ACKNOWLEDGEMENT

This M.Tech thesis required a lot of support to get completed, and luckily, I got so much help during my project work. I am glad and proud to complete my M.Tech at the Indian Institute of Technology, Indore.

First of all, I would like to express the most profound appreciation to my thesis supervisor **Dr. Srivathsan Vasudevan,** Associate Professor, Electrical Engineering, IIT Indore for his patience and faith in me, as he continually and convincingly supported me to move forward day by day. He guided me well for the project.

I would like to thank the Discipline of Electrical Engineering for providing all the facilities and resources required for the completion of this work.

Furthermore, I would like to express my sincere gratitude to my PSPC members **Dr.Satya S. Bulusu**, IIT Indore, and **Dr. Amod C. Umarikar**, IIT Indore, for their valuable suggestions and feedback at every point.

Besides, I would not forget **Miss Kritika Bhardwaj** (JRF), IIT Indore for investing all her valuable time with me. Finally, I owe a lot to my family, friends, and classmates for always encouraging and supporting me to stay motivated and focused on completing the project.

SARIKA ATHYA

Dedicated to my Family and friends

Abstract

Usually, the extensive calculations done in real life like weather forecasting, science-related massive calculations take a lot of time, due to their complexity. Here in this project, as a preliminary example, we have tried to reduce the complexity as well as reduce clock cycles and calculation time, also to reduce the hardware if needed. This project is just an attempt to do so, by simply checking it on matrix multiplication and using directives to reduce clock cycles and equipment used, namely pipelining and unrolling. Vivado High-Level Synthesis (HLS) is being used for the simulation and creation of IP, while Vivado HLx for creating a block diagram by using the same IP which we produced at HLS. Further getting the multiplied output at Xilinx SDK, this has been used for the checking of the circuit created with its calculations, timing and other parameters, so that we can implement it physically later, only if the parameters match our requirement and modify it if needed. The board used in the project is the ZYBO board with a dual core arm cortex A9 processor present in it.

Keywords- Complexity reduction, HLS, HLx, SDK, Zynq-7000

TABLE OF CONTENTS

List of figures

List of flowcharts

List of tables

List of Abbreviations

ACP	Accelerator Coherency Port
ADC	Analog to Digital Converter
ALU	Arithmetic and Logic Unit
AMBA	Advanced Microcontroller Bus Architecture
API	Application Program Interface
ARM	Advanced Reduced instruction set computer Machines
ASIC	Application Specific Integrated Circuits
AXI	Advanced Extensible Interface
BRAM	Block Random Access Memory
BSP	Board Support Package
CDT	C/C++ Development Toolkit
CLB	Configurable Logic Block
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
DDR	Double Data Rate
DMA	Direct Memory Access
DSP	Digital Signal Processing
EBR	Embedded Block RAM
EEPROM	Electrically Erasable Programmable Read Only Memory
FIFO	First In First Out
FPGA	Field Programmable Gate Array
GHz	Gigahertz
GPIO	General Purpose Input Output
HDL	Hardware Description Language
HLS	High-Level Synthesis
IDE	Integrated Development Environment
IP	Intellectual Property

IO	Input Output
JTAG	Joint Test Action Group
KB	Kilobyte
LED	Light Emitting Diode
LMB	Local Memory Bus
LUT	Look Up Table
MIO	Multiplexed Input Output
MUX	Multiplexing
NAS	Network Attached Storage
NGC	Native Generic Circuit
NGD	Native Generic Database
OCM	On-Chip Memory
OTG	On The Go
PL	Programmable Logic
PLB	Processor Local Bus
PLD	Programmable Logic Devices
PROM	Programmable Read Only Memory
PS	Processing System
RAM	Random Access Memory
RTL	Register Transfer Level
SAN	Storage Area Network
SD	Secure Digital
SDIO	Secure Digital Input Output
SDK	Software Development Kit
SoC	System on Chip
SRAM	Static Random-Access Memory
UART	Universal Asynchronous Receiver Transmitter
UCF	User Constraints File
USB	Universal Serial Bus

VHDL	Very high speed integrated circuit Hardware Description Language
XADC	Xilinx Analog to Digital Converter
XSCT	Xilinx Software Command Line Tool
ZYBO	ZYnq Board

Chapter 1
INTRODUCTION

1. INTRODUCTION

1.1 Background

Complexity in computation is a significant issue these days. There is always a need to reduce the complexity saving the time needed for the calculations used by machines. Dynamic programming may be among the causes for the complexity in the devices, to get the final output; several times, temporary data is used [1]. For doing the mentioned task, it requires a massive amount of memory space in the machine.

With the advancement in technology, we need to use lesser hardware and also speeds up the calculations. The computational techniques used up till now, need improvement. The calculations which were usually taking much time can now be done parallelly, with lesser time.

1.2 Motivation

CPUs are used for general processing, contain all in one type of processor, due to which they are slow as they perform many tasks at a time. But this may not be desired for all kind of applications. Also, we need faster processors these days to save time. However, it can perform all the tasks but not with the appropriate performance [1].CPU contains only one ALU, for performing logical tasks.

To speed up our CPU, we can either use faster circuits or arrange the hardware to perform more tasks at one time.

With advancement, there are designs of the multi-core processor. Now we can perform many logical tasks at a time. For example, our application needs ten multiplications at one time; we can do it by increasing the number of ALU.FPGA is one such device which we have used in this project.

1.3 Objective

In the project, we aim to take up a multiplication function and

- Take two matrices and get their multiplied output
- Know about the latency of the calculation
- Comparison of applying different directives and without their application
- Testing for the multiplication output using different values
- Getting the correct output using the FPGA board

Here, it is proposed to get production from an FPGA board for a 2x2 matrix multiplication. We are performing this project, for the testing of connections between Vivado HLS and the FPGA board used. Vivado HLS is used for the creation of IP and application of the directives. It uses high-level languages (C, C++, SystemC) and converts these languages to Hardware Description Languages (HDL). Vivado HLx has been used for the connections between HLS code and FPGA, while Xilinx SDK for the testing purpose.

We can relate FPGA and CPU as we are using both the devices for logical tasks, but FPGA is different as it is not a processor itself, does not run a stored program, but we can reconfigure. We can implement any logic needed and is not predefined or stored in it. An FPGA features may vary as per their vendors and models, it typically contains IO banks, clock manager, etc. As the FPGA contains the element so that it can perform any function, and can modify as per the need of the customer. This feature makes it suitable for the user to make use as per the requirement and application.

1.4 Organization of the thesis

This section contains a description of how we have organized the thesis. It mainly has the following five chapters:

Chapter 1 provides the background, motivation, and goal of the project.

Chapter 2 contains the introduction to FPGA, Vivado HLS, directives, Vivado HLx, and Xilinx

SDK.

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

Chapter 4 provides the results and discussion.

Chapter 5 presents the project summary and conclusion.

Chapter 2
LITERATURE SURVEY

2. LITERATURE SURVEY

2.1 SoC

An SoC is the one which contains all the elements required in a computer or any other electronic circuit. Parts included may be for storage, data processing, RAM, ROM, or input-output ports [2]. We can say that SoCs have replaced microcontrollers. In layman's language, SoCs are the chips which integrate all the required for a circuit on a single chip. SoC contains microprocessors, a system for communication for contacting between its modules. SoC is mainly present in devices which are small but complex [3]. It includes analog, digital, and mixed functions. SoC is the one stop for both software and hardware components in a single chip. It contains all the elements required for a complete circuit [4].

SoC requires less space, less power with excellent performance. It helps us avoid the need for multiple chips as it provides all the needed elements on one chip [5]. It is present on mobile phones, cameras, medical technology, etc

.

2.2 Field Programmable Gate Arrays (FPGAs)

Field Programmable Gate Arrays (FPGAs) consists of configurable logic blocks (CLBs) which are networked and the connection between them is made out via interconnects which are programmable. FPGAs have the feature to be reconfigured as per the needs of the consumer's application even after its manufacturing [6]. And this is the feature that makes it stand out from Application Specific Integrated Circuits (ASICs), which are manufactured only for particular tasks. However, FPGAs, which we cannot reprogram, is also available, but the one which we widely use is with the feature to be able to modify as per the user application.

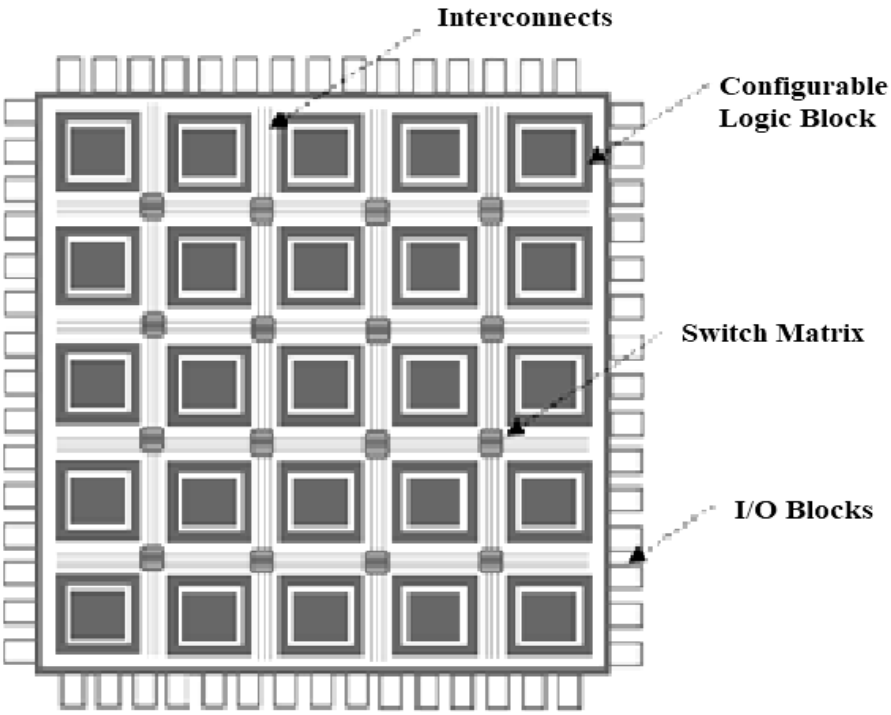


Figure 2.1: The basic architecture of FPGA [7]

We can casually say that FPGAs are chips which can be programmed to check or work on the digital designs. Advanced FPGAs have approximately 3, 30,000 logic blocks and around 1,100 inputs and outputs.

The FPGA industry has been emerged out through programmable logic devices (PLD) and the programmable read-only memory (PROM)[6]. But the FPGA allows doing the programming by the consumer itself and does not need PLD and PROM to be wired during the establishment of the board, and after we have produced the product, this feature allows the consumer to fix any bugs or add things according to the requirement.

The emergence of FPGA has brought a significant change and advancement in the industry. Earlier FPGAs were limited to networking and communication; it slowly moved to automotive,

manufacturing, day to day life and for every kind of electronic devices as well.

2.2.1 FPGA Architecture

The architecture of an FPGA is dependent on static random-access memory (SRAM), which is volatile. Once the FPGA is switched off, the data is lost. To store the data, we need to externally add electrically erasable programmable read-only memory (EEPROM) [7].

We can turn the architecture of FPGA into any hardware circuit as per the need. To turn basic logic circuits to complex architectures, we can make use of a configurable logic block (CLB).

In FPGA architecture **Configurable Logic Block (CLB)** location is shown in the figure below.

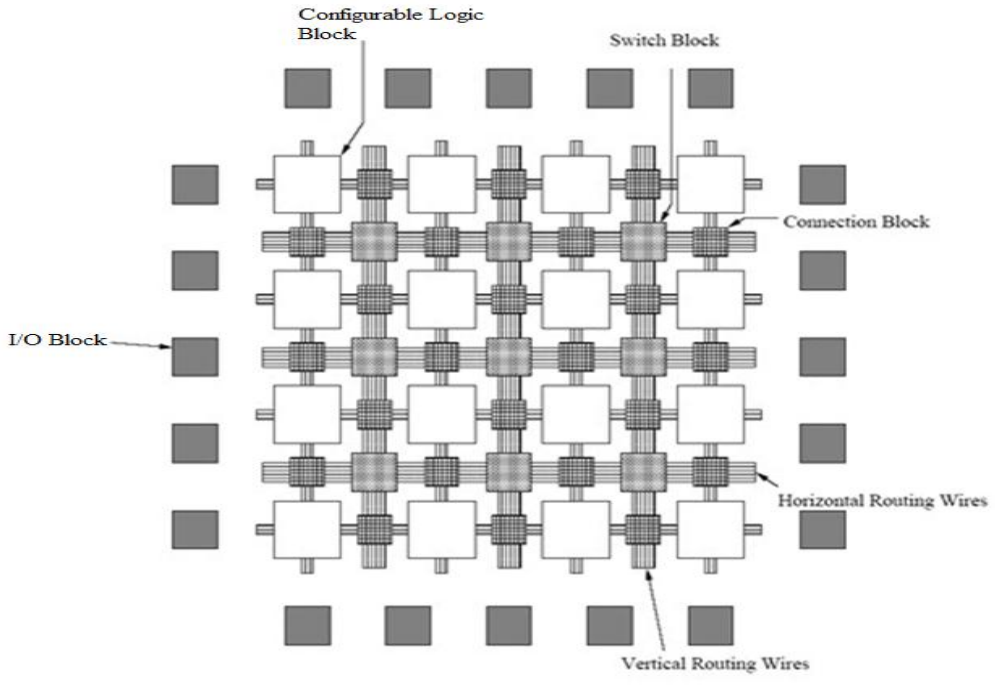


Figure 2.2: Detailed FPGA Architecture [8]

We can also make a whole new SoC using FPGA. It can be used to program hardware architecture as per our needs. This is not possible in case of a microprocessor.

2.2.2 FPGA Architecture Features

FPGA Architecture consists of features such as

- Configurable Logic Block (CLB),for logic functions
- I/O Block, for external connections
- Switching Matrix Interconnects, for internal connections

2.2.2.1 Configurable Logic Block

The programmable/configurable logic block helps with calculations and storing of data. An ideal one consists of a flip-flop, combinational logic, and logic to reduce area as well as cost.
Advanced FPGAs consists of a combination of different kind of blocks such as multiplexers, BRAM, etc. We use configuration memory for the controlling of functions of each element [7].
The input Lookup tables (LUTs) of CLB are used to implement the functionalities such as

- Combinational Logic design
- Distributed RAM
- Shift Register

We can find more circuits inside FPGA, like Multiplier, Block RAM, and many more.

CLBs have two slices, which we further divide into two logic elements.

Logic elements have

- Four input lookup Table
- Full Adder and Mux logic
- D Flip Flop

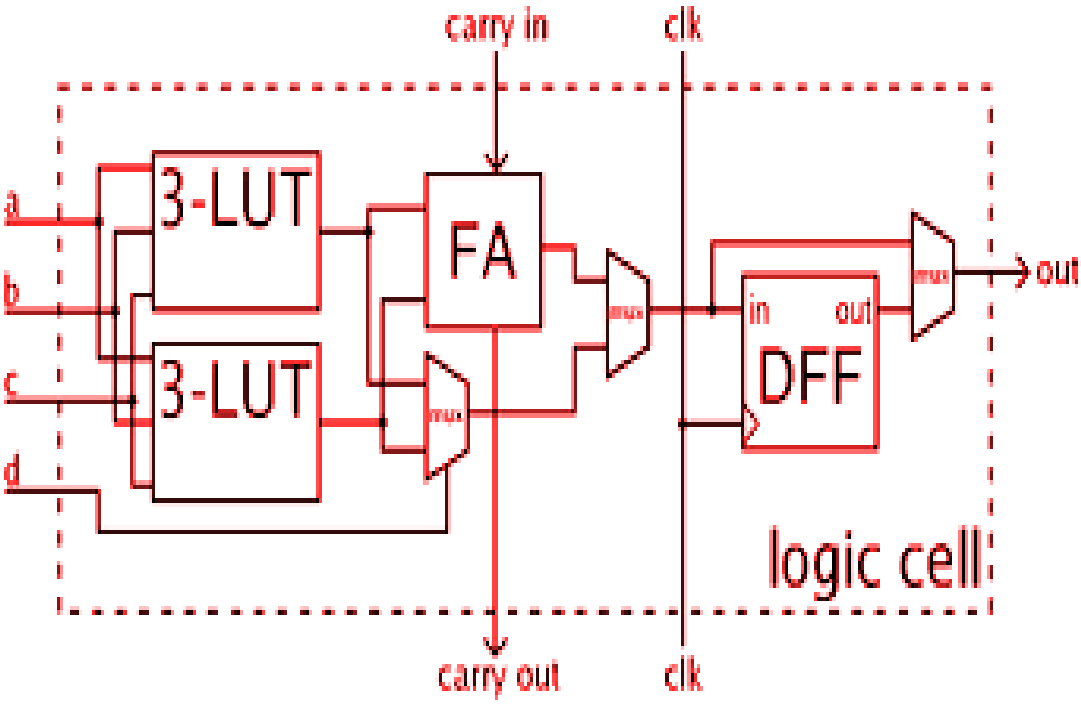


Figure 2.3: Configurable Logic Block [9]

We use Multiplier Block for the implementation of the dedicated 18×18 multipliers with signed and unsigned operations.
Block RAM is a dedicated memory that implements dual port 16kb memory.
In architectures such as zynq from Xilinx, dual-core arm cortex A9 Processor is present inside for high-performance implementation [7].

2.2.2.2 Programmable I/O

We can use the programmable I/O blocks for external connections. The I/O pad and the circuit around it together form an I/O cell.

This is a large area of an FPGA. Also, the design is complicated as the voltage applied, and the reference voltages both are different.

Selecting a vast number of standards for its design could increase the area of the chip.

With development, the architecture of FPGA now contains several programmable blocks like BRAM, multiplexers, DSP-48. Also, we attach the microprocessors to the FPGA as per its requirement [10].

2.2.2.3 Programmable Routing/Interconnects

The programmable routing is used to complete the desired circuit by connecting different blocks altogether. It has multiplexers pass transistors and tri-state buffers, used for connecting logic elements.

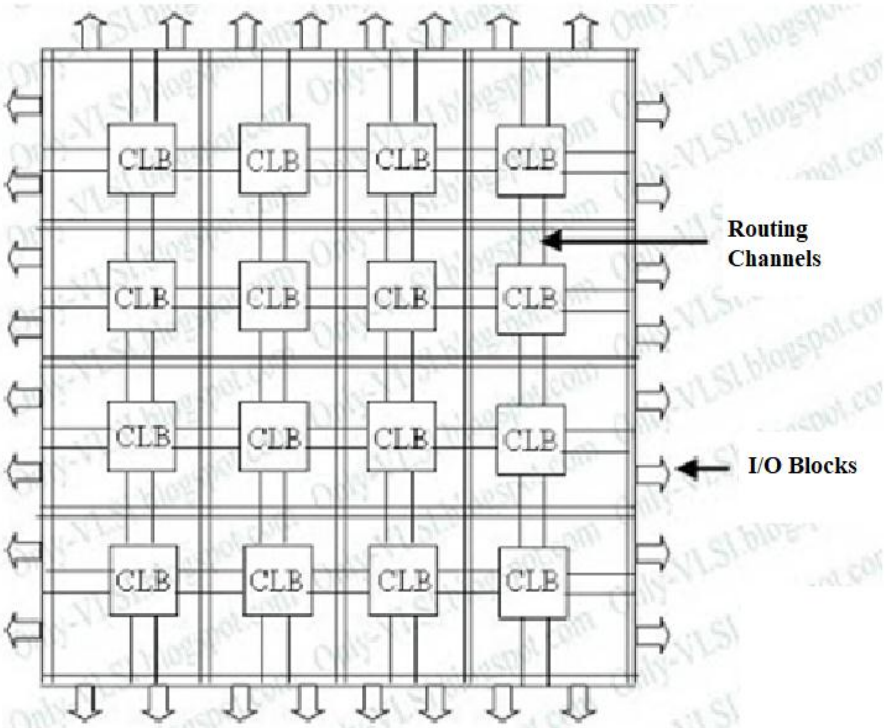


Figure 2.4: FPGA with routing channels [11]

2.2.3 FPGA Architecture Design Flow

FPGA Architecture design consists of design entry, design synthesis, design implementation, programming of the device, and verification of the design.

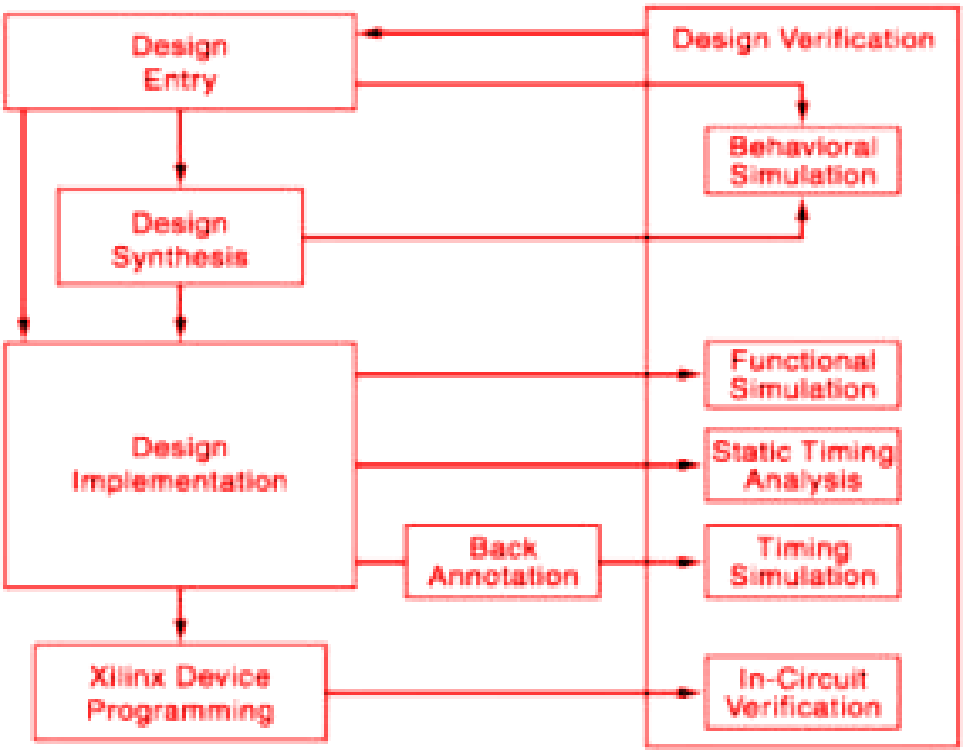


Figure 2.5: FPGA Architecture Design Flow [10]

2.2.3.1 Design verification

This includes verification of the functions and timing done during design flow. Simulation is being carried out in this step like functional, time-related simulation and also the behavioral, user-defined simulation [10].

2.2.3.2 Design Entry

The design entry can be with schematic or HDL or both. If the concept is about hardware, then the entry can be done by the schematic, and if

it is an algorithm, then HDL may be a good option. Because the schematic gives a more realistic view to the user, it is preferable [10].

2.2.3.3 Design Synthesis

Here in the design synthesis, the VHDL code converts to a circuit with logic elements. It checks for the syntax and then goes to the architecture. The architecture created is optimized, and this is called the netlist, and the file saves as a Native Generic Circuit (NGC) file [10].

2.2.3.4 Design Implementation

The design implementation needs to

- Translate
- Map
- Place and Route

Translate

Here the files are converted to NGD (Native Generic Database) file. To the ports of the architecture created, we can give physical ports like switches, LEDs, etc. and then this information is saved in User Constraints File (UCF) [10].

Map

Here the circuit is divided so that we can see it as an FPGA logic blocks. We can map it from NGD to CLBs, I/O blocks, and then an NCD file is being created. Ultimately the design is mapped to FPGA [10].

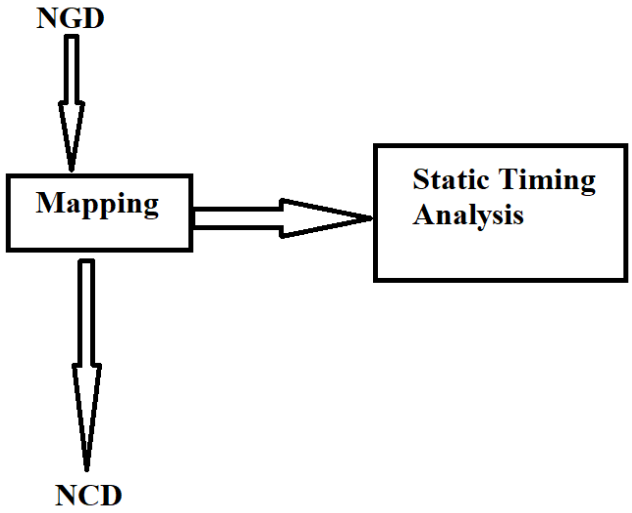


Figure 2.6: Mapping for the design implementation of FPGA [10]

Routing

This puts the divided circuit to logic blocks as per the constraints and connections are being made for the logic blocks [10].

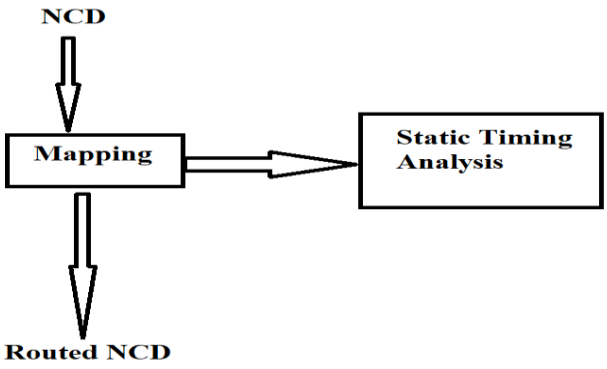


Figure 2.7: Routing for the design implementation of FPGA [10]

2.2.3.5 Device Programming

Once we have created a design as per the components of FPGA, now it must be made compatible with the FPGA. The NCD file is handed over to the BITGEN program, to create a BIT file, so that we can implement it to an FPGA [10].

2.2.3.6 Design Verification

Stages at which verification is done are;

1. Behavioral Simulation (RTL Simulation)

This is the very first step in the hierarchy. This is done to check the RTL code.

Here the breakpoints are being created, and we cross-check the function, the signals and variables are verified [10].

2. Functional Simulation

It is done once the translation simulation completes. It tells about the operation of the circuit [10].

3. Static Timing Simulation

It completes after mapping. We get to know about the path delays. Here we get to see the summary of the timings and delays taken by the design [10].

2.2.4 FPGA Applications

As we know, we can reconfigure FPGA; they are useful in many fields. By leading the industry, Xilinx provides us with FPGA devices, different software, and IP cores, which we can readily use in various areas such as:

- Aerospace & Defense - FPGAs which have the advanced property for processing of images, generation of waves and can tolerate radiation [12].

- ASIC Prototyping -For the verification purpose of software and modeling of the systems (SoC) FPGAs make it precise and faster [12].
- Audio - With lowering engineering costs, increasing flexibility Xilinx FPGAs is beneficial in fields like audio and multimedia [12].
- Automotive -For the help of the driver, increasing the comfort and ease inside the vehicle, IPs have been a great help [12].
- Broadcast & Pro-AV -fastening and increasing the lifespan of the products with Broadcast Targeted Design Platforms and also broadcast systems [12].
- Consumer Electronics -Daily use products have become more advanced with a lesser cost like networking, handsets, wireless systems, flat displays, and many more [12].
- Data Center - FPGA in the data center is helpful in storage, higher bandwidth, and providing value to cloud deployments [12].
- High-Performance Computing and Data Storage - Provides help for different kinds of storage like Network Attached Storage (NAS), Storage Area Network (SAN) and servers [12].
- Industrial -With lowering engineering costs, increasing flexibility Xilinx FPGAs are beneficial for various applications such as imaging equipment, industrial automation, and surveillance [12].
- Medical - The Virtex FPGA and Spartan® FPGA families help display and process purposes, also monitoring or identification of the diseases can be done [12].
- Security -Xilinx provides all the security needs with preventions to control and safety of the areas [12].
- Video & Image Processing -By lowering engineering costs, increasing flexibility Xilinx FPGAs help in the field of processing of videos and images [12].

- Wired Communications -Provides with the solution for the Programmable Networking Line Card Packet Processing, Framer/MAC, and more [12].
- Wireless Communications - Provides help for baseband, connection, and transfer for wireless equipment, addressing standards such as WCDMA, WiMAX, and others [12].

2.3 Complex Programmable Logic Devices (CPLDs)

CPLDs are made up of a few programmable logic arrays (PLAs) and programmable interconnection lines. We use it for the implementation of faster logic. It contains less number of registers [13].

In comparison to FPGA, CPLDs do not contain individual resources like RAM, or to do logical functions like adding or the comparators. CPLD only has the ability for small digital designs, unlike the FPGA, which posses' more significant designs. Even for large inputs CPLDs due to their limiting complexity provide only single chips with faster delays. We can use CPLDs for applications like equipment which are run by battery, where quicker and more prominent decoding is needed, use of power is less, fast switching is required, designs are not significant, and security is also an important aspect. Here in CPLDs, there is a security of the designs to be secured as they can be locked after they are being programmed [13]. Comparatively in FPGA security is an issue as the bitstream has to be loaded with power every time.

In comparison to FPGA, CPLDs are suitable for applications with high power consumption and not in battery operated applications. However, in the newly established FPGAs, it is better. As FPGA can be reconfigured at the user end as well, it is more beneficial than CPLDs, have flexible designs, RAM, the microprocessor on the chip itself, multi-gigabit transceivers and more such benefits [13].

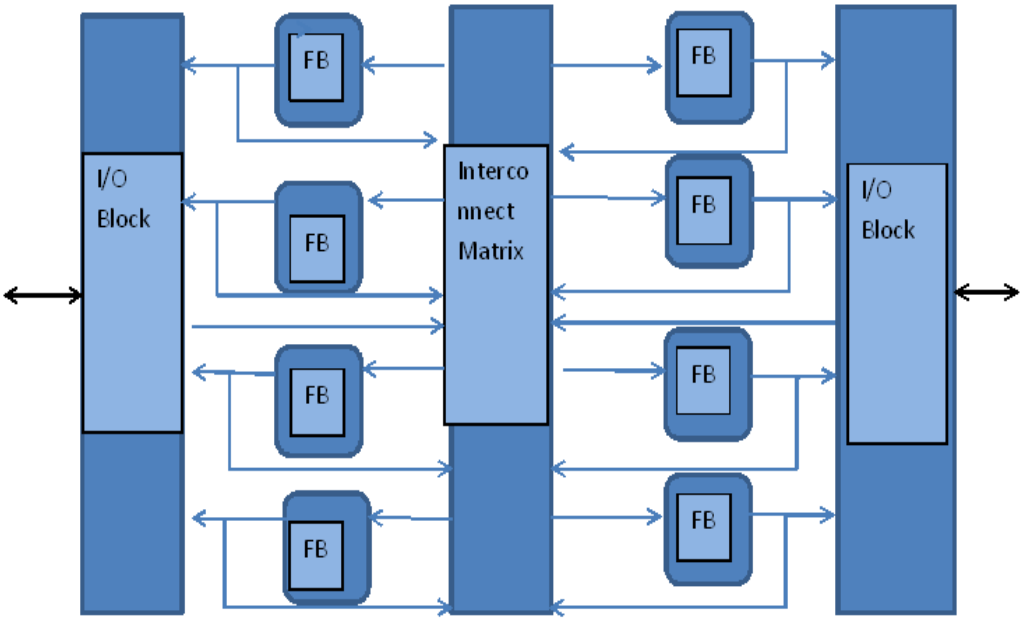


Figure 2.8: CPLD Architecture [14]

As FPGA has more registers and CPLDs have more combinational circuits, they are used for timing and control circuit respectively. Also, the synthesis report for the same code for FPGA gives different timing output every time while for CPLD it is the same. With advancement, the two devices seem to be similar. But the architecture of the CPLD still keeps it different with low cost, a configuration that is not volatile and also the timing characteristics [15].

2.4 BRAM

Block Random Access Memory (Block RAMs/BRAM) is built inside FPGA and can be used to store a massive amount of information. It is a kind of random memory on FPGA used for storing data. We consider it in the FPGA datasheet along with flip flops, LUTs and DSPs. The size of the BRAM depends on the size of the FPGA. It is an essential part of the FPGA.

A Block RAM (embedded memory, or Embedded Block RAM (EBR)), is available in a good number on the FPGA, depending on the

FPGA uniquely. We can use them as per our needs. With more and more designs we make, we get to know about the number of BRAMs we need better [16].

BRAMs have a fixed size like 4/8/16/32 KBS. Their width and depth can be changed. BRAMs are very helpful to store data.

2.4.1 Single Port BRAM Configuration

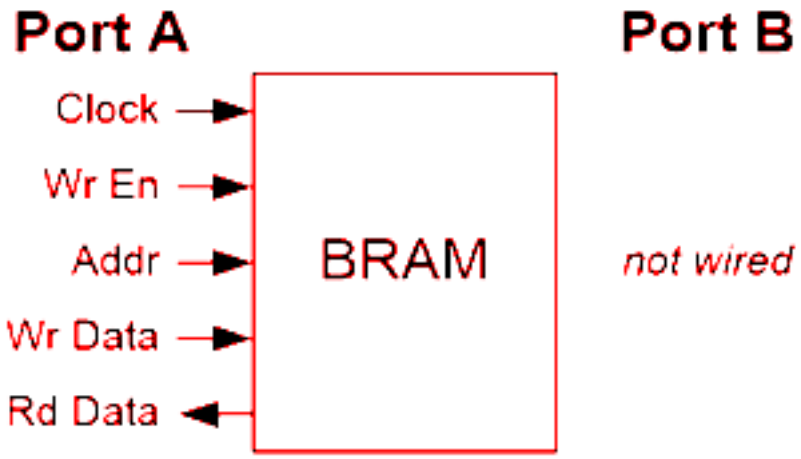


Figure 2.9: Single port BRAM [16]

If there is only one port to receive data, we can use Single Port Block RAM. This is the most straightforward configuration. We use it at places such as, we only need to read, and a fixed value is to be stored.

We can read data only at the positive edge of the clock cycle, and address is being already mentioned, for the time when the write enable signal is not on. We can read data stored when the read data signal is on. We can only read a single data at one clock cycle. So if the BRAM has a depth of 1024, it will have 1024 cycles to read [16].

Here we can only read or write data, and not both at the same time, as there is only a single port. We can write data by enabling the write data port high.

2.4.2 Dual Port BRAM Configuration

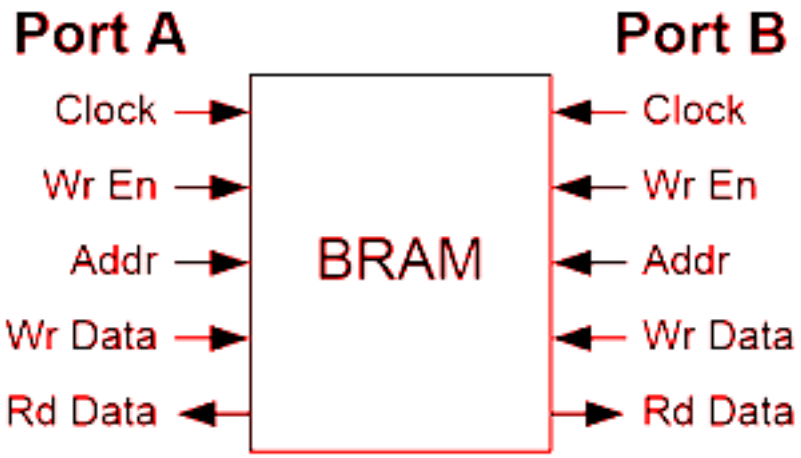


Figure 2.10: Dual port BRAM [16]

The Dual-Port Block RAM (or DPRAM) is the same as a single port only, but here we can read and write at the same time as there are two ports available. We can use any of the two ports to read or write, as both works the same. On the same clock cycle, both the ports can work simultaneously at different addresses. That is we can read from a different address and write it to another address.
This may be used at places like reading data externally or at the time of ADC conversion to store data, or we may use it as a FIFO [16].

We use BRAM for functions such as:

- With the help of local FIFOs, we can transfer data to different clock domains[17]
- With the help of DMA FIFO, we can transfer data between FPGA and any other processor[17]
- It is better than LUTs for storing data[17]

The two ports can be connected to any of the buses independently: LMB (Local Memory Bus), PLB (Processor Local Bus), and OCM (On-Chip Memory). Range of the address, number of the byte write enable define BRAM primitives [18].

2.5 AXI bus

As we have PCI for x86 architectures similarly AXI from AMBA 3 is a bus for ARM SoCs.

Here we are using AXI4, which is of 3 types, all with master and slave modes.

- AXI4 (Full) -memory mapping is needed and has high performance
- AXI4-Lite -we can read and write in the four registers contained
- AXI4-Stream - used to stream data at higher speed

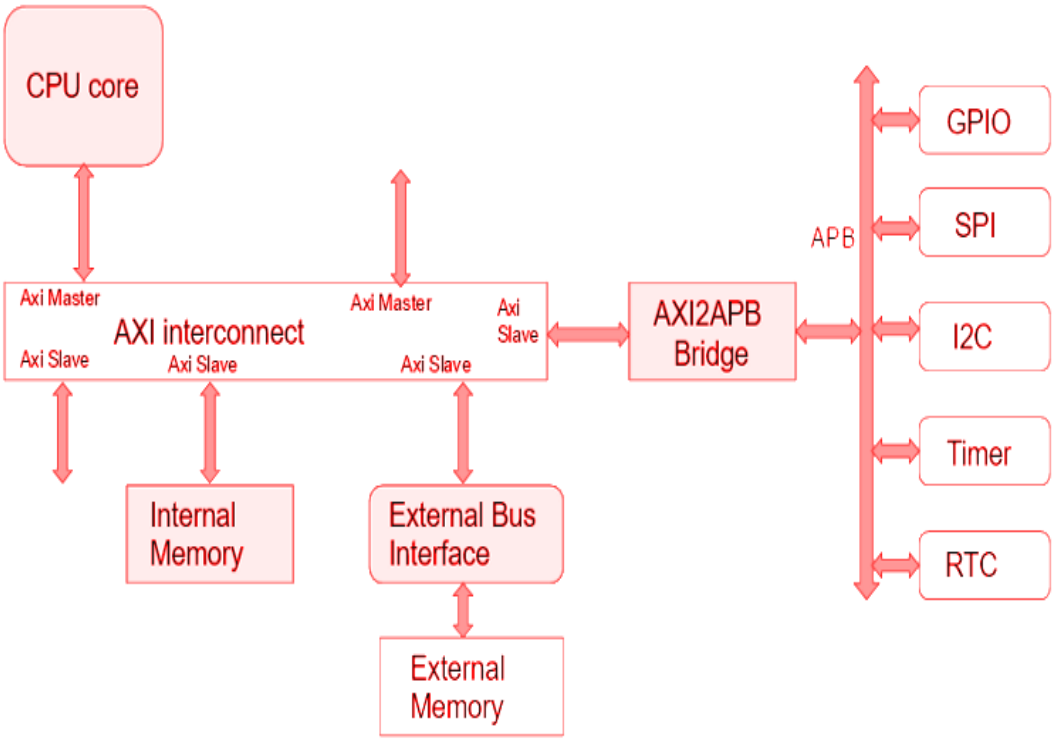


Figure 2.11: AXI bus architecture [19]

We can use the device connected as master or slave according to our need.

- If the device connected is master, it can be used to write data to DDR3
- If it is a slave, we can read from it, and which we can write to other master elements, it is exposed to 64 to 1024 bytes.

2.6 ZYBO Board

2.6.1 Introduction

It is a board by Digilent which is a Zynq-7000 family, the Z-7010, based on SoC. The board is one of the cheapest versions. With the ZYBO board, we get:

- The block design of the high level[20]
- We can program the FPGA portion of SoC into Verilog or VHDL [20]
- The interface between the AXI bus and the code[20] C code for interfacing with the wrapper through bus[20]
- Board support package[20]
- Linux kernel module[20]
- User space application[20]
- For high level block design we can use **zybo_base_system/source/vivado/hw/zybo_bsd/zybo_bsd.xpr** in vivado, because it contains information for physical connections[20]

In Zynq-7000 which we are using 2 AXI master and 2 AXI slave interfaces are present.

Next, for the connection between Zynq and a peripheral device, we need interconnect.AXI4 interconnect is being used, which contains 1 to 16 AXI master and slave interfaces each [20].

2.6.2 Features of the Zynq-7000 include:

--the processor contained is ARM Cortex-A9, which is known for its perfect performance and the per watt ratio [21].

--it supports single and double floating point [21]

--it can perform up to 1 GHz [21]

--contains the most significant amount of space,512KB cache,256KB on-chip, and supports DDR3-1866[21]

--for low power consumption and low-cost Artix-7 is being used with the excellent performance [21]

--Memory mapped components with DMA along with 2x USB 2.0, 2x Tri-mode Gigabit Ethernet, 2x SD/SDIO, 2x UART, 32b GPIO, etc. [21]

--Everything contained in a programmable, flexible, processor, interconnects, a speed which we can adjust as per the need, low power mode is also available [21]

--acceleration is provided to hardware through AXI ACP port of 64 bit, and even cache coherency for soft processors present. It offers high bandwidth with 100 GB/s between PS and PL [21]

--Hardware acceleration is provided through parallel processing with low power DSP slices [21]

--We get full safety and security is provided with anti-tamper technology and proper and safe booting system [21]

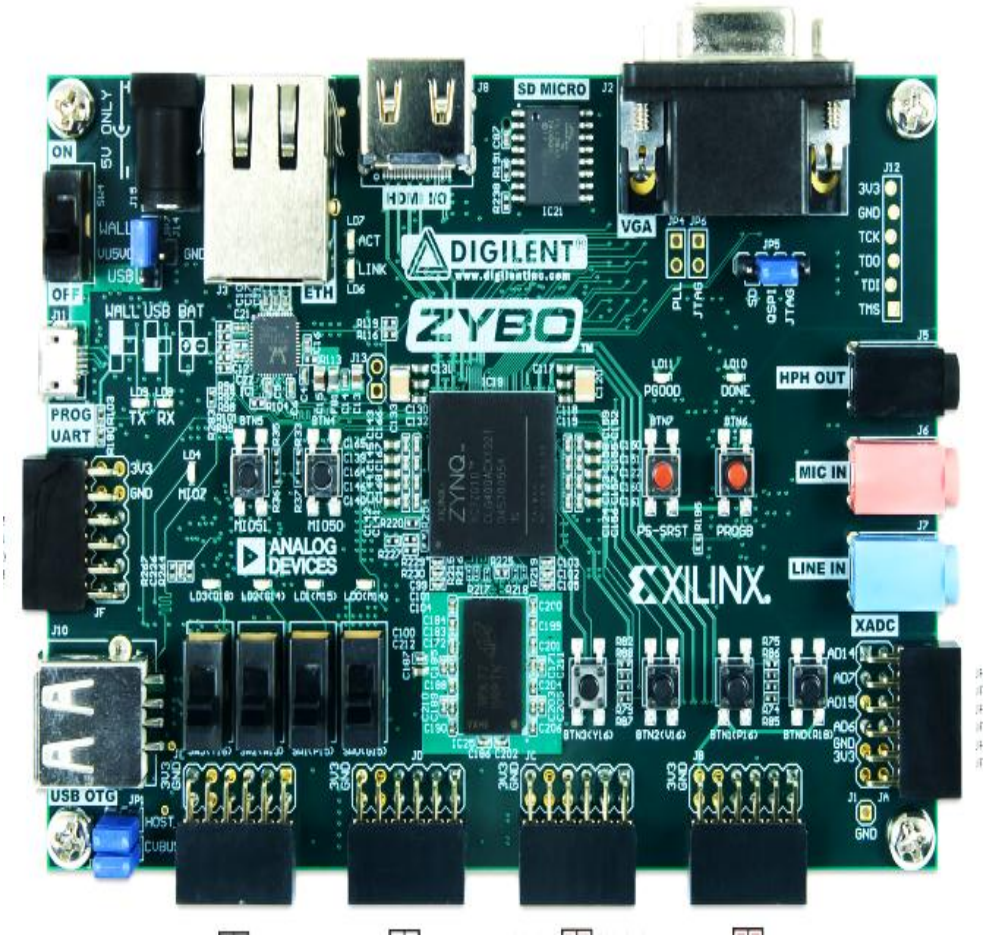


Figure 2.12: Zybo Z7-10[22]

The **XC7Z010-1CLG400C** contains:

--Power Switch [22]

--Processor Reset Pushbutton [22]

--Power Select Jumper and battery header [22]

--Logic configuration reset Pushbutton [22]

--Shared UART/JTAG USB port [22]

--Audio Codec Connectors [22]

--MIO LED [22]

--Logic Configuration Done LED [22]

--2 MIO Pushbuttons [22]

--Board Power Good LED [22]

--MIO Pmod [22]

--JTAG Port for optional external cable [22]

--USB OTG Connectors [22]

--Programming Mode Jumper [22]

--4 Logic LEDs [22]

--Independent JTAG Mode Enable Jumper [22]

--4 Logic Slide switches [22]

--PLL Bypass Jumper [22]

--USB OTG Host/Device Select Jumpers [22]

-- VGA connector [22]

--Standard Pmod [22]

--microSD connector (Reverse side)[22]

--3 High-speed Pmods [22]

--HDMI Sink/Source Connector [22]

--4 Logic Pushbuttons [22]

--Ethernet RJ45 Connector [22]

--XADC Pmod [22]

--Power Jack [22]

Chapter 3
METHODOLOGY

3. METHODOLOGY

Xilinx created software for analyzing and synthesizing designs created by HDLs, called Vivado Design Suite. It also contains some features like synthesizing it at a high level and implementing on SoC. Vivado is really scalable, accurate, properly integrated, and predictive.

3.1 Vivado HLS

The algorithms used are becoming complicated day by day. Vivado High-Level Synthesis is available as an up gradation in the HLx edition without any charges [23]. With the advent of Vivado HLS, it has become easy to create an IP with complex code, as it allows users to use C, C++ and System C languages and directly create configurations for the programmable device without the need to develop any RTL manually. Vivado HLS creates a similar system and design architects, providing a faster IP creation through:

- Absorbing the need for the algorithm from data type and interfaces [23].
- An extensive library with every kind of data type, videos, etc. [23].
- Architectures which are modified by the directives [23].
- Faster as there is no RTL created manually [23].
- The algorithm is written in simple language, and VHDL or Verilog codes are automatically created [23].
- Works with a wide range of languages [23].
- No manual work is done for the use of FPGA memory or library [23].

Vivado HLS can automatically create an implementation for RTL using the code provided. Users can use different directives available for the design. We can modify the designs as per the needs of the user for the same source using various directives.

Scheduling and Binding are essential tasks of the HLS, as scheduling tells us about the time taken for the operation and also the provision of the resources, while binding tells us about the delays of the components or directives used [24].

HLS can to do this by available defaults inside and use directives and constraints to be precise.

It calculates the timing and delays by the device, which we mentioned earlier so that it can also tell us about the area needed [24].

The advantages of using Vivado HLS can be:

--Designers can now do their work faster and more productive with lesser efforts [25].

--Software designers can now create more complex algorithms with acceleration [25].

--Verification of the design now done at a faster rate as compared to earlier processes [25]

--The use of directives creates a much-optimized design, only with the C source code, and different designs can be established [25].

--The c code is portable and understandable language [25].

--We can use the same code for different hardware [25].

--HLS makes the operations work as per the speed of the FPGA [25].

--To make a sequence of the operations, the finite state machine is being created first [25].

The synthesis of the code is being done in the following way:

--The function which is selected at the top level is converted to RTL

--Functions involved are converted to blocks, if there is a hierarchy, then it is followed in the RTL design as per the code.

The design flow of the Vivado HLS involves:

1. Firstly the compilation is done using the C simulation option, then the simulation is done to check the code and subsequently debugging [25].

2. Then the code is converted to an RTL design using a synthesis option, and we can also include directives if needed [25].

3. Reports are being created, and the design is analyzed [25].

4. The implementation of RTL is verified [25].

5. IP is then created using an export RTL option [25].

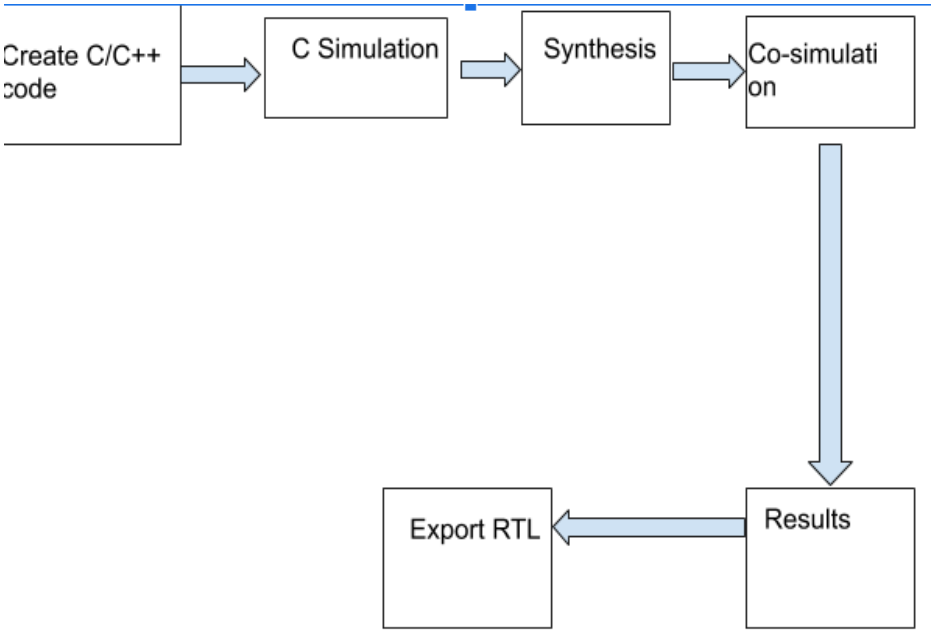


Figure 3.1: Design flow in Vivado HLS

3.1.1 DIRECTIVES

What Vivado HLS does is creates a hardware design as per the code provided. The directives option available in the software is to improve the performance, like using pipelining for the betterment of the design [26].

Sometimes it assumes that the user only needs a better performance and optimizes the design, without keeping the priority of reducing the clocks or area.

3.1.1.1 PIPELINING

Usually, in languages such as C, C++ operations are done sequentially, meaning one by one.

The Initiation Interval (II) meaning the time to start of two loops, which are one after the other, is very important for pipelining. Without pipelining the Initiation Interval are different for consecutive loops, but if we apply to the pipeline, we get only one clock cycle for both saving time and hardware, as both start at the same time[27].

To apply pipelining to a loop in Vivado HLS uses directives section and select pipelining at the desired loop. And it tries to minimize the latency.

As an example, we can see that without pipelining for two read operations, we require three clock cycles, but with pipelining, it gets reduced to just one. Overall six clock cycles are being reduced to four; the optimization done is to start iteration before the previous one ends.

Shown below is a simple example of pipelining.

In this example, we have procedures taking place, namely P1 and P2.

Table below shows how it works sequentially.

Stages	1	2	3	4	5	6	7	8
S1	P1				P2			
S2		P1				P2		
S3			P1				P2	
S4				P1				P2

Table 3.1: Table for the sequential process

Another table shows working of the procedures after applying pipelining

Stages	1	2	3	4	5
S1	P1	P2			
S2		P1	P2		
S3			P1	P2	
S4				P1	P2

Table 3.2: Table showing pipelining

Hence we can conclude that on applying pipelining, we can reduce the number of stages, procedures are being performed simultaneously with maximum use of the hardware.

Advantages of Pipelining

1. The latency is reduced.
2. The utility of the system is improved.
3. The system's reliability is increased.

Disadvantages of Pipelining

1. The use of pipelining increases hardware use, thus increasing the manufacturing cost.
2. The clock cycle for instructions increases.

In pipelining, we can perform tasks of many instructions simultaneously, by overlapping them, and not waiting for one to end. By the overlapping of the task, we increase the productivity of the system [28].

We can perform many operations at a time.

3.1.1.2 UNROLLING

Unrolling is another method for parallelism. Here many copies of a loop are made and then adjusted as per the need.

Let's take an example to understand it better.

The code written below is a rolled code, meaning no directive is applied;

```
int s=0;

for(int t=0ti<10;t++){

s+=a[t];

}
```

After applying the unrolling of a factor of two over the above code we get;

```
int s=0;

for(int t=0;t<10;t+=2){

s+=a[t];

s+=a[t+1];

}
```

The number of unrolling means that many numbers of copies of the instructions are created, and adjusted accordingly; also, the counter is updated with the iterations.

The number of operations is increased to ease the parallelism, increasing the productivity of the system.

Two types of unrolling are present, namely partial and full. Partial means the number of copies is less than the number of loops present, while if the number of loops is equal to the copies created is called full

unrolling. Full unrolling is better than the partial one as it creates more parallelism. In RTL design for fully unrolling all the loops are run simultaneously, while in partial unrolling several copies are created. Loop bounds are needed for full unrolling at compilation time [29].

To apply unrolling we can simply select unrolling from the directives section or write the code as;

```
int s=0;

for(int t=0;t<10;t++){

#pragma HLS unroll factor=2

s+=a[t];

}
```

Usually, in C/C++ rolled functions are present and performed sequentially, We create RTL designs in the same way. But we may use unrolling to increase productivity and data access [29].

HLS adds a remark at the end of a partially unrolled loop to distinguish between the original one.

3.1.2 MATRIX MULTIPLICATION

The C code used for the matrix multiplication is shown below:

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

void mmult (int A[4], int B[4], int C[4])
{
        int i,j,k,s,t;
        int Ao[2][2], Bo[2][2];

        for(i=0; i<2; i++) {
```

```
        for(j=0; j<2; j++) {

        Ao[i][j] = A[i * 2 + j];
        Bo[i][j] = B[i * 2 + j];
        }
        }

        for (i = 0; i < 2; i++) {
        for (j = 0; j < 2; j++) {

        int s = 0;
        for (k = 0; k < 2; k++) {
                int t = Ao[i][k] * Bo[k][j];
                s += t;
        }
        C[i * 2 + j] = s;
        }
        }
}
```

3.2 VIVADO HLx

In the Vivado HL Design Edition and HL System Edition, without any additional charges, the HLx Edition is a configuration to it.

The Vivado HLx helps the designers with its features to reuse optimized designs, maximize the use of the design, reuse of IPs created, automated connections, and fastening the making of a design. It is a great help for designers with the reuse of designs with much abstraction [30].

Accelerating High-Level Design [30]

- Creation of IP only with a simple C code automatically through HLS
- Integration of IP automatically based on the blocks
- Model Composer and System Generator for DSP

Accelerating Verification [30]

- Logic Simulation of Vivado
- Mixed Language Simulator
- Standalone Programming and Debug Environments
- Accelerated Verification with C, C++ or SystemC with Vivado HLS
- Verification of IP

Accelerating Implementation [30]

- Faster Implementation
- Better Design Density
- Advantage of Performance for the low and mid-range and Advantage of Power for the high-end

Vivado Design Suite HLx Editions can go further than RTL design. It is now able to the verification and creation of designs much faster than earlier RTL based methods[31].HLx also supports SDx well and together create an excellent platform for automated design creation or using all programmable devices like Zynq SoCs, 3D ICs, etc.

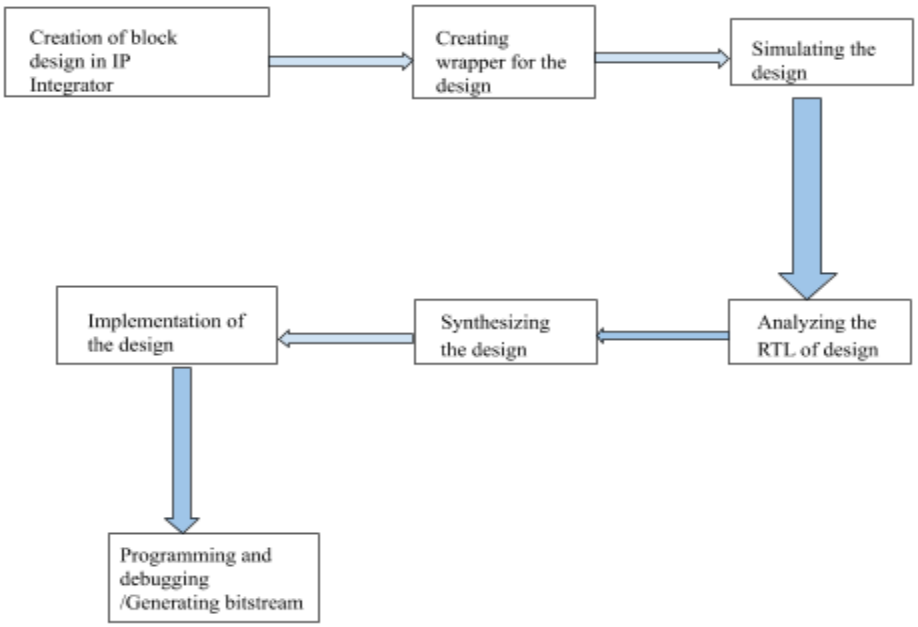


Figure 3.2: Working of Vivado HLx

RTL design cycles needed working for each block separately, and we do it for the whole design. This method took time, and any changes would make it difficult to verify it again even before the hardware setup, also the connectivity could not be stabilized. But the high-level design method covers it all, containing features such as [20]:

--Development of platform and logic are two different things now

--Connectivity is faster, with Vivado IP Integrator knowing about the board as well

--Simulation time is being reduced as compared to RTL simulation by the use of C

--From language to the chip is a faster task now with the help of HLS for better synthesis

All the steps from simulation to the FPGA are fully automated. This makes it easy to generate a bit stream and check it on the board. We get a better product by using design derivatives. Even switching the

same design to any other board is very easy; the things will be modified automatically as per the new board [31].

3.3 XILINX SDK

The Xilinx Software Development Kit (XSDK) is an environment for creating applications for any of the Xilinx's microprocessors like Zynq-7000 SoCs.

Features of the SDK [32]:

--Language used for compiling is C/C++ [32]

--Manages the projects on itself [32]

--Makefile and applications are created on its own [32]

--Inspection of errors is accessible [32]

--Boards' detection, profiling and debugging is done easily due to the environment provided [32]

--The version of the source code can be easily handled [32]

--Configuration of FPGAs is possible [32]

--Command-line tool is present which can be scripted [32]

This Integrated Development Environment (IDE) is a connection between the Xilinx's processors available and the designs created on the software. Eclipse is the base for the SDK.

Advantages of SDK include:

--Many processors like Zynq-7000 SoCs, Micro-Blaze, etc. can be used [31]

--Can be downloaded separately or with the Vivado Design Suite [31]

--Can connect the Integrated Design Environment (IDE) to the design [31]

--Debugging and co-debugging is possible for the whole of the software design [31]

--Flash memory management, compiling and JTAG debug integration [31]

--Many libraries and drivers available [31]

--RTOS for any platform is supported [31]

--Scripting possible through Xilinx Software Command Line Tool (XSCT) [31]

Using the Xilinx SDK, we can create software for the Xilinx processors available. The requirements for the same are JTAG debugger, BSPs, libraries, flash programmer, compiler, and drivers. An environment for C/C++ bare- metal and Linux development and debugging is also available. C/C++ Development Toolkit (CDT) is supported through the Eclipse platform.

The project created executes our final output, and the directory present in it contains all the required files such as source, make file, elf file, output file, etc.

The board support package (BSP) present contains all the libraries and drivers needed for the project. The APIs of our software application must be at the top of our software platform. Therefore the creation of a BSP is the primary step [33].

To get a good SoC, we need to know well about our board and the software. It's tough to get an all in one tool, to know all about performance and visualization. Xilinx System Development Kit (XSDK) can do it for us, along with optimization of the system, using System Analysis toolbox [33].

3.3.1 The C code used for getting output at the Xilinx Software Development Kit is:

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xparameters.h"
#include "xil_io.h"

int main()
{
	init_platform();
	int c[4];
print("Hello World\n\r");


	Xil_Out32(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR,0x1);

	Xil_Out32(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR+4,0x2);

	Xil_Out32(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR+8,0x1);

	Xil_Out32(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR+12,0x2);


	Xil_Out32(XPAR_AXI_BRAM_CTRL_1_S_AXI_BASEADDR,0x1);

	Xil_Out32(XPAR_AXI_BRAM_CTRL_1_S_AXI_BASEADDR+4,0x2);
```

```
        Xil_Out32(XPAR_AXI_BRAM_CTRL_1_S_AXI_BASEAD
DR+8,0x1);

        Xil_Out32(XPAR_AXI_BRAM_CTRL_1_S_AXI_BASEAD
DR+12,0x2);



c[0]=Xil_In32(XPAR_AXI_BRAM_CTRL_2_S_AXI_BASEADDR);
c[1]=Xil_In32(XPAR_AXI_BRAM_CTRL_2_S_AXI_BASEADDR+
4);
c[2]=Xil_In32(XPAR_AXI_BRAM_CTRL_2_S_AXI_BASEADDR+
8);
c[3]=Xil_In32(XPAR_AXI_BRAM_CTRL_2_S_AXI_BASEADDR+
12);

xil_printf("%d\n%d\n%d\n%d",c[0],c[1],c[2],c[3]);
                return 0;
  }
```

3.4 Design specifications

C language has been used for the implementation of the design as well for checking the output in Vivado HLS and Xilinx SDK respectively.

The synthesis and creation of IP have been done using development tools available in the Vivado HLS. And the design is being checked by the SDK Integrated Development Environment.

The targeted chip used is XC7Z010-1CLG400C.

Chapter 4
RESULTS AND DISCUSSION

4. RESULTS AND DISCUSSION

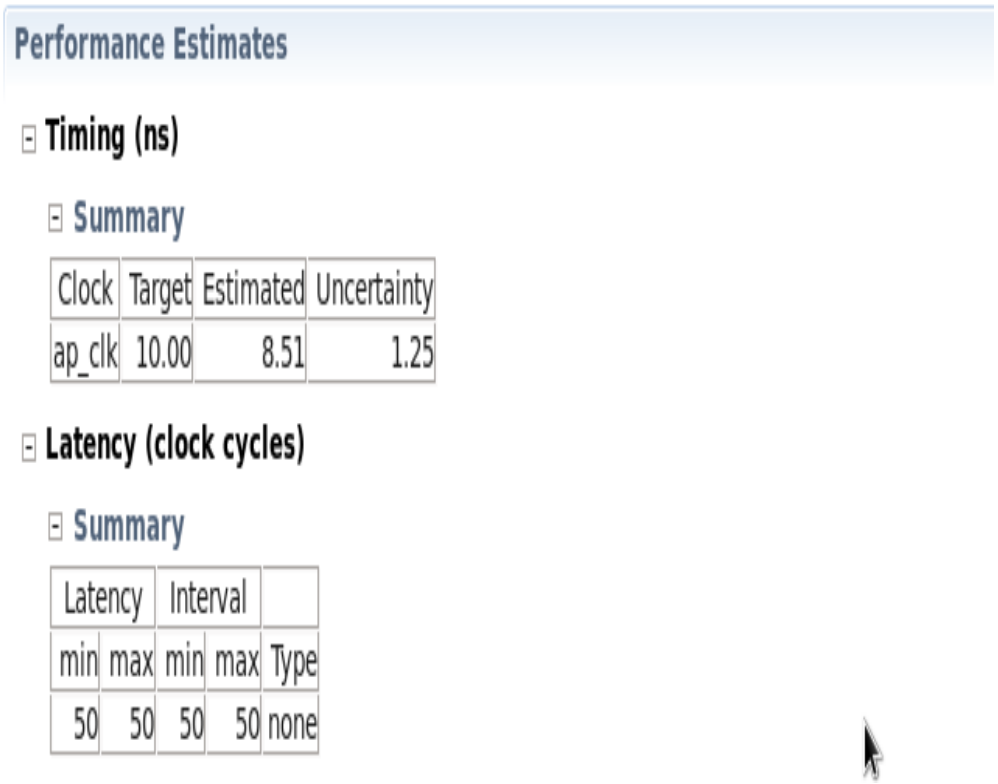
We first synthesize the C code in the Vivado HLS, and we got to know the performance and utilization estimates sequentially. Later the directives, pipelining and unrolling were applied one by one, and we calculated the estimates again by the synthesis. There is a difference in applying the directives; both of them showed different results, with an optimized design. By export RTL we have created an IP to use it in our hardware design. Using Vivado HLx Edition, we have created a block diagram to connect it up with our board through Xilinx SDK.

The results presented here show us the estimates as well as the comparison between the sequential and optimized process. Also, the block diagram created is shown up with a snip of the multiplied output in Xilinx SDK.

4.1 Sequential process

Latency report

This is the report for timing and clock cycles required for the sequential process, where no directives are applied.



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.51	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
50	50	50	50	none

Figure 4.1: Latency report of sequential process

Hardware utilization report

With this report we get to know about the hardware (LUTs, Flip-Flops) required for circuit implementation, without any directives are applied.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	3	0	878
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	101
Register	-	-	482	-
Total	0	3	482	979
Available	120	80	35200	17600
Utilization (%)	0	3	1	5

Figure 4.2: Hardware utilization of sequential process

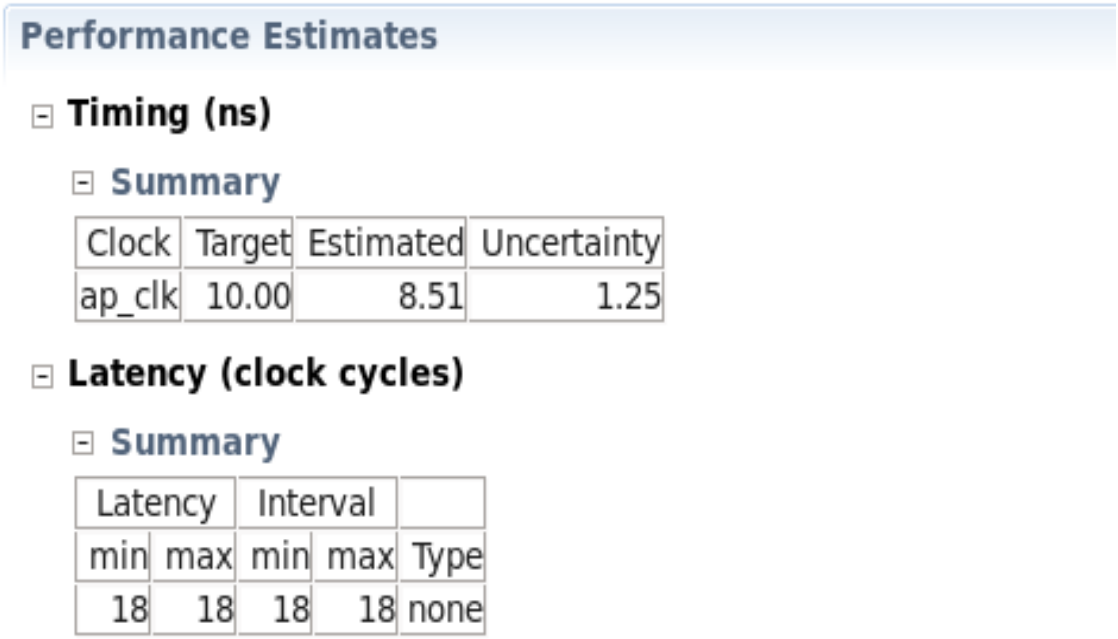
4.2 Loop Pipelining

We have applied the pipeline directive over the loops in the C code, and then the results are obtained after the simulation of the code. As we can see from the reports shown below, there is a reduction in latency and hardware utilization. Also, the application of directive at different loops gives different results. But we should know that on applying pipeline, the sub-loops gets unrolled, therefore there is no use of applying pipelining on two consecutive loops. However, below are the results for different loops.

-- For loop i

Latency report

This is the report for timing and clock cycles, where pipelining directive has been applied only on loop i, and the innermost loops as per pipelining will automatically be unrolled.



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.51	1.25

Latency (clock cycles)

Summary

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

Figure 4.3: Latency report after applying pipelining on loop i

Hardware utilization report

With this report we get to know about the hardware (LUTs, Flip-Flops) required for circuit implementation, pipelining directive being applied on loop i.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	12	0	825
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	83
Register	-	-	474	-
Total	0	12	474	908
Available	120	80	35200	17600
Utilization (%)	0	15	1	5

Figure 4.4: Hardware utilization report after applying pipelining on loop i

-- For loop j

Latency report

This is the report for timing and clock cycles, where pipelining directive has been applied on loop j and loop k will now be automatically unrolled.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.51	1.25

Latency (clock cycles)

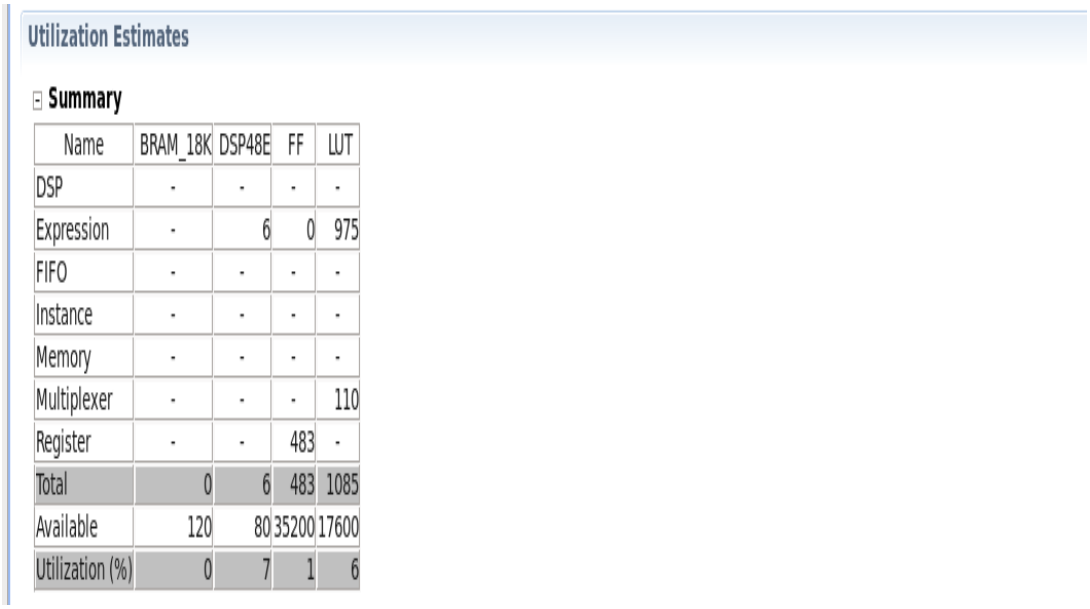
Summary

Latency		Interval		
min	max	min	max	Type
20	20	20	20	none

Figure 4.5: Latency report after applying pipelining on loop j

Hardware utilization report

With this report we get to know about the hardware (LUTs, Flip-Flops) required for circuit implementation, pipelining directive being applied only on loop j.



Utilization Estimates

Summary

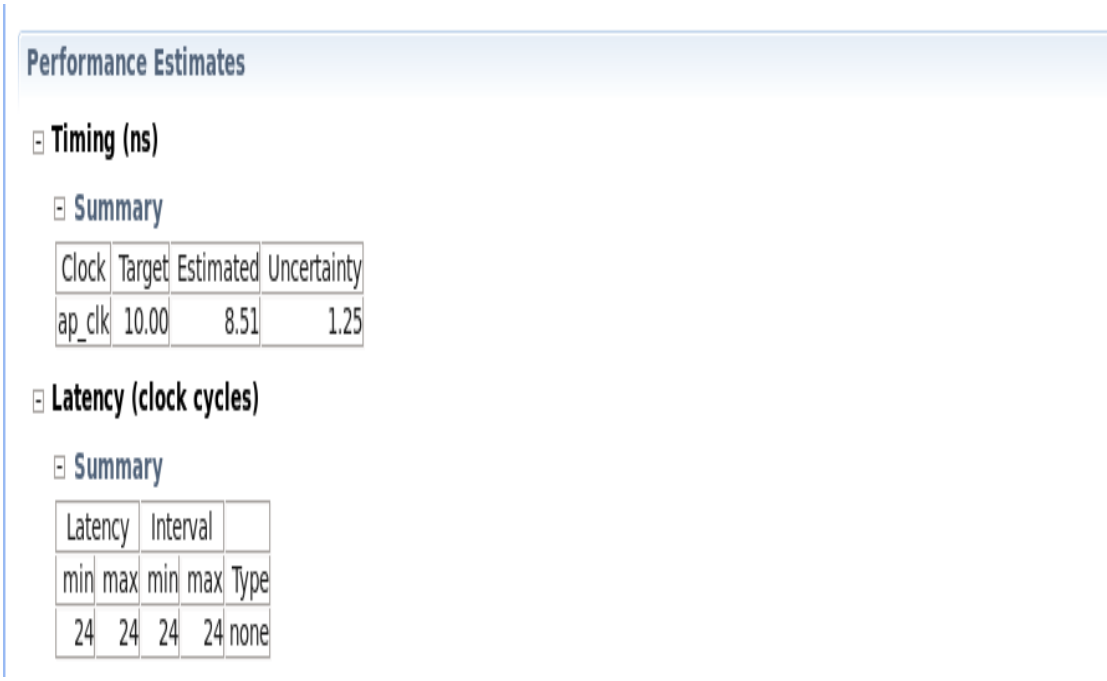
Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	6	0	975
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	110
Register	-	-	483	-
Total	0	6	483	1085
Available	120	80	35200	17600
Utilization (%)	0	7	1	6

Figure 4.6: Hardware utilization report after applying pipelining on loop j

-- For loop k

Latency report

This is the report for timing and clock cycles, where pipelining directive has been applied on loop k.



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.51	1.25

Latency (clock cycles)

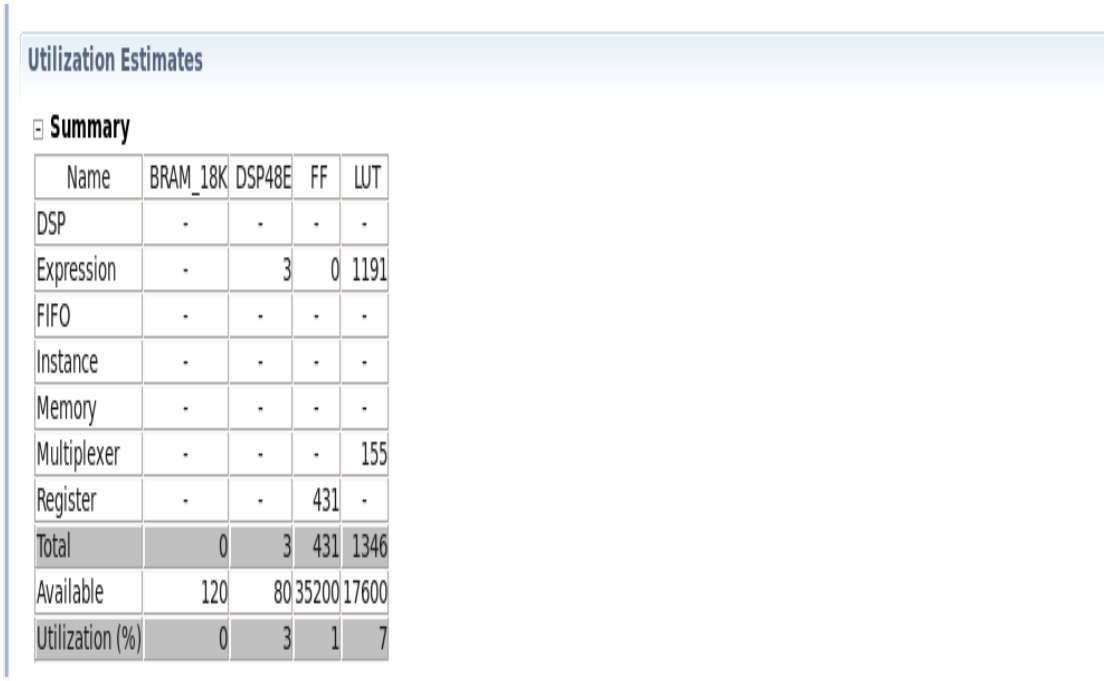
Summary

Latency		Interval		
min	max	min	max	Type
24	24	24	24	none

Figure 4.7: Latency report after applying pipelining on loop k

Hardware utilization report

With this report we get to know about the hardware (LUTs, Flip-Flops) required for circuit implementation, pipelining directive being applied only on loop k.



Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	3	0	1191
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	155
Register	-	-	431	-
Total	0	3	431	1346
Available	120	80	35200	17600
Utilization (%)	0	3	1	7

Figure 4.8: Hardware utilization report after applying pipelining on loop k

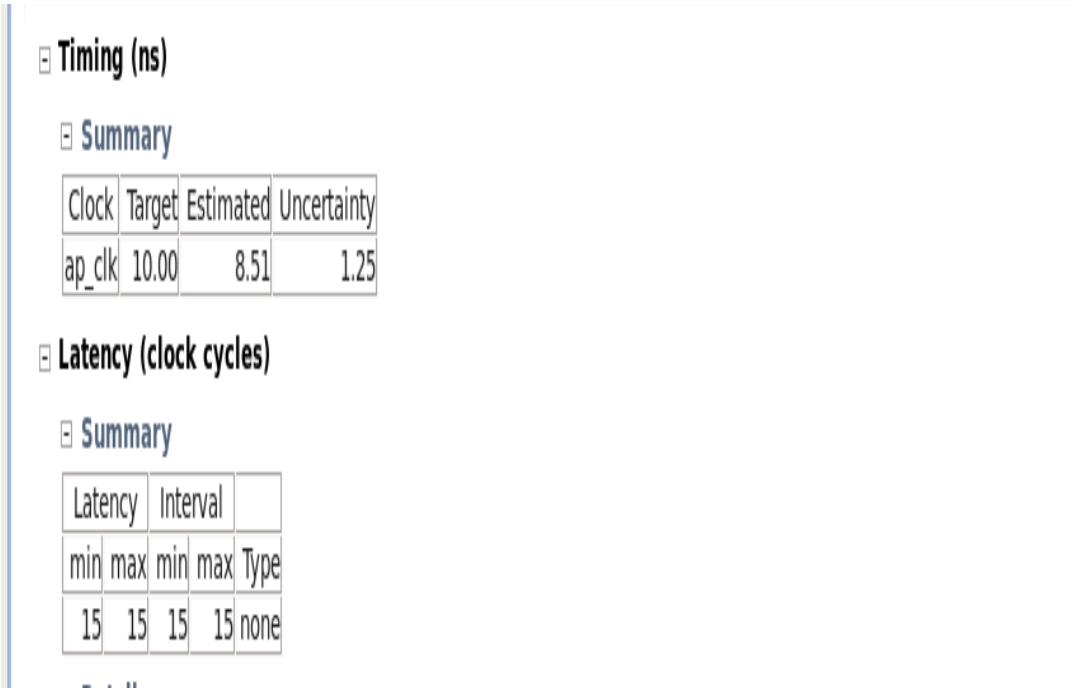
4.3 Loop Unrolling

The unrolling directive is applied to the loops of our C code, again reducing the latency but with a slight increment in hardware comparatively. Because unrolling copies the iterative part and runs in parallel, making the hardware faster.

-- For loop ijk

Latency report

This is the report for timing and clock cycles, where unrolling directive has been applied on all the three loops.



Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.51	1.25

Latency (clock cycles)

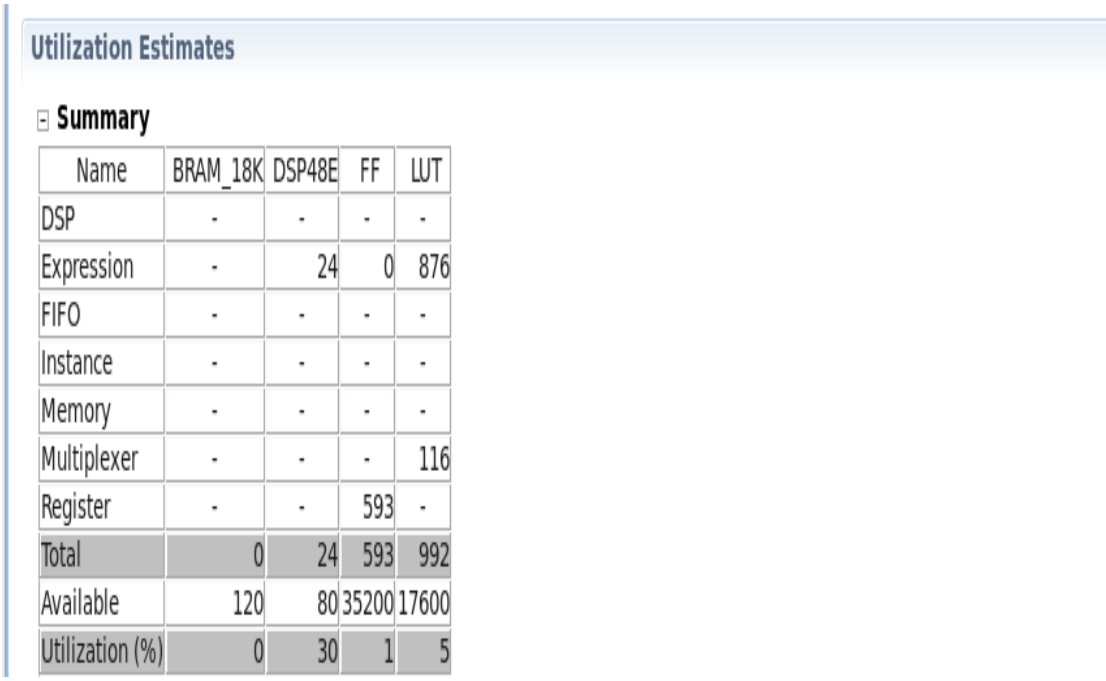
Summary

Latency		Interval		
min	max	min	max	Type
15	15	15	15	none

Figure 4.9: Latency report after applying unrolling on loop ijk

Hardware utilization report

With this report we get to know about the hardware (LUTs, Flip-Flops) required for circuit implementation, with unrolling directive being applied on loops ijk.



Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	24	0	876
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	116
Register	-	-	593	-
Total	0	24	593	992
Available	120	80	35200	17600
Utilization (%)	0	30	1	5

Figure 4.10: Hardware utilization report after applying unrolling on loop ijk

-- For loop jk

Latency report

This is the report for timing and clock cycles, where unrolling directive has been applied on only j and k loops.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.51	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
20	20	20	20	none

Figure 4.11: Latency report after applying unrolling on loop jk

Hardware utilization report

With this report we get to know about the hardware (LUTs, Flip-Flops) required for circuit implementation, unrolling directive being applied only on loop j and k.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	12	0	809
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	68
Register	-	-	470	-
Total	0	12	470	877
Available	120	80	35200	17600
Utilization (%)	0	15	1	4

Figure 4.12: Hardware utilization report after applying unrolling on loop jk

-- For loop k

Latency report

This is the report for timing and clock cycles, where unrolling directive has been applied only on the innermost loop that is k loop.

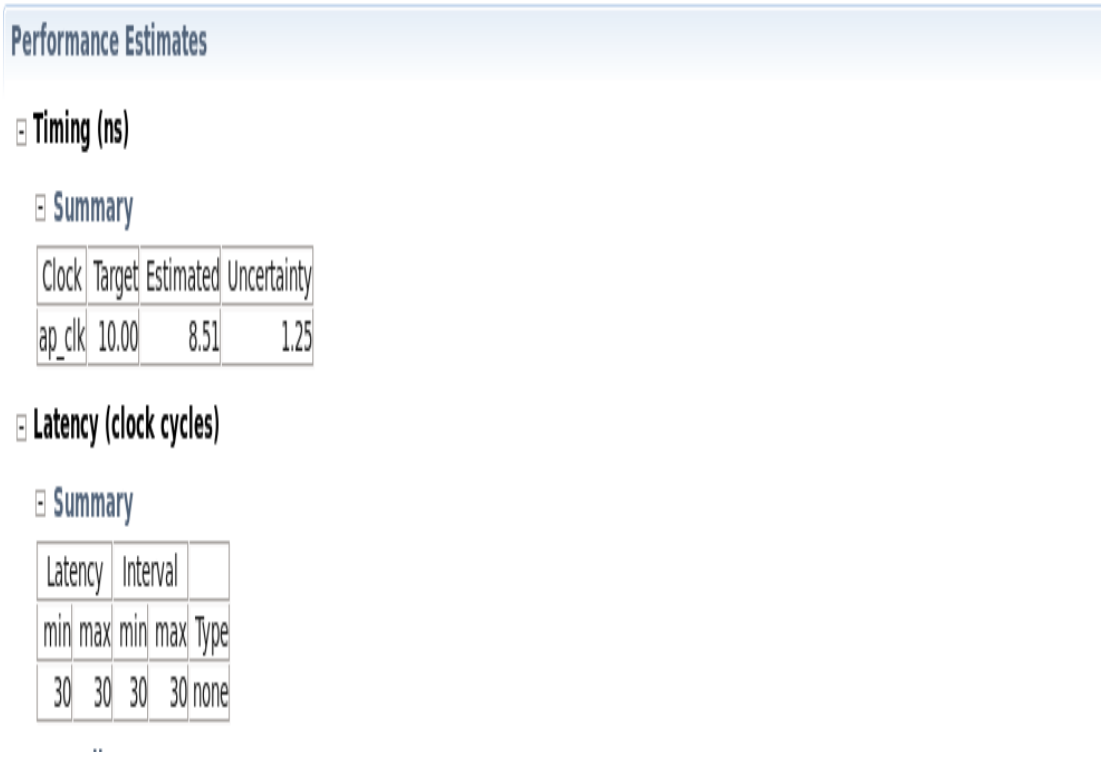


Figure 4.13: Latency report after applying unrolling on loop k

Hardware utilization report

With this report we get to know about the hardware (LUTs, Flip-Flops) required for circuit implementation, unrolling directive being applied only on loop k.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	6	0	815
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	80
Register	-	-	478	-
Total	0	6	478	895
Available	120	80	35200	17600
Utilization (%)	0	7	1	5

Figure 4.14: Hardware utilization report after applying unrolling on loop k

Comparison of parameters between sequential process, pipelining and unrolling directive

Parameters	No directive	Pipelining			Unrolling		
		i	j	k	ijk	Jk	k
Latency	50	18	20	24	15	20	30
LUTs	979	908	1085	1346	992	877	895
Registers	482	474	483	431	593	470	478
DSPs	3	12	6	3	24	12	6

Table 4.1: Comparison of parameters between sequential process, pipelining and unrolling directives

The report presented above tells us how pipelining and unrolling are helpful in comparison to the sequential process. Also, we can assume the difference between pipelining and unrolling.

We can see that latency has been reduced in both the cases, while hardware reduction is made much in the case of unrolling more effectively.

Shown below are the tables where we are comparing the reduction in delay in comparison to the sequential process and hardware utilization of unrolling and pipelining:

Pipelining			Unrolling		
i	j	k	ijk	jk	K
64%	60%	52%	70%	60%	40%

Table 4.2: Comparing the reduction in delay after applying pipelining and unrolling directives

	Pipelining			Unrolling		
	I	j	k	ijk	jk	k
LUTs	6%↓	10%↑	37%↑	2%↑	11%↓	9%↓
Registers	2%↓	0%	11%↓	23%↑	3%↑	1%↓
DSPs	4%↑	2%↑	0%	8%↑	4%↑	2%↑

Table 4.3: Comparing the hardware utilization of unrolling and pipelining

From the above two tables, we get to know that applying unrolling on all the three loops gives us a reduction in delay by 70% while pipelining on loop i provides 64% reduction in delay.

Also, we can also see that there is an 11% decrease in hardware utilization on using unrolling on j and k loop, while only 8% decrease on applying pipelining to only i loop.

Now, we can use between the two directives as per the requirement of our application.

4.4 Block Design

The following is a block diagram for the implementation of the design on hardware. We have used Vivado HLx to design it. We connect the processor to our IP created at Vivado HLS.

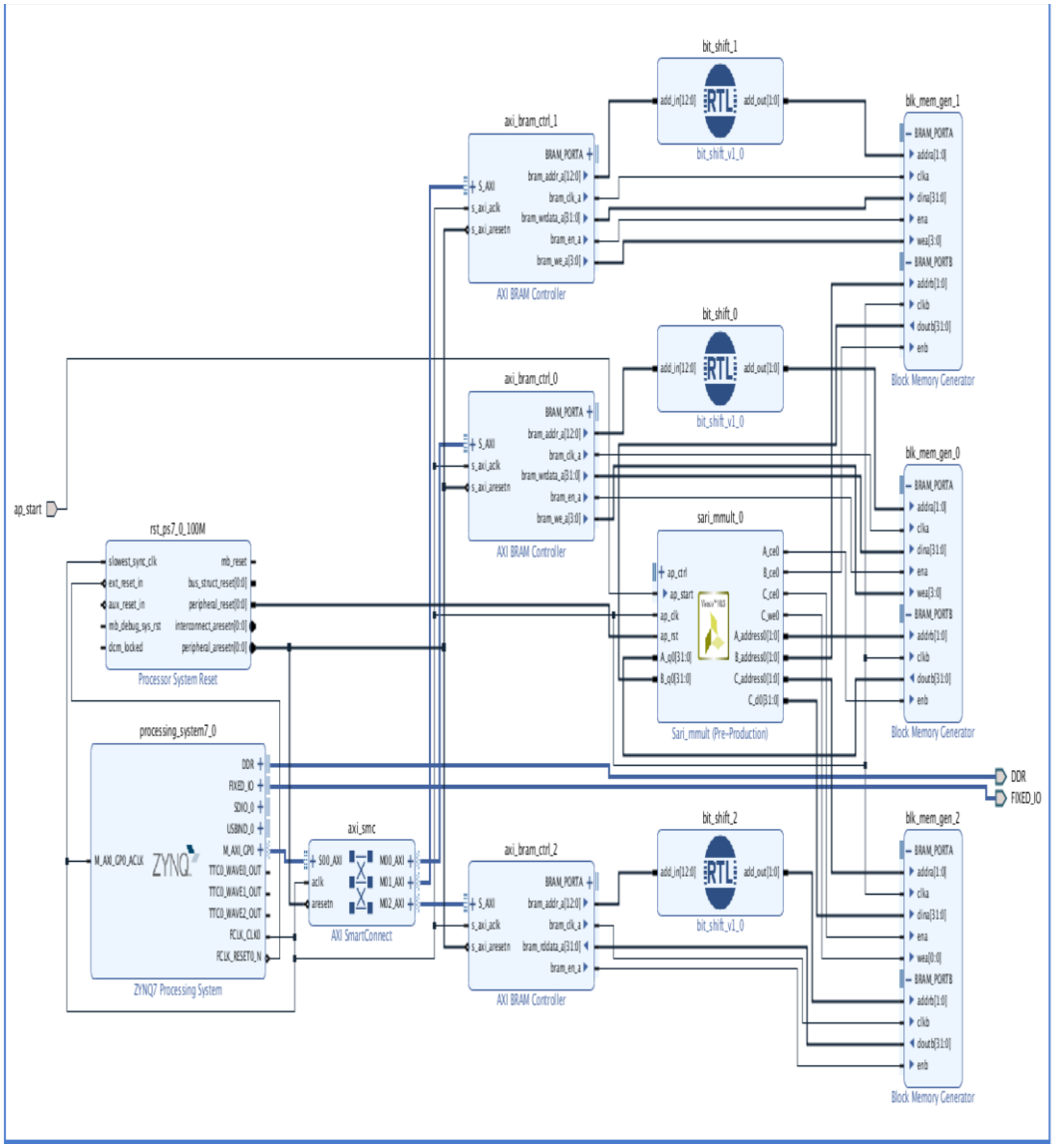


Figure 4.15: Block design for matrix multiplication

4.5 Implemented design

Shown below is the implemented design we can see at Vivado HLx. It shows the LUTs.

