithmetic unit as a component and connects all of them at the top level. The architecture of this function is shown in the Figure 5.6. GenerateGoldenRTL function calls two main functions inside.

GenerateHDLComponentFromPrm Function : This function is used to generate necessary VHDL files of the arithmetic operators of the Golden RTL. For every operator in the graph, this function fills the parameter file of the required VHDL operator and then calls the ParseAndReplace function which generates the VHDL file.

ParseAndReplace Function : In the source folder, each of arithmetic operator has parameter(prm) file and template file. Template file is a generic VHDL file of the component which has fixed places and editable places. Every template file has a parameter file which consists of the editable fields in the in the VHDL template and their corresponding values. Parse and Replace function reads the parameter values in the parameter file and fill template file with these values to create VHDL component. With this function, arithmetic operators with desired wordlengths are generated.

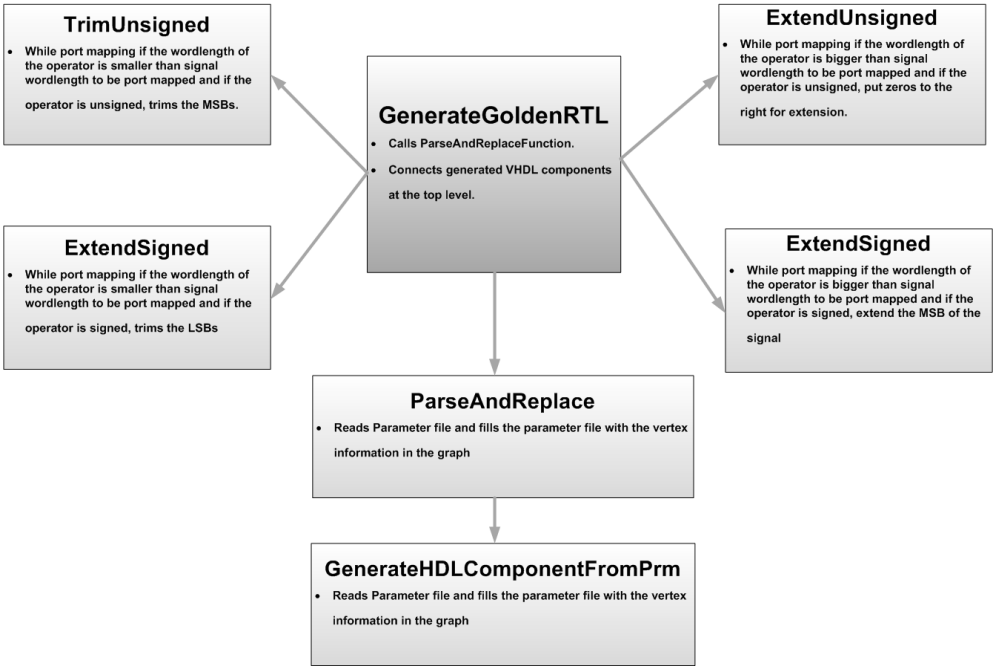

Figure 5.6. Structure of GenerateGoldenRTL Function.

With the help of GenerateHDLComponentFromPrm function and ParseAnd Replace function component VHDL files are generated. These components are then connected at the top level VHDL file. At the beginning of the top level, commenting is placed that shows which component in the VHDL corresponds to which vertex in the graph and which signal in VHDL corresponds to which edge in the graph. Entity declaration is done by using the variable vertices in the graph and component declarations are done by using the operator vertices in the graph. Edge information is used for signal declarations and port mappings are done by using the vertex and edge information. During port mapping, signals may need to be extended or trimmed before it is port mapped. There are 4 functions that is used for extending or trimming signals; TrimUnsigned, TrimSigned, ExtendUnsigned and ExtendSigned. TrimUnsigned and ExtendUnsigned are used for unsigned extension or truncation of the signal when necessary. ExtendSigned and TrimSigned is used for signed extension and truncation of the signal. As an example, extension is used for an adder with one port size of 8

bits and the other port size of 6 bits. This adder is generated as an 8 bit adder and the signal which is 6 bits length needs to be extended before port mapping. ExtendUnsigned extends the signal by putting zeros in front of the signal. ExtendSigned extends the signal by extending the most significant bit of the signal.TrimUnsigned, trims the most significant bits of the signal. TrimSigned, trims the least significant bits of the signal. The samples of extended and trimmed signals are shown below;

- **ExtendUnsigned :** d_2_e <= "00" & d_2;, here the signal is extended one bit by placing zeros as the MSBs of the signal.
- **ExtendSigned :** d_2_e <= conv_std_logic_vector(-conv_integer(d_2(7)),1)&d_2;, d_2 signal is 8 bits before extension, -conv_integer(d_2(7)),1) converts the MSB(7th bit) of the d_2 signal to one bit signal and by concatenating this signal with d_2 (-conv_integer(d_2(7)),1)&d_2), the MSB of the signal is extended.
- **TrimUnsigned :** d_2_t <= d_2(8 downto 0);, here 10 bits d_2 signal is trimmed to 9 bits by taking the LSBs of the d_2 signal.
- **TrimSigned :** d_2_t <= d_2(9 downto 1);, here 10 bits d_2 signal is trimmed to 9 bits by taking the MSBs of the d_2 signal.

The input and output registers of the Golden RTL is optional and these registers are generated if desired. There are 4 options for input and registers which are;

- neither inputs nor the outputs will be registered.
- Only inputs are registered.
- Only outputs are registered
- Both the inputs and outputs are registered.

These options are chosen by the integer input variable "GenType" of the GenerateGoldenRTL function. If this variable is zero, the Golden RTL is generated without input and output registers, if it is one, Golden RTL is generated registers with only at the inputs, if it is two, Golden RTL is generated with registers with only at the output, if it is three, Golden RTL is generated with both input and output registers.

5.3. User Constraint File(UCF) Generation Software

This function generates the UCF file to put a constraint on the clock. The UCF generation software diagram is shown in the Figure 5.7. During UCF file generation, firstly the function of GenerateFunctionFromHbd is called to calculate the delay of the operators in the graph. As explained in Section 3.4, every arithmetic operator have an HBD file which describes the delay, area and power models of the operators. These HBD files are read, parsed and then delay costs of the vertices are calculated and written into the vertex datastructure. These costs are given as edge costs and total delay of the graph is estimated by finding the longest path and summing the delay costs of the vertices in the longest path as described in Section 4.2. Longest path is found by using the directed acyclic shortest path algorithm [44] with negating these edge costs. Quickgraph library for .NET [45] is used for implementing longest path algorithm. For each variable operator, longest path algorithm is applied by selecting this variable vertex as root. The distances from the root vertex are listened with distance observer and the biggest distance corresponds to longest path from that root vertex. For all the input variable vertices, longest path is found and stored by selecting that vertex as root. Total estimated delay of the graph is the biggest value between these stored longest paths. This value is given as clock period as constraint in UCF file when it is generated. Since, constraint is given to clock, this UCF file is useful when the golden RTL is generated when both inputs and outputs are registered.

5.4. Test and Estimation Results of the Golden RTL

Delay and area estimation methodology is tested by comparing the results of estimation tool proposed in this thesis with the delay and area which are estimated by the Xilinx ISE tool. Five different benchmarks, namely Elliptic wave filter (EW), AutoRegression (AR) filter, Discrete Cosine Transform (DCT), infinite impulse response filter (IIR) and differential equation (DE) applications with various input wordlengths are given as input to the estimation tool. These graphs are shown in the Figures 5.8 to 5.12, respectively. The Golden RTL VHDL files are also generated for these applications with Golden RTL generation tool and these applications are synthesized with

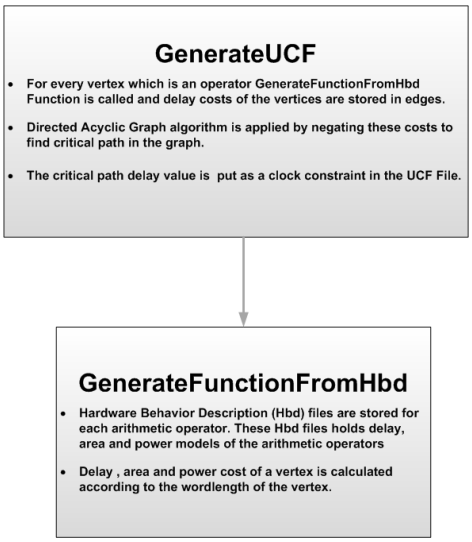

Figure 5.7. Structure of GenerateUCF Function.

Xilinx ISE 12.4 for the Spartan-3 FPGAs. During synthesis, the estimated delay of the graph is given as a clock period constraint. For measuring delay, the application is synthesized with the option of 'optimization type speed without IOB packing' and for measuring area the application is synthesized with optimization type of Area 'High' in Xilinx ISE. The measured values which is gathered from PAR report are used as reference for measuring the accuracy of our proposed estimation model. The design area is measured in slices and the delay is measured in nanoseconds. The estimated delay and area and the comparision with the Xilinx Estimator for these benchmarks are given in the following text.

The average error for these benchmarks are found 6% in delay estimation and 3.8% in area estimation. Benchmark estimations are based on aggregating standalone behaviors of individual nodes (operators like addition, multiplier etc.) which means routing between nodes are not considered and estimated accordingly. However, internal routing of each node is estimated. There is a relation between the delay error and the graph depth. If the graph depth is low, the interconnection cost between the operators becomes dominant and the estimation becomes underestimated. During delay estimation, the estimation tool assumes that one operator does not start its operation before the preceding operation finishes its operation, however in FPGAs, the succeed-

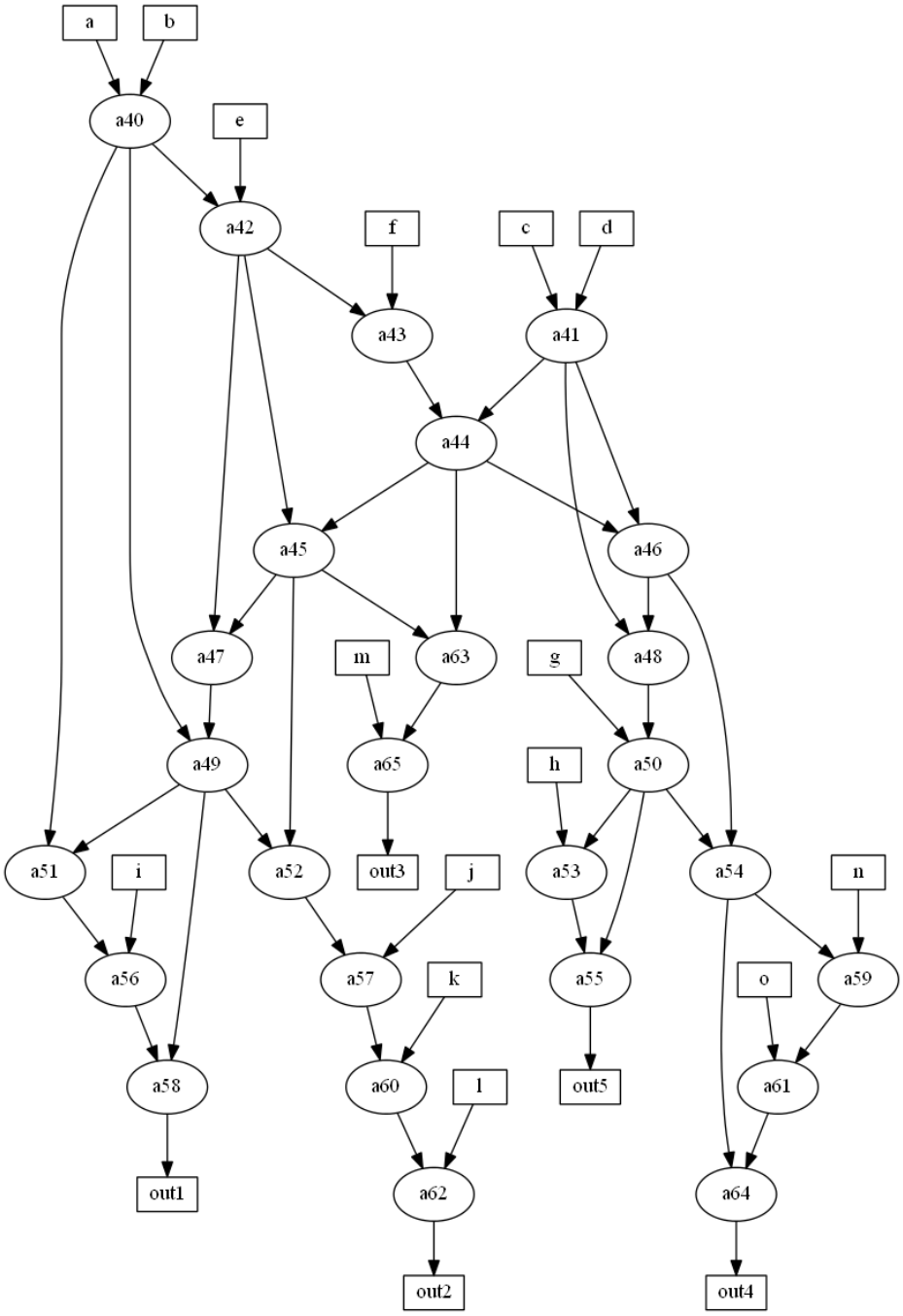

Figure 5.8. EW Filter Graph Used for Golden RTL Generation.

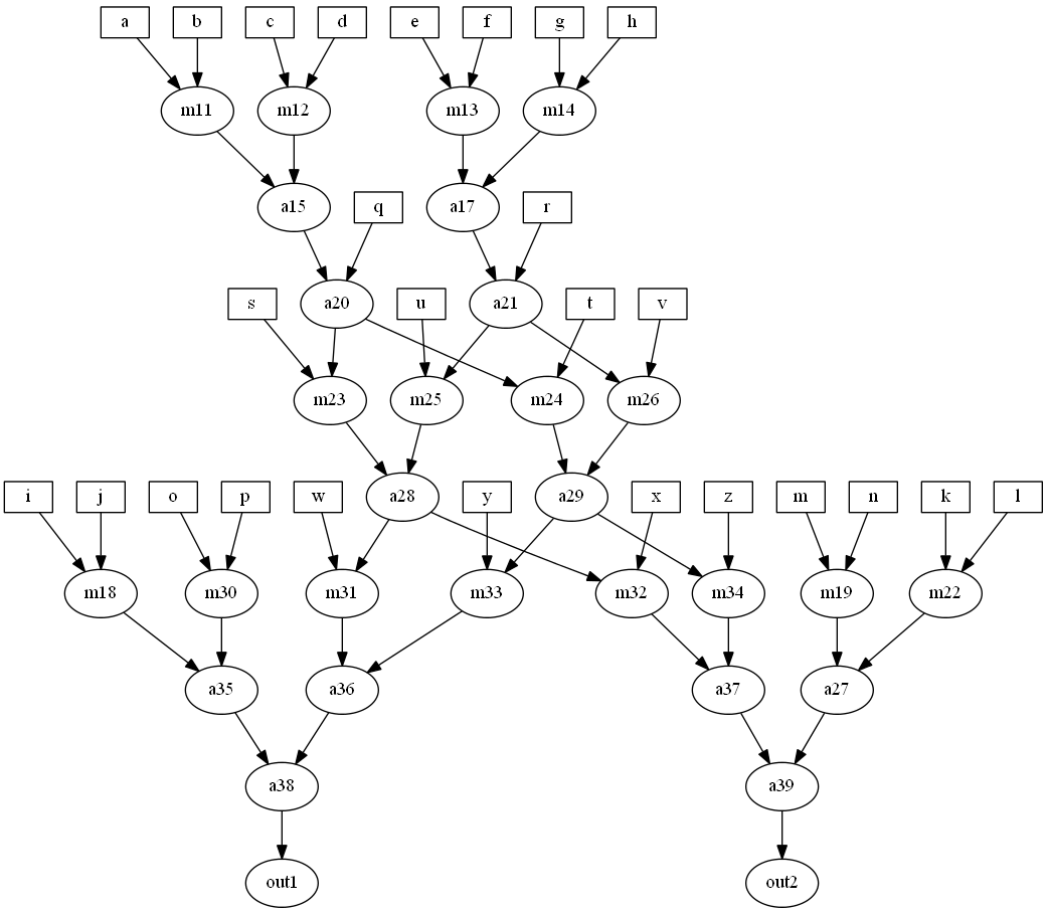

Figure 5.9. AR Filter Graph Used for Golden RTL Generation.

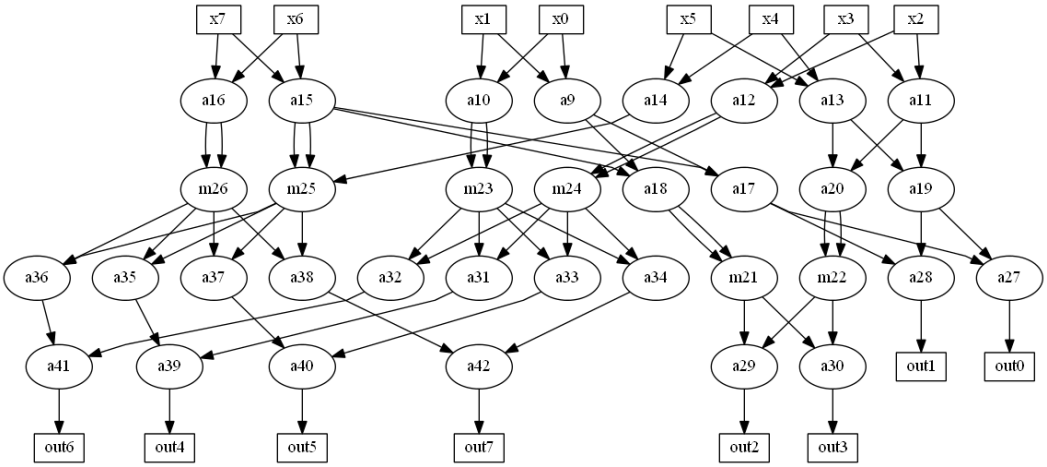

Figure 5.10. DCT Graph Used for Golden RTL Generation.

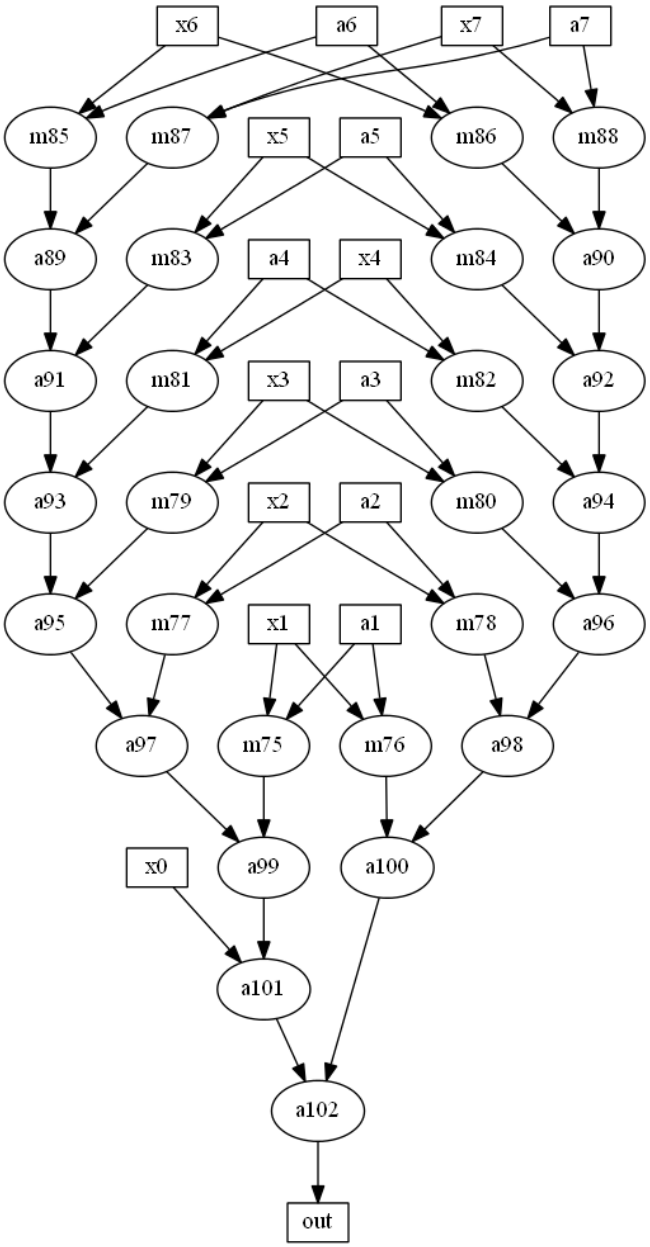

Figure 5.11. IIR Graph Used for Golden RTL Generation.

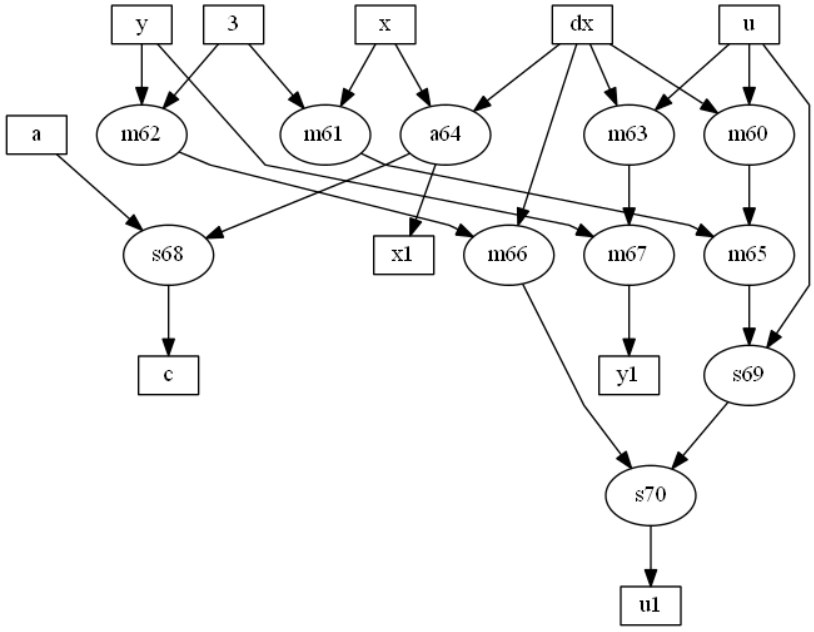

Figure 5.12. Differential Equation Graph Used for Golden RTL Generation.

ing operator starts its operation when necessary bits are ready from the preceding operator. If the graph depth is high, this assumption compensates the interconnection cost which may cause an overestimated delay by the estimation tool. There is a relation between the chip occupation which is generally as the chip occupation ratio increases the area estimation error decreases.

5.4.1. Differential Equation

The delay estimation results and estimation error is given in Tables 5.1 and 5.2. Moreover, the consumed area of the design for Spartan 3S 5000 FPGA is given in Table 5.2. Figure 5.13 shows the delay estimation and area estimation error, Figure 5.14 shows the chip occupation of the design versus area estimation error for differential equation.

Table 5.1. Differential Equation Delay Performance Estimation Results.

Input Wordlength	**Estimated Latency through Delay Model(ns)**	**Delay Result in PAR Report(ns)**	**Delay Estimation Error (%)**
4	17.86	17.819	0.230
8	24.49	24.588	0.398
12	31.97	31.78	0.623
16	37.157	37.032	0.337
24	52.71	52.567	0.272
32	56.86	56.846	0.024

Table 5.2. Differential Equation Area Estimation Results and Chip Occupation in FPGA.

Input Wordlength	**Estimated Area through Area Model(n of slices)**	**Area Result in PAR Report (n of slices)**	**Area Estimation Error (%)**	**Chip Occupation (%)**
4	112.69	124	9.120	0.372
8	432.46	454	4.744	1.364
12	950.68	977	2.693	2.935
16	1667	1707	2.343	5.129
24	3695.91	3735	1.046	11.22
32	6518.24	6556	0.575	19.69

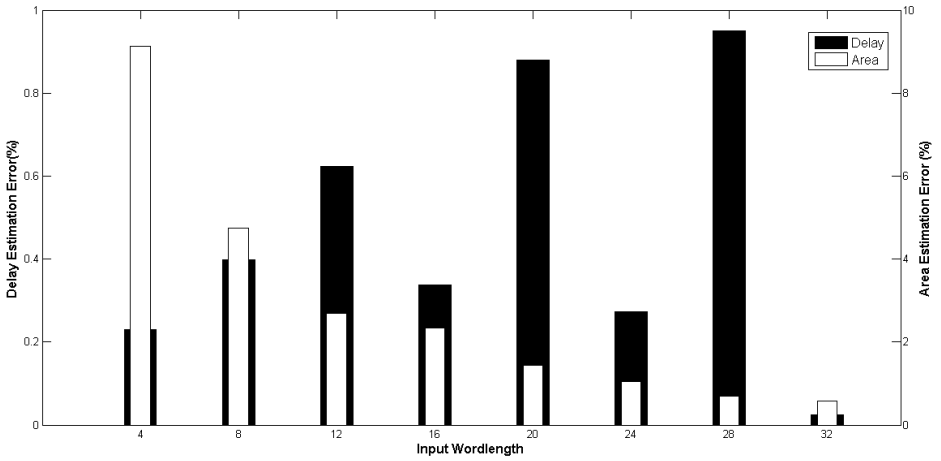

Figure 5.13. Differential Equation Delay and Area Estimation Error.

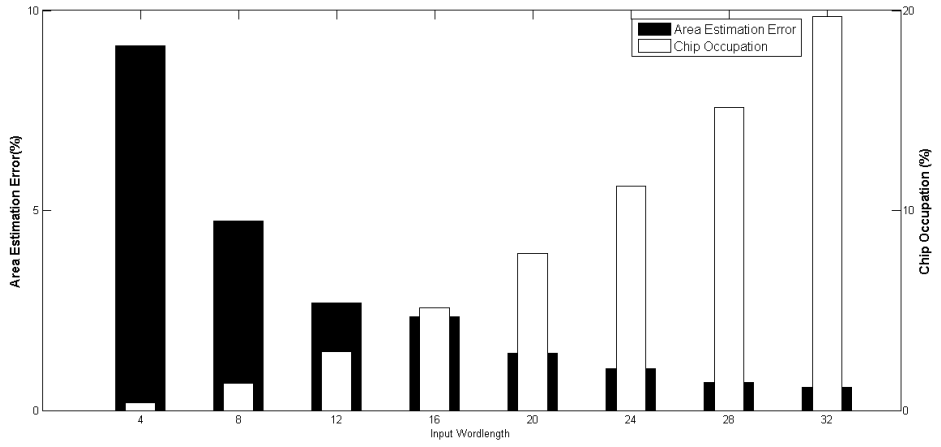

Figure 5.14. Differential Equation Chip Occupation vs Area Estimation Error.

Table 5.3. IIR Delay Performance Estimation Results.

Input Wordlength	**Estimated Latency through Delay Model(ns)**	**Delay Result in PAR Report(ns)**	**Delay Estimation Error (%)**
4	23.19	23.103	0.376
8	28.12	28.018	0.364
12	33.29	33.281	0.027
16	38.77	38.1	1.758
24	50.04	46.979	6.515
32	58.67	52.353	12.066
36	68.65	52.729	18.917
44	76.45	63.826	19.778

Table 5.4. IIR Area Estimation Results and Chip Occupation in FPGA.

Input Wordlength	**Estimated Area through Area Model(n of slices)**	**Area Result in PAR Report (n of slices)**	**Area Estimation Error (%)**	**Chip Occupation (%)**
4	176.9	207	14.541	0.621
8	584.16	641	8.876	1.926
12	1224.28	1299	5.752	3.903
16	2097.27	2209	5.057	6.637
24	4541.85	4687	3.096	14.083
32	7917.89	8103	2.284	24.347
36	9955.21	10133	1.754	30.447
44	14728.16	14685	0.917	44.666

5.4.2. Infinite Impulse Response Filter (IIR)

The delay estimation results and estimation error of IIR is given in Tables 5.3 and 5.4. Moreover, the consumed area of the design for Spartan 3S 5000 FPGA is given in Table 5.4. Figure 5.15 shows the delay estimation and area estimation error, Figure 5.16 shows the chip occupation of the design versus area estimation error.

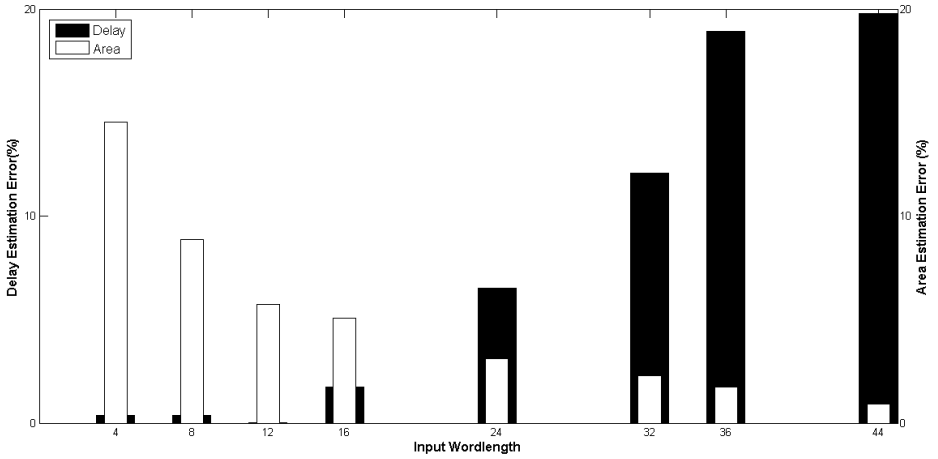

Figure 5.15. IIR Delay and Area Estimation Error.

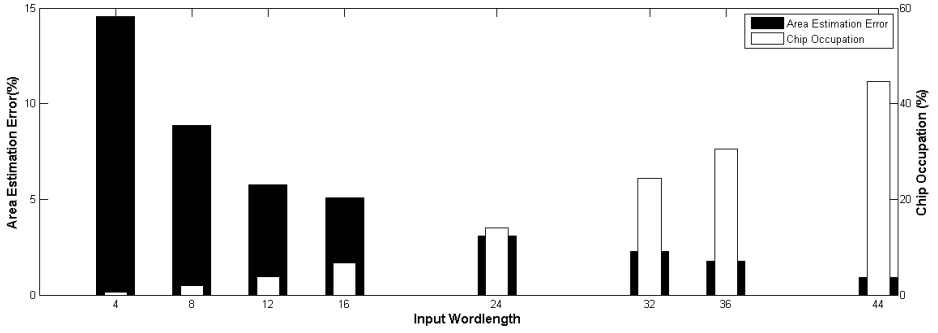

Figure 5.16. IIR Chip Occupation vs Area Estimation Error.

Table 5.5. EW Filter Delay Performance Estimation Results.

Input Wordlength	**Estimated Latency through Delay Model(ns)**	**Delay Result in PAR Report(ns)**	**Delay Estimation Error (%)**
4	22.38	25.328	11.639
8	24.84	25.367	2.077
12	27.3	27.093	0.764
16	29.705	29.448	0.872
24	34.67	33.895	2.286
32	39.59	37.098	6.717
36	42.05	41.383	1.611

Table 5.6. EW Filter Area Estimation Results and Chip Occupation in FPGA.

Input Wordlength	**Estimated Area through Area Model(n of slices)**	**Area Result in PAR Report (n of slices)**	**Area Estimation Error (%)**	**Chip Occupation (%)**
4	130.32	139	6.244	0.41
8	182.28	191	4.565	0.57
12	234.24	243	3.604	0.73
16	286.20	295	2.982	0.88
24	390.11	399	2.228	1.19
32	494.03	503	1.783	1.51
36	545.99	555	1.622	1.67

5.4.3. Elliptic Wave Filter (EW)

The delay estimation results and estimation error of EW filter is given in Tables 5.5 and 5.6. Moreover, the consumed area of the design for Spartan 3S 5000 FPGA is given in Table 5.6. Figure 5.17 shows the delay estimation and area estimation error, Figure 5.18 shows the chip occupation of the design versus area estimation error.

5.4.4. Auto Regression Filter (AR)

The delay estimation results and estimation error of AR filter is given in Tables 5.7 and 5.8. Moreover, the consumed area of the design for Spartan 3S 5000 FPGA is

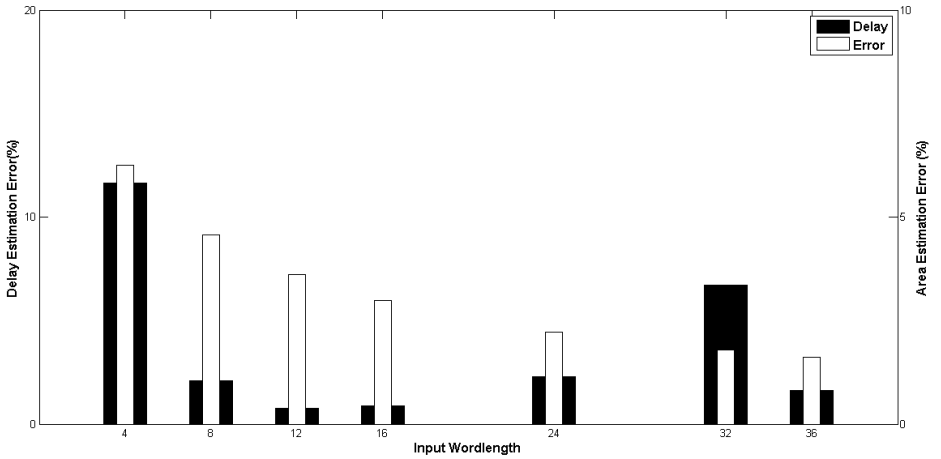

Figure 5.17. EW Filter Delay and Area Estimation Error.

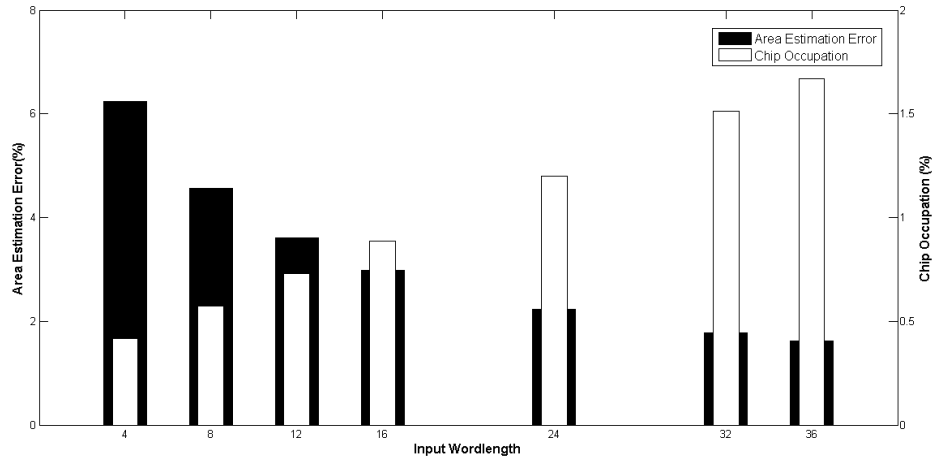

Figure 5.18. EW Filter Chip Occupation vs Area Estimation Error.

Table 5.7. AR Filter Delay Performance Estimation Results.

Input Wordlength	**Estimated Latency through Delay Model(ns)**	**Delay Result in PAR Report(ns)**	**Delay Estimation Error (%)**
8	39.03	39.17	0.357
12	49.22	49.176	0.089
16	59.94	58.439	2.568
20	70.30	70.255	0.074

Table 5.8. AR Filter Area Estimation Results and Chip Occupation in FPGA.

Input Wordlength	**Estimated Area through Area Model(n of slices)**	**Area Result in PAR Report (n of slices)**	**Area Estimation Error (%)**	**Chip Occupation (%)**
8	1126	1146	1.745	3.44
12	2377	2406	1.205	7.22
16	4104	4146	1.013	12.45
20	6244	6574	3.55	19.75

given in Table 5.8. Figure 5.19 shows the delay estimation and area estimation error, Figure 5.20 shows the chip occupation of the design versus area estimation error.

5.4.5. Discrete Cosine Transform (DCT)

The delay estimation results and estimation error of AR filter is given in Tables 5.9 and 5.10. Moreover, the consumed area of the design for Spartan 3S 5000 FPGA is given in Table 5.10. Figure 5.21 shows the delay estimation and area estimation error, Figure 5.22 shows the chip occupation of the design versus area estimation error.

5.4.6. Software Run Time Results

The software Run time for the estimation tool proposed in this thesis and Xilinx ISE software is compared. Table 5.11 shows the run time comparisons of proposed estimation tool and Golden RTL generation tool in this thesis and Xilinx ISE tool.

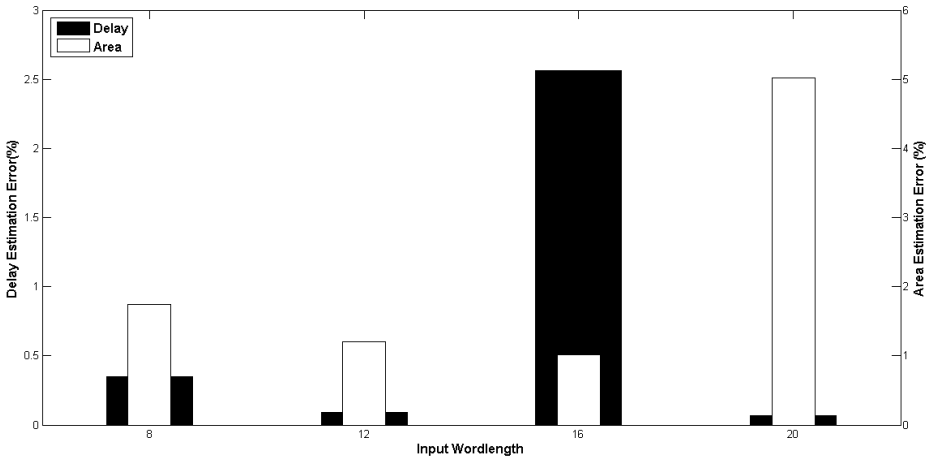

Figure 5.19. AR Filter Delay and Area Estimation Error.

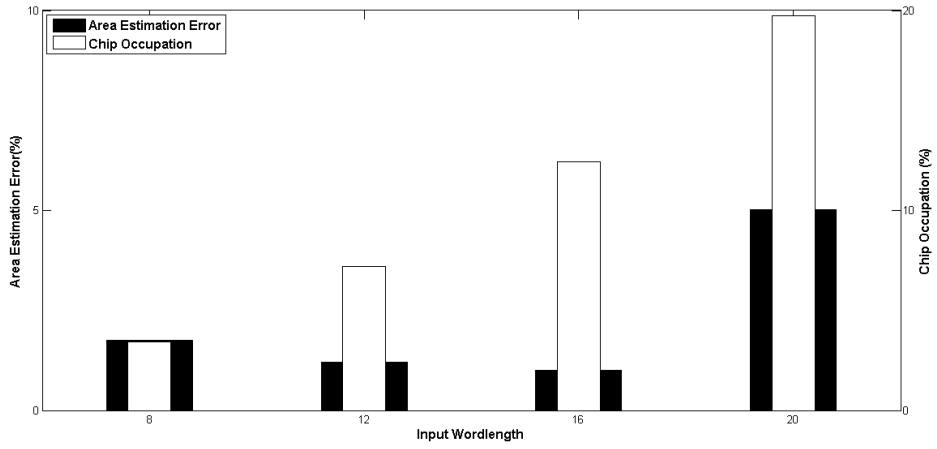

Figure 5.20. AR Filter Chip Occupation vs Area Estimation Error.

Table 5.9. DCT Delay Performance Estimation Results.

Input Wordlength	**Estimated Latency through Delay Model(ns)**	**Delay Result in PAR Report(ns)**	**Delay Estimation Error (%)**
8	14.634	23.406	37.4
12	17.415	24.497	28.9
16	20.505	28.99	29.2
24	26.77	26.795	0.01
32	36.98	36.457	1.43
36	37.51	37.379	0.35

Table 5.10. DCT Area Estimation Results and Chip Occupation in FPGA.

Input Wordlength	**Estimated Area through Area Model(n of slices)**	**Area Result in PAR Report (n of slices)**	**Area Estimation Error (%)**	**Chip Occupation (%)**
8	457.301	505	9.44	1.51
12	824.85	893	7.63	2.68
16	1292.2	1379	6.29	4.14
24	2526.3	2737	7.69	8.22
32	4159.6	4339	4.13	13.03
36	5125	5415	5.35	16.02

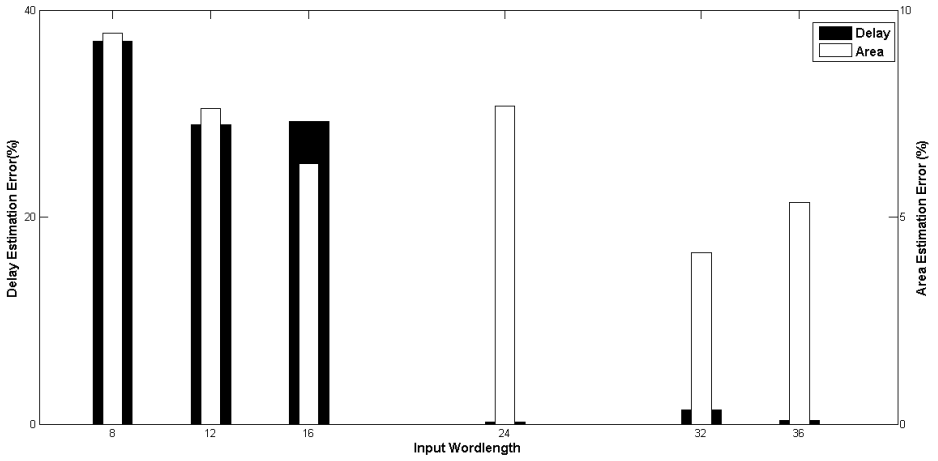

Figure 5.21. DCT Delay and Area Estimation Error.

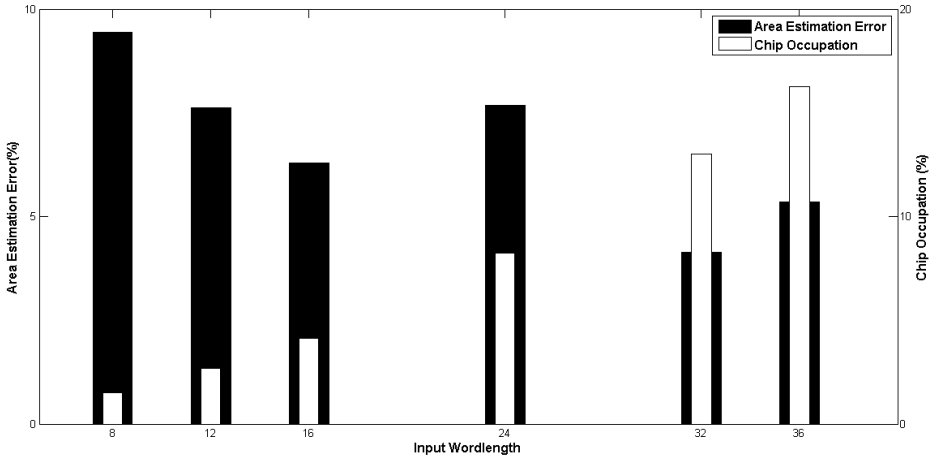

Figure 5.22. DCT Chip Occupation vs Area Estimation Error.

Table 5.11. CPU Run Time Comparison.

Application	**Input Wordlength**	**Estimation Run Time(msec)**	**Golden RTL generation Run Time(msec)**	**Xilinx Run Time (sec)**
DE	16-bit	18.72	235.56	174.62
IIR	16-bit	24.96	616.23	207.21
AR	12-bit	18.08	580.32	416.41
EW	16-bit	23.40	756.6	114.88
DCT	16-bit	20.28	822.12	107.87

Estimation Run Time(msec) corresponds to software CPU run time for processing graph, calculating the area and delay costs for every vertex in the graph using hbd files and then estimate the area and delay of the whole graph. Golden RTL generation Time(msec) corresponds to software CPU run time for generating Golden RTL VHDL files of the graph. Xilinx Run Time(sec) corresponds to CPU run time of synthesis, mapping, placing and routing of the graph in Xilinx ISE tool. As Xilinx makes estimations by passing through these steps, run time of the estimation tool implemented with this thesis is much faster while the estimation error is acceptable. All benchmarks run on Intel Core i5-480 2.67 Ghz 3GB RAM running Windows 7 64-bit OS.

6. RESCHEDULING

6.1. Introduction

As shown in the Figure 1.1, after Node Analysis module, the tool flow is splitted into two paths;

- First path includes the Performance and Area Estimation module which is explained in Chapter 4 and the Golden RTL Generation module which is explained in Chapter 5. In this path, Golden RTL VHDL files of the application design is generated and the total delay and consumed area of the design is estimated.
- In the second path, application graph firstly passed through the Optimization and Allocation & Schedule of CDFG modules which optimize the application graph, make resource allocation and scheduling according to the optimization results. Secondly, the graph is rescheduled in the Rescheduling module to find the best clock period for the design and to reschedule the application graph according to this clock period. In the Datapath Generation module, the VHDL files of the optimized and rescheduled graph is generated for Xilinx FPGAs.

The implementation of the Optimization and Allocation & Schedule of CDFG modules are out of the scope of this thesis. However, some inputs are given from the blocks implemented in this thesis and outputs of these blocks are used. These blocks are explained briefly in the following subsections as their outputs are used in the Rescheduling module.

6.1.1. Optimization and Allocation & Schedule of CDFG

In Optimization and Allocation & Schedule of the CDFG blocks, a mathematical model is used to solve the multiplexer and resource scheduling binding problem [3]. In comparison with traditional models, it solves the multiplexer allocation and binding problem as well as the resource scheduling and binding problem in an integrated

Name	ComponentID	Instance	StartTime	Wordlength
SE1	00001	2	0.0	9.00
SE2	00001	3	1.2	10.00
SE3	02002	4	3.4	18.00
SE4	02002	1	3.4	18.00
SE5	00001	1	6.2	19.00
SE6	02002	1	7.1	27.00
SE7	00001	1	8.8	28.00
SE8	00001	5	9.9	55.00

Figure 6.1. An example output file of the Allocation & Scheduling of the CDFG block.

fashion. In traditional mathematical models, overhead of sharing components (multiplexers) are not taken into account however, in FPGAs, area, power and latency costs of a multiplexer are comparable to that of a functional resource which may cause area and power overhead up to 70% in FPGAs.

In this mathematical model, the real delay and area values of operators and multiplexers which are obtained in Node Analysis module (see Figure 1.1) are used as inputs. The multiplexer and resource allocation binding problem is solved with minimum latency under minimum area constraint. The model is firstly solved with the objective of minimum area while keeping the latency constraint maximum. After the minimum area is found, the model is solved with the objective of minimum latency while the area found in the first step is given as area constraint. If the optimal solution is not found in five hours time, suboptimal solution is the output of the optimization tool.

After the model is solved with minimum latency under minimum area constraint, the operators which use the same functional hardware units, selected hardware units, the shared operators, and minimum starting times of the each operator in the CDFG are determined in Allocation & Scheduling module. This information is used in Rescheduling block as input. An example output file of the Allocation & Scheduling of the CDFG block is shown in the Figure 6.1.

In Figure 6.1, the first column corresponds to the name of the vertex, second column corresponds to the component ID of the selected operator, third column corresponds to instance of the operator, the fourth column corresponds to minimum start time of the operation and fifth column corresponds to output wordlength. If the two vertices has the same component ID and same instance that means these vertices share the same functional hardware unit. The component ID of '00001' corresponds to RCA adder and '02002' corresponds to multiplier, so SE4 and SE6 use the same multiplier and SE5 and SE7 use the same adder for this example.

6.2. Rescheduling Methodology

In the Optimization and Allocation & Scheduling of the CDFG modules, the graph is optimized and operator sharing information and starting times of the vertices are found. The output of these modules are given as input to the Rescheduling module. The starting times found in these modules are the minimum start times for all operations so the minimum latency of the graph is found. However, this value is unattainable if there are shared resources. Even though the delay due to multiplexers is taken into consideration, the switching between inputs of the multiplexer requires a clock and the delay of one multiplexer + operator pair might not be equal to another multiplexer + operator pair. Hence, a common clock period has to be extracted so that the delay of every multiplexer + operator pair is a multiple of that clock period. The clock period can be made very small but this increases the number of control steps. If there is no sharing, then the clock period can be as big as the minimum latency found by the Optimization module. So, there are a lot of ways to select the clock period. In this thesis, we decided to select the clock period in a way to reduce the control steps while keeping the overall latency as close to the found minimum latency as possible. Then, the CDFG is rescheduled according to the selected clock period. Clock period extraction and rescheduling of the CDFG are realized in Rescheduling module.

6.2.1. Rescheduling Algorithm

The diagram for the rescheduling algorithm is shown in the Figure 6.12. Firstly, the output file of the Optimization and Allocation & Scheduling block is parsed. As described in Section 6.1.1, this file holds the sharing information between the vertices and the start time of the operators. With parsing this file, sharing information and start time of the operators are recorded in the vertex datastructure of the graph. In the AddSharedComponentInformation method, the latencies of the vertices which share the same hardware unit are updated. The latencies of the shared vertices are changed to the sum of maximum latency between the shared operators and the multiplexer delay. In the DetermineClockValues method, the graph is traversed and the delay of the each vertex is stored in an array to use it for rescheduling clock period value. In the AddEdgeBetweenSharedOperators method, edges are placed between the shared operators. The edges are placed from the vertex which has a lower start time to the vertex which has a higher start time. Then, for each clock period value found in the DetermineClockValues method, FindChainableOperators and RescheduleGraph methods are applied. In the FindChainableOperators method, the operators which can be chained in one clock cycle are found for that clock period. An important note to add is that, the operators cannot be chained and cascaded in the same clock period if one of the operators is shared. In the RescheduleGraph method, the graph is rescheduled with using chainable operator information. In RescheduleGraph method , the register delay is added to the rescheduling clock period and also register delays are also added to the delay cost of vertices in the graph. The register delay value is read from the technology parameter file which holds the delay of the FPGA. The technology file of Spartan-3 FPGA is given in Figure 6.11. The register costs are added because the delay models explained in Section 3.3 does not include register costs however in optimized RTL datapath registers are placed between operations. The total delay of the graph for that clock period is found by multiplying the clock period with the number of control steps. At the end, the clock period which gives the lowest total delay is chosen as the clock period for that graph and the graph is rescheduled for that value before datapath generation. The detailed description of the algorithm is given

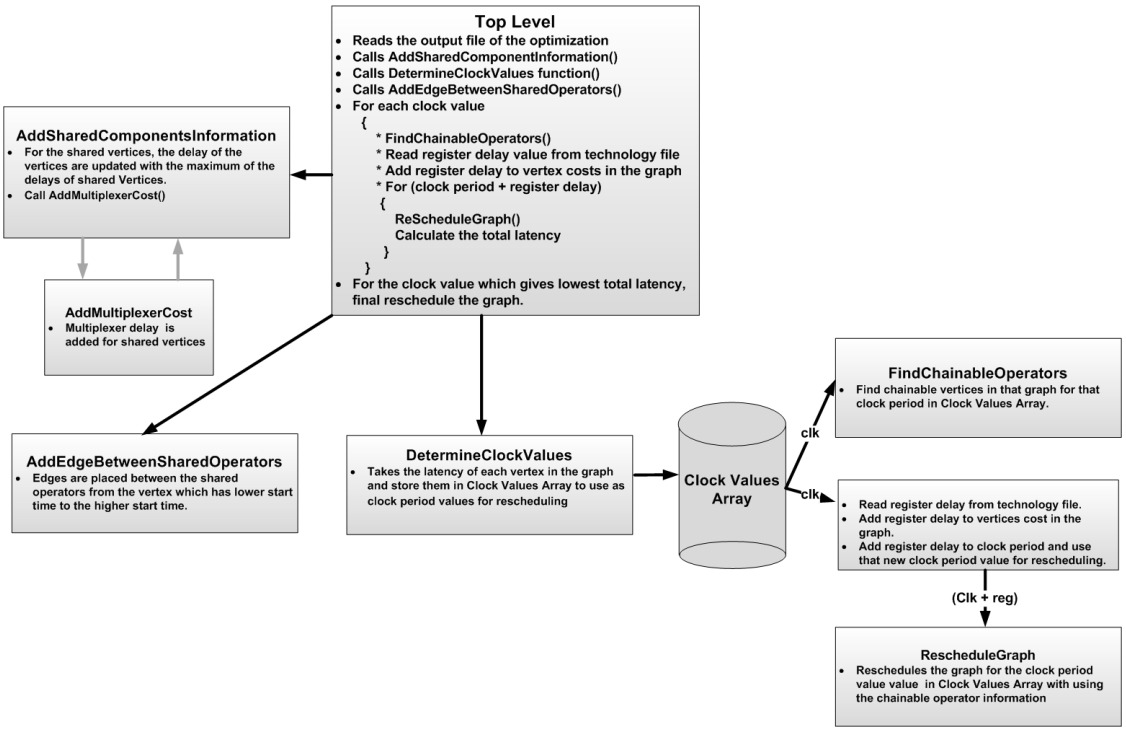

Figure 6.2. Block Diagram of the Rescheduling Algorithm.

in the following subsections.

6.2.1.1. Top Level G(V,E). The pseudocode of the Top level algorithm is shown in the Figure 6.3. The application graph is the input of the software. The output file of the Optimization and Allocation & Scheduling block is parsed and vertex datastructure is filled according to that file. Then, AddSharedComponentsInformation function, DetermineClockValues function, AddEdgeBetweenSharedOperators function are called respectively. With DetermineClockValues function the delay values of the vertices in the CDFG are gathered and these values are hold in ClockValuesArray to use as clock period value. For each clock period value in ClockValuesArray, FindChainableOperators function is called to find chainable operators. Chainable operator information is used during Rescheduling the graph. After, Technology parameter file is read and register delay is added to the clock period value and also the delay costs of the vertices in the graph. The technology file of Spartan-3 FPGA is given in Figure 6.11. The rescheduling clock period value becomes the clock period + register delay. Reschedule-

Graph function is called to reschedule the graph with that clock period. For each clock period value, the total latency of the graph is calculated as clock period multiplied by the number of control steps. The clock value which gives the minimum latency is selected as the clock period value.

6.2.1.2. AddSharedComponentsInformation (G(V,E))). AddSharedComponentsInformation algorithm updates the latency costs of the vertices by changing the latency cost of the vertices which share the same hardware resource to sum of maximum latency of the vertex between shared vertices and the multiplexer delay cost. The pseudocode of the AddSharedComponentsInformation is shown in the Figure 6.4.

6.2.1.3. DetermineClockValues (G(V,E)). Determine Clock Values algorithm traverses the graph and takes the latencies of each vertex in the graph and store these values in ClockValuesArray. The values in this array is later used as clock period value during rescheduling. The pseudocode of the DetermineClockValues algorithm is shown in the Figure 6.6.

6.2.1.4. AddEdgeBetweenSharedOperators(G(V,E)). Add Edge Between Shared Operators algorithm add edges between the operators which shares the same hardware resource. These edges are added from the vertices which have lower start times to the vertices which have higher start times. The pseudocode of the AddEdgeBetweenSharedOperators algorithm is shown in the Figure 6.7. The algorithm firstly traverses the graph and finds the shared vertices. For each shared vertex, a Temporary Array is defined that holds the vertices and their starting times. After the graph is traversed again and the vertices that shares the same resource with that vertex is added to Temporary Array and this array is sorted according to starting times of the vertices in ascending way. Then, edges are defined and added between the vertices in Temporary Array.

1: G (V, E) ▷ *Directed Acyclic Graph which is composed of Vertices and Edges is the input, Application is the input*
2: ▷ *Initialization*
3: ▷ *Read from a file and fill the below fields in Vertex datastructure*
4: Latency[v] ▷ *Latency Costs of the vertices are filled*
5: Isshared[v] ▷ *Determines whether vertex appears on a shared resource or not.*
6: OtherSharingVertices[v] ▷ *holds the indices of the vertices that share the same resource.*
7: dSchedulingtime[v] ▷ *holds the starting times of the vertices before they are rescheduled.*
8: **AddSharedComponentsInformation(G(V,E))** ▷ *Recorrects the Latency Costs of the vertices*
9: **DetermineClockValues(G(V,E))**▷ *Assigns the clock Values which is used to Reschedule to an Array from Vertex Latencies*
10: **AddEdgeBetweenSharedOperators(G(V,E))**
11: **for each** element in ClockValuesArray **do**
12: **FindChainableOperators(G(V,E),clock period value)** ▷ *find chainable vertices for that clock, this returns ZeroEdgeArray which holds the edges that will have the cost of 0 during rescheduling*
13: **RescheduleGraph(G(V,E), clock period value, ZeroEdgeArray)** ▷ *Reschedule Graph and store the total latency*
14: **end for**
15: Choose the clock period value which gives the minimum total latency for that graph and Reschedule the graph with that clock value.

