B. TECH. PROJECT REPORT

On

PERFORMING MATHEMATICAL OPERATIONS AND PARALLELISING THEM ON FPGA

BY

PALASH DURUGKAR
SUNIL KUMAR PAL

DISCIPLINE OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY INDORE
December 2017

PERFORMING MATHEMATICAL OPERATIONS AND PARALLELISING THEM ON FPGA

A PROJECT REPORT

Submitted in partial fulfillment of the requirements for the award of the degrees

of
BACHELOR OF TECHNOLOGY
in
ELECTRICAL ENGINEERING

Submitted by:
PALASH DURUGKAR
SUNIL KUMAR PAL

Guided by:
Dr. SRIVATHSAN VASUDEVAN, Faculty, Electrical Engineering
DR. SATYA BULUSU, