THESIS

ON

REDUCING COMPUTATIONAL COMPLEXITY OF MATHEMATICAL FUNCTIONS USING FPGA

A THESIS

To be submitted by

NEHA GOUR, ABMTV16030

Under the supervision of

Prof. ARUP BANERJEE
RAJA RAMANNA CENTRE FOR ADVANCED TECHNOLOGY,
INDORE-452013

For the award of degree

Of

M.TECH.

VLSI DESIGN

DEPARTMENT OF ELECTRONICS,
BANASTHALI VIDYAPITH
BANASTHALI – 304022
MAY-2018

Department of Electronics Engineering
Banasthali University, Banasthali

CERTIFICATE

This is to certify that the thesis entitled ***"Reducing computational complexity of mathematical functions using FPGA"*** submitted by **Neha Gour (ABMTV16030)**, in partial fulfilment of the requirements for the award of degree of **Master of Technology** in **VLSI Design** is a bonafide piece of work carried out by her under the supervision of **Dr. Arup Banerjee, Head, HRDS at Raja Ramanna Centre for Advanced Technology, Indore**. To the best of my knowledge the matter embodied in the thesis has not been submitted by her to any other University/Institute for the award of degree.

Dr. Ritu Vijay

Head, Department of Electronics
Banasthali University

Department of Electronics Engineering
Banasthali University, Banasthali

CERTIFICATE OF APPROVAL

This is to certify that dissertation ***"Reducing computational complexity of mathematical functions using FPGA"*** submitted by **Neha Gour (ABMTV16030)**, is accepted towards partial fulfilment for the award of degree of **Master of Technology** with specialization in **VLSI Design**.

Internal Examiner

External Examiner

Date:

Government of India
Department of Atomic Energy
Raja Ramanna Centre for Advanced Technology

CERTIFICATE

This is to certify that the dissertation entitled "***Reducing computational complexity of mathematical functions using FPGA***" submitted by **Neha Gour (ABMTV16030)**, to **Banasthali University**, is hereby approved as credible work carried out by her during the period from July 18th, 2017 to April 30th, 2018 at Human Resource Development Section (HRDS), under my supervision and is recommended towards the partial fulfilment of the requirement for the award of the degree of **Master of Technology** in **VLSI Design**. Her dedication and sincerity are praiseworthy and we wish her all professional success in her future.

Dr. Arup Banerjee
Scientific Officer "H"
Head, HRDS
RRCAT, Indore

DECLARATION

I hereby declare that this thesis report entitled **"*Reducing computational complexity of mathematical functions using FPGA*"** is an authentic record of my own work carried out as the requirement for the award of degree of **Master of Technology** with specialization in **VLSI Design** under the guidance of Dr. Arup Banerjee, Head, HRDS, Raja Ramanna Centre for Advanced Technology, Indore.

DATE

NEHA GOUR

ABMTV16030

It is certified that the above statement made by the student is correct to the best of my knowledge and belief.

Dr. Arup Banerjee
HRDS
RRCAT, INDORE

ACKNOWLEDGEMENT

The final outcome and success of this M.Tech thesis required the guidance and support from many people and I am extremely fortunate to have got this all along the completion of my project work. I feel credited to complete this thesis at the Human Resource Development Section, **Raja Ramanna Centre for Advanced Technology**, Indore.

I would like to thank **Dr. P. A. Naik, Director,** Raja Ramanna Centre for Advanced Technology for giving me this wonderful opportunity to work in Human Resource Development Section, RRCAT Indore during the entire length of my project work.

I would like to express my gratitude to my project guide, **Prof. Arup Banerjee**, SO/H, HRDS, RRCAT Indore for his excellent guidance, patience, and providing me an excellent atmosphere for doing research.

I would also like to express my sincere thanks to **Dr. Srivathsan Vasudevan,** IIT Indore, **Dr. Satya S. Bulusu** IIT Indore, **Dr. Amod C. Umarikar,** IIT indore who always gave the valuable suggestion and guidance for this work. The advice and suggestions from their side helped me to complete the project work smoothly.

In addition, I am also very thankful to **Dr. Ritu Vijay (**HOD) Department of Electronics, Banasthali University and **Mr. Vipan Sharma**, Department of Electronics, Banasthali University, Rajasthan for supporting and providing me exposure to the industrial field.

I would not forget to remember **Miss Megha Sharma** research scholar, RRCAT, **Mr. Abhijeet Gorey** research scholar, IIT Indore and **Miss Kritika Bhardwaj** JRF, IIT Indore for their encouragement and more over for their timely support and guidance till the completion of project. Finally, there is still a debt I owe to my family and all my well-wishers who are directly or indirectly involved in this project.

NEHA GOUR

TABLE OF CONTENTS

List of figures

List of flowcharts

List of tables

List of abbreviations

ALU	Arithmetic Logic Unit
CLB	Configurable Logic Blocks
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HLS	High-Level Synthesis
IOB	Input Output Blocks
IP	Intellectual Property
PAL	Programmable Array Logic
PLA	Programmable Logic Array
PLD	Programmable Logic Device
PROM	Programmable Read Only Memory
RTL	Register Transfer Level
SPLD	Simple Programmable Logic Device
LSI	Large Scale Integration
VLSI	Very Large Scale Integration
VHSIC	Very High Speed Integrated Circuits
VHDL	VHSIC Hardware Description Language

Abstract

Computationally intensive applications such as weather forecasting, computational biology, Big Data etc. require a lot of time to execute a task, because usually the task is sequentially executed. The objective of this project is to explore the concept of reduction in latency of computationally complex algorithms while execution. Usually, complex algorithms demand less execution time (time complexity) and low storage space (space complexity). Storage space is not an issue in digital systems. Existing algorithms do not attempt to minimize the execution time. This project explores matrix additions and matrix multiplication as two problems and tries to reduce the latency of these two operations during execution. The main goal of our work is to reduce latency using directives such as loop unrolling and loop pipelining. High-Level Synthesis (HLS) is used to apply and verify these different directives. HLS is an automated generation of hardware circuit of a digital system from its behavioral description. Our initial simulation results demonstrate the utility of these optimization techniques in reducing the total execution time of parallel integer and floating point number additions and parallel matrix multiplication in comparison to the existing algorithms. In addition, loop unrolling and loop pipelining also results in the increase of hardware.

Keywords- *Computationally intensive applications, Time complexity, FPGA, HLS.*

****

Chapter 1

INTRODUCTION

1. Introduction

1.1 Background

High-performance computing is a rapidly growing field in computer and electronics engineering. Many of these high performance computing applications are generally performed in computers/servers that operate on digital logic (binary logic 1 and 0). Digital systems are getting faster, smaller in size and more power efficient with time. The main characteristics to attain these features is a choice of suitable architecture. Here, the main goal is to achieve the execution speed by reducing latency in any computing operation. Central Processing Units (CPU), used in conventional computers generally work sequentially. Therefore, high computing applications that utilize CPUs take a long time for performing these computations.

1.2 Motivation

CPUs contain only one Arithmetic Logic Unit (ALU) and some registers to perform various computing operations [1]. Over the recent years, developments in Very Large Scale Integration (VLSI) technologies have allowed these processors to execute operations at very high speed (high clock frequency) and having multiple cores of processors[2]. However, each core will have one ALU and associated registers [1].

For many of the high computing applications, in addition to speed, reconfigurable computing can make these applications faster. For example, an application requiring 1000 additions can be performed faster by having 1000 adders in the processing unit compared to having one adder in the ALU, as is the case in a conventional CPU. One such solution of achieving reconfigurable computing is by utilizing Field Programmable Gate Arrays (FPGA) which has been explored in this project work [3].

1.3 Project goal

The aim of this project is to take up a simple mathematical operation and

- Understand the latency of this operation
- Speedup this operation by applying directives using FPGA
- Testing this mathematical operation using integer as well as real numbers
- Comparison of latency and resources used in sequential and parallel operations.

In this project, it is proposed to use FPGA for addition of two numbers (integers and floating points) in a loop of 8 and performing matrix multiplication of two 3x3 matrices. These two operations will be the fundamental step for exploring reconfigurable computing using FPGA. The application of directives (to reduce latency) is possible with the help of Vivado High Level Synthesis (HLS). Vivado HLS is software provided by Xilinx which converts high level language(C, C++) to Hardware Description Language (HDL).In the present work, Xilinx VIVADO (2017.2) software was used [3].

1.4 Organization of the thesis

This section describes the organization of thesis. The thesis is divided into five chapters and presented as follows:
Chapter 1 deals with the background and motivation behind the problem and project goal. Chapter 2 Gives an introduction to FPGA, HLS, and directives.

Chapter 3 describes the methodology to implement the proposed project work.

Chapter 4provides results and discussion.

Chapter 5 presents project summary and conclusion.

Chapter 2

LITERATURE SURVEY

2. Literature survey

2.1 History of FPGA

During the initial stages of electronics development, programmable memory devices came into picture, where the user is given the advantage of storing data into memory as well as electrically / UV erasing it and re-writing it[4]. This memory contains programmable AND and fixed OR gates that can be wired. This helps in implementing sum –of –product forms of digital logic.

Subsequently, electronic circuits evolved which can be programmed. One such application is Programmable Logic Arrays (PLAs). PLAs consist of large number of inputs and connected to the AND plane, where different combinations are logically ANDed. Outputs of logically ANDed combination go into the OR plane, where all are ORed together and outputs are produced. They generally have faster outputs [5].

Further, Programmable Array Logic (PAL) came into existence in the electronic industry. Programmable Array Logic (PAL) is a kind of variation in PLAs. In the PALs, programmable AND are used for ANDing inputs together and fixed OR plane is used to limit the number of terms. PALs are extremely fast. PLAs and PALs were developed and grouped as Simple Programmable Logic Devices (SPLD) [5].

With the advancement in fabrication technology, SPLDs were integrated on a single chip and interconnects were provided by programming to the SPLD blocks. These were called Complex Programmable Logic Devices (CPLDs). CPLDs are as fast as PALs but they are more complex then PALs [5]. In general, CPLDs consist of functional blocks, interconnect matrix and I/O blocks [5].The architecture of a general CPLD is shown in Fig. 2.1[5], where the different PALs are connected through interconnection wires.

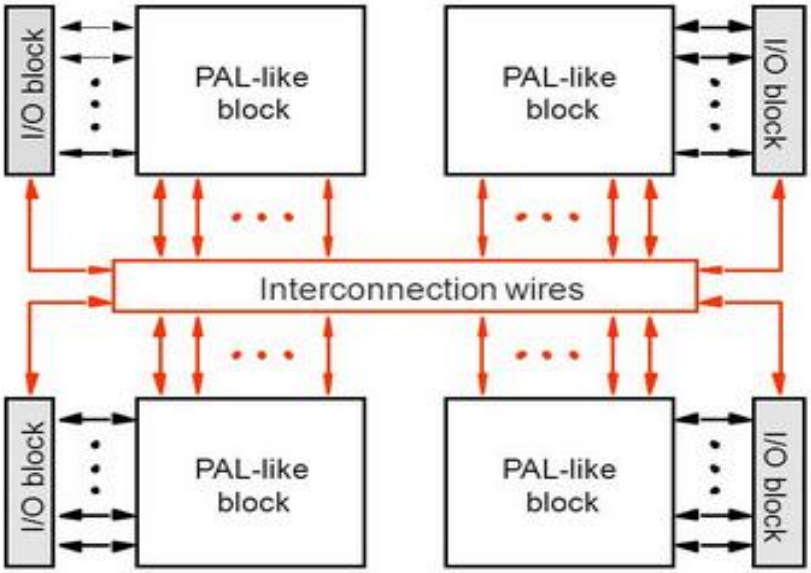

Figure 2.1: CPLD architecture [5]

2.2 Field Programmable Gate Array (FPGA)

FPGA is an integrated circuit that can be programmed in laboratory after manufacture. A FPGA is defined as a matrix of Configurable Logic Blocks (CLBs) and connected to each other by a programmable interconnection [6]. The memory cells, control logic blocks and interconnects are all connected within the FPGA. The word "field" in name refers to ability of gate array to be programmed by the user at the field. The word "array" is used to indicate the series of columns and rows of logic gates that can be programmed by end users.

When FPGA is configured, the internal circuitry is connected in a way that creates the hardware implementations for software applications FPGA devices deliver the high performance, low power, short time to market, reliability, high end productivity and flexibility in re-programmability in the hardware, because of their parallel nature [6].The flexibility offered by FPGA improves the performance by reducing the time complexity of system and possibility to implement a complex logic in real time. Our requirement of applications will decide which FPGA family is suitable for the specific application.

FPGA are manufactured by mainly two companies namely Xilinx and Altera. Xilinx FPGA families are Spartan(45nm), Virtex(28nm), Kintex (28nm), Artix(28nm), VirtexUltrascale (20nm), VirtexUltrascale+ (16nm). Since this project uses Artix FPGA, its architecture is explained further in the next section.

2.3 Architecture of an FPGA

Architecture of FPGA consists of three basic components [5]; 1. Configurable Logic Blocks (CLBs) which implement the logic functions. 2. Programmable interconnects, which provide the interconnection between logic blocks. 3. I/O blocks to complete the design as shown in Fig. 2.2[5]. It consists of multiplexers, pass transistors and buffers that form the desired connection. The next basic component is programmable Input / Output Blocks (IOBs) that provides a programmable interface between internal array of logic blocks (CLBs) and devices external package pins. CLBs perform user specified logic functions and interconnect resources carry signals among the blocks [5].

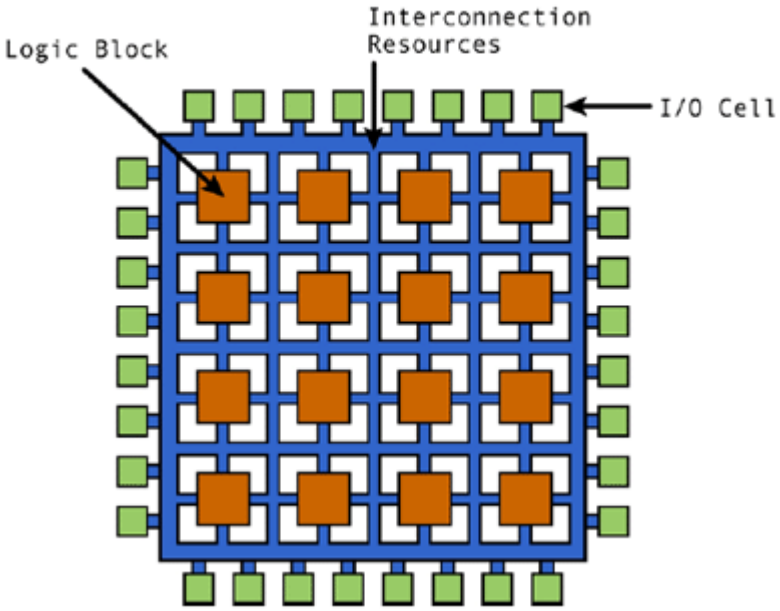

Figure 2.2: Basic structure of FPGA [5]

2.3.1 Configurable Logic Blocks (CLBs)

CLBs are the basic building blocks that are the functional elements through which the user can construct the logic [5]. It contains Lookup Tables (LUTs) for creating arbitrary logic functions. It also contains Flip-Flops (FF) for clocked storage elements. CLB also contains routing elements for routing signals from one block to another. They provide the physical support for an implemented and downloaded design. CLBs have inputs on each side, and this versatility makes them flexible for the mapping and partitioning of logic [7]. A brief architectural diagram of CLB is provided in Fig. 2.3[5].

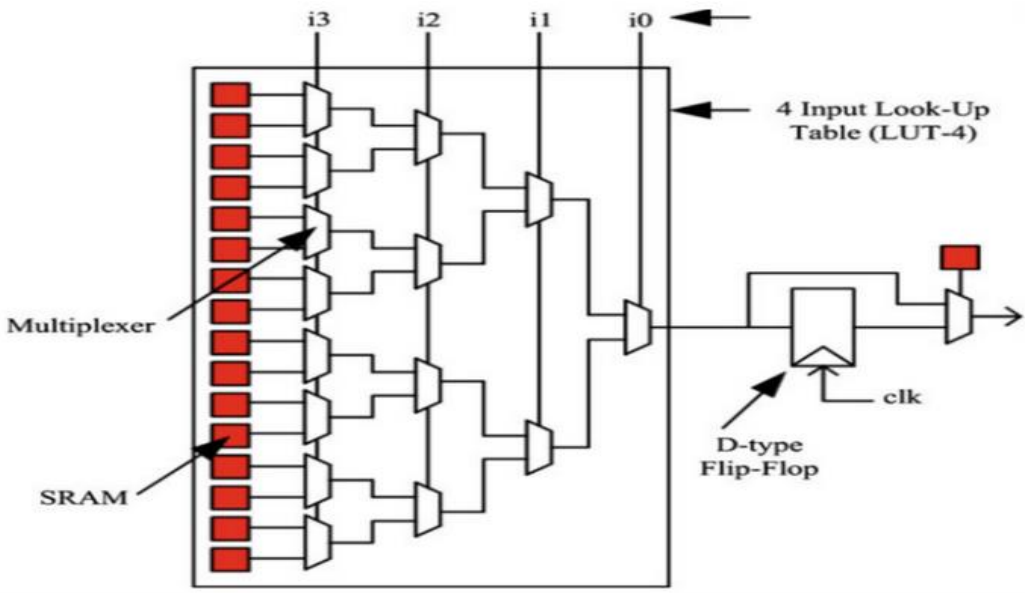

Figure 2.3: Configurable Logic Blocks [5]

2.3.2 Programmable Interconnects

Interconnects perform routing operations between different logic blocks and I/O devices. They can be reprogrammed by the user. Interconnect is a very large programmable switch matrix that allows signals from all parts of the device go to all other parts of the device. While no switch can connect all internal function blocks to all other function blocks. Figure 2.4[5] provides an overview of the architecture of the interconnects. There are different technologies are used to implement the programmable interconnects. The common technologies are SRAM and anti-fuse technology.

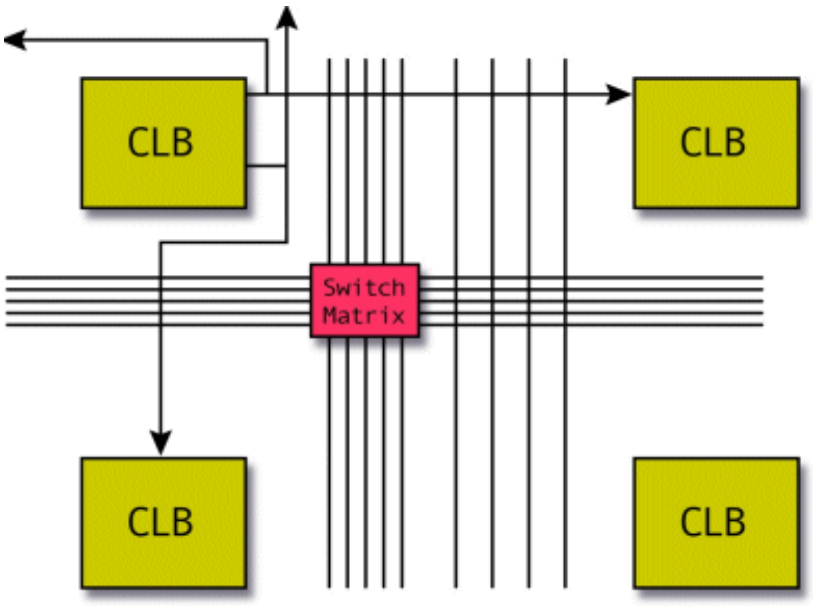

Figure 2.4: Programmable Interconnects [5]

2.3.3 Programmable IOBs

A configurable I/O block is used to bring a signal on chip from outside world and send them back again. It consist of an input buffer and an output buffer with tri-state and open collector output control. Typically there are pull-up resistor on outputs and sometimes pull-down resistors. In addition there is often a FF on output that can act as a buffer before connecting it to external boards or instruments [5]. Figure 2.5[5] provides an overview of IO blocks.

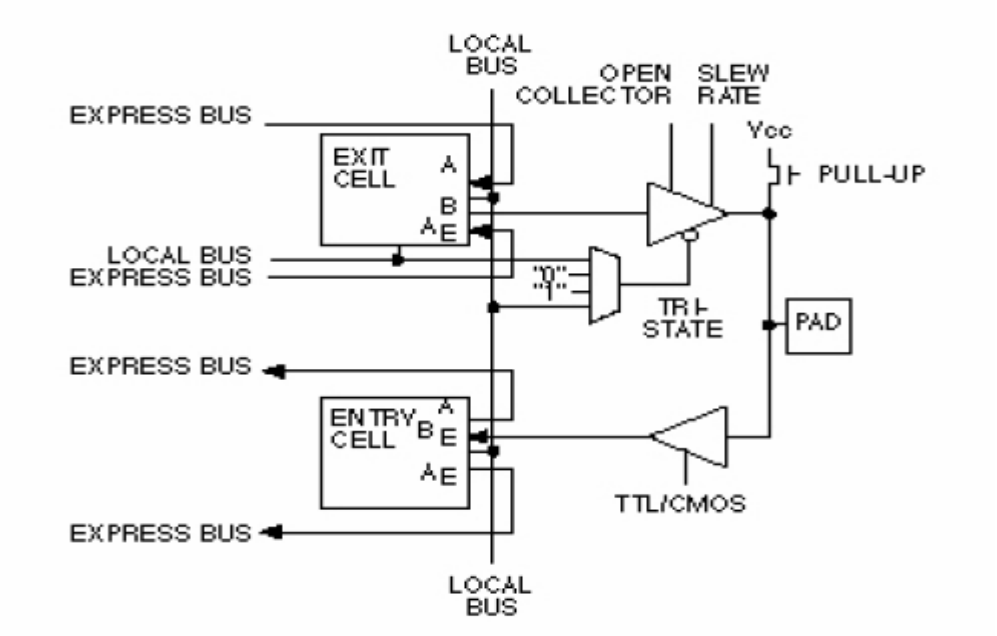

Figure 2.5: Programmable Interconnects [5]

2.4 Basys 3 Artix-7 FPGA

The basys 3 board is a complete, ready to use digital circuit development platform based on Artix-7FPGA with its high capacity FPGA (Xilinx part number XC7A35T-1CPG236C), low overall cost and collection of many ports. The basys 3 works with a new high-performance Vivado tool [8]. Fig. 2.6 is a photograph of the board used in this project. The basys 3 offers an improved collection of ports and peripherals, including [8]:

- 16 user switches
- 16 user LEDs
- 5 user pushbuttons
- 4 digit 7 segment display
- Three Pmod ports
- Pmod for XADC signal
- 12 bit VGA output

- USB-UART bridge
- Serial flash
- USB-JTAG port
- USB HID host

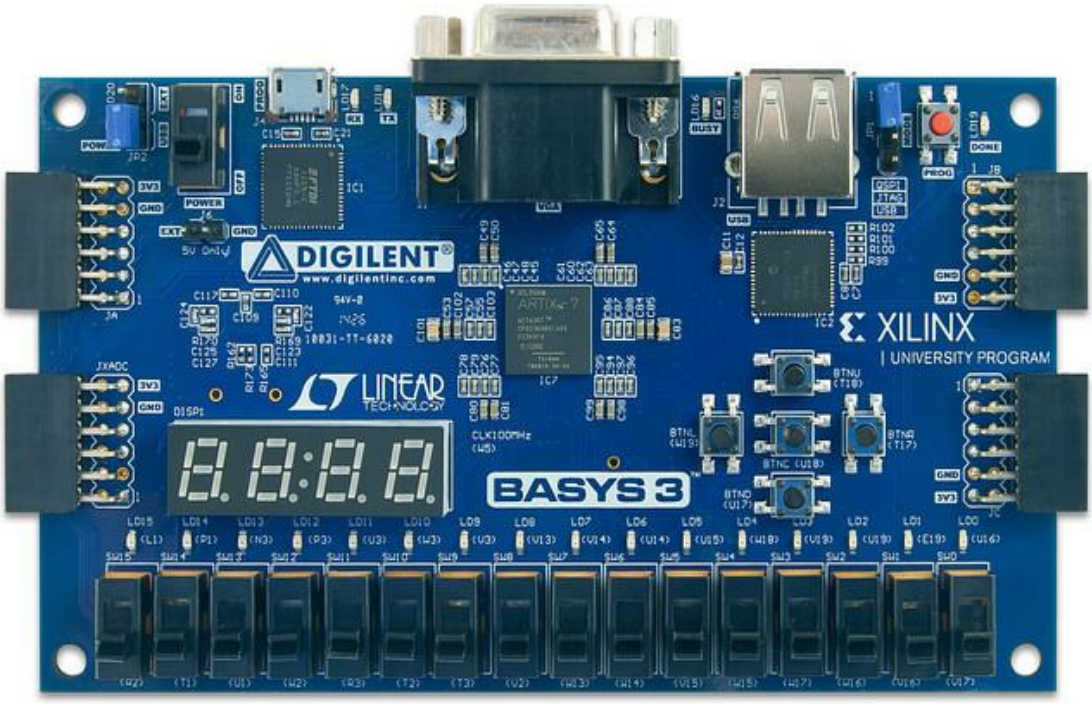

Figure 2.6: The basys 3 board [8]

Artix-7 FPGA family devices are mainly designed for low power and low cost. The Xilinx Artix-7 family of FPGA offers the fastest line rates for cost-sensitive applications. Supporting up to sixteen 6.6 GB /s transceiver that has been optimized for low power. The Artix-7 FPGA features a 50%smaller package at equivalent density compared to the previous FPGA families [12].Artix-7FPGA is optimized for high-performance logic and offers more capacity, higher and more resources than earlier designs. An Artix-7 35T feature includes [8]:

- 33,280 logic cells in 5200 slices (each slice contains four 6-input LUTs and 8 flip-flops)
- 1.800 Kbits of fast block RAM.
- Five clock management tiles, each with a phase-locked loop (PLL)
- 90 DSP slices
- Internal clock speeds exceeding 450 MHz
- On-chip Analog to digital converter (XADC).

2.5 Representation of a number in a computer

The computer uses binary digits (logic 0 and logic 1) to represent both integers and real numbers. The real numbers are represented in the floating point format. Integer data type is used widely in digital signal processing (DSP) and game applications, where performance is sometimes more important than precision. Integer numbers are faster and more area efficient. Floating point format represents a number in wider dynamic range, which allows a single data-type to be used through long sequences of calculations that are required by many algorithms [14]. Floating-point unit support is provided by Xilinx's Vivado HLS [14] and by System Generator for DSP. Both the supports include single precision and double precision floating point representations [11].

2.5.1 Real number representation using Floating point format

The unique representation of a real number has 3 components in a model within single precision and double precision representation. An IEEE 754 standard floating point binary word consists of a sign bit, exponent, and a mantissa as shown in the fig 2.7 [13].

Fig 2.7: IEEE -754 format [13]

Chapter 3

METHODOLOGY

3. Methodology

3.1 High Level Synthesis (HLS)

HLS allows the designer to work at a higher level of abstraction, starting with an algorithmic description of high-level languages such as C, C++, and System C and transform that functional code into RTL implementation [9]. HLS is increasingly popular for the design of high-performance FPGA systems. The primary benefit of HLS for hardware designers is improved productivity and performance (optimize code for latency and power) of the software design. Using the HLS design methodology, we can re-target C program into different FPGA devices. Fig. 3.1 is a flow chart of HLS that shows all the steps in the software for converting C/C++ code to RTL implementation and then to FPGA hardware implementation.

At the system specification level it is important to specify the requirement without restricting the design. The objective is to describe the purpose of design including all aspects such as timing constraints, power requirement etc.

At algorithm level specification, the input is written in some high level language (C/C++/ System C) or hardware language like VHDL, Verilog or in form of any graph. From here onwards we need the help of EDA tools. Transforming the code from lower level of abstraction to higher level of abstraction is possible by Vivado HLS tool.

RTL specification is done using HDLs. RTL description is simulated to test functionality. RTL description is then converted to a gate-level netlist using logic synthesis tools.

A gate level netlist is a description of the circuit in terms of gate and connections between them, which are made in such a way that they meet the timing, power and area specifications. Finally, a physical layout is made, which will be verified and then sent to fabrication

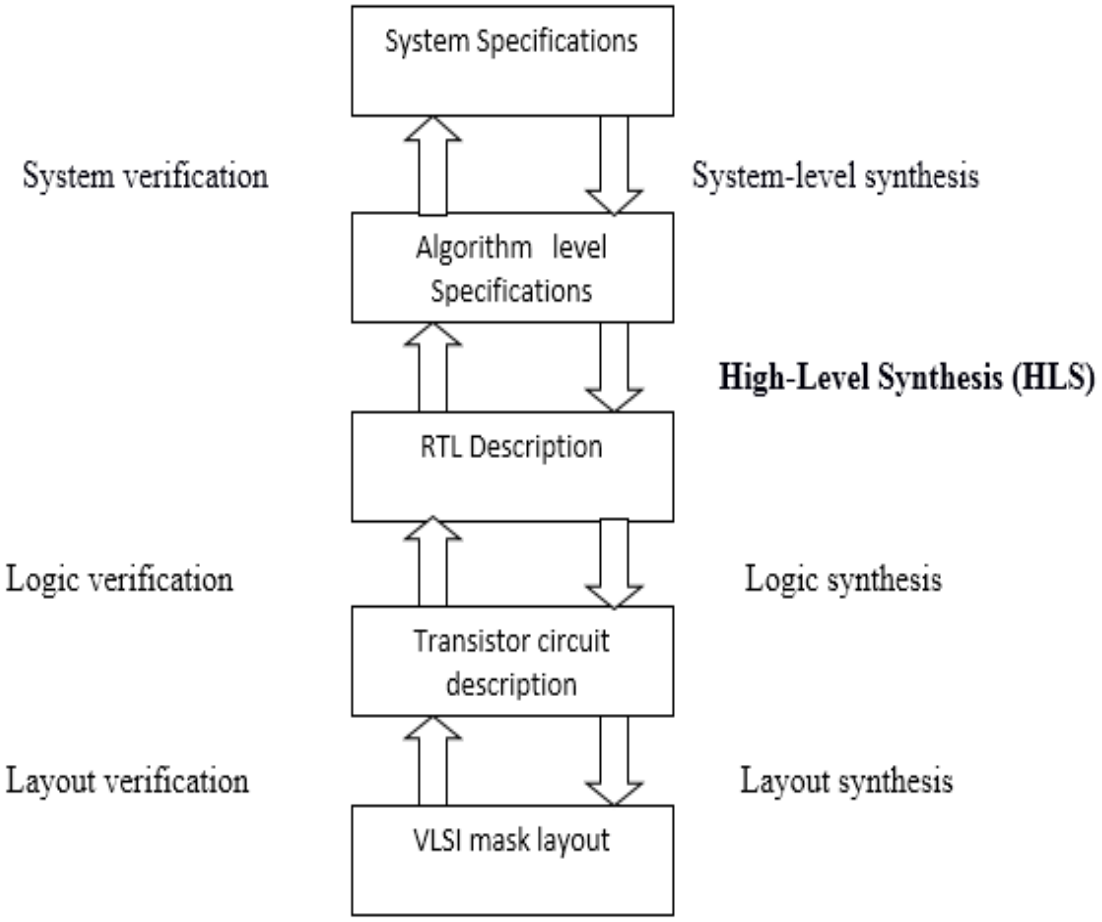

Figure 3.1: CAD Technology in VLSI

3.2 HLS working principle

Figure 3.2 provides basic introduction of High-Level Synthesis using Vivado HLS [10] tool flow. We will use the Vivado HLS in GUI mode to create a project. This flow comprises of 7 steps: First, we will create a new project in Vivado HLS, run simulation, run debug, synthesize the design using Vivado HLS, open an analysis perspective, run RTL co-simulation, view simulation results using Vivado, and export and implement the design in Vivado.

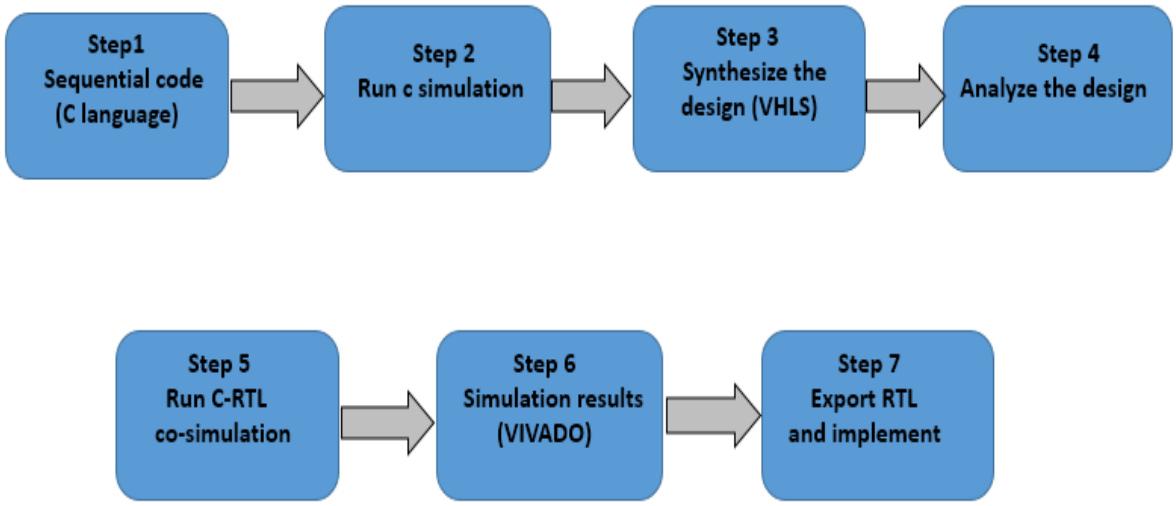

Figure 3.2: Vivado HLS tool flow [10]

3.3 Directives

Vivado HLS provides various pragmas (directives) that can be used to optimize the design by reducing latency, improving throughput performance, reducing area and device resource utilization of the resulting RTL code. These pragmas can be added directly to the source code [11]. Vivado HLS also provides Tcl set directive commands that can be passed to the tool at run time to control performance and optimization of the hardware [11].

To understand the latency by using different directives, in the present study, we considered two simple mathematical operations. 1. Addition of two 1x8 matrices, using both integers and real numbers. 2. Matrix multiplication of two 3x3 matrices involving only integers.

Usually programs written in C/C++ for computations using mathematical operations are generally sequential. To achieve speed up of these computations by reducing latency, we are applying different directives to the sequential program. Additionally, with the help of schematics and using Finite State Machine (FSM), obtained by VHDL/Verilog code, we can differentiate the changes in hardware after applying different directives to the sequential program.

3.3.1 Addition of two 1x8 matrices

3.3.1.1 Sequential looping

C/C++ programming languages provide different type of loops to handle the looping demand such as while loop, for loop, nested loops, do while loops. An example of adding two 1x8 matrices using sequential for loop is provided in figure 3.4:

```
C=0;
For (I =0; I <8; I ++)
C [I] =a [I] +b [I];
```

Figure 3.3: Original loop

The for loop is commonly used when the number of iterations is known.

Syntax:

```
For (Initialization; condition; update expression)
{
Statements;
}
```

How does loop work?

The initialization statement is executed only once. Then, the condition is evaluated. If the condition is false, for loop is terminated. If condition is true, code inside the body of loop will execute and output of loop is updated. This process repeats until the test expression is false. For example:

```
int i, A [8], B [8], C [8];
For ( i =0; i < 8; i++)
{
C [i] = A [i] +B [i];
}
```

In the above example, initialization of loop is 0(I=0), test condition is less than eight(I<8), that means statement will becomes false at I=8, every execution value should be incremented (I ++) after completion of task till condition becomes wrong. Next, operation of the loop will execute only when first iteration is completed. The state diagram shows that how many states are required to execute the task.

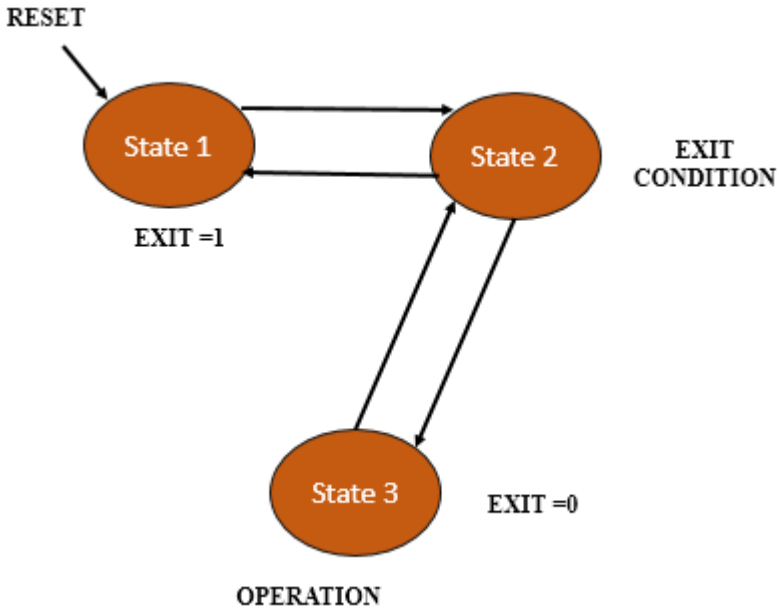

Figure 3.4: State diagram of FSM of integer numbers sequential addition

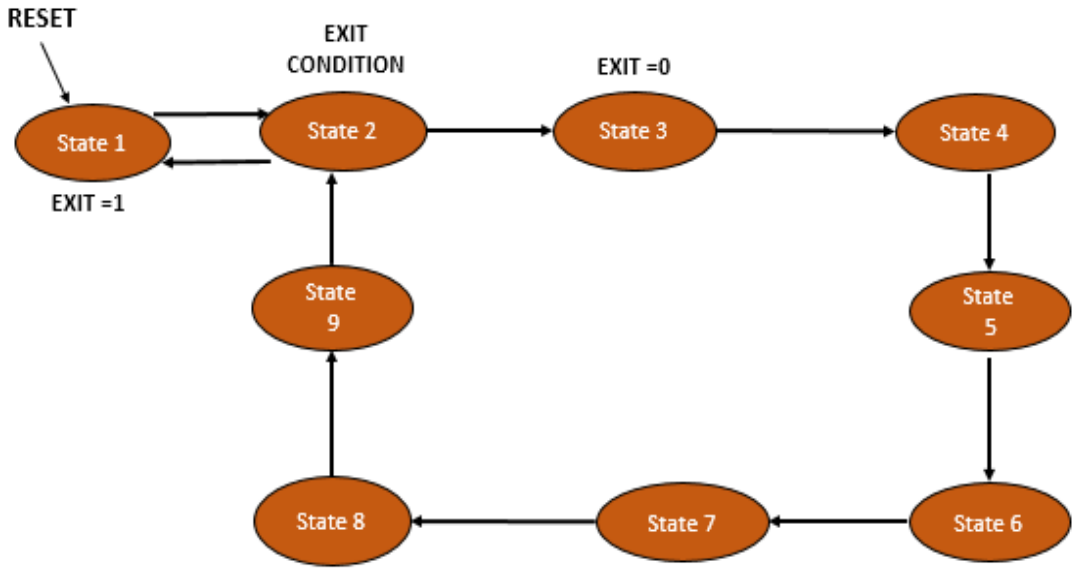

Figure 3.5: State diagram of FSM of real number sequential addition

3.3.1.2 Loop unrolling

The loop unroll transforms the loop by creating the multiple copies of loop body in the RTL design which allow the some or all loop iterations to occur in parallel[11]. Using the unroll pragma we can unroll the loop to improve the latency. Complete unrolling of above sequential loop is shown in figure 3.6.

```
For (I =0; I <8; I ++)
   {
     C [I] =a [I] +b [I];
     C [I+1] =a [I+1] +b [I+1];
     C [I+2] =a [I+2] +b [I+2];
.
      .
      .
     C [I+7] =a [I+7] +b [I+7];
   }
```