Figure 6.3. Top Level algorithm of the Rescheduling.

```
▷ Takes the Graph as input (passed by reference) and corrects the latencies of the Vertices. Recorrection for the Latency costs of the shared vertices by changing the Latency cost with the highest value between the shared vertices, and adding multiplexer cost.
for each vertex v in V do
  if Isshared[v] = true ▷ if the vertex is shared then
    temp ← Latency[v]
    for each vertex u in OtherSharingVertices[v] do
      ▷ find the highest Latency cost between the vertices sharing the same resource with v
      if Latency[u] >temp then
        temp ← Latency[u]
      end if
    end for
    Cost[v] ← temp
    AddMultiplexerCost(v) ▷ Change the cost of the vertices with the highest latency and adding multiplexer cost to it.
  end if
end for
```

Figure 6.4. AddSharedComponentsInformation Algorithm.

```
▷ Takes the vertex and Graph as input and returns the graph with the latency of that vertex is corrected.
▷ Multiplexer Latency Costs are added to the vertex latency if the vertex is shared. Multiplexer Latency depends on the port size of the Multiplexer, which corresponds to the size of OtherSharingVertices field in vertex datastructure.
for each vertex v in V do
  if Isshared[v] = true ▷ if the vertex is shared then
    PortSize ← OtherSharingVertices[v].Count() + 1 ▷ if the vertex is shared port size is the size of the OtherSharingVertices + 1
    GenerateFunctionfromhbd() ▷ Read the HBD file of multiplexer and calculate latency values
  end if
end for
```

Figure 6.5. AddMultiplexerCost Algorithm.

```
▷ Takes the graph as input and returns ClockValuesArray
▷ Defines Array of clock values and the latencies of the vertices in that array.
ClockValuesArray []; ▷ Define array to hold clock values
for each vertex v in V do
  if Cost[v]∉ ClockValuesArray then
    Add cost[v] to ClockValuesArray
  end if
end for
return ClockValuesArray[];
```

Figure 6.6. Determine Clock Values Algorithm.

6.2.1.5. FindChainableOperators(G(V,E), clockvalue) . Find Chainable Operators algorithm takes a clock period from ClockValuesArray and the graph as input. It finds the chainable operators in one clock cycle for that clock period value and it returns an edge array which consist of edges between chainable operators. Two or more operators can be chained and cascaded in the same clock period if only none of the operators are shared. The pseudocode of the FindChainableOperators algorithm is shown in the Figure 6.9. The algorithm is explained below;

(i) EdgeCostArray is defined to hold the costs of edges and ZeroEdgeArray is defined to store the edges which will have the cost of 0.

(ii) For each edge in the graph, the delay of the source vertex of that edge is given as edgecost.

(iii) An array of CombinedVertices is defined to hold vertices that can be combined in one clock period.

(iv) For each vertex in the graph, if the vertex is not shared and is not combined before, Temporary array is defined to store chainable vertices and CallCheck function is called. CallCheck function extracts the paths from the given vertex until a shared vertex is reached and marks these vertices as true. Then, DagLongestPath algorithm is applied while that vertex is given as root vertex and a listener is used to observe distances to other vertices in the graph from that root vertex. DagLongestPath algorithm is simply Dag Shortest Path algorithm [44]

```
▷ Graph is given as input and Graph with added edges returns back.
▷ Adding Edge between the shared Vertices by first sorting these vertices by their
starting time and then adding edges between the vertices that have lower starting times
to vertices that have higher starting times.
for each vertex v in V do
  Checked[v] ← false
end for
for each vertex v in V do
  if Checked[v] = false then
    if Isshared[v] = true  then
      ▷ if vertex is shared
      TempArray[] ▷ define temporary Array that hold vertices
      TempArray[v] ← dSchedulingtime[v] ▷ Starting times of the vertices are added
      to array
      for each vertex u in OtherSharingVertices[u]▷ for each shared vertices with the
      v do
        TempArray[u] ← dSchedulingtime[u] ▷ add starting times of these vertices to
        array
      end for
      TempArray.Sort()▷ Sort the array that holds the starting times of the shared
      vertices as ascending
      for j = 0 to Length[TempArray] do
        Addedge(TempArray[j], TempArray[j+1] ▷ Add edges from the vertex which
        has lower starting time to higher starting time
      end for
      for each vertex u in TempArray do
        Checked[u] ← true
      end for
    end if
  end if
end for
```

Figure 6.7. Add Edge Between Shared Operators Algorithm.

where edge costs are negated before applying Dag Shortest Path algorithm. For each vertex in the graph if the latency+distance is smaller than the clock period, this vertex can be chainable and it is added in the TempArray. After, for each edge in the graph, if the source vertex and target vertex of that edge exist in the TempArray, then the cost of this edge should be zero so it is placed in the ZeroEdgeArray. Moreover, the source and target vertices are placed in CombinedVerticesArray so as to prevent the chaining of these vertices again. The vertices are marked as false again for the other iterations.

6.2.1.6. RescheduleGraphG(V,E), ClockValue, ZeroEdgeArray. RescheduleGraph algorithm reschedules the graph for that clock value with using the chainability information. It takes the graph, Clock period value from ClockValues array and ZeroEdgeArray which is filled in FindChainableOperators function. The algorithm is explained below;

(i) ClockOccupationArray is defined to store how many clock cycles a vertex occupies. There may be multicycle operators which occupies more than 1 clock cycles. For example, if latency of an operator is 4 ns, and if the clock period is 2 ns, this operation is multicycle operation which occupies two clock cycles.

(ii) Cost array is defined to hold the cost of edges during DagLongestPath algorithm.

(iii) Technology parameter file is read and parsed to get register delay for the FPGA. The technology parameter file of Spartan-3 FPGA is shown in Figure 6.11. The register file is the sum of the input_register_delay and output_register_delay in the technology file. Technology file has the extension of “tech”. The rescheduling clock period is changed by adding the register delay to the clock period.

(iv) The register delay is added to the vertex delay cost of the each vertex in the graph.

(v) For each vertex in the graph, Clock occupation value is calculated and it is stored in ClockOccupationArray.

(vi) For each edge in the graph, if the edge exist in ZeroEdgeArray, ‘0’ is given as edgecost. If not, the clock occupation value of the source vertex is given as

```
ZeroEdgeArray[] ▷ Define Array for edges to hold which will have 0 cost
EdgeCostArray[] ▷ Define Array for holding the costs of edges
for each vertex v in V do
   Checked[v] ← false
end for
for each edge e in V do
   EdgeCostArray[e] ← Latency[V_t(e)] ▷ Edge costs are the latency costs of the source
   vertex of the edge (Source(e) = V_t and Target(e) = V_h , e = (V_t,V_h )
end for
CombinedVertices[]▷ Array for holding Vertices that can be combined in one clock
for each vertex v in V do
   if Isshared[v] = false & v ∉ CombinedVertices then
      TempArray[]▷ Array for holding Vertices
      CallCheck(G(V,E),v) ▷ Check vertices as true for all the paths with root vertex as
      v until shared vertex is reached.
   end if
   DagLongestPath(G(V,E),v) ▷ Longest path algorithm with root vertex as v
   for each vertex u in V do
      if Checked[u] = true ▷ if the vertex is checked then
         if  distance[u] + Latency[u] <clockvalue  then
            TempArray ← u ▷ add vertex to Temporary Array
         end if
      end if
   end for
   for each edge e in E do
      ▷  Traverses the edges and if both source and target of the edge is in the array, put
      this edge in EdgeArray
      if V_t(e) ∈ TempArray & V_h(e) ∈ TempArray then
         ZeroEdgeArray ← e
      end if
   end for
end for
```

Figure 6.8. FindChainableOperators Algorithm.

```
▷ Takes the graph as input and a vertex as root.
▷ With the given Vertex as root, finds the paths until a shared Vertex is reached and
marks these vertices as Checked
for each vertex v in Adj(root) do
  if Isshared[v] = false then
    Checked[v] ▷ true
  end if
  if Checked[u] = true & Precedencelevel[u] >Precedencelevel[root] then
    CallCheck(u)
  end if
end for
```

Figure 6.9. Call Check Algorithm.

edgecost. For example, during rescheduling, all edges are controlled whether they exist in the ZeroEdgeArray or not. If it doesn't exist, the number of clocks occupied by the source vertex of the edge is given as edge cost to that vertex.

(vii) A new variable vertex node, named 'z', is created. All the input variable vertices in the graph are connected to that vertex by inserting edges from node 'z' to these input variable vertices. This 'z' vertex is used during the application of DagLongestPath algorithm as the root vertex so it will not be required to traverse all input vertices as root vertex and find the root vertex that results with the maximum path value.

(viii) DaglongestPath is applied by selecting this 'z' vertex as root vertex. A listener is used to observe distances of the vertices in the graph from that vertex. Maximum distance corresponds to number of control steps for that clock period. Number of control steps multiplied with the clock period corresponds to the total latency of the graph for that clock period.

6.2.2. Illustrative Example

In this section, the rescheduling methodology is explained by example. For that reason, the graph in Figure 5.2 is rescheduled according to the optimization results in

```
▷ Graph, ClockValue, ZeroEdgeArray is given as input
▷ Reschedule graph with the Clock Value and Chainability Information
ClockOccupationArray[] ▷ Define array to hold clock occupation value
Cost[] ▷ Define Array to hold edge costs
Read Technology File () ▷ Read register delay from technology parameter file
clockvalue ← clockvalue + register delay ▷ new rescheduling clock is the (clock period
+ register delay)
for each vertex v in V do
   Latency[v] ← Latency[v] + register delay ▷ add register delay to vertex costs
end for
for each vertex v in V do
   ClockOccupationArray[v] ← Ceil(Latency[v] / Clockvalue ▷ Clock occupation is cal-
   culated, by ceiling the Latency value of the vertex divided by clock value
end for
for each vertex v in V do
   if e ∈ ZeroEdgeArray then
      Cost[e] = 0 ▷ if the edge is in the EdgeArray then its cost is 0
   else
      Cost[e] = Latency[V_t(e)] ▷ else edge cost is the clock occupation of source vertex
   end if
end for
Create a new node z ▷ Creating a node source node which all input variable vertices are
to be connected
for each vertex v in V do
   ▷Add edge from the node z to vertex if the vertex is input variable
   if Precedencelevel[v] = 1 & Isoperator[v] = false then
      Addedge(z,v)
   end if
end for
DagLongestPath(G,z)
```

Figure 6.10. Schedule Graph Algorithm.

```
@tech
*Property = delay
*Input_register_delay=1.726
*Output_register_delay=0.176
*IBUF_delay=1.691
*OBUF_delay=5.59
*Net_delay=0.976
@endtech
```

Figure 6.11. Spartan-3 Technology Parameter File.

Figure 6.1. According to these results SE4 and SE6 shares the same hardware resource and SE5 and SE7 shares the same hardware resource as depicted in Figure 6.12.

In Table 6.1, the delays of the vertices, the register delay for Spartan-3 FPGA and vertex delays with registers are given. The graph is rescheduled for each vertex+register cost in the table by selecting that cost as the clock period. In Figure 6.13, the rescheduling of the graph with clock period 3.756 ns is shown. The total number of control steps for that clock period is 14 which leads to 3.756 x 14 = 52.584 ns total latency. In Figure 6.14, the rescheduling of the graph with clock period 3.815 ns is shown. The total number of control steps for that clock period is 13 which leads to 3.815 x 13 = 49.595 ns total latency. In Figure 6.15, the rescheduling of the graph with clock period 3.868 ns is shown. The total number of control steps for that clock period is 13 which leads to 3.868 x 13 = 50.284 ns total latency. In Figure 6.16, the rescheduling of the graph with clock period 5.297 ns is shown. The total number of control steps for that clock period is 12 which leads to 5.297 x 12 = 63.564 ns total latency. In Figure 6.17, the rescheduling of the graph with clock period 6.328 ns is shown. Here, the multicycle operators(SE4 and SE6) and chained operators(SE1 and SE2) can be seen. The total number of control steps for that clock period is 7 which leads to 6.328 x 7 = 44.296 ns total latency. In Figure 6.18, the rescheduling of the graph with clock period 10.878 ns is shown. The total number of control steps for that clock period is 6 which leads to 10.878 x 6 = 65.268 ns total latency. The clock period value which gives the lowest total delay of the graph is 6.328 ns. The total

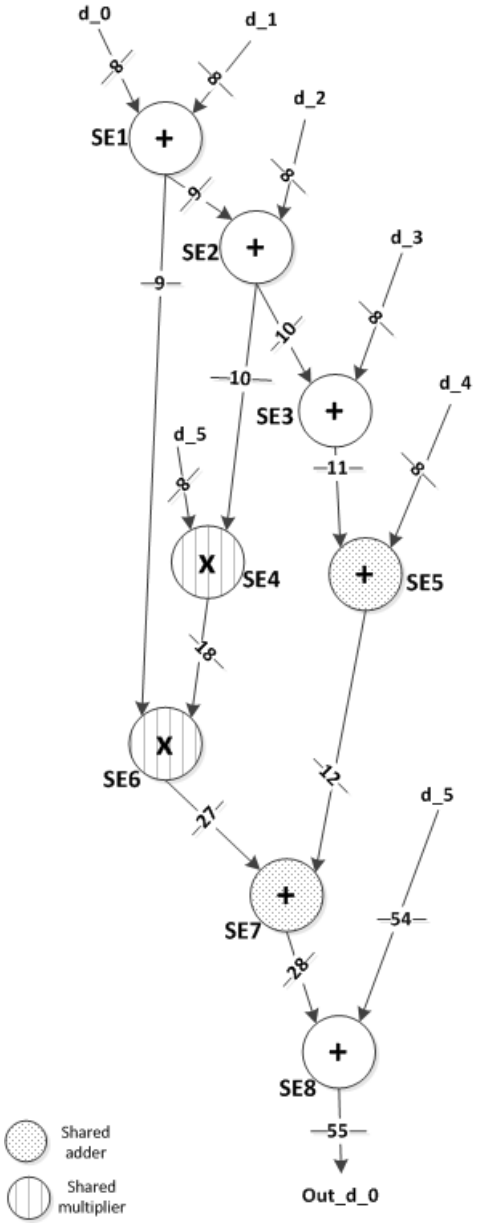

Figure 6.12. Example Graph to be Rescheduled.

Table 6.1. Delay Costs of the vertices in the example graph.

Vertex Name	**Delay Cost(ns)**	**Register Delay(ns)**	**Vertex with Register Delay (ns)**
SE1	1.978	1.778	3.756
SE2	2.037	1.778	3.815
SE3	2.09	1.778	3.868
Shared Multiplier	9.1	1.778	10.878
Shared Adder	3.519	1.778	5.297
SE8	4.55	1.778	6.328

optimized latency of the graph before rescheduling is 30.284 ns. The total latency after rescheduling is 44.296 ns. This overhead caused by the registers which are not taken into account in the optimization model.

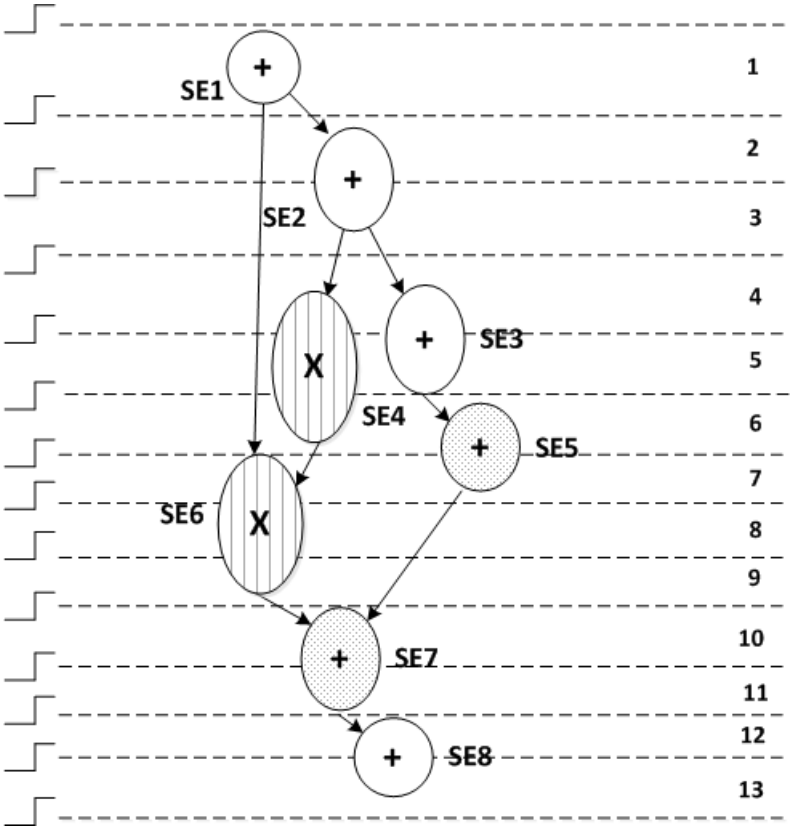

Figure 6.13. Rescheduling the graph with clock period = 3.756 ns.

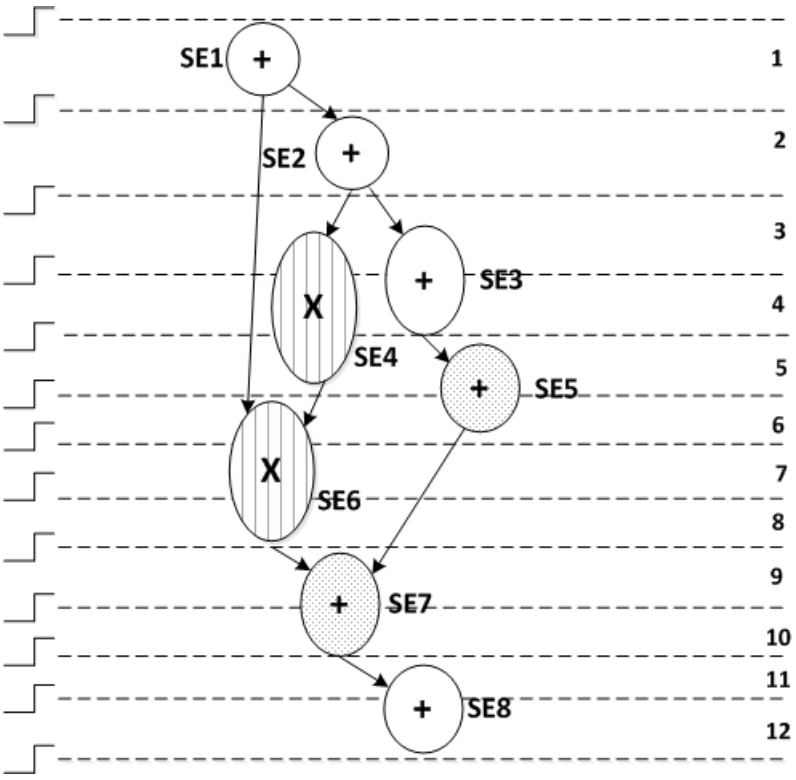

Figure 6.14. Rescheduling the graph with clock period = 3.815 ns.

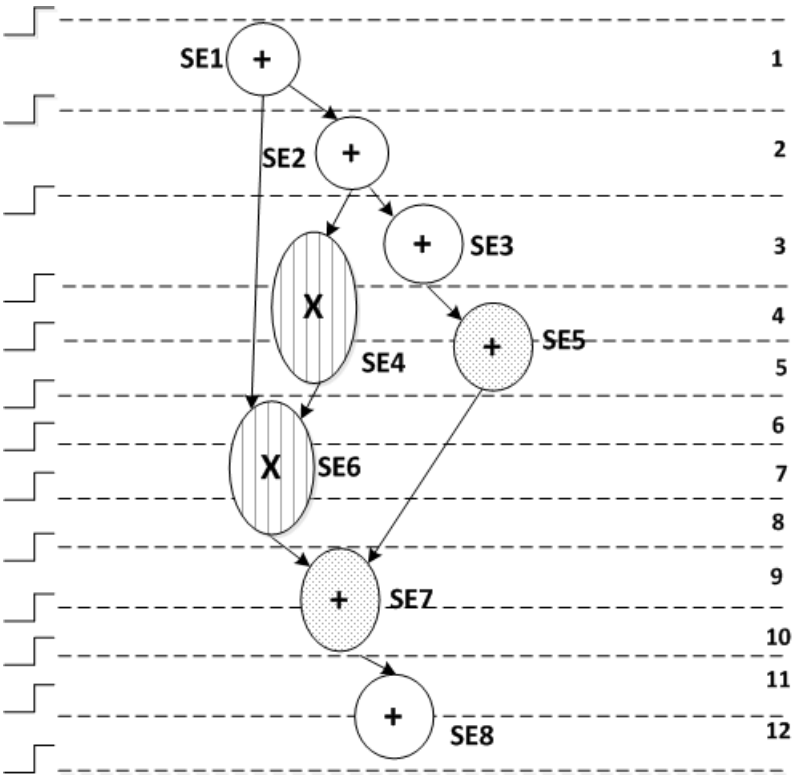

Figure 6.15. Rescheduling the graph with clock period = 3.868 ns.

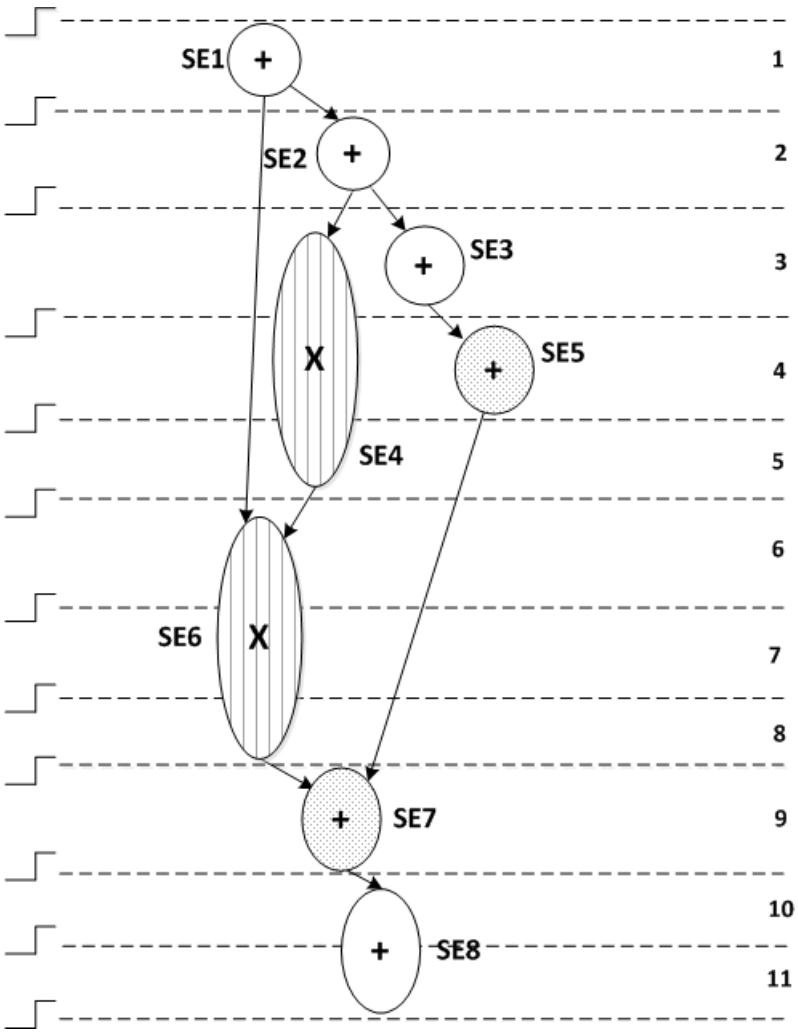

Figure 6.16. Rescheduling the graph with clock period = 5.297 ns.

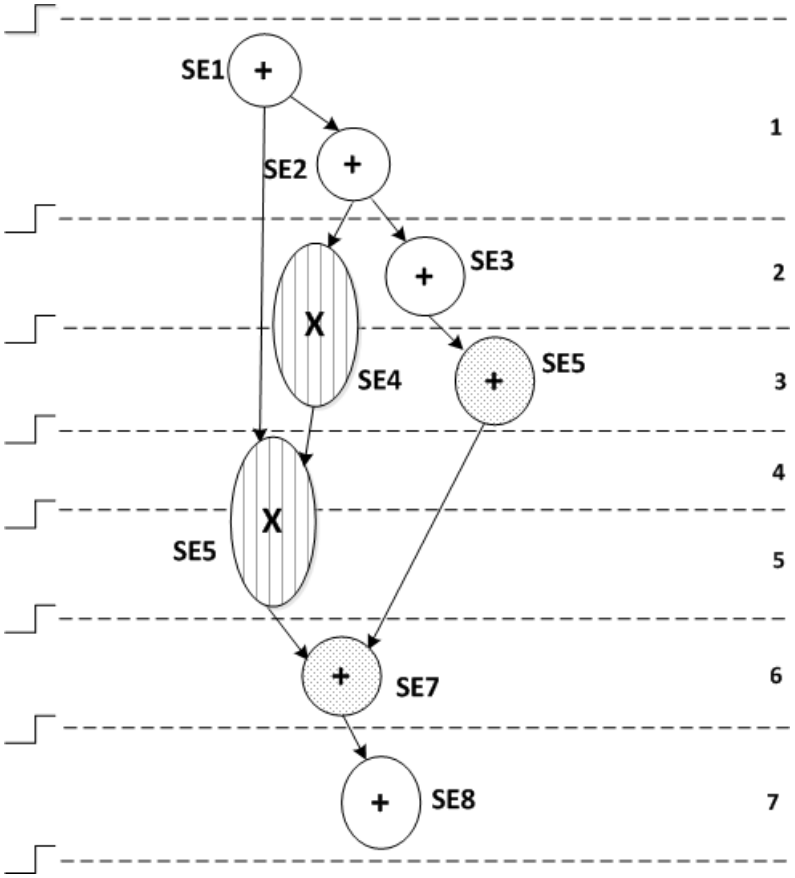

Figure 6.17. Rescheduling the graph with clock period = 6.328 ns.

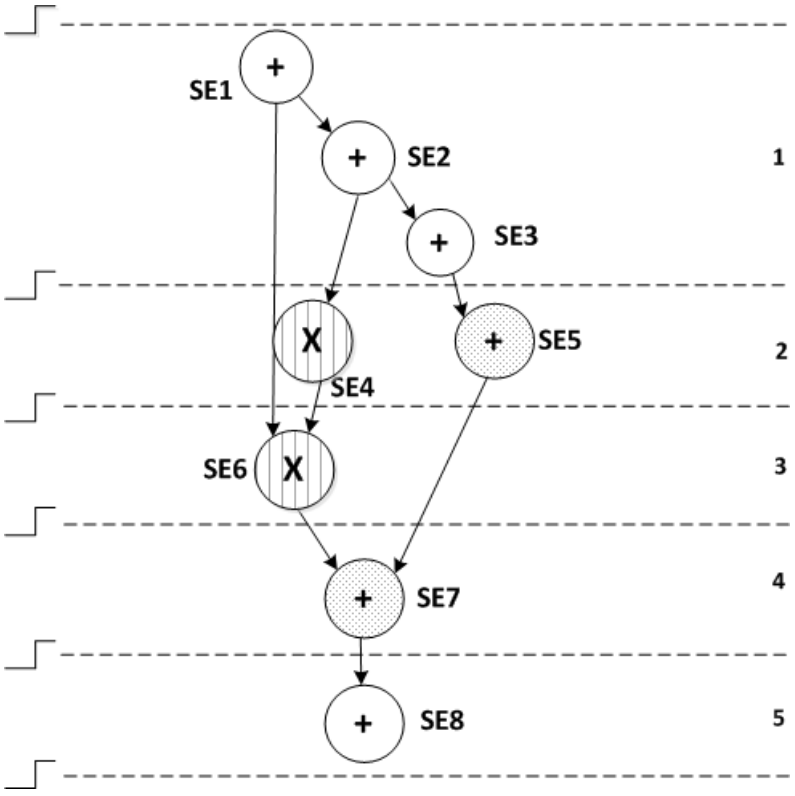

Figure 6.18. Rescheduling the graph with clock period = 10.878 ns.

7. OPTIMIZED RTL GENERATION

As shown in the Figure 1.1, the Optimized RTL Generation module comes after the graph passes through the optimization, scheduling & allocation module and rescheduling module. In Optimized RTL Generation module, the datapath, controller and register files of the optimized and rescheduled graph are generated. In addition to the VHDL files of the datapath, controller and registers, the TCL file is also generated to synthesize and place & route of the design for the Xilinx FPGAs.

7.1. Optimized RTL Generation Methodology

The generated optimized RTL of the example graph in Figure 6.12, is given here. The graph is rescheduled with clock period of 4.55 ns as in Figure 6.17. The rescheduled graph is given in Figure 7.1 including wordlengths. The number on the edges represent the signal wordlengths between vertices. The generated optimized RTL is given in Figure 7.2.

7.1.1. Optimized RTL Generation Rules

7.1.1.1. Unshared Operators. If the operator vertex is an unshared operator, then there must be a register at the end of the operator. If two or more unshared operators are chained and cascaded in the same clock cycle, then these operators can be connected to each other without registering. The registers are put at the end of the clock cycle if necessary. For example, if two operators are chained and cascaded in the same clock cycle, these are connected to each other without any registers between them. But if the first operator is also connected to other operator, it should be registered at the end of that clock cycle.

- For instance, in Figure 7.2, SE8 is not a shared operator, and there is a register at the end of the operator.
- For instance, as can be seen in figure 7.1, SE1 and SE2 are chained in one clock

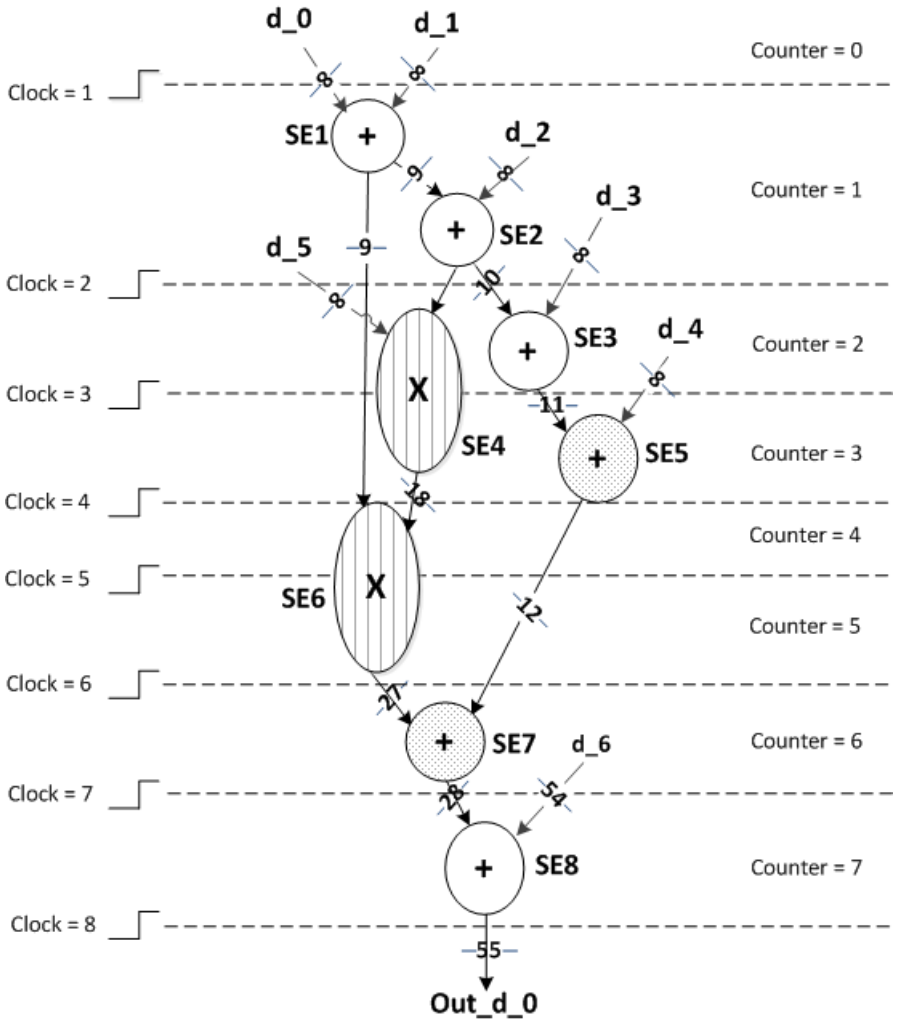

Figure 7.1. Example Graph.

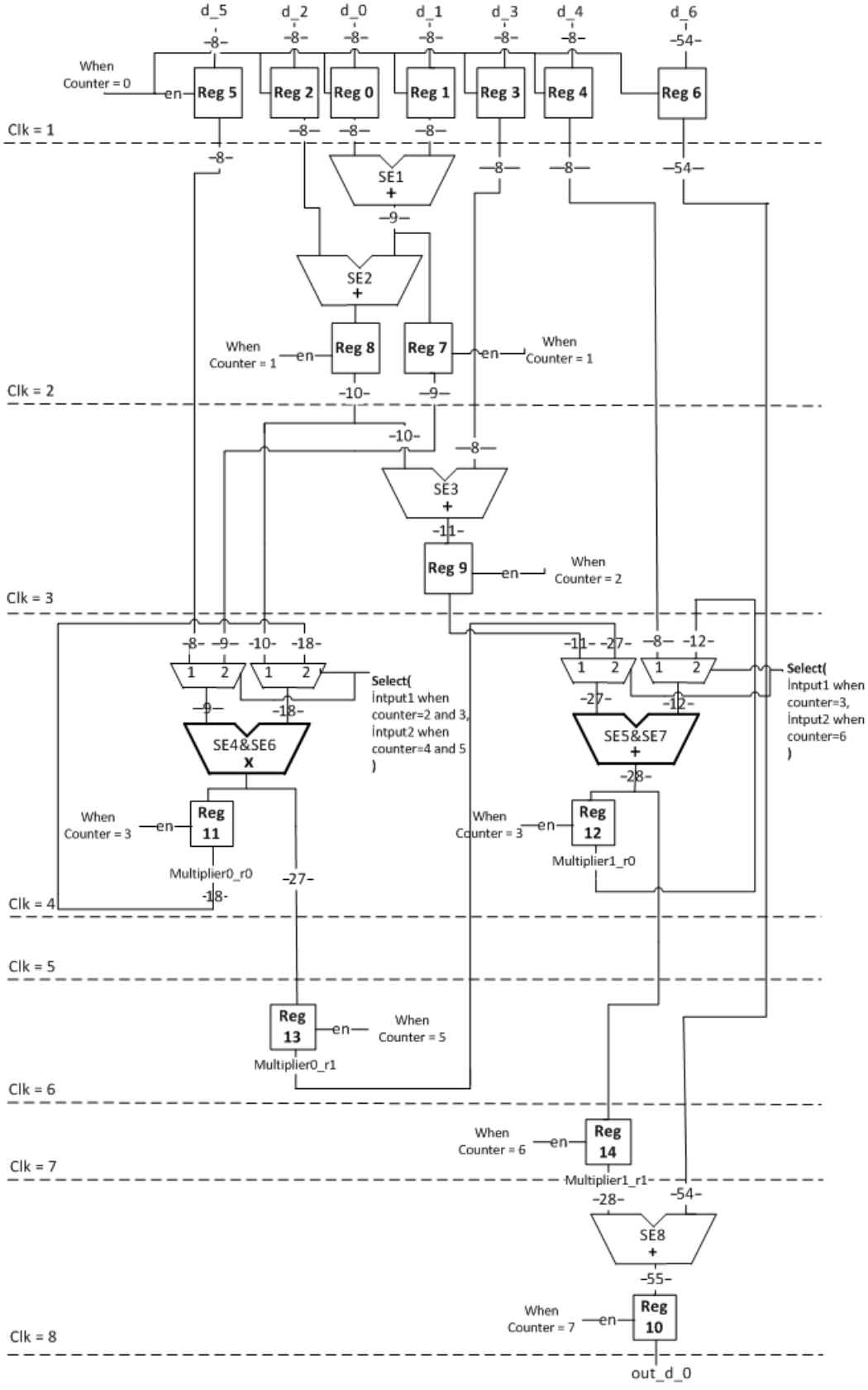

Figure 7.2. Datapath Generation of the Example Graph.

cycle. These operators are connected to each other without registering as shown in Figure 7.2. Outputs of the both operators are registered at the end of clock cycle and these registers are controlled with the same enable signal.

7.1.1.2. Shared Operators. There are separate registers at the output of the shared operators. The number of the registers is the number of operations sharing that operator. These registers are enabled separately. For example, in the example graph in Figure 7.1, SE4 and SE6 operators are shared and they use the same multiplier. And, in the datapath of the graph which is shown in Figure 7.2, there are two registers at the output where each register holds the result of one operator. These registers are enabled only at the specified scheduled instances of the shared operator.

- For instance, for the graph in Figure 7.1, SE4 and SE6 operators are shared and they use the same multiplier resource. In the generated RTL, shown in Figure 7.2, there are two registers at the output of the Shared Multiplier(SE4&SE6), where Reg11 holds the output of SE4 operation and Reg13 holds the output of SE6 operation. Reg11 is enabled when the counter is 3 and reg13 is enabled when the counter is 13.
- For instance, for the graph in Figure 7.1, SE5 and SE7 operators are shared and they use the same adder resource. In the generated RTL, shown in Figure 7.2, there are two registers at the output of the Shared Adder(SE5&SE7), where Reg12 holds the output of SE5 operation and Reg14 holds the output of SE7 operation. Reg12 is enabled when the counter is 3 and reg14 is enabled when the counter is 14.

For shared operators, vertices are arranged according to their execution clock time. A dictionary holds the operator's execution time count. For example, if an operator is shared for 4 times. The first operation to be executed has the value of 0, the second has the value 1 etc. These values differentiates the output signals of the shared operators, by naming like <signal_name>_r<number>.

- For instance, in the example graph, SE4 is the first operator to be executed and SE6 is the second operator for the shared multiplier resource. So, SE4 has the value of 0 and SE6 has the value of 1. If the component name of the multiplier which corresponds to SE4&SE6 in Figure 7.2 is Multiplier0, then the output signal name for the SE5 operator is <signalname>_r0 (e.g Multiplier0_r0) and SE6 is <signalname>_r1 (e.g Multiplier0_r1) in the datapath.

7.1.1.3. Multiplexers. In FPGAs, the fastest multiplexers are obtained when the number of ports is a nonnegative power of two. For example, if three vertices share the same operator, a multiplexer with four ports is faster in delay and lower in area than a multiplexer with three ports. For that reason, in datapath multiplexers with port-size of 2, 4, 8, 16, 32 and 64 are used. If a resource is shared by 3 operators, then for that operator multiplexer with 4 ports is used where one of the input port is left unconnected.

7.1.1.4. Controller Part. The registers and multiplexers must be controlled in order to provide the functionality in the graph. There is a clock time for every graph that the output of the graph is ready. A counter holds the clock time and resets itself periodically when the output of the graph becomes ready. This counter is also a fundamental part of the controller. For example, for the example graph in Figure 7.1, the counter value will be reset after the value of 8. With counter, we know each operator's execution time.

- For instance, for the example graph in Figure 7.1, the graph executes for 8 clock cycles. This means that output of the graph is ready in every 8 clocks. For that reason, the counter counts up to 7 and resets itself to zero after that.

The execution time of the each operator is the graph is known. This counter is used to control execution of the operators enabling the registers at the output of the operators and selecting port of the multiplexers according to the counter value.

- Each register is controlled with a different signal. Counter is used for enabling these register. Registers are enabled while the operator which is the input of that register is being executed.
  (i) For instance, SE8 operator in Figure 7.1 is executed when the counter = 7. So, Reg10 register at the output of the SE8 operator in Figure 7.2 is enabled when the counter is 7.
- Select signals of the multiplexers of the shared operators are controlled with counter as well.
  (i) For instance, for the graph in Figure 7.1 and for the generated RTL in Figure 7.2, SE5 and SE7 share the same adder. The SE5 operation is executed when the counter is 3 and SE7 operation is executed when the counter is 6. So, the select signal of the multiplexer in front of the shared adder(SE5&SE7), selects the first port when the counter is 3 and selects the second port when the counter is 6. The SE4 operation is executed when the counter is 2 and 3, SE7 operation is executed when the counter is 4 and 5. So, the select signal of the multiplexer in front of the shared multiplier(SE4&SE6), selects the first port when the counter is 2 and 3 and selects the second port when the counter is 4 and 5.

- Extension or Truncation of the signals might be necessary while connecting the signals to the operators. For instance, for the graph in Figure 7.1 and for the generated RTL in Figure 7.2, one port of the SE8 adder is 28 bits and the other port is 54 bits. This adder is generated with 54 bit wordlength so, the signal with the wordlength of 28 bits is extended as explained in Section 7.1.2.2.

7.1.2. Optimized RTL Generation Software

Datapath Generation software flow is shown in the Figure 7.3. The graph is firstly processed with ModifyGraphWithMux function. The graph passes through the ModifyGraphWithMux, GenerateDatapath, GenerateController, and GenerateTCL functions respectively.

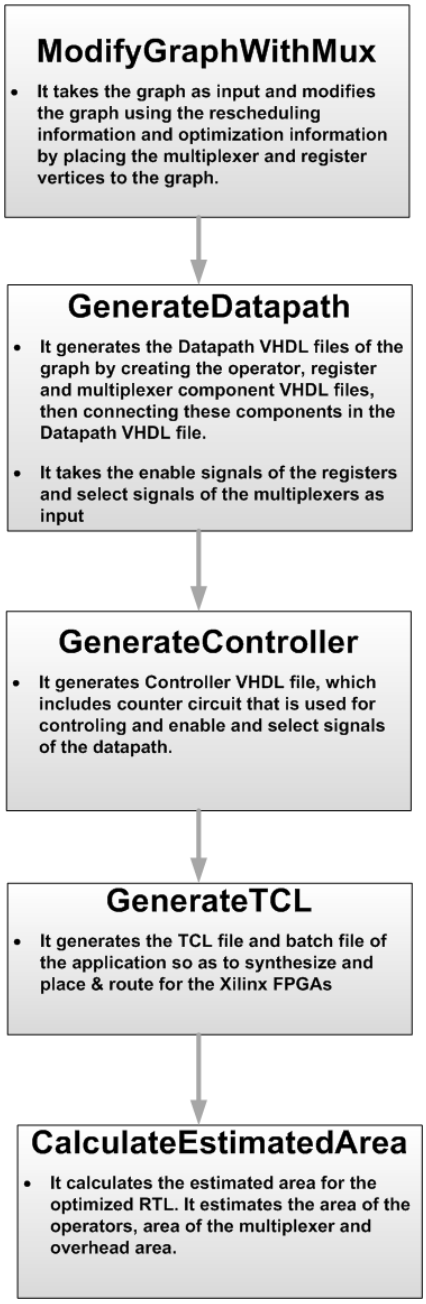

Figure 7.3. Datapath Generation Software Flow.

7.1.2.1. ModifyGraphWithMux Function. Before the VHDL file generation, the graph is firstly processed to insert multiplexer and register vertices into the graph and connect them with operator vertices. Rescheduling information and sharing information which is determined by the optimization tool is used during the modification of the graph. In Figure 7.4, the modified graph of the example graph in Figure 7.1 is shown.

- Registers are placed at the end of unshared operators as well as after input variables and before output variables.
- Multiplexers are placed before the shared operators (shown in gray).
- Shared multiplier in Figure 7.4 corresponds to SE4 and SE6 in Figure 7.1. The inputs of the SE4 and SE6 are connected to the multiplexers in front of the shared multiplier. Two registers are placed at the end of the shared multiplier which holds the outputs of the SE4 and SE6 operations. The output edges of SE4 are connected as the outgoing edges of reg10 vertex and the outgoing edges of SE6 vertex are connected as the outgoing edges of reg12 vertex in Figure 7.4.
- Shared adder in Figure 7.4 corresponds to SE5 and SE7 in Figure 7.1.The inputs of the SE5 and SE7 are connected to the multiplexers in front of the shared adder. Two registers are placed at the end of the shared adder which holds the outputs of the SE5 and SE7 operations. The output edges of SE5 are connected as the outgoing edges of reg11 vertex and the output edges of SE7 are connected as the outgoing edges of reg13 vertex in Figure 7.4.

7.1.2.2. GenerateDatapath Function. GenerateDatapath function takes the modified graph as input and generates the VHDL files of the datapath of the graph by following the steps below;

(i) Firstly, the VHDL files of the arithmetic components, registers and multiplexers are generated. The arithmetic components are generated using the parameter files and template files which is explained in Section 3.5. For multiplexer and register parameter files are not necessary, the generic VHDL files for registers and multiplexers are used where the input wordlengths are adjusted during port-

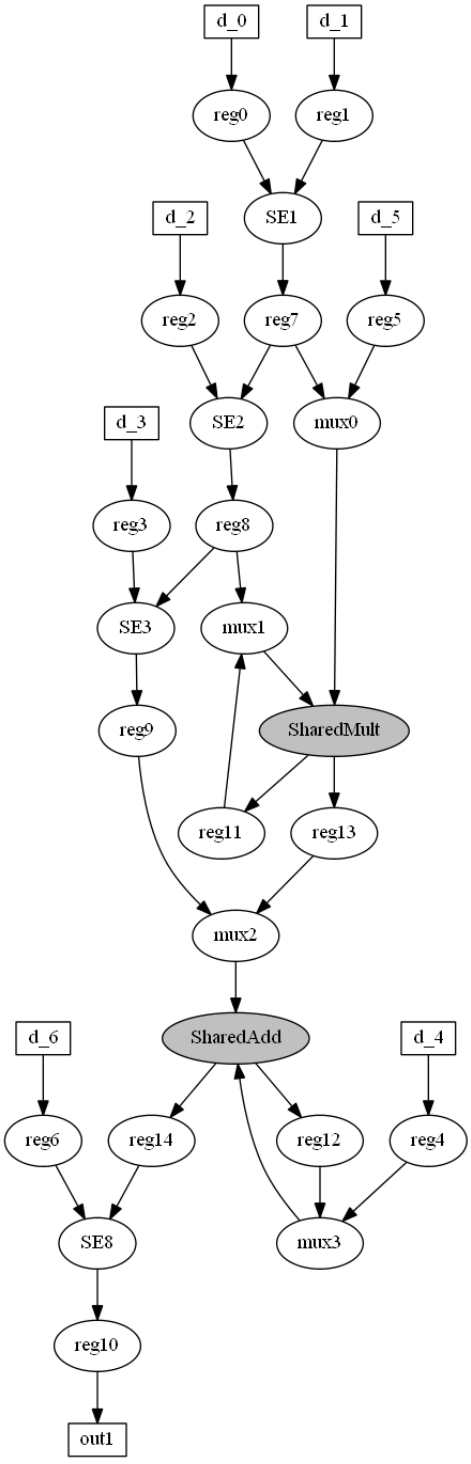

Figure 7.4. Example Graph after modified with multiplexers.

mapping. The generic VHDL register and multiplexer template are added to the project.

(ii) Secondly, XCO files of the used components from the Xilinx Coregen library are generated. XCO file records all the customization parameters used to create the core. During synthesis, Xilinx reads the XCO file of the used core and generates the coregen file which is required to synthesize the design. XCO file of multiplier is given in Figure 7.5 as example. As it can be seen in Figure 7.5, the properties of the multiplier is defined in XCO file. In this thesis, multiplier and dividers are used from the Xilinx Coregen library. For multiplier, coregen multiplier version 11.2 and for divider coregen divider version 3.0 are used.

(iii) Thirdly, in the commenting section of the datapath VHDL file, The CDFG vs VHDL part is created where, which operators in the CDFG corresponds to which component in the VHDL file and which edge in the CDFG corresponds to which signal in the VHDL file. An example of this section is shown in Figure 7.6. Here, CDFG part includes the vertex and edge names in the modified graph and VHDL part includes the corresponding component and signals in the VHDL file.

(iv) Fourthly, Datapath VHDL file is generated which connect generated operators, registers, multiplexer functions at the top level datapath VHDL file. During datapath generation, following functions are called respectively.

- *DatapathEntityDeclare :* This function makes the entity declaration of the datapath VHDL file. The graph is traversed, data signals are declared for variable vertices, enable signals are declared for register vertices and select signals are declared for multiplexer vertices. The enable signals of the registers, the select signals of the multiplexers, clock signal and input variables are given as input to the Datapath VHDL file and output variable is the output of the Datapath VHDL file.
- *ComponentDeclare :* This function makes the component declarations of the generated operators, register and multiplexers in the architectural part of the VHDL files.
- *SignalDeclare :* This function makes the signal declarations of the used signals in the Datapath VHDL file. It traverses the edges and makes signal

```
SET addpads = false
SET device = xc3s5000
SET devicefamily = spartan3
SET flowvendor = Foundation_ISE
SET implementationfiletype = Ngc
SET package = fg1156
SET speedgrade = -5
SET verilogsim = true
SET vhdlsim = true
CSET ccmimp=Distributed_Memory
CSET clockenable=false
CSET component_name=mult_0
CSET constvalue=129
CSET multiplier_construction=Use_LUTs
CSET optgoal=Speed
CSET outputwidthhigh=71
CSET outputwidthlow=0
CSET pipestages=0
CSET portawidth=36
CSET portbwidth=36
CSET roundpoint=0
CSET syncclear=false
```

Figure 7.5. Multiplier XCO File.

```
---------- CDFG vs VHDL --------------
--- node d_0 = d_0
--- node d_1 = d_1
--- node SE1 = RCAAdder0
--- node d_2 = d_2
--- node SE2 = RCAAdder1
--- node d_3 = d_3
--- node d_4 = d_4
--- node SE4 = Multiplier0
--- node SE3 = RCAAdder2
--- node d_5 = d_5
--- node SE6 = Reg0
--- node SE5 = RCAAdder3
--- node SE7 = Reg1
--- node SE8 = RCAAdder4
--- node out1 = out_d_0
--- node d_6 = d_6
--- node reg0 = reg0
--- node reg1 = reg1
--- node reg2 = reg2
--- node reg3 = reg3
--- node mux0 = mux0
--- node mux1 = mux1
--- node mux2 = mux2
--- node mux3 = mux3
--- edge e0 = d_0
--- edge e1 = d_1
--- edge e2 = RCAAdder0_d0
--- edge e3 = d_2
--- edge e4 = RCAAdder1_d0
--- edge e5 = d_3
--- edge e6 = d_4
--- edge e7 = Multiplier0_d0_t18
--- edge e8 = Multiplier0_d0
--- edge e9 = RCAAdder2_d0
--- edge e10 = d_5
--- edge e11 = Multiplier0_r2
--- edge e12 = RCAAdder3_d0_t12
--- edge e13 = RCAAdder3_d0
--- edge e15 = RCAAdder4_d0
--- edge e16 = RCAAdder1_r1_e18
--- edge e17 = RCAAdder2_r1_e27
--- edge e18 = RCAAdder4_r1
--- edge e19 = mux0_d0
--- edge e20 = mux1_d0
```

Figure 7.6. Datapath Comment of CDFG vs VHDL.

declaration of each edge with using wordlength information of the source vertex of the edge. The signals that is required to extend or trimmed are also declared in this function.

- *SignalExtend :* After all the declarations finish, this function extends and trims the required signals before port-mapping. Signals are extended or trimmed as shown below;

  Multiplier2_d0_t32 <= Multiplier2_d0(31 downto 0); : In "t32" t means this signal is truncated and 32 means the wordlength of the signal after it is truncated.

  Multiplier2_r2_e65 <="00"&Multiplier2_r2; : In "e65" e means this signal is extended and 65 means the wordlength of the signal after it is extended.
- *PortMap :* This function connects the operators, registers and multiplexers with port-mapping.

(v) In this function, a dictionary file is also generated which stores the register names with respect to name of their enable signals and multiplexer names with respect to the name of their select signals. This dictionary file is used during controller generation.

(vi) In this function, the entity declaration of Top Level file is also created since, the entity signals of the Top level is the data signals of the datapath VHDL file.

7.1.2.3. GenerateController Function. This function generates the controller VHDL file of the graph. The clock is the input of the controller VHDL file and enable signals of the registers and select signals of the multiplexers are the output. In the controller file, there is a counter which resets itself when the output of the graph becomes ready. From the rescheduling of the graph, the execution time of each operation is specified, and from the dictionary file,which is generated during datapath generation, the enable signals of the registers and select signals of the multiplexers are known. Using these information, the controller VHDL file is created as explained below;

(i) Firstly, the entity of the controller VHDL file is declared. Clock and reset signal is the input and enable signals of the register and select signals of the multiplexers

```
counter :  process(clk,rst)
begin
if ( rst = '1') then
pre_count <= ''0000";
elsif (clk='1' and clk'event ) then
if ( pre_count <''1101" ) then
pre_count <= pre_count + 1;
else
pre_count <= ''0000";
end if;
end if;
end process;
count <= pre_count;
```

Figure 7.7. Counter Example.

in the datapath VHDL file is the output of the controller VHDL file.

(ii) Secondly, a counter is defined which counts up to maximum number of control steps of the graph and resets itself when the output of the graph is ready. In Figure 7.7, a counter VHDL code is given as an example for a graph that the output becomes ready after 12 clock cycles. So the counter counts up to 12 and reset itself.

(iii) Thirdly, enable signals of the registers are created. In the dictionary file that is generated, the register and their enable signals are stored. Using that file and the information when the registers are enabled, the enable signals are created according to the counter. An example enable signal creation VHDL code is given in Figure 7.8. This enable signal is active when the counter is 8.

(iv) Fourthly, the select signals of the multiplexers are created. The dictionary file created during datapath generation is used which signal selects which multiplexer in the graph. An example multiplexer signal creation VHDL code is given in Figure 7.9.

(v) In this function, controller and datapath is connected with port-mapping in the Top Level VHDL file.

```
enable0 :  process(count)
begin
CASE count IS
WHEN ‘‘1000" =>en_0 <= ‘1’;
WHEN OTHERS =>en_0 <= ‘0’;
END CASE;
end process;
```

Figure 7.8. Enable Signal Creation.

```
enselect0 :  process(count)
begin
CASE count is
WHEN ‘‘0011’’ =>sel_0 <= ‘‘0’’;
WHEN ‘‘0100’’ =>sel_0 <= ‘‘0’’;
WHEN ‘‘0101’’ =>sel_0 <= ‘‘0’’;
WHEN ‘‘0110’’ =>sel_0 <= ‘‘1’’;
WHEN ‘‘0111’’ =>sel_0 <= ‘‘1’’;
WHEN ‘‘1000’’ =>sel_0 <= ‘‘1’’;
WHEN OTHERS =>sel_0 <= ‘‘0’’;
END CASE;
end process;
```

Figure 7.9. Select Signal Creation.

7.1.2.4. GenerateTCL Function. After all the VHDL files are generated for the graph, TCL file is generated which is used for synthesizing, mapping, place & routing the design in the Xilinx ISE for Xilinx FPGAs. In addition to TCL file, batch file is generated so as to run TCL file in Windows OS. TCL file does the following actions respectively;

- sets the synthesis directory and working path
- sets the VHDL files to be synthesized
- deletes the synthesis directory if exist
- creates a new Xilinx ISE project
- sets FPGA family, device, package and speed
- sets synthesis options
- add each VHDL file into the project
- Regenerate all coregen components
- Synthesizes, maps, places & routes the design
- archives the folder by zipping it

An example TCL file is shown in the Figure 7.10.

7.1.2.5. CalculateEstimatedArea Function. The clock period for the application graph is already extracted during rescheduling of the graph in Section 6. This function estimates the consumed area of the optimized RTL. It separately estimates the area of the operators, multiplexer and overhead caused by registers. An example output of the estimation results is given in Figure 7.11. In Figure 7.11, Datapath Area corresponds to area of operators in the graph, Mux Area corresponds to area of multiplexers in the graph, Overhead Area corresponds to area of overhead caused by the registers in the graph, Total Area corresponds to the sum of all. Overhead area is caused by the shared operators. During Datapath Generation, if a hardware resource is shared by more than one operator, the number of register files placed at the end of the hardware resource is the number of how many operators share this resource. First register file does not cause an overhead, because the flip-flops in the used slices are used. For the

```
 set synth_directory synthExampleGraph_8_latency_1_1
 set working_path ‘‘D:/RHPlus/DatapathGeneratedFiles’’
 set project_name ExampleGraph_8_latency_1_1
 # input source files:
 set hdl_files [ list
 D:/RHPlus/DatapathGeneratedFiles/ExampleGraph_8_latency_1_1/mult_0.xco
 D:/RHPlus/DatapathGeneratedFiles/ExampleGraph_8_latency_1_1/Mux2_1.vhd
 D:/RHPlus/DatapathGeneratedFiles/ExampleGraph_8_latency_1_1/Reg.vhd
 D:/RHPlus/DatapathGeneratedFiles/ExampleGraph_8_latency_1_1/TopLevel.vhd
]
 # synth directory
 file delete -force ./$synth_directory
 if [![file isdirectory ./$synth_directory]] [
 file mkdir ./$synth_directory ]
 project new $working_path/synth/$project_name.ise
 project set family Spartan3
 project set device xc3s1000
 project set package fg676
 project set speed -5
 project set ‘‘Optimization Goal" ‘‘Speed" -process ‘‘Synthesize - XST"
 project set ‘‘Optimization Effort" ‘‘High" -process ‘‘Synthesize - XST"
 project set ‘‘Place & Route Effort Level (Overall)" ‘‘High" -process
‘‘Place & Route"
 foreach filename $hdl_files [
 xfile add $filename
 puts ‘‘Adding file $filename to the project" ]
 process run ‘‘Regenerate All Cores"
 process run ‘‘Synthesize - XST"
 process run ‘‘Implement Design"
 project archive $working_path/synthExampleGraph_8_latency_1_1.zip
 project clean
 file delete -force $working_path/$synth_directory
```

Figure 7.10. TCL File Example.

```
Estimated Datapath Area :  48.24
Estimated Mux Area :  17
Estimated Overhead Area :  17
Estimated Total Area :  82.24
```

Figure 7.11. An example output for the area estimation of Optimized RTL.

other register files, new slices are used which cause an overhead in the area.

7.2. VHDL File Example of 8 bit Differential Equation

The Generated VHDL files of a 8-bit Differential Equation is given. In Figures 7.12 to 7.17, the generated Datapath VHDL file is given. In Figures 7.18 to 7.22, the generated Controller VHDL file is given. In Figures 7.23 to 7.25, the generated Top Level VHDL file is given.

```
---------- CDFG vs VHDL --------------
--- node 3 = d_0
--- node x = d_1
--- node u = d_2
--- node dx = d_3
--- node y = d_4
--- node SE61 = Multiplier0
--- node SE60 = Reg0
--- node SE62 = Reg1
--- node SE63 = Reg2
--- node SE64 = RCAAdder0
--- node SE65 = Multiplier1
--- node SE66 = Multiplier2
--- node SE67 = Reg3
--- node SE68 = RCASubtractor0
--- node a = d_5
--- node SE69 = RCASubtractor1
--- node SE70 = RCASubtractor2
--- node u1 = out_d_0
--- node x1 = out_d_1
--- node y1 = out_d_2
--- node c = out_d_3
--- edge e1 = d_0
--- edge e2 = d_1
--- edge e3 = d_2
--- edge e4 = d_3
--- edge e5 = d_4
--- edge e6 = Multiplier0_d0
--- edge e7 = Multiplier0_d0
--- edge e8 = Multiplier0_d0
--- edge e9 = Multiplier0_d0
--- edge e10 = Multiplier0_r2
--- edge e11 = Multiplier0_r3
--- edge e12 = Multiplier0_r4
--- edge e13 = RCAAdder0_d0
--- edge e14 = RCAAdder0_d0
--- edge e15 = Multiplier1_d0
--- edge e16 = Multiplier2_d0
--- edge e17 = Multiplier2_d0
--- edge e23 = Multiplier2_r2
--- edge e18 = RCASubtractor0_d0
--- edge e19 = d_5
--- edge e21 = RCASubtractor1_d0
--- edge e20 = RCASubtractor2_d0
--- edge e22 = d_0_r1
--- edge e23 = d_0_r1
```

Figure 7.12. Differential Equation Datapath VHDL File 1.

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
library UNISIM;
use UNISIM.VComponents.all;
entity
diff_eq is
port (
 clk:  in std_logic;
 rst:  in std_logic;
 en_0:  in std_logic;
 en_1:  in std_logic;
 en_2:  in std_logic;
 en_3:  in std_logic;
 en_4:  in std_logic;
 en_5:  in std_logic;
 en_6:  in std_logic;
 en_7:  in std_logic;
 en_8:  in std_logic;
 en_9:  in std_logic;
 en_10:  in std_logic;
 en_11:  in std_logic;
 en_12:  in std_logic;
 sel_0:  in std_logic_vector(1 downto 0);
 en_13:  in std_logic;
 sel_1:  in std_logic_vector(1 downto 0);
 en_14:  in std_logic;
 sel_2:  in std_logic_vector(0 downto 0);
 sel_3:  in std_logic_vector(0 downto 0);
 en_15:  in std_logic;
 d_0:  in std_logic_vector(7 downto 0);
 d_1:  in std_logic_vector(7 downto 0);
 d_2:  in std_logic_vector(7 downto 0);
 d_3:  in std_logic_vector(7 downto 0);
 d_4:  in std_logic_vector(7 downto 0);
 d_5:  in std_logic_vector(7 downto 0);
 out_d_0:  out std_logic_vector(33 downto 0);
 out_d_1:  out std_logic_vector(8 downto 0);
 out_d_2:  out std_logic_vector(23 downto 0);
 out_d_3:  out std_logic_vector(9 downto 0));
end entity;
```

Figure 7.13. Differential Equation Datapath VHDL File 2.

```
architecture Structural of diff_eq is
component Multiplier_002_4_i1_16_i2_16 is
generic(
g_BITSIZE_0 :  integer := 16;
g_BITSIZE_1 :  integer := 16 );
port (
d_0 :  IN std_logic_VECTOR(g_BITSIZE_0-1 downto 0);
d_1 :  IN std_logic_VECTOR(g_BITSIZE_1-1 downto 0);
out_d_0:  OUT std_logic_VECTOR(g_BITSIZE_0+g_BITSIZE_1-1 downto 0));
end component ;
component Multiplier_002_4_i1_16_i2_8 is
generic(
g_BITSIZE_0 :  integer := 16;
g_BITSIZE_1 :  integer := 8 );
port (
-- clk :  IN std_logic;
d_0 :  IN std_logic_VECTOR(g_BITSIZE_0-1 downto 0);
d_1 :  IN std_logic_VECTOR(g_BITSIZE_1-1 downto 0);
out_d_0:  OUT std_logic_VECTOR(g_BITSIZE_0+g_BITSIZE_1-1 downto 0));
end component ;
component Multiplier_002_4_i1_8_i2_8 is
generic(
g_BITSIZE_0 :  integer := 8;
g_BITSIZE_1 :  integer := 8 );
port (
-- clk :  IN std_logic;
d_0 :  IN std_logic_VECTOR(g_BITSIZE_0-1 downto 0);
d_1 :  IN std_logic_VECTOR(g_BITSIZE_1-1 downto 0);
out_d_0:  OUT std_logic_VECTOR(g_BITSIZE_0+g_BITSIZE_1-1 downto 0));
end component ;
component mux2_1 is
generic (depth :  integer := 8 );
port ( in0, in1 :  in std_logic_vector (depth-1 downto 0);
s :  in std_logic_vector (0 downto 0);
o :  out std_logic_vector (depth-1 downto 0)
);
end component;
```

Figure 7.14. Differential Equation Datapath VHDL File 3.

```
component mux4_1 is
generic (depth :  integer := 8 );
port (in0,in1, in2, in3 :  in std_logic_vector (depth-1 downto 0);
s :  in std_logic_vector (1 downto 0);
o :  out std_logic_vector (depth-1 downto 0));
end component;
component RCAAdder_001_8_i1_8_i2_8 is
generic (g_BITSIZE: integer := 8 );
port (d_0 :  in std_logic_vector(g_BITSIZE-1 downto 0);
d_1 :  in std_logic_vector(g_BITSIZE-1 downto 0);
out_d_0 :  out std_logic_vector(g_BITSIZE downto 0));
end component;
component RCASubtractor_123_110_i1_32_i2_32 is
generic (g_BITSIZE: integer := 32 );
port ( d_0 :  in std_logic_vector(g_BITSIZE-1 downto 0);
d_1 :  in std_logic_vector(g_BITSIZE-1 downto 0);
out_d_0 :  out std_logic_vector(g_BITSIZE downto 0));
end component;
component RCASubtractor_123_110_i1_33_i2_33 is
generic (g_BITSIZE: integer := 33 );
port ( d_0 :  in std_logic_vector(g_BITSIZE-1 downto 0);
d_1 :  in std_logic_vector(g_BITSIZE-1 downto 0);
out_d_0 :  out std_logic_vector(g_BITSIZE downto 0));
end component;
component RCASubtractor_123_110_i1_9_i2_9 is
generic (g_BITSIZE: integer := 9 );
port ( d_0 :  in std_logic_vector(g_BITSIZE-1 downto 0);
d_1 :  in std_logic_vector(g_BITSIZE-1 downto 0);
out_d_0 :  out std_logic_vector(g_BITSIZE downto 0));
end component;
component Reg is
generic ( len :  integer := 16 );
Port ( clk :  in STD_LOGIC;
i1 :  in STD_LOGIC_VECTOR (len-1 downto 0);
en :  in STD_LOGIC;
rst :  in STD_LOGIC;
o1 :  out STD_LOGIC_VECTOR (len-1 downto 0));
end component;
signal Multiplier0_d0 :  std_logic_vector(15 downto 0);
signal Multiplier0_r2 :  std_logic_vector(15 downto 0);
signal Multiplier0_d0 :  std_logic_vector(15 downto 0);
signal Multiplier0_r2 :  std_logic_vector(15 downto 0);
```

Figure 7.15. Differential Equation Datapath VHDL File 4.

```
 signal Multiplier0_r3 :  std_logic_vector(15 downto 0);
 signal Multiplier0_r4 :  std_logic_vector(15 downto 0);
 signal RCAAdder0_d0 :  std_logic_vector(8 downto 0);
 signal Multiplier1_d0 :  std_logic_vector(31 downto 0);
 signal Multiplier2_d0 :  std_logic_vector(23 downto 0);
 signal Multiplier2_r2 :  std_logic_vector(23 downto 0);
 signal RCASubtractor0_d0 :  std_logic_vector(9 downto 0);
 signal RCASubtractor1_d0 :  std_logic_vector(32 downto 0);
 signal RCASubtractor2_d0 :  std_logic_vector(33 downto 0);
 signal d_0_r1 :  std_logic_vector(7 downto 0);
 signal d_1_r1 :  std_logic_vector(7 downto 0);
 signal d_2_r1 :  std_logic_vector(7 downto 0);
 signal d_3_r1 :  std_logic_vector(7 downto 0);
 signal d_4_r1 :  std_logic_vector(7 downto 0);
 signal d_5_r1 :  std_logic_vector(7 downto 0);
 signal RCAAdder0_r1 :  std_logic_vector(8 downto 0);
 signal Multiplier1_r1 :  std_logic_vector(31 downto 0);
 signal RCASubtractor0_r1 :  std_logic_vector(9 downto 0);
 signal mux0_d0 :  std_logic_vector(7 downto 0);
 signal RCASubtractor2_r1 :  std_logic_vector(33 downto 0);
 signal mux1_d0 :  std_logic_vector(7 downto 0);
 signal Multiplier0_r1 :  std_logic_vector(15 downto 0);
 signal mux2_d0 :  std_logic_vector(15 downto 0);
 signal mux3_d0 :  std_logic_vector(7 downto 0);
 signal Multiplier2_r1 :  std_logic_vector(23 downto 0);
 signal d_5_r1_e9 :  std_logic_vector(8 downto 0);
 signal d_2_r1_e32 :  std_logic_vector(31 downto 0);
 signal Multiplier2_r1_e33 :  std_logic_vector(32 downto 0);
 begin
 ER19 :  d_5_r1_e9<="0"&d_5_r1;
 ER8 :  d_2_r1_e32<="00000000000000000000000"&d_2_r1;
 ER40 :  Multiplier2_r1_e33<="000000000"&Multiplier2_r1;
 SE61 :  Multiplier_002_4_i1_8_i2_8 port map ( d_0 =>mux0_d0 , d_1 =>mux1_d0 , out_d_0
=>Multiplier0_d0 );
 SE60 :  Reg generic map (16) port map ( clk => clk , i1 =>Multiplier0_d0 , en =>en_0 , rst =>rst
, o1 =>Multiplier0_r2 );
 SE62 :  Reg generic map (16) port map ( clk =>clk , i1 =>Multiplier0_d0 , en =>en_1 , rst =>rst
, o1 =>Multiplier0_r3 );
 SE63 :  Reg generic map (16) port map ( clk =>clk , i1 =>Multiplier0_d0 , en =>en_2 , rst =>rst
, o1 =>Multiplier0_r4 );
 SE64 :  RCAAdder_001_8_i1_8_i2_8 port map ( d_0 =>d_1_r1 , d_1 =>d_3_r1 , out_d_0 =>RCAAdder0_d0 );
 SE65 :  Multiplier_002_4_i1_16_i2_16 port map ( d_1 =>Multiplier0_r2 , d_0 =>Multiplier0_r1 ,
out_d_0 =>Multiplier1_d0 );
 SE66 :  Multiplier_002_4_i1_16_i2_8 port map ( d_0 => mux2_d0 , d_1 =>mux3_d0 , out_d_0
=>Multiplier2_d0 );
```

Figure 7.16. Differential Equation Datapath VHDL File 5.

```
 SE67 :  Reg generic map (24) port map ( clk =>clk , i1 =>Multiplier2_d0 , en =>en_3 , rst =>rst
, o1 =>Multiplier2_r2 );
 SE68 :  RCASubtractor_123_110_i1_9_i2_9 port map ( d_0 =>RCAAdder0_d0 , d_1 =>d_5_r1_e9 , out_d_0
=>RCASubtractor0_d0 );
 SE69 :  RCASubtractor_123_110_i1_32_i2_32 port map ( d_1 =>d_2_r1_e32 , d_0 =>Multiplier1_r1 ,
out_d_0 =>RCASubtractor1_d0 );
 SE70 :  RCASubtractor_123_110_i1_33_i2_33 port map ( d_1 =>RCASubtractor1_d0 , d_0
=>Multiplier2_r1_e33 , out_d_0 =>RCASubtractor2_d0 );
 reg0 :  Reg generic map (8) port map ( clk =>clk , i1 =>d_0 , en =>en_4 , rst =>rst , o1
=>d_0_r1 );
 reg1 :  Reg generic map (8) port map ( clk =>clk , i1 =>d_1 , en =>en_5 , rst =>rst , o1
=>d_1_r1 );
 reg2 :  Reg generic map (8) port map ( clk =>clk , i1 =>d_2 , en =>en_6 , rst =>rst , o1
=>d_2_r1 );
 reg3 :  Reg generic map (8) port map ( clk =>clk , i1 =>d_3 , en =>en_7 , rst =>rst , o1
=>d_3_r1 );
 reg4 :  Reg generic map (8) port map ( clk =>clk , i1 =>d_4 , en =>en_8 , rst =>rst , o1
=>d_4_r1 );
 reg5 :  Reg generic map (8) port map ( clk =>clk , i1 =>d_5 , en =>en_9 , rst =>rst , o1
=>d_5_r1 );
 reg6 :  Reg generic map (9) port map ( clk =>clk , i1 =>RCAAdder0_d0 , en =>en_10 , rst =>rst ,
o1 =>RCAAdder0_r1 );
 reg7 :  Reg generic map (32) port map ( clk =>clk , i1 =>Multiplier1_d0 , en =>en_11 , rst
=>rst , o1 =>Multiplier1_r1 );
 reg8 :  Reg generic map (10) port map ( clk =>clk , i1 =>RCASubtractor0_d0 , en =>en_12 , rst
=>rst , o1 =>RCASubtractor0_r1 );
 mux0 :  mux4_1 generic map (8) port map ( in0 =>d_0_r1 , in1 =>d_2_r1 , in2 =>d_0_r1 , in3
=>d_2_r1 , s =>sel_0 , o =>mux0_d0 );
 reg10 :  Reg generic map (34) port map ( clk =>clk , i1 =>RCASubtractor2_d0 , en =>en_13 , rst
=>rst , o1 =>RCASubtractor2_r1 );
 mux1 :  mux4_1 generic map (8) port map ( in0 =>d_1_r1 , in1 =>d_3_r1 , in2 =>d_4_r1 , in3
=>d_3_r1 , s =>sel_1 , o =>mux1_d0 );
 reg11 :  Reg generic map (16) port map ( clk =>clk , i1 =>Multiplier0_d0 , en =>en_14 , rst
=>rst , o1 =>Multiplier0_r1 );
 mux2 :  mux2_1 generic map (16) port map ( in0 =>Multiplier0_r3 , in1 =>Multiplier0_r4 , s
=>sel_2 , o =>mux2_d0 );
 mux3 :  mux2_1 generic map (8) port map ( in0 =>d_3_r1 , in1 =>d_4_r1 , s =>sel_3 , o =>mux3_d0
);
 reg12 :  Reg generic map (24) port map ( clk =>clk , i1 =>Multiplier2_d0 , en =>en_15 , rst
=>rst , o1 =>Multiplier2_r1 );
 out_d_0 <= RCASubtractor2_r1;
 out_d_1 <= RCAAdder0_r1;
 out_d_2 <= Multiplier2_r2;
 out_d_3 <= RCASubtractor0_r1;
 end Architecture;
```

Figure 7.17. Differential Equation Datapath VHDL File 6.

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
library UNISIM;
use UNISIM.VComponents.all;
entity
Controller is
port (
clk:  in std_logic;
rst:  in std_logic ;
en_0 :  out std_logic;
en_1 :  out std_logic;
en_2 :  out std_logic;
en_3 :  out std_logic;
en_4 :  out std_logic;
en_5 :  out std_logic;
en_6 :  out std_logic;
en_7 :  out std_logic;
en_8 :  out std_logic;
en_9 :  out std_logic;
en_10 :  out std_logic;
en_11 :  out std_logic;
en_12 :  out std_logic;
sel_0 :  out std_logic_vector(1 downto 0);
en_13 :  out std_logic;
sel_1 :  out std_logic_vector(1 downto 0);
en_14 :  out std_logic;
sel_2 :  out std_logic_vector(0 downto 0);
sel_3 :  out std_logic_vector(0 downto 0);
en_15 :  out std_logic);
end entity;
architecture Behavioral of Controller is
signal count :  std_logic_VECTOR(2 downto 0);
signal pre_count :  std_logic_VECTOR(2 downto 0);
begin
counter :  process(clk,rst)
begin
if ( rst = '1') then
pre_count <= ''000";
elsif (clk='1' and clk'event ) then
if ( pre_count <''110" ) then
pre_count <= pre_count + 1;
else
pre_count <= ''000";
```

Figure 7.18. Differential Equation Controller VHDL File 1.

```
 end if;
 end if;
 end process;
 count <= pre_count;
enable0 :  process(count)
 begin
 CASE count IS
 WHEN ‘‘010" =>en_0 <= ‘1’;
 WHEN OTHERS =>en_0 <= ‘0’;
 END CASE;
end process;
enable1 :  process(count)
 begin
 CASE count IS
 WHEN ‘‘011" =>en_1 <= ‘1’;
 WHEN OTHERS =>en_1 <= ‘0’;
 END CASE;
end process;
enable2 :  process(count)
 begin
 CASE count IS
 WHEN ‘‘100" =>en_2 <= ‘1’;
 WHEN OTHERS =>en_2 <= ‘0’;
 END CASE;
end process;
enable3 :  process(count)
 begin
 CASE count IS
 WHEN ‘‘101" =>en_3 <= ‘1’;
 WHEN OTHERS =>en_3 <= ‘0’;
 END CASE;
end process;
enable4 :  process(count)
 begin
 CASE count IS
 WHEN ‘‘000" =>en_4 <= ‘1’;
 WHEN OTHERS =>en_4 <= ‘0’;
 END CASE;
end process;
enable5 :  process(count)
 begin
 CASE count IS
 WHEN ‘‘000" =>en_5 <= ‘1’;
 WHEN OTHERS =>en_5 <= ‘0’;
 END CASE;
```

Figure 7.19. Differential Equation Controller VHDL File 2.

```
end process;
enable6 :  process(count)
begin
CASE count IS
WHEN ‘‘000" =>en_6 <= ’1’;
WHEN OTHERS =>en_6 <= ‘0’;
END CASE;
end process;
enable7 :  process(count)
begin
CASE count IS
WHEN ‘‘000" =>en_7 <= ‘1’;
WHEN OTHERS =>en_7 <= ‘0’;
END CASE;
end process;
enable8 :  process(count)
begin
CASE count IS
WHEN ‘‘000" =>en_8 <= ‘1’;
WHEN OTHERS =>en_8 <= ‘0’;
END CASE;
end process;
enable9 :  process(count)
begin
CASE count IS
WHEN ‘‘000" =>en_9 <= ‘1’;
WHEN OTHERS =>en_9 <= ‘0’;
END CASE;
end process;
enable10 :  process(count)
begin
CASE count IS
WHEN ‘‘001" =>en_10 <= ‘1’;
WHEN OTHERS =>en_10 <= ‘0’;
END CASE;
end process;
enable11 :  process(count)
begin
CASE count IS
WHEN ‘‘100" =>en_11 <= ‘1’;
WHEN OTHERS =>en_11 <= ‘0’;
END CASE;
end process;
```

Figure 7.20. Differential Equation Controller VHDL File 3.

```
enable12 :  process(count)
begin
CASE count IS
WHEN ‘‘001" =>en_12 <= ‘1’;
WHEN OTHERS =>en_12 <= ‘0’;
END CASE;
end process;
enable13 :  process(count)
begin
CASE count IS
WHEN ‘‘101" =>en_13 <= ‘1’;
WHEN OTHERS =>en_13 <= ‘0’;
END CASE;
end process;
enable14 :  process(count)
begin
CASE count IS
WHEN ‘‘001" =>en_14 <= ‘1’;
WHEN OTHERS =>en_14 <= ‘0’;
END CASE;
end process;
enable15 :  process(count)
begin
CASE count IS
WHEN ‘‘100" =>en_15 <= ‘1’;
WHEN OTHERS =>en_15 <= ‘0’;
END CASE;
end process;
enselect0 :  process(count)
begin
CASE count is
WHEN ‘‘001" =>sel_0 <= ‘‘00";
WHEN ‘‘010" =>sel_0 <= ‘‘01";
WHEN ‘‘011" =>sel_0 <= ‘‘10";
WHEN ‘‘100" =>sel_0 <= ‘‘11";
WHEN OTHERS =>sel_0 <= ‘‘00";
END CASE;
end process;
```

Figure 7.21. Differential Equation Controller VHDL File 4.

```
enselect1 :  process(count)
begin
CASE count is
WHEN ‘‘001" =>sel_1 <= ‘‘00";
WHEN ‘‘010" =>sel_1 <= ‘‘01";
WHEN ‘‘011" =>sel_1 <= ‘‘10";
WHEN ‘‘100" =>sel_1 <= ‘‘11";
WHEN OTHERS =>sel_1 <= ‘‘00";
END CASE;
end process;
enselect2 :  process(count)
begin
CASE count is
WHEN ‘‘100" =>sel_2 <= ‘‘0";
WHEN ‘‘101" =>sel_2 <= ‘‘1";
WHEN OTHERS =>sel_2 <= ‘‘0";
END CASE;
end process;
enselect3 :  process(count)
begin
CASE count is
WHEN ‘‘100" =>sel_3 <= ‘‘0";
WHEN ‘‘101" =>sel_3 <= ‘‘1";
WHEN OTHERS =>sel_3 <= ‘‘0";
END CASE;
end process;
end architecture;
```

Figure 7.22. Differential Equation Controller VHDL File 5.

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
library UNISIM;
use UNISIM.VComponents.all;
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
library UNISIM;
use UNISIM.VComponents.all;
entity TopLevel is
port (
clk:  in std_logic;
rst:  in std_logic;
d_0:  in std_logic_vector(7 downto 0);
d_1:  in std_logic_vector(7 downto 0);
d_2:  in std_logic_vector(7 downto 0);
d_3:  in std_logic_vector(7 downto 0);
d_4:  in std_logic_vector(7 downto 0);
d_5:  in std_logic_vector(7 downto 0);
out_d_0:  out std_logic_vector(33 downto 0);
out_d_1:  out std_logic_vector(8 downto 0);
out_d_2:  out std_logic_vector(23 downto 0);
out_d_3:  out std_logic_vector(9 downto 0));
end entity;
architecture Behavioral of TopLevel is
component
diff_eq is
port (
clk:  in std_logic;
rst:  in std_logic;
en_0:  in std_logic;
en_1:  in std_logic;
en_2:  in std_logic;
en_3:  in std_logic;
en_4:  in std_logic;
en_5:  in std_logic;
en_6:  in std_logic;
en_7:  in std_logic;
en_8:  in std_logic;
en_9:  in std_logic;
en_10:  in std_logic;
en_11:  in std_logic;
en_12:  in std_logic;
sel_0:  in std_logic_vector(1 downto 0);
en_13:  in std_logic;
```

Figure 7.23. Differential Equation Top Level VHDL File 1.

```
sel_1:  in std_logic_vector(1 downto 0);
en_14:  in std_logic;
sel_2:  in std_logic_vector(0 downto 0);
sel_3:  in std_logic_vector(0 downto 0);
en_15:  in std_logic;
d_0:  in std_logic_vector(7 downto 0);
d_1:  in std_logic_vector(7 downto 0);
d_2:  in std_logic_vector(7 downto 0);
d_3:  in std_logic_vector(7 downto 0);
d_4:  in std_logic_vector(7 downto 0);
d_5:  in std_logic_vector(7 downto 0);
out_d_0:  out std_logic_vector(33 downto 0);
out_d_1:  out std_logic_vector(8 downto 0);
out_d_2:  out std_logic_vector(23 downto 0);
out_d_3:  out std_logic_vector(9 downto 0));
end component;
component
Controller is
port (
clk:  in std_logic;
rst:  in std_logic ;
en_0 :  out std_logic;
en_1 :  out std_logic;
en_2 :  out std_logic;
en_3 :  out std_logic;
en_4 :  out std_logic;
en_5 :  out std_logic;
en_6 :  out std_logic;
en_7 :  out std_logic;
en_8 :  out std_logic;
en_9 :  out std_logic;
en_10 :  out std_logic;
en_11 :  out std_logic;
en_12 :  out std_logic;
sel_0 :  out std_logic_vector(1 downto 0);
en_13 :  out std_logic;
sel_1 :  out std_logic_vector(1 downto 0);
en_14 :  out std_logic;
sel_2 :  out std_logic_vector(0 downto 0);
```

Figure 7.24. Differential Equation Top Level VHDL File 2.

```
sel_3 :  out std_logic_vector(0 downto 0);
en_15 :  out std_logic);

end component;
signal en_0 :  std_logic;
signal en_1 :  std_logic;
signal en_2 :  std_logic;
signal en_3 :  std_logic;
signal en_4 :  std_logic;
signal en_5 :  std_logic;
signal en_6 :  std_logic;
signal en_7 :  std_logic;
signal en_8 :  std_logic;
signal en_9 :  std_logic;
signal en_10 :  std_logic;
signal en_11 :  std_logic;
signal en_12 :  std_logic;
signal en_13 :  std_logic;
signal en_14 :  std_logic;
signal en_15 :  std_logic;
signal sel_0 :  std_logic_vector( 1 downto 0);
signal sel_1 :  std_logic_vector( 1 downto 0);
signal sel_2 :  std_logic_vector( 0 downto 0);
signal sel_3 :  std_logic_vector( 0 downto 0);
begin
inst_Controller :  Controller port map ( clk =>clk, rst =>rst, sel_0 =>sel_0, sel_1 =>sel_1,
sel_2 =>sel_2, sel_3 =>sel_3, en_0 =>en_0, en_1 =>en_1, en_2 =>en_2, en_3 =>en_3, en_4 =>en_4,
en_5 =>en_5, en_6 =>en_6, en_7 =>en_7, en_8 =>en_8, en_9 =>en_9, en_10 =>en_10, en_11 =>en_11,
en_12 =>en_12, en_13 =>en_13, en_14 =>en_14, en_15 =>en_15);
inst_Datapath :  diff_eq port map ( clk =>clk, rst =>rst , en_0 =>en_0, en_1 =>en_1, en_2
=>en_2, en_3 =>en_3, en_4 =>en_4, en_5 =>en_5, en_6 =>en_6, en_7 =>en_7, en_8 =>en_8, en_9 =>en_9,
en_10 =>en_10, en_11 =>en_11, en_12 =>en_12, en_13 =>en_13, en_14 =>en_14, en_15 =>en_15, sel_0
=>sel_0, sel_1 =>sel_1, sel_2 =>sel_2, sel_3 =>sel_3, d_0 =>d_0, d_1 =>d_1, d_2 =>d_2, d_3 =>d_3,
d_4 =>d_4, d_5 =>d_5, out_d_0 =>out_d_0, out_d_1 =>out_d_1, out_d_2 =>out_d_2, out_d_3 =>out_d_3);
end Architecture;
```

Figure 7.25. Differential Equation Top Level VHDL File 3.

8. TEST RESULTS OF RESCHEDULING AND OPTIMIZED RTL GENERATION

For testing Rescheduling and Optimized RTL Generation modules, the differential equation, the autoregression (AR) filter and Infinite Impulse Response (IIR) filter applications are used. The application graphs are shown in Figures 8.2 to 8.4 respectively. In the figures, the name of vertices beginning with "m" corresponds to multiplier, with "a" corresponds to adder and with "s" corresponds to subtractor. The numbers after m, s and a corresponds to the name of the vertices. For instance, 'm62' vertex in the Figure 8.2 is multiplier vertex with the name of 'SE62'.

These applications are rescheduled using different optimization models as input as described in Section 6 and Optimized RTL VHDL files are generated as described in Section 7. These applications are tested for 8-bits and 16-bits where all the inputs of the graph have 8-bit or 16-bit wordlength. The VHDL files are synthesized, Placed & Routed in Xilinx Spartan-3 FPGA using Xilinx ISE 12.4 [1]. During Placing & Routing, "Timing Performance" is selected as design goal, and "Performance without IOB packing" is selected as design strategy. The Input/Output buffers are removed during synthesis by using the synthesis process properties. The clock period that is obtained during rescheduling is given as clock period constraint to datapath. Multicycle operations are also defined in UCF file. An example UCF File is given in Figure 8.1. Here, output signals of the registers are grouped and multicycle clock constraints are defined from register to registers. The delay, and area are taken from the Place and Route report of the Xilinx ISE. The clock period extracted during rescheduling is compared with the clock period in the Place and Route report. The estimated area during optimized RTL generation is also compared with the area in the Place and Route report. These delay and area values are also compared with the delay and area value found in the optimization module. The delay is measured in nanoseconds and area is measured in number of slices.