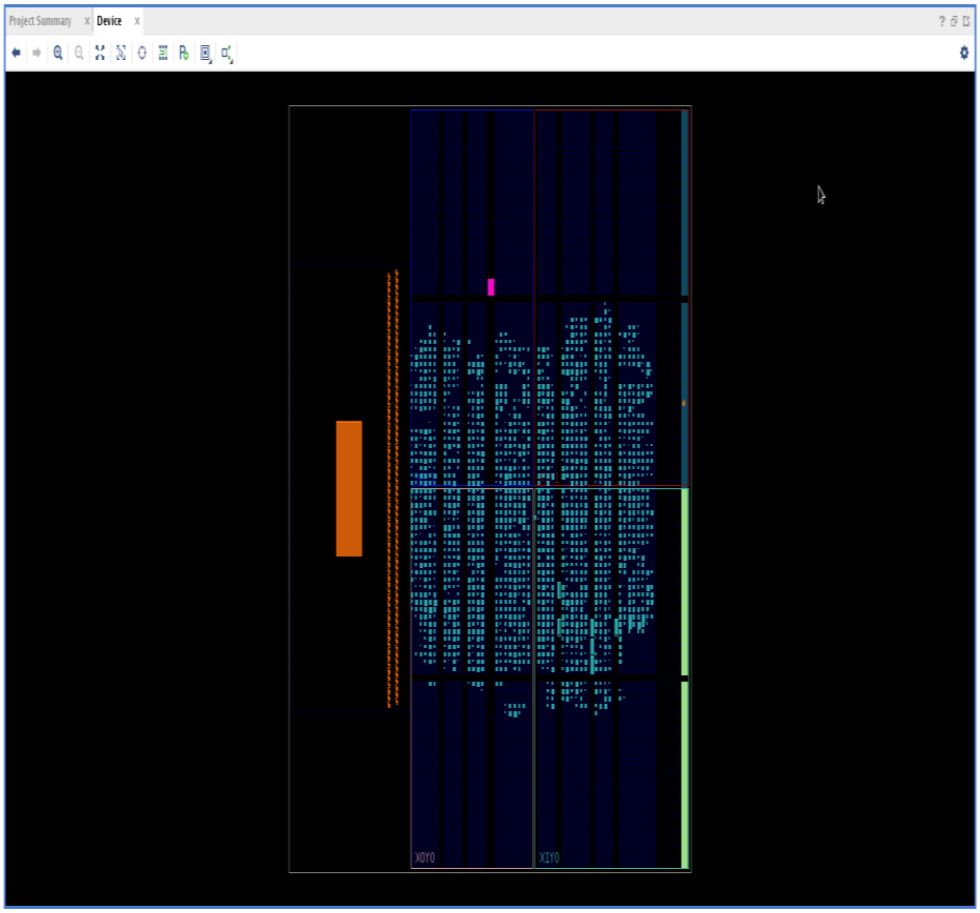


Figure 4.16: Implemented design for matrix multiplication

The implemented design provides us with information about the hardware; the circuit will contain 5108 LUTs count, 4536 Flip-Flops count, 1.50 BRAM count, and 3 DSP count.

4.6 SDK Output

Shown below is the output we got for the multiplication of two matrices as per our code, through Xilinx SDK, in the SDK Terminal. Here we have multiplied [1,2,1,2] x [1,2,1,2] and got the output [3,6,3,6].

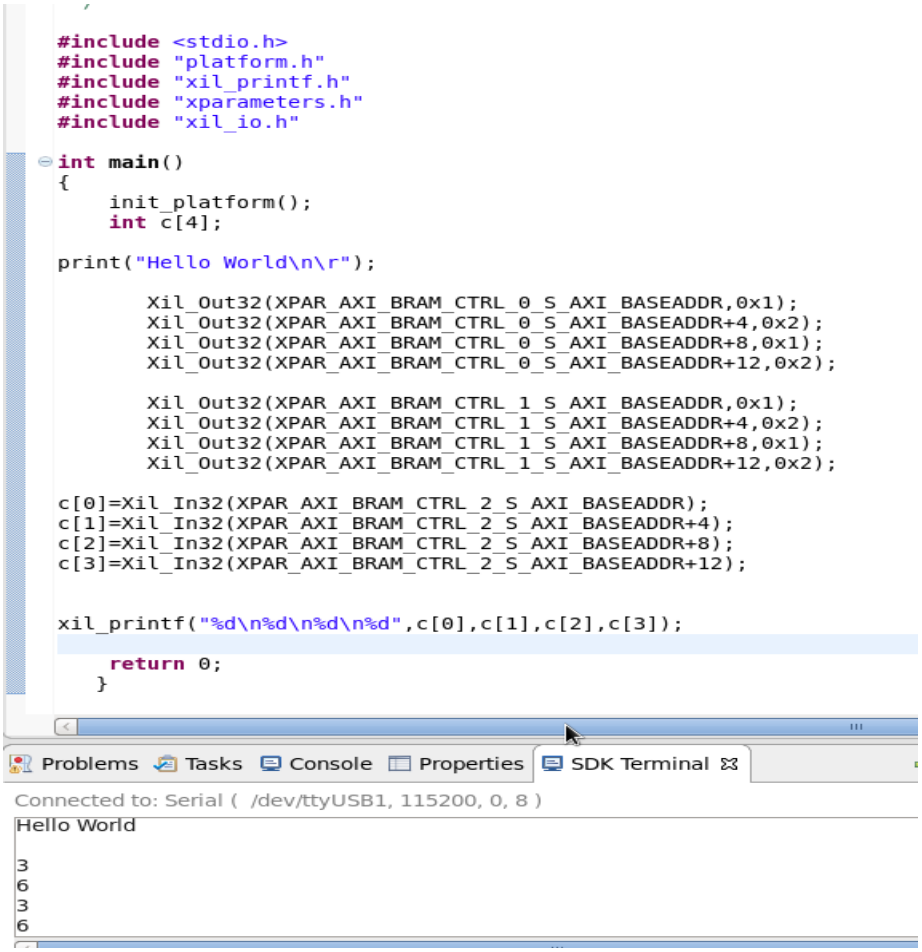


Figure 4.17: SDK output for matrix multiplication

Chapter 5

SUMMARY AND FUTURE SCOPE

5. SUMMARY AND FUTURE SCOPE

In this project, we have tried to reduce the complexity during calculations, reducing the latency as well as the use of the hardware using different directives and tried to get the output of the multiplication of two matrices by using the device, the FPGA board.

The two directives used were pipelining and unrolling. The use of directives lets us do the calculations faster, reducing the latency as compared to our sequential process, also allowing us to use lesser hardware. The two have been explained above, along with the software used. We have created a C code for matrix multiplication, done the synthesis, created an IP for the same, implemented a block design for its hardware implementation, and got the multiplied output using the board.

The future work we can apply to this project can be:

1. As we have implemented only two directives, we can apply more available directives and compare all of them and use as per the need of our application.

Directives can be:

- Loop Flatten: helps change nested or sequential loops to a single loop.
- Dataflow: helps start a function or loop before the previous one ends.

2. Using multi-core processor or other boards for faster calculations then now.

3. Calculation of distance between atoms, applying the logic for distance calculation in a quicker way.

REFERENCES

References

1. Gaur, N. (2018). *Reducing computational complexity of Mathematical functions using FPGA*. [online] Slideshare.net. Available at: https://www.slideshare.net/nehagaur339/reducing-computational-complexity-of-mathematical-functions-using-fpga-104973922 [Accessed 1 Jun. 2019].
2. En.wikipedia.org. (n.d.). *System on a chip*. [online] Available at: https://en.wikipedia.org/wiki/System_on_a_chip [Accessed 15 Jun. 2019].
3. Margaret Rouse (2015). *system-on-a-chip (SoC).* [ONLINE] Available at: https://internetofthingsagenda.techtarget.com/definition/system-on-a-chip-SoC. [Accessed 15 June 2019].
4. What is a System on a Chip (SoC)? - Definition from Techopedia. *Techopedia.com*. Available at: https://www.techopedia.com/definition/702/system-on-a-chip-soc [Accessed June 18, 2019].
5. SoC. *What is SoC? Webopedia Definition*. Available at: https://www.webopedia.com/TERM/S/SoC.html [Accessed June 18, 2019].
6. HardwareBee. (2018). *Field Programmable Gate Array (FPGA) History and Applications - HardwareBee*. [online] Available at: http://hardwarebee.com/field-programmable-gate-array-fpga-history-applications/ [Accessed 13 May 2019].
7. Akthar, S. (2014). *FPGA Architecture*. [online] All About FPGA. Available at: https://allaboutfpga.com/fpga-architecture/ [Accessed 18 May 2019].
8. Slideplayer.com. (n.d.). *FPGA PLB Evaluation using Quantified Boolean Satisfiability Andrew C. Ling M.A.Sc. Candidate University of Toronto Deshanand P. Singh Ph.D. Altera Corporation. - ppt download*. [online] Available at: https://slideplayer.com/slide/8349497/ [Accessed 2 Jun. 2019].
9. En.m.wikipedia.org. (n.d.). *Logic block*. [online] Available at: https://en.m.wikipedia.org/wiki/Logic_block [Accessed 5 Jun. 2019].

10. Agarwal, T. (n.d.). *Basic FPGA Architecture and its Applications*. [online] Edgefx.in. Available at: https://www.edgefx.in/fpga-architecture-applications/ [Accessed 18 May 2019].
11. Only-vlsi.blogspot.com. (2008). *Field-Programmable Gate Array*. [online] Available at: http://only-vlsi.blogspot.com/2008/05/field-programmable-gate-array.html?m=1 [Accessed 4 Jun. 2019].
12. Xilinx.com. (n.d.). *What is an FPGA? Field Programmable Gate Array*. [online] Available at: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html [Accessed 16 May 2019].
13. Trausmuth, R. (2006). *Introduction to Field Programmable Gate Arrays*. [online] Cis.upenn.edu. Available at: http://www.cis.upenn.edu/~lee/06cse480/lec-fpga.pdf [Accessed 15 May 2019].
14. ElProCus - Electronic Projects for Engineering Students. (n.d.). *Applications of Complex Programmable Logic Device (CPLD)*. [online] Available at: https://www.elprocus.com/complex-programmable-logic-device-cpld-architecture-applications/ [Accessed 4 Jun. 2019].
15. Idc-online.com. (2007). [online] Available at: http://www.idc-online.com/technical_references/pdfs/electronic_engineering/Difference_between_FPGA_and_CPLD.pdf [Accessed 12 May 2019].
16. Nandland.com. (n.d.). *What is a Block RAM in an FPGA? For Beginners.*. [online] Available at: https://www.nandland.com/articles/block-ram-in-fpga.html [Accessed 20 May 2019].
17. Ni.com. (n.d.). *Block RAM (BRAM) on an FPGA - LabVIEW Communications System Design Suite 3.1 Manual - National Instruments*. [online] Available at: http://www.ni.com/documentation/en/labview-comms/latest/fpga-targets/block-memory/ [Accessed 21 May 2019].
18. Xilinx.com. (2011). [online] Available at: https://www.xilinx.com/support/documentation/ip_documentation/bram_block.pdf [Accessed 22 May 2019].
19. AnySilicon. (2016). *Understanding AMBA Bus Architechture and Protocols - AnySilicon*. [online] Available at: https://anysilicon.com/understanding-amba-bus-architechture-protocols/ [Accessed 5 Jun. 2019].
20. Lauri Võsandi, l. (2014). *Lauri's blog | Vivado 2014.1 vs ZYBO*. [online] Lauri.xn--vsandi-pxa.com. Available at: https://lauri.xn--

vsandi-pxa.com/hdl/zynq/vivado-2014.1-vs-zybo.html [Accessed 22 May 2019].

21. Xilinx.com. (n.d.). [online] Available at: https://www.xilinx.com/support/documentation/product-briefs/zynq-7000-product-brief.pdf [Accessed 23 May. 2019].
22. Reference.digilentinc.com. (2015). *zybo:zybopins.png [Reference.Digilentinc]*. [online] Available at: https://reference.digilentinc.com/_detail/zybo/zybopins.png?id=reference%3Aprogrammable-logic%3Azybo%3Areference-manual [Accessed 5 Jun. 2019].
23. Xilinx.com. (n.d.). *Vivado High-Level Synthesis*. [online] Available at: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html [Accessed 24 May 2019].
24. Users.ece.utexas.edu. (n.d.). [online] Available at: http://users.ece.utexas.edu/~gerstl/ee382v_f14/soc/vivado_hls/VivadoHLS_Overview.pdf [Accessed 26 May 2019].
25. Iitg.ac.in. (n.d.). [online] Available at: http://www.iitg.ac.in/ckarfa/Course/2018/CS526/ug902-vivado-high-level-synthesis.pdf [Accessed 26 May 2019].
26. China.xilinx.com. (n.d.). [online] Available at: https://china.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1270-vivado-hls-opt-methodology-guide.pdf [Accessed 26 May 2019].
27. Xilinx.com. (n.d.). *Loop Pipelining and Loop Unrolling*. [online] Available at: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/sdsoc_doc/topics/calling-coding-guidelines/concept_pipelining_loop_unrolling.html [Accessed 28 May 2019].
28. Studytonight.com. (n.d.). *Concept of Pipelining | Computer Architecture Tutorial | Studytonight*. [online] Available at: https://www.studytonight.com/computer-architecture/pipelining [Accessed 28 May 2019].
29. Japan.xilinx.com. (n.d.). *pragma HLS unroll*. [online] Available at: https://japan.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/uyd1504034366571.html [Accessed 27 May 2019].
30. Xilinx.com. (n.d.). *Vivado Design Suite*. [online] Available at: https://www.xilinx.com/products/design-tools/vivado.html [Accessed 29 May 2019].

31. Xilinx.com. (n.d.). [online] Available at: https://www.xilinx.com/support/documentation/backgrounders/vivado-hlx.pdf [Accessed 29 May 2019].
32. Xilinx.com. (n.d.). *Xilinx Software Development Kit (XSDK)*. [online] Available at: https://www.xilinx.com/products/design-tools/embedded-software/sdk.html [Accessed 29 May 2019].
33. Xilinx.com. (n.d.). *Getting Started with Xilinx SDK*. [online] Available at: https://www.xilinx.com/html_docs/xilinx2018_3/SDK_Doc/sdk_getting_started/sdk_getting_started.html [Accessed 30 May 2019].
34. Xilinx.com. (n.d.). *Using Xilinx SDK*. [online] Available at: https://www.xilinx.com/html_docs/xilinx2018_2/SDK_Doc/index.html [Accessed 31 May 2019].