Figure 3.6: unrolled loop

The Sequential loop requires approximately 15 clock cycles to add two 1x8 matrices, ignoring the initialization part. When a loop is unrolled, the new loop executes in roughly 4 clock cycles. There is performance improvement of 74% by unrolling the loop.

To improve the performance of unrolled loop we will use different directives such as array partition, dataflow etc. However, unrolling leads a code size increases which affects the hardware design of the system.

#pragma HLS unroll factor = < integer > [11]

Factor = <integer>- that non-zero integer indicating the partial unrolling is requested and the loop body will repeat this number of times.

Loop unrolling is an optimization which can reduce overhead of a loop. That means, number of instructions of checking the loop termination conditions are removed. State diagram shows the reduction in loop latency. When we are applying loop unrolling into the fixed point addition design fetch the data together and gives the result in parallel manner. There is no need to check the loop condition unlike conventional processing. Floating point addition with an optimization directive will take more cycles to complete the task.

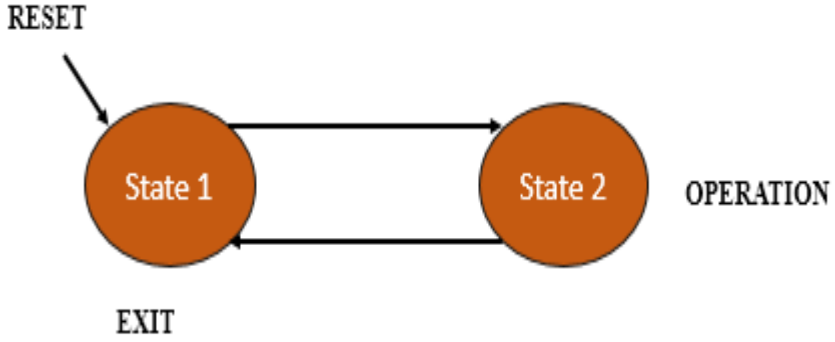

Figure 3.7: State diagram of FSM of integer number addition using unrolling

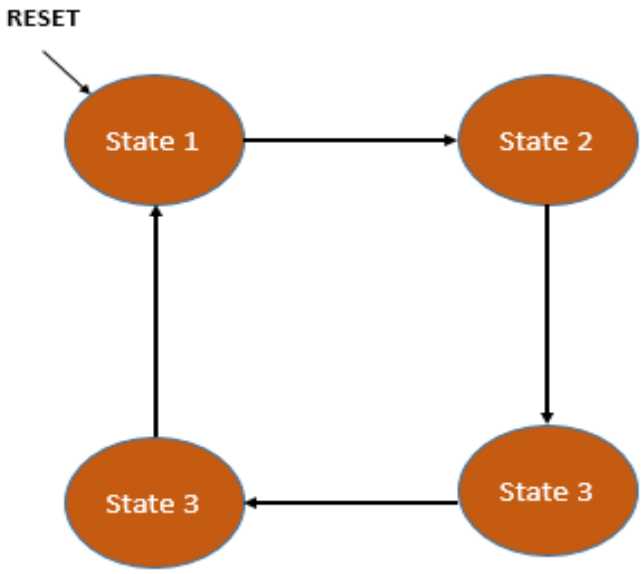

Figure 3.8: State diagram of FSM of real number addition using unrolling

3.3.1.3 Array Partition

Here the array is partitioned into smaller arrays or individual elements of the array [11]. The partitioning results in dividing large chunk of memory into multiple registers or into small memory elements in the RTL implementation. Another advantage of array partitioning is that it increases the amount of read and write ports for the storage. Array partition potentially improves the latency of the design [11].

Using array partitioning the performance of unrolled loop can be improved by increasing the amount of read and write ports for the storage. There is performance improvement of 85% and 11% by unrolling the loop with array partition compared to the sequential loop and unrolling without array partition respectively. In addition, unrolling with array partition increases in the hardware requirement. The pragma should be placed in C source with the boundary of required location.

#pragma HLS ARRAY_PARTITION variable = <name> factor = <integer> dim = <integer>[11]

Variable = <name>: a required argument that specifies the array variable to be partitioned.

Factor = <integer>: Specifies the number of smaller arrays that are to be created.

Dim = <integer>: Specifies which dimension of a multi-dimensional array to partition.

3.3.1.4 Loop pipeline

Pipelining is another directive to increase the performance of function by using the concept of concurrency. In pipelining, the next iteration started before the completion of current iteration. Pipelining divides the execution of operations. Each execution in a processor divided into 5 stages, Instruction Fetch (IF), Instruction Decode (ID), Operand fetch (OF), Execute (EX), and Write Back (WB).This one requires buffering between each pair of consecutive stages with a delay. Since in pipelining one instruction will fetched at one cycle, the average cycle per instruction is equal to 1.

The command we are using in VHLS to optimize the loop with the help of pipelining concept is:

#pragma HLS pipeline initiation interval (II) =<integer> [8].

Loop will process new inputs every N clock cycle, where N is the initiation interval. Initiation interval is the number of clock cycles between the start times of consecutive loop iterations. The default initiation interval (II) is 1.

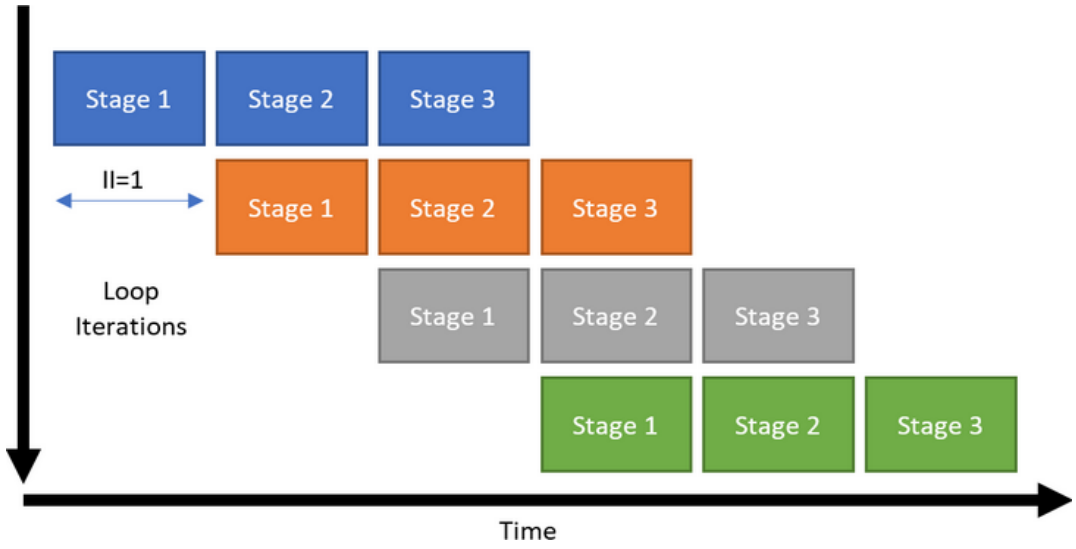

Figure 3.9: Loop pipelining [4]

Following is state diagram for pipelining that shows the multi-processing steps:

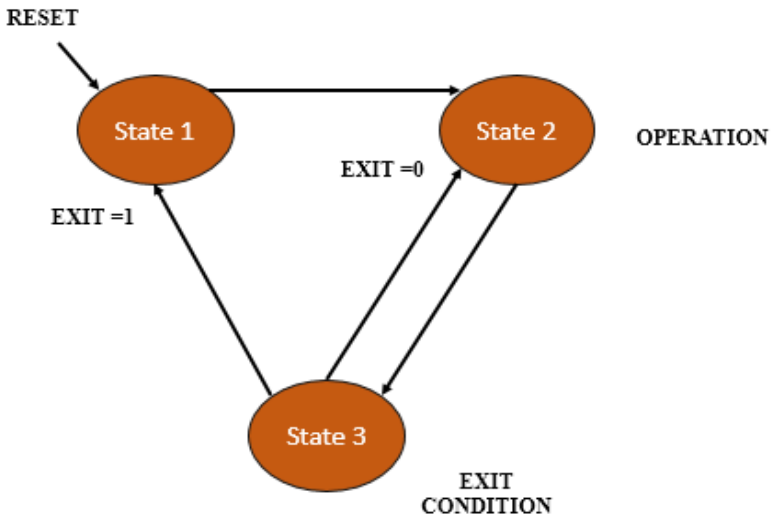

Figure 3.10: State diagram of FSM of integer number addition using pipelining

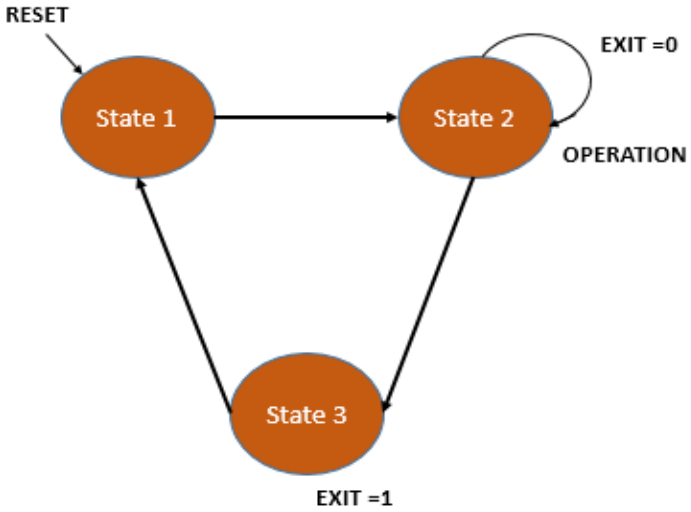

Figure 3.11: State diagram of FSM of real number addition using pipelining

3.3.2 Matrix multiplication

Following is the C program of matrix multiplication:

```
#include<stdio.h>
Void matrixmul (
A [MAT_A_ROWS] [MAT_A_COLS],
B [MAT_B_ROWS] [MAT_B_COLS],
C [MAT_A_ROWS] [MAT_B_COLS])
{
Row: for (Int I = 0; I < MAT_A_ROWS; I ++)
{
Col: for (Int j = 0; j < MAT_B_COLS; j++)
```

```
                {
                C [i] [j] = 0;
Product: for (Int k = 0; k < MAT_B_ROWS; k++)
{
C [i] [j] += A [i] [k] * B[k][j];
}
}
}
}
```

In order to achieve a parallel matrix multiplication, the data should be read in parallel according to the computing processes of matrix multiplication. As the row size of matrix A equals to column size of matrix B, the data from them can be read simultaneously to perform the multiply-accumulation operation and calculate the corresponding elements of matrix C. If the number of the column of matrix A is 3, the sequence of the data read from matrices A and B from clock cycle 1 to 3, then these data will be fed into accumulator in this order. To achieve the high speed matrix multiplication each row of matrix A and each column of matrix B in memory A array and memory B array should be stored, respectively.

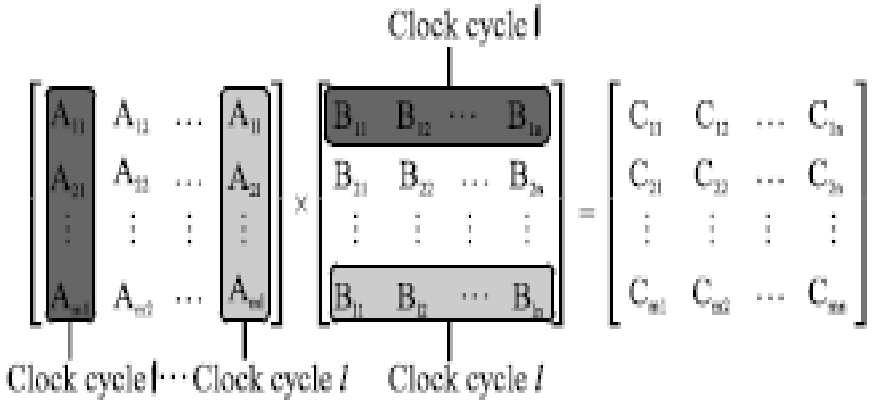

Figure 3.12: Reading process of data sequence [20]

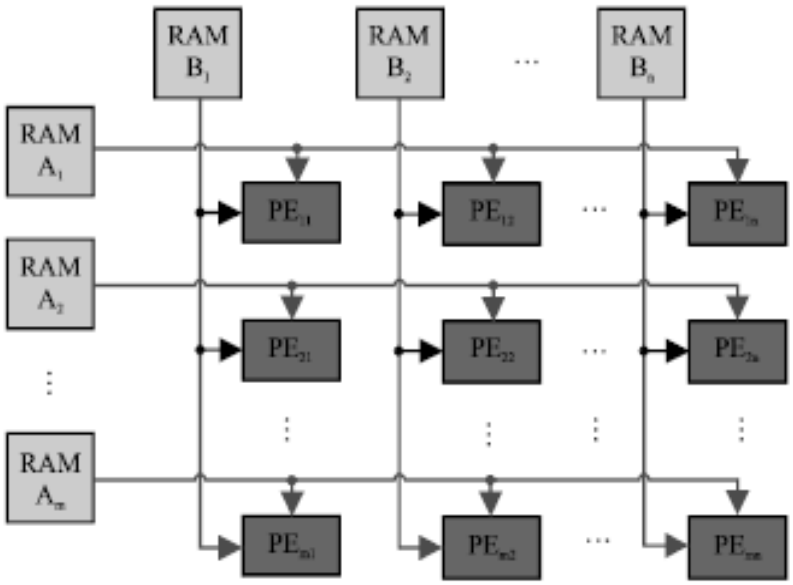

Figure 3.13: Parallel architecture of Matrix multiplication [20]

Since there is no intercommunication between the neighboring Processing Elements (PEs), they complete their own computation task independently, so the architecture can be applied to matrix multiplication with any size.

3.4 Design specifications

We used C language to implement both conventional and proposed architecture. Various code implementations were synthesized using Vivado High-Level Synthesis development tools. The target FPGA is Artix-7 xc7a35tcpg236-2 chip.

Chapter 4

RESULT AND DISCUSSION

4. Result and discussion

Upon applying directives, namely loop unrolling and pipelining, there is a significant improvement in the latency of the execution of addition and matrix multiplication. The work carried out was first simulated to determine the clock timings and then later on implemented through the Vivado software. The results presented in this chapter contain the simulation results first followed by reports that highlight the latency and the hardware resources used. We present the simulation results with hardware realization of integer and Floating point addition. The following report shows the improvement in latency and increase in hardware part of integer addition followed by floating point addition and comparison of these two reports for different directives.

4.1 Integer Addition

4.1.1 Sequential process

1. Simulation report

To start with, a 'for loop' addition of 8 different numbers is proposed. The C program is written in HLS and then synthesized. The synthesized code would contain the detailed codes available in the hardware description language. Figure 4.1 provided below gives an overview of the clock cycles involved in the process of addition. Since the FPGA board in place contains 100MHz clock, a 10 ns clock is provided and a latency of 17 clock cycles is obtained as shown in the report below (Figure 4.2).

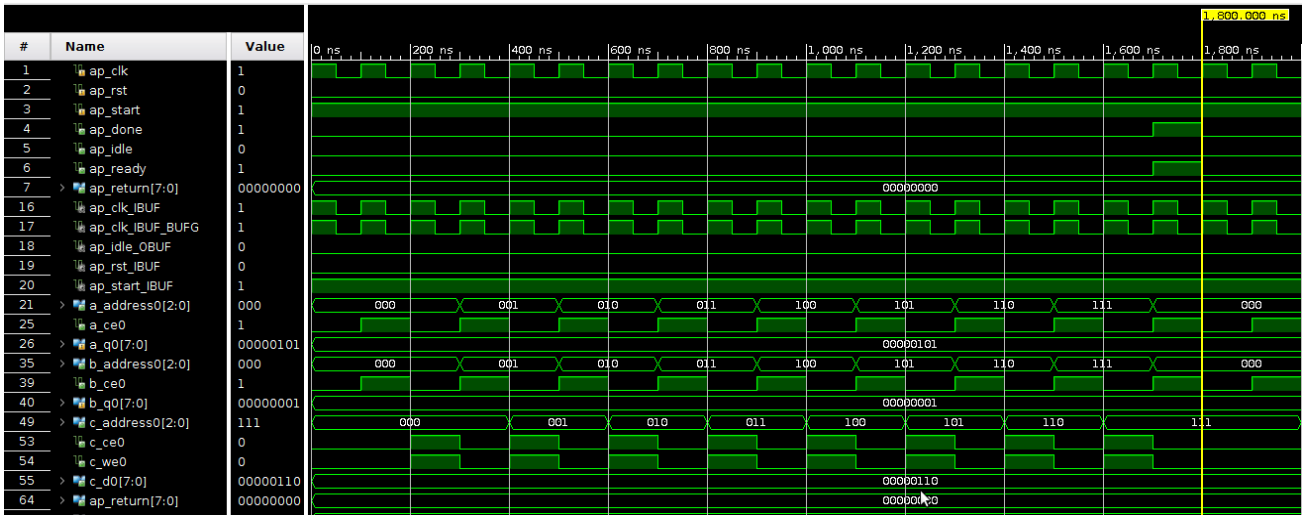

Figure 4.1 Simulation results of integer addition using sequential process

2. Latency report

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	5.39	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
17	17	18	18	none

Figure 4.2: Latency report of integer addition using sequential process

3. Hardware utilization report

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	118	48
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	30
Register	-	-	15	-
Total	0	0	133	78
Available	100	90	41600	20800
Utilization (%)	0	0	~0	~0

Figure 4.3: Resources utilization for integer addition using sequential process

4. RTL design

The RTL design contains the different hardware elements realized in the FPGA for the logic written in the C program in HLS. Figure 4.3 provided the estimated utilization of resources. It can be noticed that every loop condition is checked through equating symbols and the use of registers

are used to store the different variables in binary form. RTL design of the integer addition by sequential processing shown as figure 4.4.

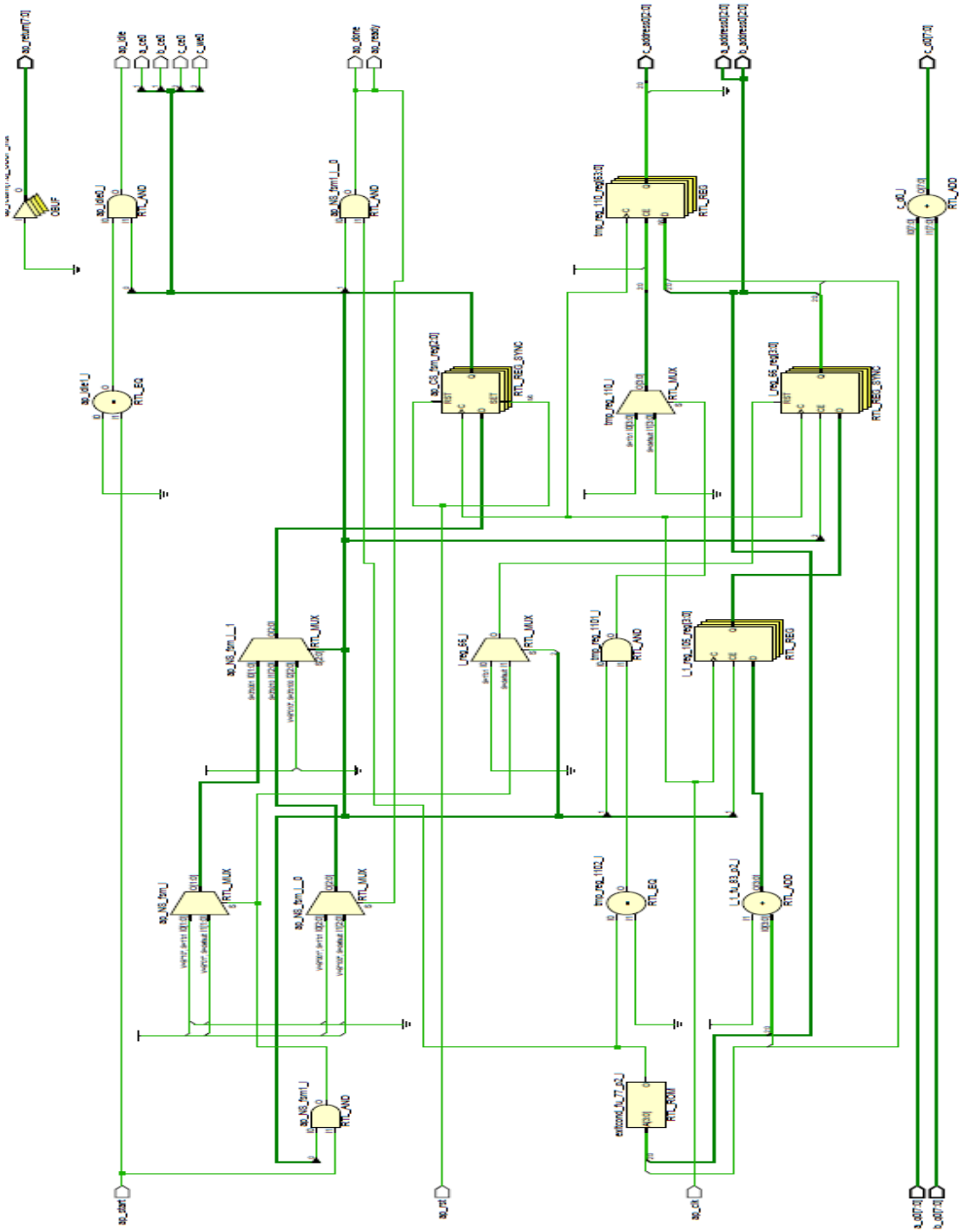

Figure 4.4: RTL design of integer addition using sequential process

4.1.2 Loop pipeline directive on integer addition

To start with, a for loop addition of 8 different numbers is proposed here with pipeline directive initialized in the program. The C program is written in HLS and then synthesized. The synthesized code would contain the detailed codes available in the hardware description language. Figure 4.5 provided below gives an overview of the clock cycles involved in the process of addition. Since the FPGA board in place contains 100MHz clock, a 10 ns clock is provided and a latency of 10 clock cycles is obtained as shown in the report below (figure 4.6). Hence, there is a reduction in clock cycle from 17 clock cycles to 10 clock cycles for the execution of the task. Figure 4.7 provided below estimated utilization of resources report by pipeline process for addition of integer number.

1. Simulation report

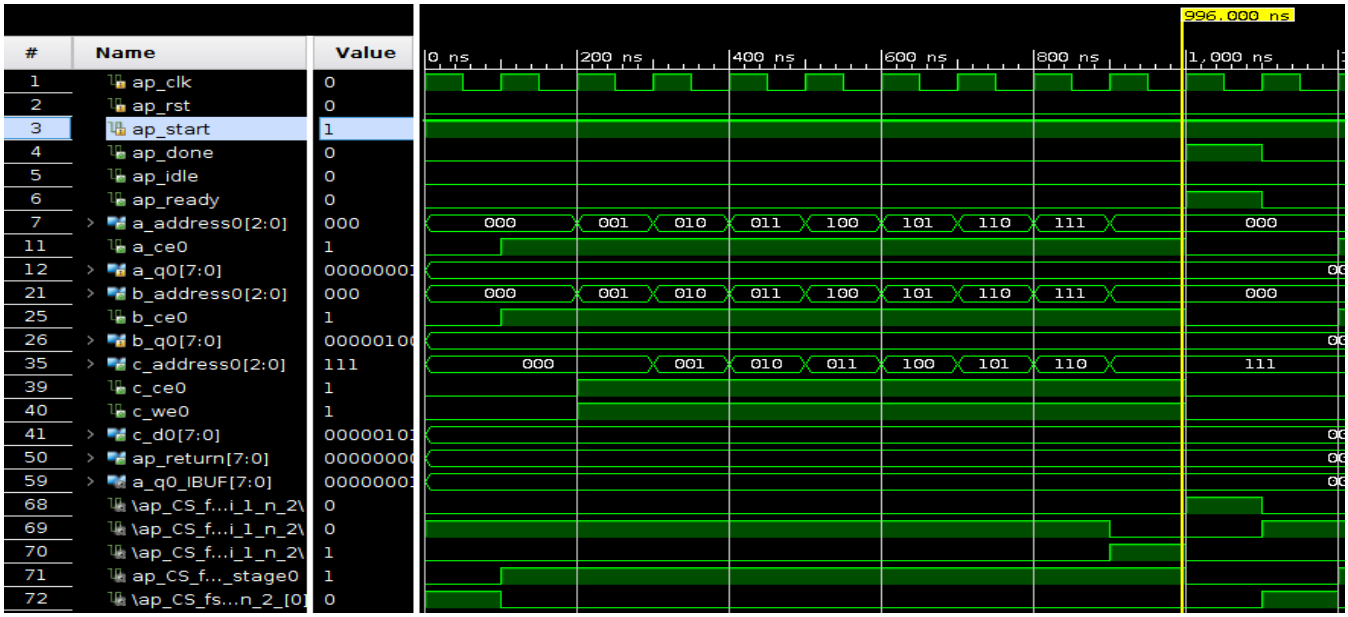

Figure 4.5: Simulation results of integer addition after applying pipeline directive

2. Latency report

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	5.39	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
10	10	11	11	none

Figure 4.6: Latency report of integer addition using pipelining

3. Hardware utilization report

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	118	52
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	45
Register	-	-	14	-
Total	0	0	132	97
Available	100	90	41600	20800
Utilization (%)	0	0	~0	~0

Figure 4.7: Resources utilization for integer addition using pipelining

3. RTL design

The RTL design shows the estimated utilization of hardware for the addition where the latency reduced compared to sequential as shown in figure 4.8

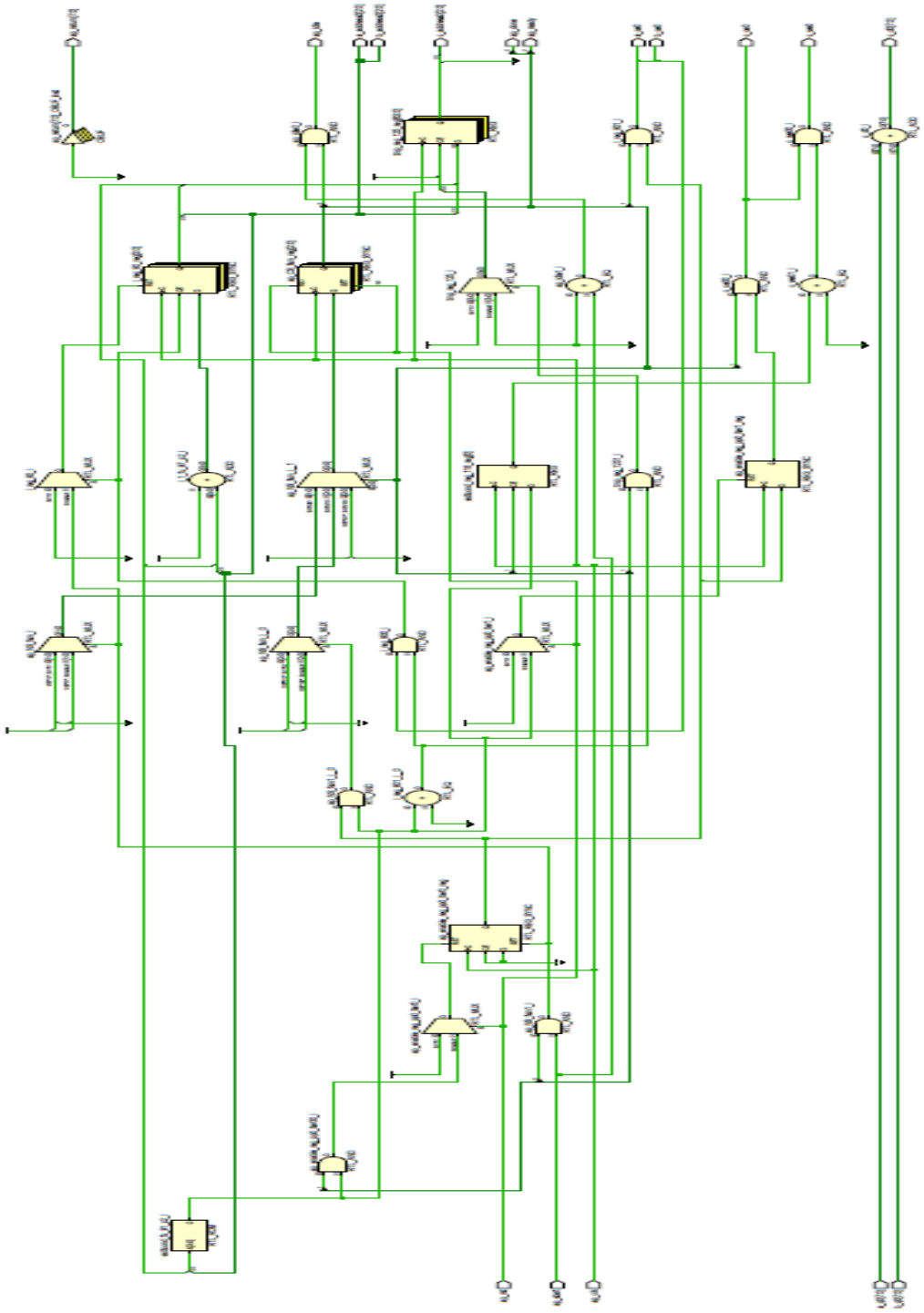

Figure 4.8: RTL design of integer addition using pipelining

4.1.3 Loop Unroll directive on integer addition

Loop unrolling is a process where copies of the code inside a loop are made such that the hardware can perform all these operations in a parallel manner in the FPGA. Figure 4.9 provided below gives an overview of the clock cycles involved in the process of addition. Upon applying loop unrolling in the addition program, the latency report shows (figure 4.10) 1 clock cycle for performing the operations. The RTL design shows the hardware involved for the same where the for loop checking has been dissolved, but the hardware implementation has increased.

1. Simulation report

Figure 4.9: Simulation results of integer addition after applying unrolling directive

2. Latency report

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	5.39	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1	1	2	2	none

Figure 4.10: Latency report of integer addition after applying unrolling

3. Hardware utilization report

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	808	296
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	15
Register	-	-	2	-
Total	0	0	810	311
Available	100	90	41600	20800
Utilization (%)	0	0	1	1

Figure 4.11: Resources utilization for integer addition after applying unrolling

4. RTL design

Since the loop unrolling completely makes the hardware replicate for the operations, many of the multiplexers used for “for loop” condition checking have been removed from the RTL design shown below figure 4.12.

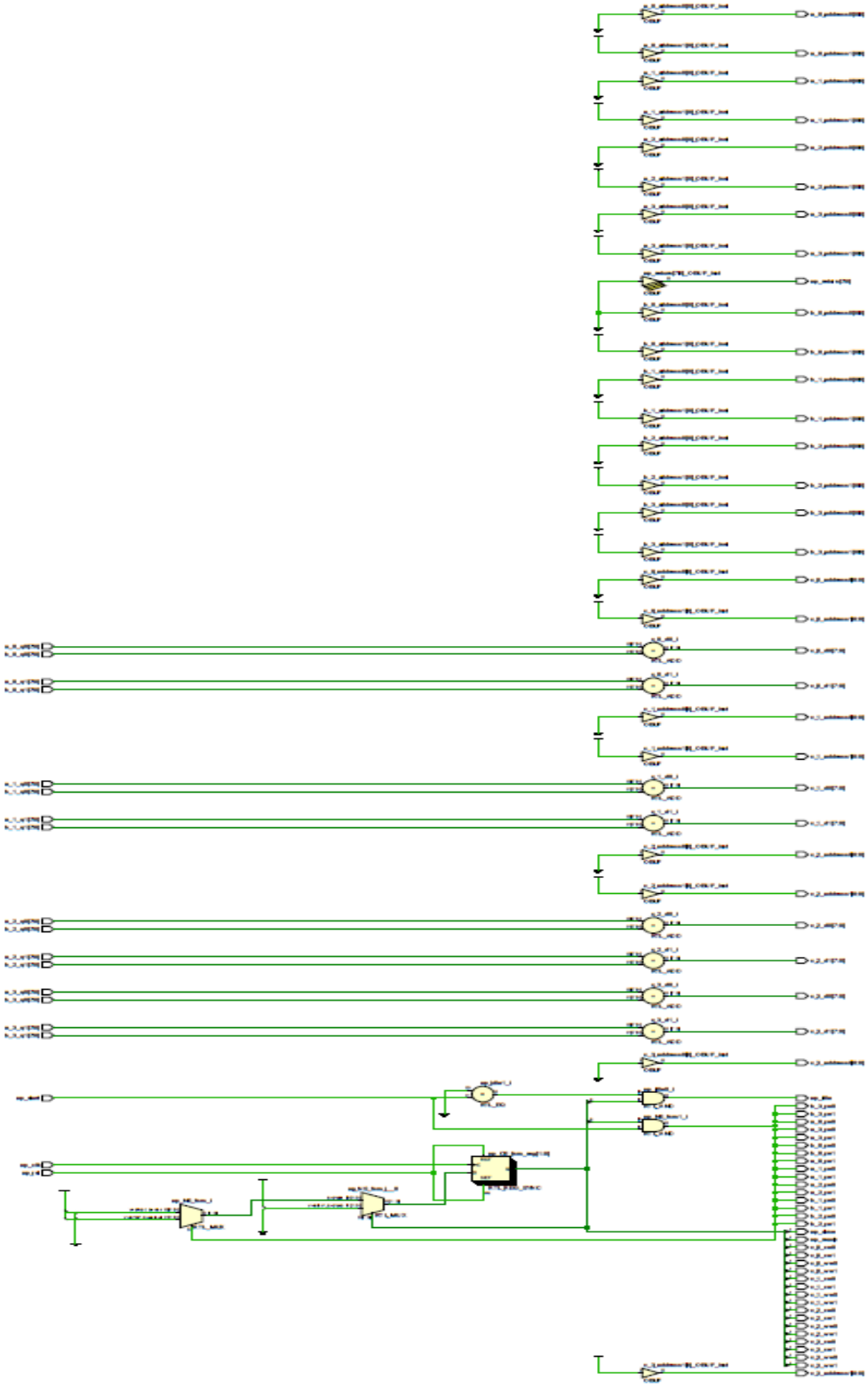

Figure 4.12: RTL design of integer addition after applying unrolling

4.1.4 Comparison of performance of directives for integer number addition

The report below shows how the performance of the sequential loop changes while applying to directives. It can be clearly seen that the latency has been reduced to 1 clock cycle, obviously with an increase in hardware resources such as LUTs and I/O ports.

PARAMETERS	Sequential	Pipeline	Unroll
Loop latency (cycle)	17	10	1
LUTs	78	97	117
I/O ports	280	367	487

4.2 Comparison of performance of directives for real number addition

While the addition has been successful for integer numbers, often the programs requiring high performance computing, would be using floating point number representation of decimals. Hence the same procedure is repeated for 8 loop addition for floating point number representation. The hardware resources increase and the time taken for the task completion also increases because of increase in the number of bits. A comparison of the three directives on the floating point addition is provided below for reference.

PARAMETER	Sequential	Pipeline	Unroll
Latency	65	14	3
LUTs	272	326	1771
I/O ports	104	205	708
DSP48E	2	2	16

Chapter 5

SUMMARY AND FUTURE SCOPE

5. Summary and future scope

In summary, the exploration of utilizing FPGA for parallel processing of computing operations were performed in this project. Two mathematical operations namely addition (integer and real numbers with a looping of 8 times) and matrix multiplication (integer) were investigated in this project.

Two main directives were applied onto the program taken for consideration. They were loop pipelining and loop unrolling. The definitions of these two directives were explained and the simulations were carried out to ascertain whether they were in order. Subsequently, the hardware implementation of the same in FPGA has been completed for the addition program. Matrix multiplication has also been explored on the same lines. The software analysis of matrix multiplication has been performed. The hardware realization of the same would be the future work. Real number matrix multiplication would be performed using directives and the hardware realization would be ascertained. This would form the future work of this project.

REFERENCES

References

[1] J.F. Li, "*Basic structure of computers*," Dept. Elect. Eng., National Central Univ., Jungli, Taiwan.

[2] O. Johnson and O. Dinyo, *"Comparative analysis of single core and multicore systems*," International Journal of Computer Science & Information Technology (IJCSIT), (vol. 7) no. 6, Dec. 2015.

[3] *UG998, "Introduction to FPGA design with Vivado High-Level Synthesis," v2013.1.*

[4] S. Habinc, et al. "*Lesson learned from FPGA development*," Sweden, Gaisler Research, Tech.Rep. TR.001 (01)-2, Sept. 2002.

[5] B. Zeidman, "*Introduction to FPGA design,* "Embedded system conference Europe.1999, 304-314.

[6] A. Al-Mohmood and M.O. Agyeman,"*A study of FPGA based system on chip designs for real time industrial application,*" IJCA (0975-8887)-163, April 2017.

[7] CLB <FPGA Editor> Xilinx.

[8] Diligent-Basys 3™ FPGA Board Reference Manual.

[9] J.A. Abranam,"*High-Level Synthesis,"* SoC Design, EE382v, July 2009.

[10] Vivado HLS design flow lab.

[11] SDAccel Environment Reference Guide.

[12] https://www.stat.berkeley.edu/~nolan/stat133/Spr04/chapters/representations.pdf

[13] *ANSI/IEEE, IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-2008. IEEE-754.*

[14] J. Hrica, *"Floating point Design with Vivado HLS,"* Xilinx, XAPP599 (v1.0) Sept. 2012.

[15] *UG902, Vivado Design Suit User Guide: High-Level Synthesis v2012.2.*