```
# define clock constraint
NET ‘‘clk" TNM_NET = "CLK1";
TIMESPEC "TS_CLK1" = PERIOD "CLK1" 3.715 ns HIGH 50%;
# group all the output signals of register0
INST ‘‘reg0/o1_0" TNM = reg0;
INST ‘‘reg0/o1_1" TNM = reg0;
INST ‘‘reg0/o1_2" TNM = reg0;
# group all the output signals of register1
INST ‘‘reg1/o1_0" TNM = reg1;
INST ‘‘reg1/o1_1" TNM = reg1;
INST ‘‘reg1/o1_2" TNM = reg1;
INST ‘‘reg1/o1_3" TNM = reg1;
# group all the output signals of register2
INST ‘‘reg2/o1_0" TNM = reg2;
INST ‘‘reg2/o1_1" TNM = reg2;
INST ‘‘reg2/o1_2" TNM = reg2;
# group all the output signals of register3
INST ‘‘reg3/o1_0" TNM = reg3;
INST ‘‘reg3/o1_1" TNM = reg3;
INST ‘‘reg3/o1_2" TNM = reg3;
# define multicycle register from reg0 to reg1 with constraint clk x 2
TIMESPEC TS_1 = FROM ‘‘reg0" TO ‘‘reg1" TS_CLK1 * 2;
# define multicycle register from reg2 to reg3 with constraint clk x 3
TIMESPEC TS_1 = FROM ‘‘reg2" TO ‘‘reg3" TS_CLK1 * 3;
```

Figure 8.1. An example UCF File.

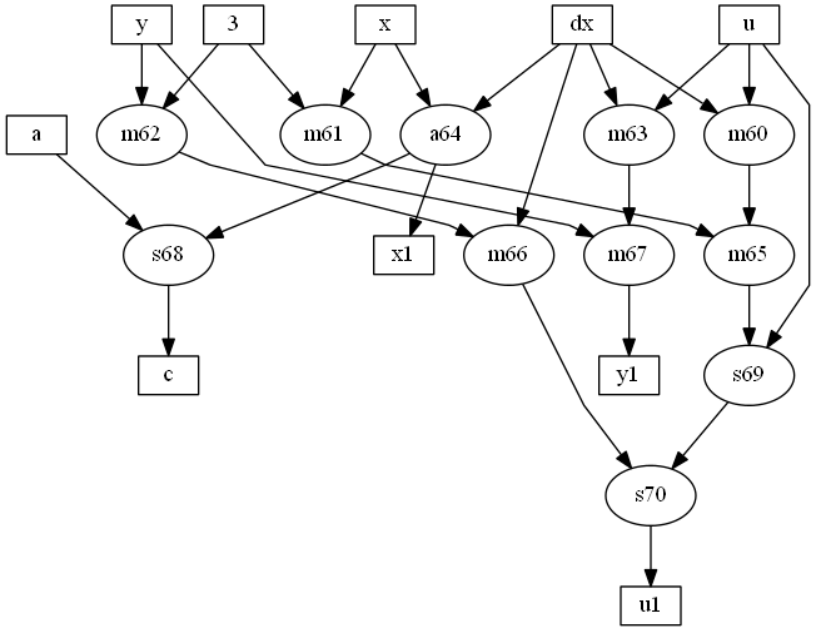

Figure 8.2. Differential Equation Graph for Optimized RTL Generation.

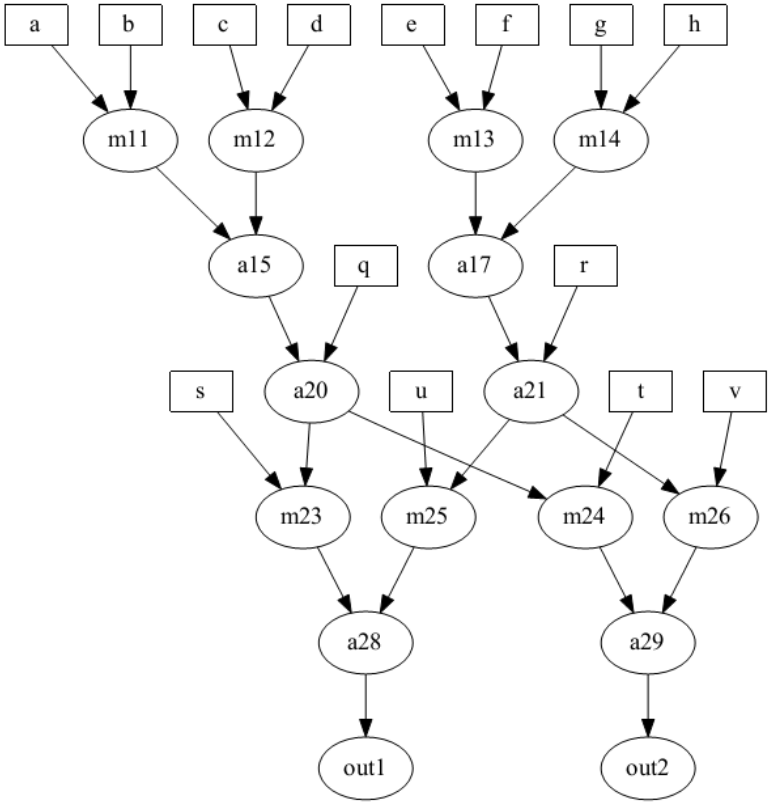

Figure 8.3. AR Filter Graph for Optimized RTL Generation.

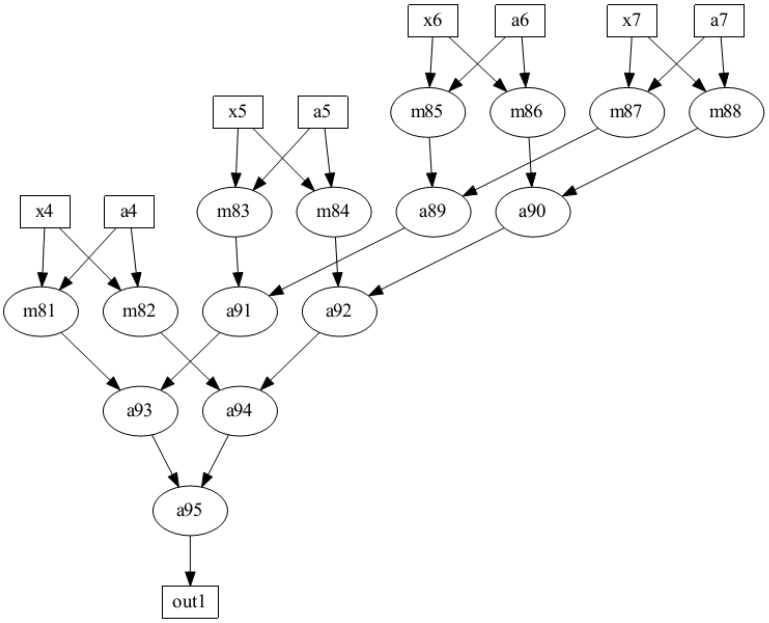

Figure 8.4. IIR Filter Graph for Optimized RTL Generation.

8.1. Differential Equation Application Results

8.1.1. Differential Equation application for 8-bits

The delay results of the 8-bit differential equation application are given in Table 8.1. Here, Optimization Latency and Optimization Area correspond to values found in optimization module. Rescheduling clock period corresponds to the extracted clock period in Rescheduling Module and number of control steps corresponds to the total number of clock steps. Total Rescheduling Latency corresponds to rescheduling clock period multiplied by number of control steps. PAR clock period is the clock period that is taken from the Xilinx Place and Route report. PAR total latency corresponds to PAR clock period multiplied by number of control steps. In the last column of Table 8.1, the estimation error of the rescheduling latency with respect to PAR Latency is given.

The area results are given in Table 8.2. The optimization area is the area found in the optimization module. Estimated operator area is the estimated area of the operators in the graph, estimated Mux area is the area of the multiplexers in the graph,

Table 8.1. Differential Equation Delay Results of Optimized RTL Generation for 8-bits.

Optimization Latency (ns)	**Optimization Area (n# of Slices)**	**Rescheduling Clock Period (ns)**	**Number of Control Steps**	**Rescheduling Total Latency (ns)**	**PAR Clock Period (ns)**	**PAR Total Latency (ns)**	**Latency Error of Rescheduling vs PAR (%)**
15.03	381	3.748	11	41.228	3.92	43.12	4.589
15.51	364	10.308	4	41.232	11.362	46.528	12.844
16.45	356	10.618	4	42.472	11.591	46.364	9.163
16.72	339	10.618	4	42.472	11.286	45.144	6.291
19.04	331	10.308	5	51.54	11.09	55.45	7.586
23.32	305	5.098	12	61.167	5.7	68.4	11.808
49.01	304	13.508	7	94.556	16.252	113.74	20.313

and Register overhead area is the overhead area caused by the registers in the datapath where the calculation of the register overhead area is described in Section 7.1.2.5. Total Estimated Area is the sum of operator area, multiplexer area and the register overhead area. PAR Area is the area value taken from the Place and Route report of Xilinx ISE. In the last column, the estimation error of the area with respect to Xilinx Place and route area is given.

Figure 8.5 and 8.6 show the delay and area comparision results of the optimization, estimation and Place & Route. The Golden RTL delay of 8-bit differential equation is 24.49 ns and the golden RTL area is 432 number of slices.

When the optimization delay is compared with the delay value found with rescheduling for 8-bit differential equation, the average ratio of the rescheduling delay with respect to optimization delay is 2.5. This overhead is caused by the registers. The average area overhead for 8-bit differential equation is 10%. This area overhead is calculated by comparing the area found in optimization module and the estimated area during optimized RTL generation.

Table 8.2. Differential Equation Area Results of Optimized RTL Generation for 8-bits.

Optimization Latency (ns)	**Optimization Area (n# of Slices)**	**Estimated Operator Area (n# of Slices)**	**Estimated Mux Area (n# of Slices)**	**Register Overhead Area (n# of Slices)**	**Total Estimated Area (n# of Slices)**	**PAR Area (n# of Slices)**	**Area Error of Estimated vs PAR (%)**
15.03	381	335	40	20	395	400	1.265
15.51	364	302	56	28	386	418	8.29
16.45	356	302	48	56	406	387	4.679
16.72	339	270	64	36	370	336	9.189
19.04	331	270	56	36	362	343	5.248
23.32	305	205	96	48	349	305	12.607
49.01	304	173	128	56	357	273	23.52

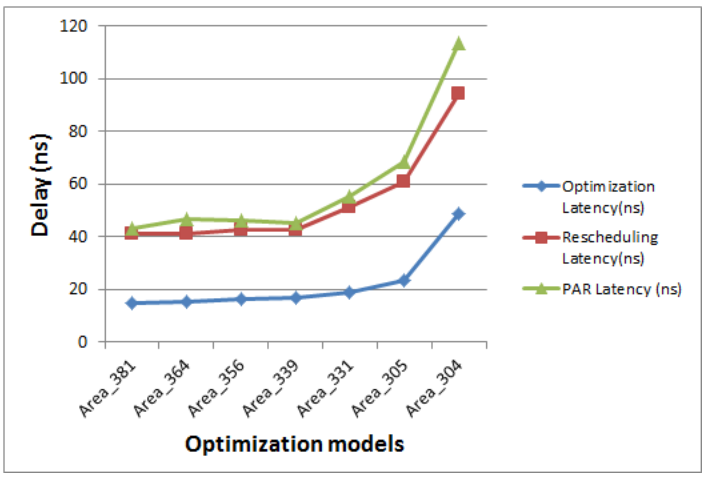

Figure 8.5. Delays of 8-bit Differential Equation for Area Values Obtained from Optimization Model.

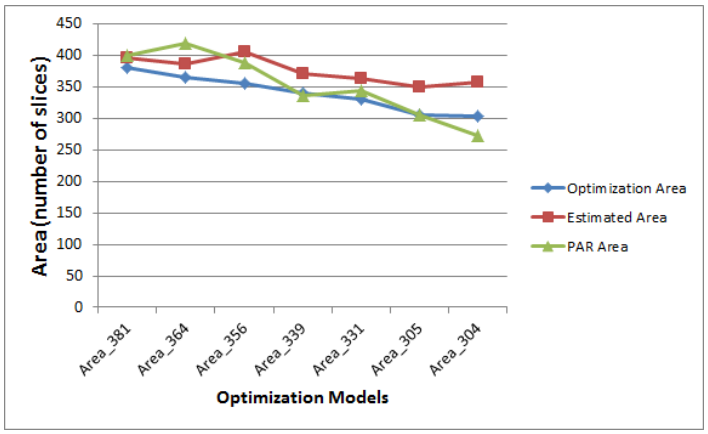

Figure 8.6. Areas of 8-bit Differential Equation for Area Values Obtained from Optimization Model.

8.1.2. Differential Equation application for 16-bits

The delay and area results of the 16-bit differential equation application are given in Table 8.3 and 8.3.

Figure 8.7 and 8.8 show the delay and area comparision results of the optimization, estimation and Place & Route. The Golden RTL delay of 16-bit differential equation is 37.157 ns and the golden RTL area is 1667 number of slices.

When the optimization delay is compared with the delay found with rescheduling for 16-bit differential equation, the average ratio of the rescheduling delay with respect to optimization delay is 1.23. This overhead is caused by the registers. The average area overhead for 16-bit differential equation is 5.4%. This area overhead is calculated by comparing the area found in optimization module and the estimated area during optimized RTL generation.

Table 8.3. Differential Equation Delay Results of Optimized RTL Generation for 16-bits.

Optimization Latency (ns)	**Optimization Area (n# of Slices)**	**Rescheduling Clock Period (ns)**	**Number of Control Steps**	**Rescheduling Total Latency (ns)**	**PAR Clock Period (ns)**	**PAR Total Latency (ns)**	**Latency Error of Rescheduling vs PAR (%)**
37.16	1501	6.938	7	48.566	6.938	48.566	0
37.637	1485	6.938	7	48.566	7	49	0.89
41.923	1270	18.018	3	54.054	17.996	53.988	0.122
43.714	1202	18.488	3	55.464	18.483	55.449	0.027
49.387	1170	6.938	9	62.442	7.3	65.7	5.217
68.135	939	18.808	4	75.232	18.792	75.168	0.085
121.32	871	19.108	7	133.756	19.5	136.5	2.051

Table 8.4. Differential Equation Area Results of Optimized RTL Generation for 16-bits.

Optimization Latency (ns)	**Optimization Area (n# of Slices)**	**Estimated Operator Area (n# of Slices)**	**Estimated Mux Area (n# of Slices)**	**Register Overhead Area (n# of Slices)**	**Total Estimated Area (n# of Slices)**	**PAR Area (n# of Slices)**	**Area Error of Estimated vs PAR (%)**
37.16	1501	1402	96	32	1530	1590	3.921
37.637	1485	1402	80	32	1514	1520	0.396
41.923	1270	1140	128	56	1324	1320	0.302
43.714	1202	1008	192	72	1272	1210	4.874
49.387	1170	1008	160	56	1224	1173	4.166
56.266	1102	875	224	88	1187	1052	11.373
68.135	939	745	192	48	985	921	6.497
121.32	871	613	256	112	981	909	7.339

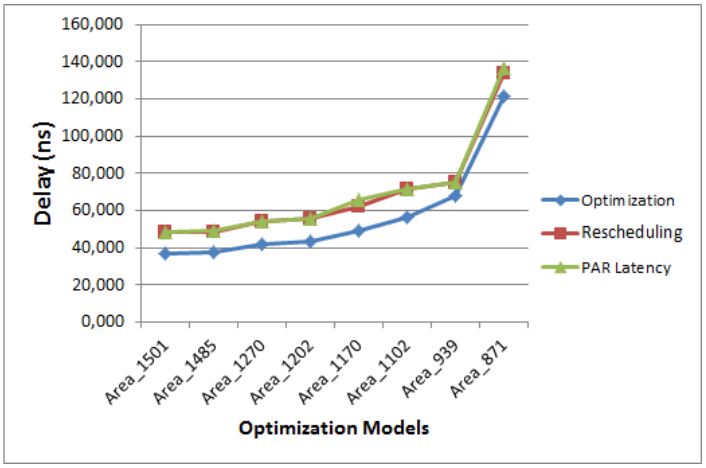

Figure 8.7. Delays of 16-bit Differential Equation for Area Values Obtained from Optimization Model.

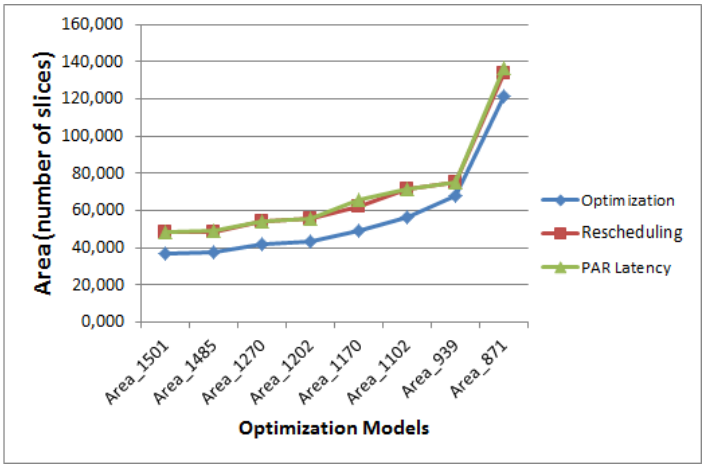

Figure 8.8. Areas of 16-bit Differential Equation for Area Values Obtained from Optimization Model.

Table 8.5. AR Filter Delay Results of Optimized RTL Generation for 8-bits.

Optimization Latency (ns)	**Optimization Area (n# of Slices)**	**Rescheduling Clock Period (ns)**	**Number of Control Steps**	**Rescheduling Total Latency (ns)**	**PAR Clock Period (ns)**	**PAR Total Latency (ns)**	**Latency Error of Rescheduling vs PAR (%)**
23.26	457	4.758	8	38.064	4.174	37.712	0.924
25.27	439	10.468	4	41.872	11.744	46.976	12.189
32.43	347	4.758	11	52.338	5	55	5.086
33.54	341	4.758	11	52.338	5.09	55.99	6.977
39	312	10.778	5	53.89	17.46	87.3	61.99
51.07	274	10.778	7	75.446	17.099	119.693	58.64
51.86	258	10.778	7	75.446	15.681	109.767	45.49
59.81	251	10.778	8	86.224	17.19	137.52	59.49
68.1	246	11.078	8	88.624	18.677	149.416	68.59

8.2. AR Filter Application Results

8.2.1. AR Filter application for 8-bits

The delay and area results of the 8-bit AR Filter application are given in Table 8.5 and 8.6.

Figure 8.9 and 8.10 show the delay and area comparision results of the optimization, estimation and Place & Route. The Golden RTL delay of 8-bit AR filter is 23.26 ns and the golden RTL area is 480 number of slices.

When the optimization delay is compared with the delay found with rescheduling for 8-bit AR Filter, the average ratio of the rescheduling delay with respect to optimization delay is 1.48. This overhead is caused by the registers. The average area overhead for 8-bit AR filter is 4.6%. This area overhead is calculated by comparing the area found in optimization module and the estimated area during optimized RTL generation.

Table 8.6. AR Filter Area Results of Optimized RTL Generation for 8-bits.

Optimization Latency (ns)	**Optimization Area (n# of Slices)**	**Estimated Operator Area (n# of Slices)**	**Estimated Mux Area (n# of Slices)**	**Register Overhead Area (n# of Slices)**	**Total Estimated Area (n# of Slices)**	**PAR Area (n# of Slices)**	**Area Error of Estimated vs PAR (%)**
23.26	457	480	0	0	480	534	11.25
25.27	439	384	78	24	486	503	3.497
32.43	347	302	68	34	404	424	4.95
33.54	341	270	94	42	406	424	4.43
39	312	394	104	42	540	563	4.259
51.07	274	324	68	47	439	460	4.78
51.86	258	291	84	110	485	579	19.38
59.81	251	259	84	84	427	404	5.38
68.1	246	259	104	63	426	441	.521

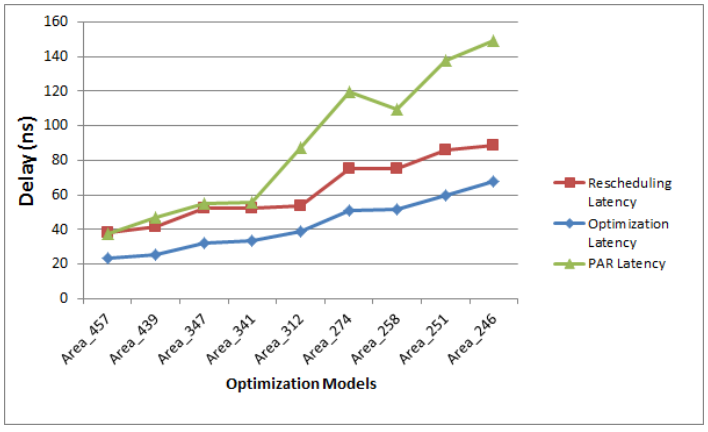

Figure 8.9. Delays of 8-bit AR Filter for Area Values Obtained from Optimization Model.

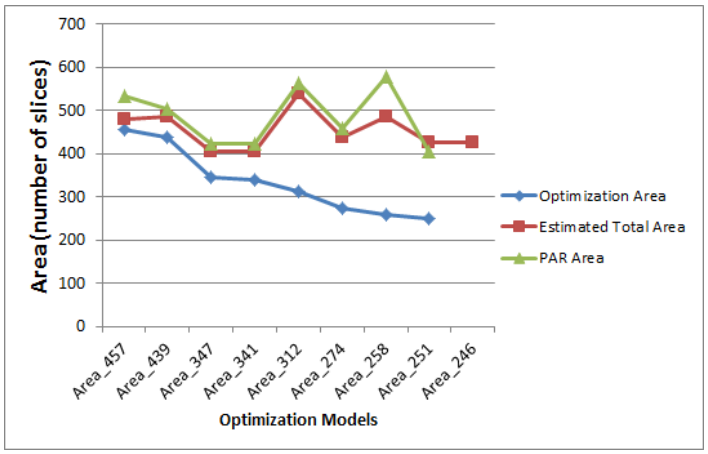

Figure 8.10. Areas of 8-bit AR Filter for Area Values Obtained from Optimization Model.

8.2.2. AR Filter application for 16-bits

The delay and area results of the 16-bit AR Filter application are given in Table 8.7 and 8.8.

Figure 8.11 and 8.12 show the delay and area comparision results of the optimization, estimation and Place & Route. The Golden RTL delay of 16-bit AR filter is 35.037 ns and the golden RTL area is 1757 number of slices.

When the optimization delay is compared with the delay found with rescheduling for 16-bit AR Filter, the average ratio of the rescheduling delay with respect to optimization delay is 1.4. This overhead is caused by the registers. The average area overhead for 16-bit AR filter is 3.6%. This area overhead is calculated by comparing the area found in optimization module and the estimated area during optimized RTL generation.

Table 8.7. AR Filter Delay Results of Optimized RTL Generation for 16-bits.

Optimization Latency (ns)	**Optimization Area (n# of Slices)**	**Rescheduling Clock Period (ns)**	**Number of Control Steps**	**Rescheduling Total Latency (ns)**	**PAR Clock Period (ns)**	**PAR Total Latency (ns)**	**Latency Error of Rescheduling vs PAR (%)**
35.04	1658	5.148	10	51.48	5.308	53.08	3.108
38.72	1412	6.098	9	54.882	6.098	54.882	0
49.99	1038	6.098	12	73.176	7.2	86.4	18.0714
50.47	1020	6.098	12	73.176	6.092	73.104	0.0983
59.18	1002	5.148	17	87.516	5.31	90.27	3.146
60.98	874	6.098	13	79.274	6.098	79.274	0
78.32	824	6.098	18	109.764	7.2	129.6	18.071
90.55	692	15.938	8	127.504	21.04	168.32	32.011
120.03	596	16.238	9	146.142	23.596	212.364	45.31346225

Table 8.8. AR Filter Area Results of Optimized RTL Generation for 16-bits.

Optimization Latency (ns)	**Optimization Area (n# of Slices)**	**Estimated Operator Area (n# of Slices)**	**Estimated Mux Area (n# of Slices)**	**Register Overhead Area (n# of Slices)**	**Total Estimated Area (n# of Slices)**	**PAR Area (n# of Slices)**	**Area Error of Estimated vs PAR (%)**
35.040	1658	1757	0	0	1757	1872	6.545
38.72	1412	1361	150	48	1559	1672	7.248
49.99	1038	1581	200	81	1862	1904	2.255
59.18	1002	936	164	82	1182	1106	6.429
60.98	874	672	200	114	986	943	4.361
78.32	824	1112	132	66	1310	1373	4.809
90.55	692	848	164	8	1020	1146	12.352
120.030	596	715	200	123	1038	926	10.789

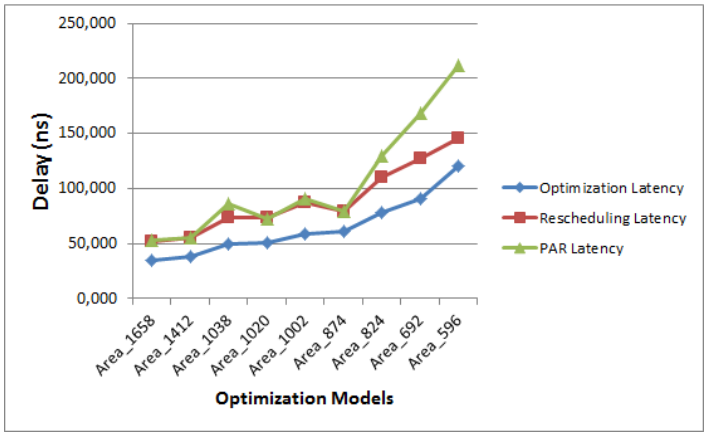

Figure 8.11. Delays of 16-bit AR Filter for Area Values Obtained from Optimization Model.

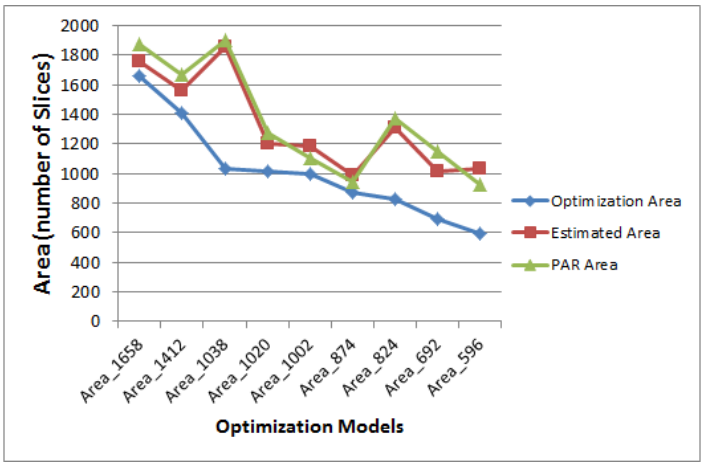

Figure 8.12. Areas of 16-bit AR Filter for Area Values Obtained from Optimization Model.

Table 8.9. IIR Filter Delay Results of Optimized RTL Generation for 8-bits.

Optimization Latency (ns)	**Optimization Area (n# of Slices)**	**Rescheduling Clock Period (ns)**	**Number of Control Steps**	**Rescheduling Total Latency (ns)**	**PAR Clock Period (ns)**	**PAR Total Latency (ns)**	**Latency Error of Rescheduling vs PAR (%)**
12.46	296	4.368	8	34.944	4.362	34.896	0.13
13.42	262	9.408	4	37.632	11.851	47.404	25.9
15.84	245	9.408	5	47.04	11.489	57.445	22.1
18	196	9.718	6	58.308	11.663	69.978	20
33.24	163	8.24	10	82.4	11.745	117.45	42.5

8.3. IIR Filter Application Results

8.3.1. IIR Filter Applications for 8-bits

The delay and area results of the 8-bit IIR Filter application are given in Table 8.9 and 8.10.

Figure 8.9 and 8.14 show the delay and area comparision results of the optimization, estimation and Place & Route. The Golden RTL delay of 8-bit IIR filter is 17.195 ns and the golden RTL area is 317 number of slices.

When the optimization delay is compared with the delay found with rescheduling for 8-bit IIR Filter, the average ratio of the rescheduling delay with respect to optimization delay is 2.85. This overhead is caused by the registers. The average area overhead for 8-bit IIR Filter is 13%. This area overhead is calculated by comparing the area found in optimization module and the estimated area during optimized RTL generation.

Table 8.10. IIR Filter Area Results of Optimized RTL Generation for 8-bits.

Optimization Latency (ns)	**Optimization Area (n# of Slices)**	**Estimated Operator Area (n# of Slices)**	**Estimated Mux Area (n# of Slices)**	**Register Overhead Area (n# of Slices)**	**Total Estimated Area (n# of Slices)**	**PAR Area (n# of Slices)**	**Area Error of Estimated vs PAR (%)**
12.46	296	253	32	16	301	326	8.30
13.42	262	189	64	32	285	303	6.31
15.84	245	157.2	64	40	261.2	250	4.28
18	196	125	64	48	237	244	2.95
33.240	163	92.97	64	56	212.97	241	13.16

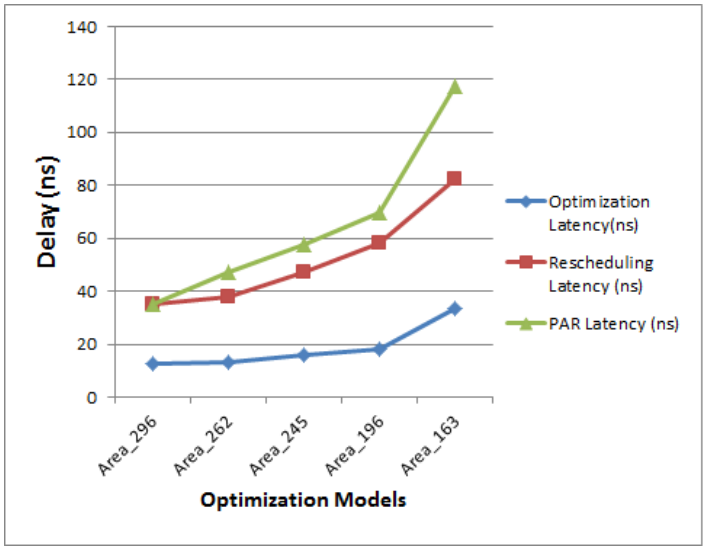

Figure 8.13. Delays of 8-bit IIR Filter for Area Values Obtained from Optimization Model.

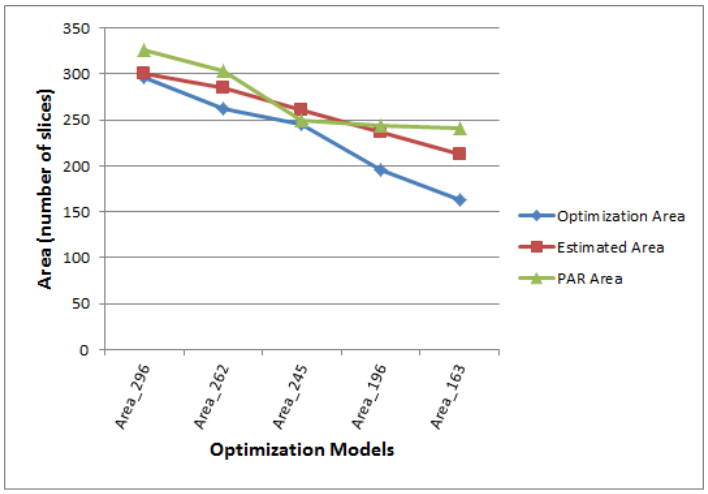

Figure 8.14. Areas of 8-bit IIR Filter for Area Values Obtained from Optimization Model.

8.3.2. IIR Filter Application for 16-bits

The delay and area results of the 16-bit IIR Filter application are given in Table 8.11 and 8.12.

Figure 8.11 and 8.16 show the delay and area comparision results of the optimization, estimation and Place & Route. The Golden RTL delay of 8-bit IIR filter is 24.25 ns and the golden RTL area is 1174 number of slices.

When the optimization delay is compared with the delay found with rescheduling for 16-bit IIR Filter, the average ratio of the rescheduling delay with respect to optimization delay is 2.14. This overhead is caused by the registers. The average area overhead for 16-bit IIR Filter is 11%. This area overhead is calculated by comparing the area found in optimization module and the estimated area during optimized RTL generation.

Table 8.11. IIR Filter Delay Results of Optimized RTL Generation for 16-bits.

Optimization Latency (ns)	**Optimization Area (n# of Slices)**	**Rescheduling Clock Period (ns)**	**Number of Control Steps**	**Rescheduling Total Latency (ns)**	**PAR Clock Period (ns)**	**PAR Total Latency (ns)**	**Latency Error of Rescheduling vs PAR (%)**
19.070	982	5.258	8	42.064	5.42	43.36	3.081
23.07	780	5.258	9	47.322	5.374	48.366	2.206
27.03	679	5.258	11	57.838	5.43	59.73	3.271
33.73	514	5.258	14	73.612	5.7	79.8	8.406
62.93	381	13.508	10	135.08	15.192	151.92	12.466

Table 8.12. IIR Filter Area Results of Optimized RTL Generation for 16-bits.

Optimization Latency (ns)	**Optimization Area (n# of Slices)**	**Estimated Operator Area (n# of Slices)**	**Estimated Mux Area (n# of Slices)**	**Register Overhead Area (n# of Slices)**	**Total Estimated Area (n# of Slices)**	**PAR Area (n# of Slices)**	**Area Error of Estimated vs PAR (%)**
19.070	982	910	64	32	1006	1118	11.133
23.07	780	645	64	64	773	878	13.583
27.03	679	513.47	160	80	753.47	798	5.909
33.73	514	381.24	128	96	605.24	629	3.925
62.93	381	249	112	112	473	478	1.057

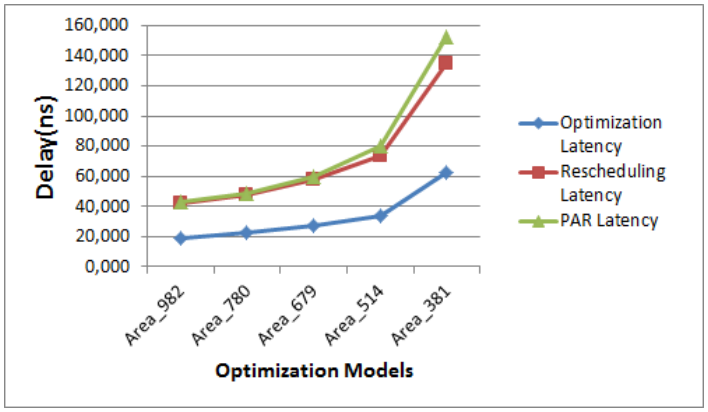

Figure 8.15. Delays of 8-bit IIR Filter for Area Values Obtained from Optimization Model.

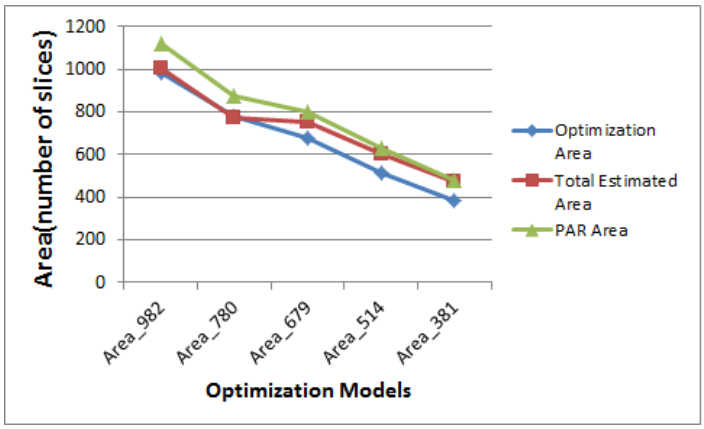

Figure 8.16. Areas of 8-bit IIR Filter for Area Values Obtained from Optimization Model.

8.4. Conclusion

Since the optimization model does not contain register information, there is overhead in delay and area when the delay and area found in optimization block is compared with the delay and area found in the Rescheduling Module or with the delay and area value found with Place& Route of Xilinx. Generally, the latency and area trends of the optimization module are similar to Estimated or Place & Route delay and area. However, there are some peaks in the Estimated and Place& Route area and delay especially for AR filter as can be seen in Figures 8.9, 8.10, 8.7, 8.7. These peaks are also caused by the registers which are not taken into account in the Optimization Module.

When the estimated delay in rescheduling module is compared with the delay found with Place & Route, the average error of delay is measured as 14.46% for the applied graphs. The estimation error increases when the operator sharing increases in the optimization model. According to the optimized RTL generation methodology, the number of registers at the output of the shared operator is the count of how many times the operator is shared. So, as the sharing increases, the registers increase which lead to increase routing delay caused by the registers. When the estimated area is compared with the Place & Route area, the average error is found 6.88 % for the application graphs. The error in area is lower because, the register area is calculated and added during area estimation.

9. CONCLUSIONS AND FUTURE WORK

In this study, an RTL generator tool for the RH(+) HLS tool has been developed and implemented. The RH(+) framework takes the input application which is described with an high level language such as LRH(+) or C converts the application to the CDFG representation. RTL generator tool takes this CDFG representation as input and generates the VHDL files of the RTL as output. This tool has the capabilities of Golden RTL generation and Optimized RTL generation.

The tool can generate the Golden RTL VHDL files of the input application. Golden RTL generation is the generation of RTL without any optimization and resource sharing. It also has the capability of delay and consumed area estimation of the unoptimized application for Xilinx FPGAs. This delay and area estimator is tested with five different applications for Xilinx Spartan 3 FPGA. The delay, area estimation results for the tool proposed in this thesis is compared with the delay and area results gathered from the Place&Route report of Xilinx ISE. The average error for these tests are found 6% in delay estimation and 3.8% in area estimation. However, Xilinx ISE makes these estimations by passing through the synthesis, mapping, place & route steps which take considerably long time. The CPU run times of Xilinx ISE tool and the estimator tool proposed in this thesis are compared. Our estimation tool works in the order of milliseconds whereas Xilinx's process flow generates the expected results in the order of seconds. On average, our estimation tool is 300 times faster than Xilinx with acceptable error margins. The Generated Golden RTL files are used to verify Optimized RTL by comparing the simulation results of the Golden RTL and Optimized RTL VHDL files.

This tool can generate the Optimized RTL VHDL files of the input application. Optimized RTL generation is the generation of the Optimized RTL where resource sharing exist. Optimization is not in the scope of this thesis, optimization information is used only as input. According to the optimization results, the optimum clock period for the application is found, graph is rescheduled and RTL VHDL files are generated for that clock period. The consumed area is also estimated for the optimized RTL

in Spartan3 FPGA. The estimated delay found with rescheduling is compared with the delay value after the design is Placed & Routed in Xilinx Spartan-3 FPGA. The area estimated during optimized RTL generation is also compared with the area value found after the design is Placed & Routed in Xilinx Spartan-3 FPGA. The average delay error for the tested applications is 14.4 % and average area error is 6.88 %. The error is caused by the routing delays in FPGA. Moreover, the estimated delay and area is also compared with the delay and area found in the optimization module. Since the optimization model does not take registers into account, it is found that there is an overhead in the delay after the design is rescheduled. However, the delay and area trends are similar with the delay and area found with the optimization module.

As future work, the automated RTL verification should be designed and implemented since the simulation results are compared manually for the Golden RTL and Optimized RTL descriptions.

APPENDIX A: VHDL TEMPLATE GUIDELINES

A.1. File Names

File names guideline is given in Table A.1.

A.2. Entity Names

Entity names guideline is given in Table A.2.

- Names should clarify what the block code does.
- Names should be lowercase and use underscores to separate words.
- Group the ports of the entity as : 1 Reset, 2 Clocks, 3 Input Group, 4 Control Signal Group (e.g., Select), 5 Output Group.
- Port types should be std_logic_vector and std_logic. Other port types such as bit, bitvector, std_ulogic are not allowed.

A.3. Port Types

Port Types are given in Table A.3.

- Port Names should be compatible with the syntax of LRH(+). The variable order in each subexpression should be considered while naming dataports. For example, for the expression ".+(a,b)" "a" should be connected to d_0 and "b"

Table A.1. File Names.

File : <**entityname**>**.vhd**
Test Bench : **tb_**<**entityname**>**.vhd**
Package Name : **pkg_**<**package_name**>**.vhd**

Table A.2. Entity Names.

<template_file_name>_<number>

Table A.3. Port Types.

Input names : **<portname>** – input port name
Output names : **out_<portname>** – output port name
Parameter names : **mode_<parametername>** –parameter port
Registered output : **ro_<portname>**
Registered input : **ri_<portname>**

should be connected to d_1.

- Parameter port is used for a case of the data which is input to the entity does not change and used only for one kind of operation such as choosing mode, initialization of counters etc.
- Registered output is used for the output from the entity is coming directly from a register.
- Registered input port is used for the input coming to the entity directly from a register.

A.4. PORT NAMES

Port names guideline is given in Table A.4.

A.4.1. Usage of some port names

- **Write and Read Enable** are used for memory modules such as RAMs
- **Write and Read Address** are used for memory modules to address memory locations.

Table A.4. Port Names.

Clock : **clk_<number>**
Clock divided : **clk_div<division_amount>_<number>**
Clock multiplied : **clk_x<multiplication_amount>_<number>**
Reset : **rst** — for multiple resets, reset vectors are used.
Asynchronous Reset : **rst_a**
Reset Active Low : **rst_l**
Asynchronous Reset Active Low : **rst_al**
Data :**d_<number>**
Write Enable : **we_<number>**
Read Enable : **re_<number>**
Write Enable active low : **we_l_<number>**
Read Enable active low : **re_l_<number>**
Data Enable : **de_<number>**
Enable : **en_<number>**
Write Address : **w_adr_<number>**
Read Address : **r_adr_<number>**
Write/Read Address : **wr_adr_<number>**
Select : **sel_<number>**
Acknowledge : **ack_<number>**
Request : **rqt_<number>**
Load : **ld_<number>**
Status :**stat_<number>**

Table A.5. Generics and Constants Names.

Generic : **g_<generic_name>**
Generate Statement Name : **gen_<component_name>**
Component instance : **<name of the vertex in the graph>**
Constant Name : : **const_<constant_name>**
Type Name : **t_<type_name>**

- **Enable** is used to activate and de-activate all operations of any module
- **Data Enable** indicates that there is valid data on the data line of the module
- **Select** is used to select the output of an active module such as multiplexers
- If multiple clocks are necessary, clocks should be given as an array from the top level and they can be numerated inside if necessary.
- All port names are numerated starting from 0. For example we_0, we_1, r_adr_0, r_adr_1 etc.
- Data naming starts from 0 such as d_0 , d_1 , d_2 etc. The inputs and outputs of the modules are named separately. For example, for inputs d_0, d_1, d_2 .. are given as name, for outputs out_d_0, out_d_1, out_d_2. . . .

A.5. Generics and Constants

Generics and constants guideline is given in Table A.5.

A.5.1. Some Rules to Consider

- For constants and generic names upper case letters should be used. (e.g., WIDTH, RAM_DEPTH, BLOCK_SIZE)
- The component instances are named according to name of the vertex in the graph while generating(Sub Expression Names such as SE56).

A.6. Signal Names and Types

Signal names and types guideline are given in Table A.6.

Table A.6. Signal Names and Types.

Signal Names : **<last module which the signal is the output of><instance number>_<portname / portnumber>**
Registered Signal : **signalname>_r<number_of_clocked_delays>**
Extended Signal : **port name>_e**
Trimmed Signal : **<port name>_t**

A.6.1. Some Rules to Consider

- Signals are named according to the last module which they are output of. If signal is registered; signal is named as <signalname>_ r<number> where number indicates how many times it is registered.
- For active-low signals, using '<*signal_name*>_*l*' is preferred.
- If the signal is the output of an adder of instance 1 it is named as ; add1_d0, if instance 2; add2_d0, d0 here indicates the port name of the module which the signal is coming from, It may be d0, d1, d2..
- Signal Name should start with uppercase letter.
- When a signal is extended or reduced it must be assigned the same signal again ,however if it is a port signal it can't be assigned to same signal so new signal should be declared and it is naming should be <port name>_e if extended, port name>_t when reduced. If that signal is extended or reduced several times, first operation is important. If it is extended first and then reduced the signal name is <portname>_e.

Table A.7. Architecture Names.

Behavioral : **Behavioral**
Structural : **Structural**
Mixed : **Mixed**

Table A.8. Signal Order.

Type Declaration
Constant Declaration
Declaration of Components
Declaration of Signals

A.7. Architecture Names

Architecture names guideline is given in Table A.7.

A.8. Declarations

Component and Signal Declaration should be ordered as groups that is given in Table A.8 ;

A.9. CDFG vs VHDL

At the beginning of the VHDL file of the top module, a CDFG vs VHDL part exist as a comment. It gives the information about which vertex in the CDFG corresponds to which component in VHDL, which edge in CDFG corresponds to which signal in VHDL. Below, there is an example CDFG and the VHDL vs CDFG section. In the comment section of the graph in Figure A.1 is given in Table A.9.

Table A.9. CDFG vs VHDL section of example graph.

```
---------- CDFG vs VHDL --------------
--- node a = d_0
--- node b = d_1
--- node SE86 = RCAAdder_212_100_i1_4_i2_4
--- node c = d_2
--- node d = d_3
--- node SE87 = RCAAdder_212_101_i1_4_i2_4
--- node e = d_4
--- node SE88 = RCAAdder_212_102_i1_5_i2_5
--- node SE90 = RCAAdder_212_103_i1_6_i2_6
--- node out = out_d_0
--- edge e0 = d_0
--- edge e1 = d_1
--- edge e5 = RCAAdder0_d0
--- edge e2 = d_2
--- edge e3 = d_3
--- edge e4 = RCAAdder1_d0
--- edge e10 = d_4_e
--- edge e9 = RCAAdder2_d0
--- edge e11 = RCAAdder3_d0
```

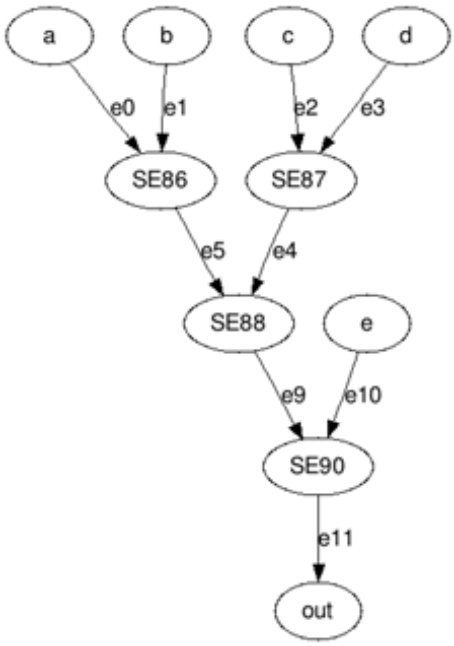

Figure A.1. An Example Graph.

A.10. General Rules to Consider

- Lowest level can be both behavioral and structural. In higher levels, the modules are connected by structural vhdl coding style however, behavioral coding can be used for registers and multiplexers in higher levels too.
- **Entity Names :**
  - (i) Names should clarify what the block code does.
  - (ii) Names should be lowercase and use underscores to separate words.
  - (iii) Group the ports of the entity as : 1 Reset, 2 Clock, 3 Input Group, 4 Control Signal Group (e.g., Select), 5 Output Group.
  - (iv) Port types should be std_logic_vector and std_logic. Other port types such as bit, bitvector, std_ulogic are not allowed.
    - **Port Names :**
      - (i) **Write and Read Enable** are used for memory modules such as RAMs
      - (ii) **Write and Read Address** are used for memory modules to address memory locations.

(iii) **Enable** is used to activate and de-activate all operations of any module.

(iv) **Data Enable** indicates that there is valid data on the data line of the module.

(v) **Select** is used to select the output of an active module such as multiplexers.

(vi) Others sections are the port names that cannot be described by the port names included which may be necessary.

(vii) If multiple clocks are necessary, clocks should be given as an array from the top level and they can be numerated inside if necessary.

(viii) Data naming starts from 0 such as d_0 , d_1 , d_2 etc. The inputs and outputs of the modules are named separately. For example, for inputs d_0, d_1, d_2 .. are given as name, for outputs out_d_0, out_d_1, out_d_2. . . .

- **Signal Names :**

(i) If signals are related to behavioral coding style, there should be r and c in front of signal name to define it is combinational or registered signal. If signals are related to structural coding style r and c shouldn't be used.

(ii) For active-low signals, using '<*signal_name*>_*l*' is preferred.

(iii) Portname or portnumber defines which port of the module that the signal is coming from. For example " sum" can be used for defining sum port of an adder; or 2 can be used as defining second port of an Adder. If the module has only one port, port number or port name may not be used.

A.11. Other Rules to Consider

- Maximum Length of a code line must be 80 with comments.(Divide lines for easy understanding of code)
- Do not exceed 16 characters for the length of signal and instance names, whenever

possible.

- Names of objects with a hardware equivalent (e.g., signals, entities/architectures, components) start with an upper-case letter.
- Names of objects with no hardware equivalent (e.g., variables, types) and labels start with a lower-case letter.
- For constants and generic names upper case letters should be used. (e.g., WIDTH, RAM_DEPTH, BLOCK_SIZE)
- Generic names should be BITSIZE for the bitwidth of the port, BLOCKSIZE for compressor bit size in adders.
- For signal assignments, port mappings and component instantiations, the general rule is one line per signal. This rule will increase the readability and understandability of your code.
- For signal assignments, vector signals should be increasing vector size. For example;
  (i) Signal Signalname1 : std_logic_vector(2 downto 0);
  (ii) Signal Signalname2 : std_logic_vector(3 downto 0);
  (iii) Signal Signalname3 : std_logic_vector(4 downto 0); goes like that.

- Port mapping should not be done by listing, it should be done by explicitly as port map (input1 => i_1, input2 => i_2, output => o_1).

- For enumeration of coding states in FSM, do not use s0,s1,a,b,state0; naming should clarify the state machines. The standards for state machines are given below. If necessary coding name is not listed here, it can be used extra suitable name which is related to operation.
  (i) Idle<number>
  (ii) Read<number>
  (iii) Write<number>
  (iv) Start<number>
  (v) Reset<number>
  (vi) Wait<number>

(vii) Send<number>

(viii) Load<number>

(ix) Shift<number>

(x) Check<number>

- The following FPGA resource names are reserved and should not be used to name nets or components.
  (i) Components (Comps), Configurable Logic Blocks (CLBs), Input/Output Blocks (IOBs), Slices, basic elements (bels), clock buffers (BUFGs), tristate buffers (BUFTs), oscillators (OSC), CCLK, DP, GND, VCC, and RST.
  (ii) CLB names such as AA, AB, SLICE_R1C2, SLICE_X1Y2, X1Y2, and R1C2.
  (iii) Primitive names such as TD0, BSCAN, M0, M1, M2, or STARTUP.

- Do not use pin names such as P1 and A4 for component names.
- Do not use pad names such as PAD1 for component names.
  (i) Align the columns and port types in the entity, signal declaration and port mapping.
  (ii) Commenting and Indentation should be as below;

COMMENTS :

Comments should include a header template for each entity-architecture pair and for each package and package-body pair. The purpose should include a brief description of the functionality of each lower block instantiated within it. An example comment for headers are given in Table A.2. An example comment for entities are given in Table A.3. An example comment for processes are given in Table A.4. An example comment for statements are given in A.5. Within processes and functions commenting is important so that the reader may understand not only that you have made a signal assignment, but also your reasoning behind it with respect to the entire code block. You need not comment every statement, but those that are key to the behaviour of the component should generally be commented.

```
--------------------------------------------------------------------------------
-- Author:                                          Copyright CASLAB
--
-- Create Date:    01/13/2011
--Revision History          Date          Author        Comments
--                        01/13/2011      CASLAB        created
--                        01/15/2011      CASLAB        changed entity port adress
--------------------------------------------------------------------------------
-- Module Description
-- File: header.vhd
-- Module Name(s): (entity/architecture/package/etc.)
-- Constraints: (critical path delay, etc.)
-- Limitations: (bus functional only, partial implementation, etc.)none
-- I/O Format(s): (I/O formats and/or unusual interface characteristics)
--------------------------------------------------------------------------------
-- Description:
-- This entity architecture pair is a block level with 4 sub-blocks.This is the processor control
--interface for the block level <block_level> ...
-- Additional Comments:

--------------------------------------------------------------------------------
```

Figure A.2. Header Commenting.

```
entity <entity-name> is
begin
port(clk: in std_logic; --system clock
data: out std_logic_vector(7downto 0);--data output sent to next component
end <entity-name>;
```

Figure A.3. Entity Commenting.

```
-------------------------------------------
-- ProcessName:
-- A description of the process and
-- what it accomplishes goes here
-------------------------------------------
ProcessName: process ()
begin
```

Figure A.4. Process Commenting.

```
data <= (others => '0'); -- reset the data to 0's
```

Figure A.5. Statement Commenting.

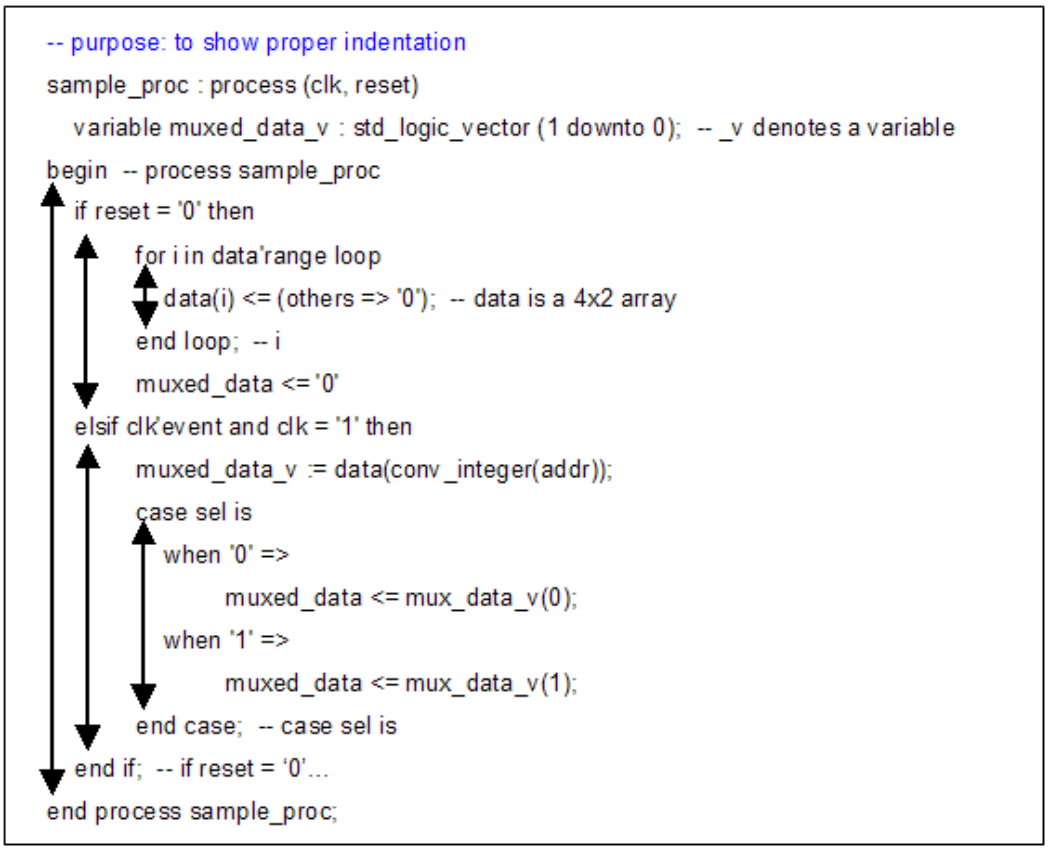

Figure A.6. Indentation.

INDENTATION

- Any section of code that is marked by an end keyword (begin, for ,if, generate; they all end with end) should have its body intended. It includes component declarations, process statements, if statements, case statements, entity declarations and architectures.
- Begin of a block and end of it should be in the same indentation. Indentation must be with 2 space characters.

An example for indentation is given in Figure A.6.

APPENDIX B: SCENARIOS

In the Figure B.1, single port RAM is shown. Write enable (we_0) , Read enable (re_0), data port (d_0), write/read adres port (wr_adr_0) is and clock (clk) is shown from the top module.

- we_0 : is used for enabling the writing to the single port RAM.
- re_0 : is used for enabling the reading from the single port RAM.
- D_0 : is the data to be read or write.
- out_d_0 : is the output data of the top module.
- wr_adr_0 : is the address for reading from and writing to the single port RAM.
- clk : is the clock of the single port RAM

The important thing is the input and output ports are named separately, Each of them starts from d_0 for example, D_0, D_1, D_2 for inputs; Out_d_0, Out_d_1, Out_d_2 for outputs. All port names enumerated starting from 0

The figure B.2, shows a dual port RAM. Write enable (we_0, we_1) , Read enable (re_0, re_1), data port (d_0, d_1), write/read adres port (wr_adr_0,wr_adr_1) and clock (clk) is shown from the top module. Clock port is a bus in the top level clk (1 downto 0) and it is splitted inside the module as clk_1, clk_2.

- we_0 : write enable of the first port of the dual port RAM.
- we_1 : write enable of the second port of the dual port RAM.
- re_0 : read enable of the first port of the dual port RAM.
- re_1 : read enable of the second port of the dual port RAM.
- D_0 : data to be read from or write to first port of the dual port RAM.
- D_1 : data to be read from or write to second port of the dual port RAM.
- out_d_0 : output data of the top module.
- out_d_1 : output data of the top module.
- wr_adr_0 : Read and write address of the first port of the dual port RAM.
- wr_adr_1 : Read and write address of the second port of the dual port RAM.

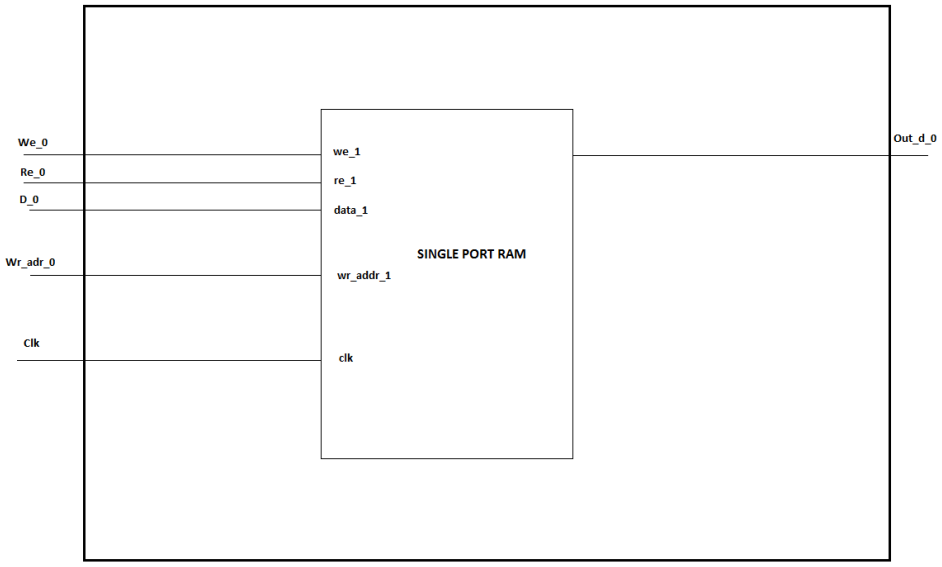

Figure B.1. Single Port RAM.

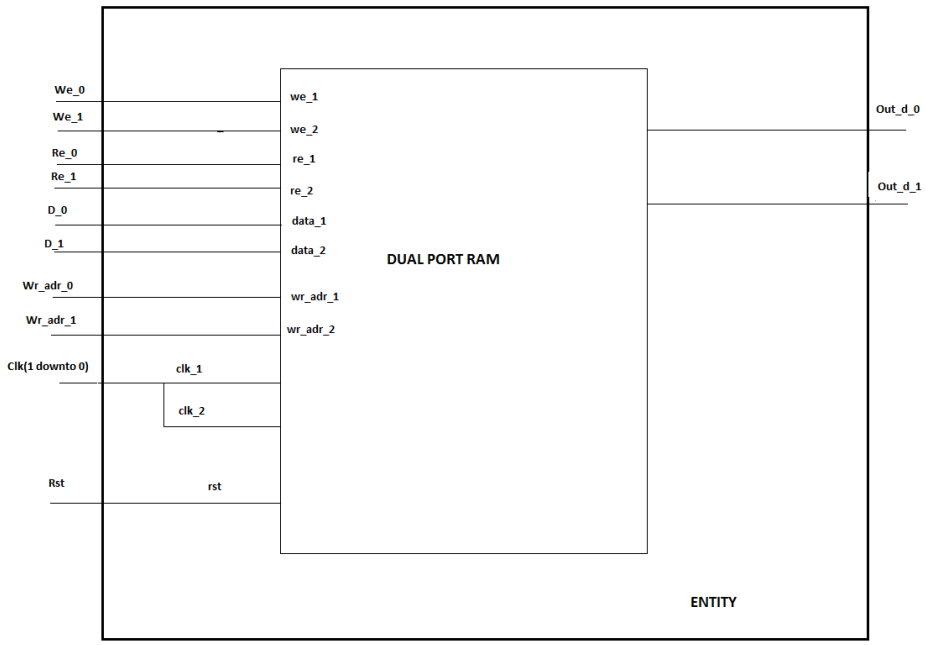

Figure B.2. Dual Port RAM.

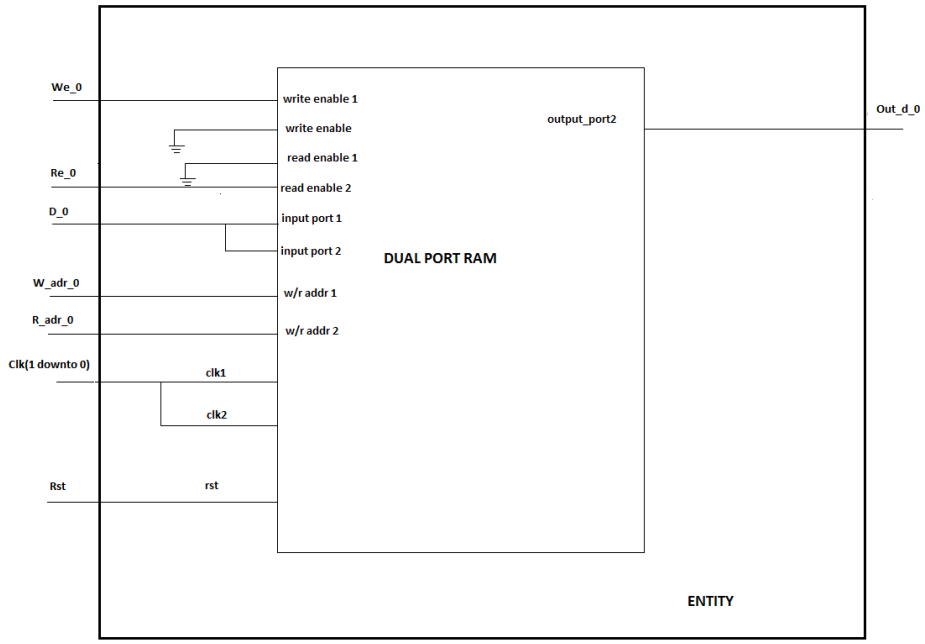

Figure B.3. Dual port RAM is used as a single port RAM with separate addresses for write and read.

- clk : clock signal of the top module is a bus and it is splitted to clk1 and clk2 as the clock input to the first and second ports of the dual port RAM.
- rst : reset port.

Dual port RAM is used as a single port RAM which has different addresses for write and read. The figure B.3 below shows that, Dual Port RAM is connected such that, data ports are merged, write enable of port 2 and read enable of port 1 is grounded. Write enable of port 1 is used as write enable, read enable of the port 2 is used as read enable of the single port RAM. Write/Read address for port 1 is used as write address and write/read address for port 2 is used as read address. Output of the port 2 of the dual port RAM becomes the output of the single port RAM.

- we_0 : write enable signal of the Single Port RAM
- re_0 : read enable signal of the Single port RAM.
- d_0: data signal to be written into the RAM.

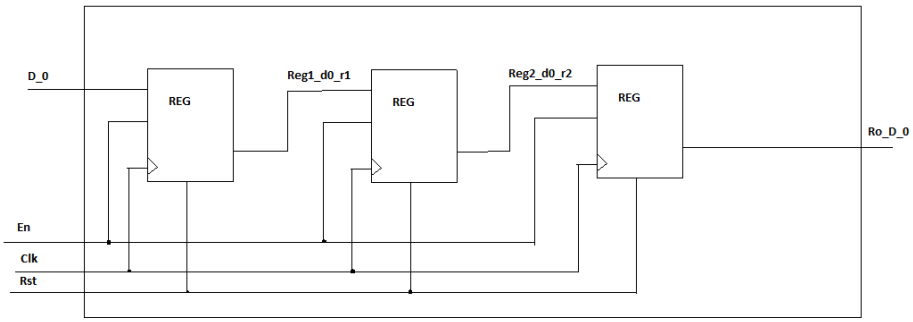

Figure B.4. Registered Signals.

- w_adr_0 : write address of the RAM.
- r_adr_0 : read address of the RAM
- clk(1 downto 0) : clk port of the RAM, it is a bus signal and it is splitted if we want different speeds for write and read.
- rst: reset signal
- out_d_0 : output data of the RAM which is read data from the port.

Figure B.4 shows the naming example of the signals. If a signal registered it is named as <signalname>_r1, <signalname>_r2, <signalname>_r3 etc.

- d_0 : data port of the top module.
- en : enable port of the module.
- clk : clock port.
- rst : reset port.
- Reg1_d0_r1 : is the signal name. It is registered one time so there exists " r1" at the end of signalname. "1" after reg defines the instance of the component.
- d0 is the port name which the signal is coming from.
- Reg2_d0_r2: is the signal that goes from 2^{nd} register to 3^{rd} register. r2 comes after the signal name because it is registered 2 times."2" after Reg defines instance of the register.

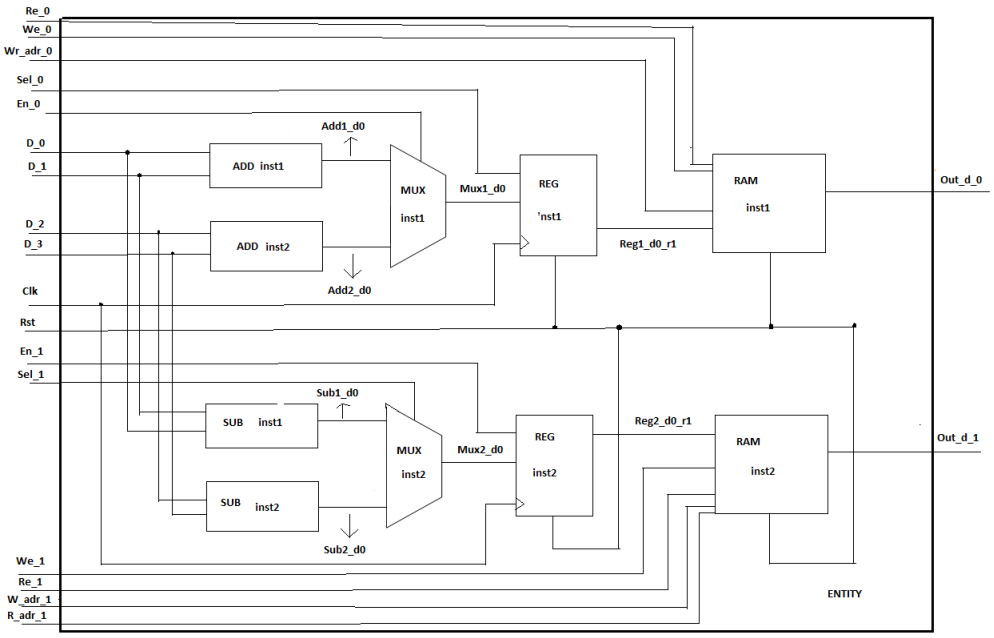

Figure B.5. Blocks with RAMs.

- ro_d_2 : output port of the module. Because of the output port comes directly from a register.

The Figure B.5 shows an example of blocks with memories and registers. Multiplexer with 2 adder units is a block and it is enabled with enable signal (en_0). Multiplexer with 2 subtractors is another unit and it is enabled with en_1.Sel_0 and sel_1 are used for Multiplexers.We_0 and we_1 are write enables for RAMs. wr_adr_0, w_adr_1 and r_adr_1 are Write/read, write and read addresses of the RAMs. Ports d_0, d_1, d_2 and d_3 are data ports and goes to adders and subtractors.

- en_0 : is used for enabling the block 1 which consists of 2 adders and 1 multiplexer.
- en_1 : is used for enabling the block 2 which consists of 2 subtractors and 1 multiplexer.
- sel_0 : is used as a select signal for multiplexer 1.
- sel_1: is used as a select signal for multiplexer 2.

- d_0,d_1,d_2,d_3 : data ports
- re_0 : is used for read enabling of the RAM 1.
- we_0 : is used for write enabling of the RAM1.
- re_1 : is used for read enabling of the RAM 1.
- we_1 : is used for write enabling of the RAM1.
- wr_adr_0 : is used for addressing for writing to and reading from RAM1. (one address port is used for addressing for both reading and writing)
- w_adr_1 : is used for addressing writing to the RAM2.
- r_adr_1 : is used for addressing reading from the RAM2. (For RAM2 write and read address ports are separated for RAM2)
- clk : clock port of the top module.
- rst : reset port of the top module.
- out_d_0 : output data port
- out_d_1 : output data port

Signals are named according to the module which they are output of. For example;

- add1_d0: it shows that it is a combinational signal, it is the output of instance 1 of the adder. Number after add indicates the instance of the adder. D0 indicates the port of the component which the signal is output.
- reg_inst1_o<number> : it shows the signal after register, and it is the output of the instance 1 of the register. If the signal was registered more than once It would be like r<number>_<signalname> where number indicates how many times it is registered.

In Figure B.5, all of the modules can be connected structurally from the top module. However, some components such as multiplexer or registers can be behaviourally coded and connected if necessary.

The Figure B.6 shows an example of mod port. There is a counter and it loads a number when load=1 which determines start number. Because of this port is stable

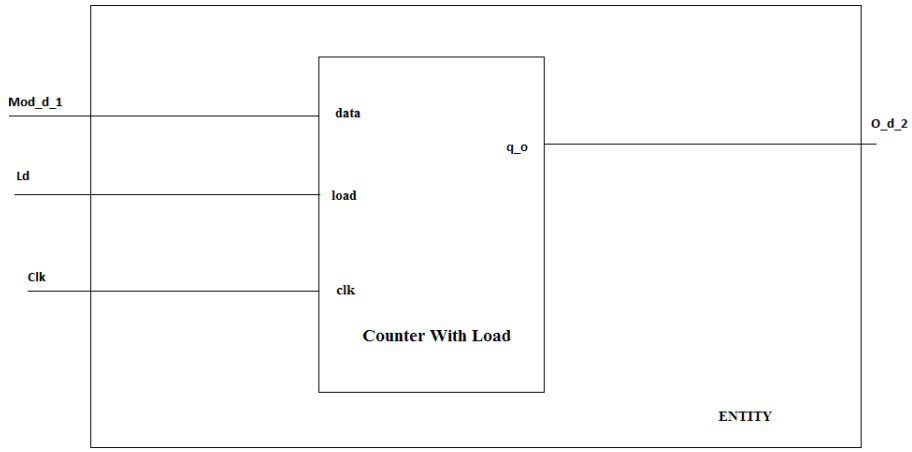

Figure B.6. Mod_<portname> is used for loading counters

and only used for loading counters it is used as mod port.

- mod_d_1 : Data port which is loaded to the counter when load=1. It is a mod port because it is stable and only used for loading counters.
- ld: load data when load=1
- clk : clock port
- out_d_2 : data output of the top module.

The Figure B.7 below shows data enable and request - acknowledge signals. The signals are shown from the top level in the Figure B.7

- o_d_1 : output data of the sender, which is input of the receiver.
- d_1 : input data of the receiver
- o_de_1: output data enable signal which is equal to 1 when sending data to the receiver.
- de_1: input port of data enable of the receiver.
- o_ack : acknowledge signal which is output of the receiver and input of the sender.
- o_rqt : request signal which is the output of the sender and input of the receiver.

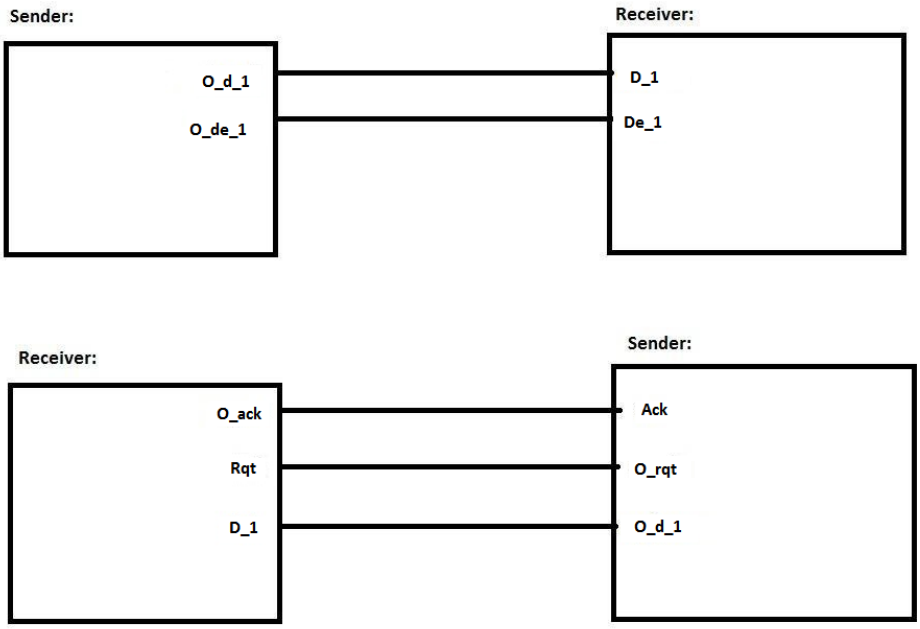

Figure B.7. Requests and Acknowledge signals.

- d_1 ,o_d_1 : data signal

The Figure B.8 shows the usage of clocks. DCM can be used for division and multiplication of the clocks.

- clk_1: is the input clock of the DCM.
- rst : reset of the DCM
- o_clk_1_x2 : Clock is multiplied by 2 and it is the output of the DCM module. When it enters to an another module it is used as clk_1_x2.
- o_clk_1_div<amount> : clock is divided by the division amount and it is the output of the DCM module. When it enters to an another module it is used as clk_1_div<amount>
- o_clk_1_x<amount> : clock is multiplied by the multiplication amount and it is the output of the DCM module. When it enters to an another module it is used as clk_1_x<amount>.

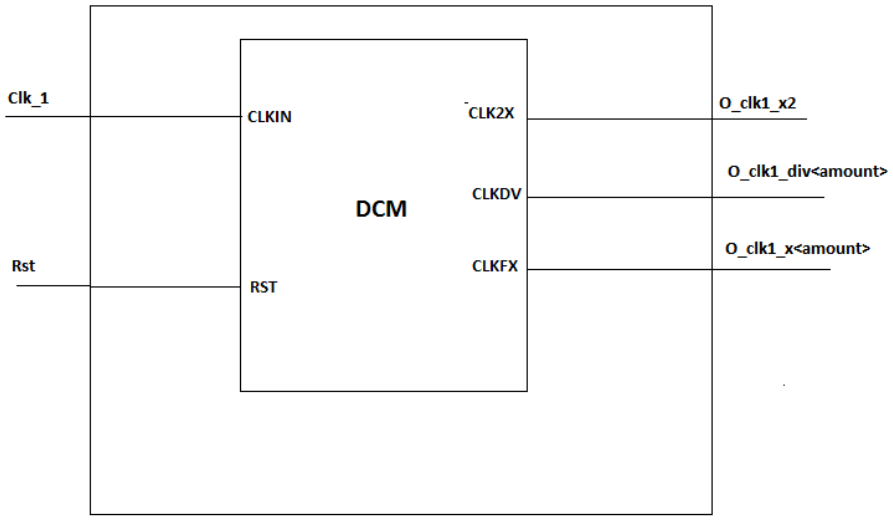

Figure B.8. Clocks are divided and multiplied with DCM of the FPGA.

B.1. Generics

Usage of generics , generate statement and instance is shown with the code of an RCA adder as an example in Figure B.9.

As shown in the Figure B.9, “bit size” is coded as “g_BITSIZE”.Generate statement is coded as “gen_adder” because here adder is the component to be generated.

```
--------------------------------------------------------------------------------
-- Author :  Ender Culha Copyright :  CASLAB
--------------------------------------------------------------------------------
--Module Description :
--File RCA_adder.vhd
--- Module Name :  LUT_2, MUXCY, XORCY is used
--------------------------------------------------------------------------------
--Description :  This entity is the RCA Adder which uses the carry chain structure of the FPGA.
--The Code is written in low level structural style so as to use carry chain for every bit size.
--LUT 2s, MUXCYs and XORCYs are connected structurally.
--------------------------------------------------------------------------------------------------------------
----------libraries-----------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
-----------libraries for low level primitives----------------------------------
library UNISIM;
use UNISIM.VComponents.all;
use IEEE.STD_LOGIC_ARITH.ALL;
------------entity ------------------------------------------------------------
entity RCAAdder_001_5_i1_8_i2_8 is
generic (g_BITSIZE: integer := 8 );
port (d_0 :  in std_logic_vector(g_BITSIZE-1 downto 0);
     d_1 :  in std_logic_vector(g_BITSIZE-1 downto 0);
     out_d_0 :  out std_logic_vector(g_BITSIZE downto 0));
end entity;
--------------------------------------------------------------------------------
architecture Structural of RCAAdder_001_1_i1_8_i2_8 is
..
..
..

end architecture;
```

Figure B.9. Example use of Generics in VHDL File.

REFERENCES

1. Xilinx, "Xilinx ISE Design Suite", `http://www.xilinx.com/`, accessed at April 2013.

2. Kurumahmut, B., G. Kabukcu, R. Ghamari and A. Yurdakul, "Design Automation Model for Application-Specific Processors on Reconfigurable Fabric", *Forum on specification and Design Languages, FDL 2009, September 22-24, 2009, Sophia Antipolis, France, Proceedings. IEEE 2009*, pp. 1–6.

3. Güncan, A., *Database Design and Optimal Module Selection for RH(+)*, Undergraduate Project, Bogazici University, Istanbul, 2011.

4. Barbacci, M. R., "Instruction Set Processor Specifications (ISPS): the Notation and its Applications", *IEEE Transactions on Computers*, Vol. 30, No. 1, pp. 24–40, 1981.

5. Kowalski, T. J. and D. E. Thomas, "The VLSI Design Automation Assistant: What's in a Knowledge Base", *Proceedings of the 22nd ACM/IEEE Design Automation Conference*, DAC '85, pp. 252–258, IEEE Press, Piscataway, NJ, USA, 1985.

6. Trickey, H., "Flamel: A High-Level Hardware Compiler", *IEEE Transaction on Computer-aided Design of Integrated Circuits and Systems*, Vol. 6, No. 2, pp. 259–269, 1987.

7. Girczyc, E., *Automatic Generation of Micro-sequenced Data Paths to Realize ADA Circuit Descriptions*, Ph.D. Thesis, Carleton University, 1984.

8. Paulin, P. G. and J. P. Knight, "Force-directed Scheduling in Automatic Data Path Synthesis", *Proceedings of the 24th ACM/IEEE Design Automation Conference*, DAC '87, pp. 195–202, ACM, New York, NY, USA, 1987.

9. Hwang, C. T., J. H. Lee and Y. C. Hsu, "A Formal Approach to the Scheduling Problem in High Level Synthesis", *IEEE Transaction on Computer-aided Design of Integrated Circuits and Systems*, Vol. 10, No. 4, pp. 464–475, 2006.

10. Landwehr, B., P. Marwedel and R. Dömer, "OSCAR: Optimum Simultaneous Scheduling, Allocation and Resource Binding Based on Integer Programming", *Proceedings of the Conference on European Design Automation*, EURO-DAC '94, pp. 90–95, IEEE Computer Society Press, Los Alamitos, CA, USA, 1994.

11. T.C. Wilson, M. G., N. Mukherjee and D. K. Banerji, "An ILP Solution for Optimum Scheduling, Module and Register Allocation, and Operation Binding in Datapath Synthesis", *VLSI Design*, 1995.

12. Marwedel, P., I. F. Informatik and P. Mathem, "The MIMOLA Design System: A Design System which Spans Several Levels", *in Methodologies of Computer System Design*, pp. 223–237, 1985.

13. Blackman, T., J. Fox and C. Rosebrugh, "The Silc Silicon Compiler: Language and Features", *Proceedings of the 22nd ACM/IEEE Design Automation Conference*, DAC '85, pp. 232–237, IEEE Press, Piscataway, NJ, USA, 1985.

14. Knapp, D., *Behavioral Synthesis: Digital System Design Using the Synopsys Behavioral Compiler*, Electronic and digital design, Prentice Hall PTR, 1996.

15. Hemani, A., B. Karlsson, M. Fredriksson, K. Nordqvist and B. Fjellborg, "Application of High-Level Synthesis in an Industrial Project", *VLSI Design*, pp. 5–10, 1994.

16. Elliott, J. P., *Understanding Behavioral Synthesis: A Practical Guide to High-Level Design*, Kluwer Academic Publishers, Norwell, MA, USA, 1999.

17. Bollaert, T., *"Catapult Synthesis: A Practical Introduction to Interactive C Synthesis*, Springer, 2008.

18. Meredith, M., "High-Level SystemC Synthesis with Forte's Cynthesizer High-Level Synthesis", P. Coussy and A. Morawiec (Editors), *High-Level Synthesis*, chap. 5, pp. 75–97, Springer Netherlands, Dordrecht, 2008.

19. Zhang, Z., Y. Fan, W. Jiang, G. Han, C. Yang and J. Cong, "AutoPilot: A Platform-Based ESL Synthesis System", P. Coussy and A. Morawiec (Editors), *High-Level Synthesis*, pp. 99–112, Springer Netherlands, 2008.

20. Oliveira, M. F., C. Kuznik, H. M. Le, D. Grosse, F. Haedicke, W. Mueller, R. Drechsler, W. Ecker and V. Esen, "The System Verification Methodology for Advanced TLM Verification", *Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '12, pp. 313–322, ACM, New York, NY, USA, 2012.

21. Wakabayashi, K. and B. C. Schafer, " " All-in-C" Behavioral Synthesis and Verification with CyberWorkBench High-Level Synthesis", P. Coussy and A. Morawiec (Editors), *High-Level Synthesis*, chap. 7, pp. 113–127, Springer Netherlands, Dordrecht, 2008.

22. Edwards, S. A., "High-Level Synthesis from the Synchronous Language Esterel", *IWLS*, pp. 401–406, 2002.

23. Haldar, M., A. Nayak, A. Choudhary and P. Banerjee, "A System for Synthesizing Optimized FPGA Hardware from Matlab", *in Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design ICCAD'01*, pp. 314–319, 2001.

24. Gupta, S., N. Dutt, R. Gupta and A. Nicolau, "SPARK: A High-level Synthesis Framework for Applying Parallelizing Compiler Transformations", *VLSID '03: Proceedings of the 16th International Conference on VLSI Design*, pp. 461–466, 2003.

25. Ku, D. and G. DeMicheli, *HardwareC – A Language for Hardware Design (Version*

2.0), Stanford, CA, USA, 1990.

26. Agility Design Solutions, *Handel-C language reference manual*, 2007.

27. IEEE Computer Society, *IEEE Standard SystemC Language Reference Manual*, 2005.

28. Altera, "Altera Quartus II Design Suite", `http://www.altera.com/`, accessed at April 2013.

29. Kulkarni, D., W. A. Najjar, R. Rinker and F. J. Kurdahi, "Compile-time Area Estimation for LUT-based FPGAs", *ACM Transactions on Design Automation of Electronic Systems*, Vol. 11, p. 2006, 2006.

30. Enzler, R., T. Jeger, D. Cottet and G. Tröster, "High-Level Area and Performance Estimation of Hardware Building Blocks on FPGAs", *Proceedings of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications*, FPL '00, pp. 525–534, Springer, 2000.

31. Deng, L., K. Sobti and C. Chakrabarti, "Accurate Models for Estimating Area and Power of FPGA Implementations", *Proceedings of the 34th IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP 2008, Las Vegas, Nevada, USA*, pp. 1417–1420, IEEE, 2008.

32. Yang, Z., G. Ma and S. Zhang, "Formal Verification of High-level Data-flow Synthesis Designs Using Relational Modeling and Symbolic Computation", *Integration the VLSI Journal*, Vol. 43, No. 1, pp. 101–112, Jan. 2010.

33. Sudipta Kundu, R. K. G., Sorin Lerner, *High-Level Verification Methods and Tools for Verification of System-Level Designs*, Springer, New York, NY, 2011.

34. Ganai, M., *SAT-Based Scalable Formal Verification Solutions*, Springer, New York, NY, 2007.

35. Ganai, M. K., A. Gupta and P. Ashar, "DiVer: SAT-based Model Checking Platform for Verifying Large Scale Systems", *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pp. 575–580, Springer-Verlag, Berlin, Heidelberg, 2005.

36. Kundu, S., S. Lerner and R. Gupta, "Validating High-Level Synthesis", *Proceedings of the 20th international conference on Computer Aided Verification*, CAV '08, pp. 459–472, Springer-Verlag, Berlin, Heidelberg, 2008.

37. Devadas, S., H.-K. T. Ma and A. R. Newton, "On the Verification of Sequential Machines at Differing Levels of Abstraction", *Proceedings of the 24th ACM/IEEE Design Automation Conference*, DAC '87, pp. 271–276, ACM, New York, NY, USA, 1987.

38. Malazgirt, G. A., E. Culha, A. Sen, F. Baskaya and A. Yurdakul, "A Verifiable High Level Data Path Synthesis Framework", *Proceedings of the 2012 15th Euromicro Conference on Digital System Design*, DSD '12, pp. 397–404, IEEE Computer Society, 2012.

39. Culha, E., G. A. Malazgirt, A. Sen, F. Baskaya and A. Yurdakul, "Sayısal İşaret İşleyen Sistemler İçin Yüksek Seviye Doğrulanabilir Veriyolu Üretimi", *Sinyal İşleme ve İletişim Uygulamaları Kurultayı(SİU'12), 2012, Fethiye, Muğla.*

40. "NuSMV v2.5 Tutorial", `http://nusmv.fbk.eu/NuSMV/tutorial/index.html`, accessed at April 2013.

41. "Scripting Xilinx ISE using TCL", `http://www.doulos.com/knowhow/tcltk/xilinx/`, accessed at April 2013.

42. "Reconfigurable Processors", `http://brass.cs.berkeley. edu/rpsum.html`, accessed at April 2013.

43. Deschamps, J.-P., G. J. A. Bioul and G. D. Sutter, *Synthesis of Arithmetic Cir-*

cuits: FPGA, ASIC and Embedded Systems, Wiley-Interscience, 2006.

44. "Directed Acyclic Graph Shortest Path Algorithm", `http://en.wikipedia.org/wiki/Dijkstra's_algorithm`, accessed at April 2013.

45. "Quickgraph Graph Datastructures and Algorithms for .NET", `http://quickgraph.codeplex.com/`, accessed at April 2013.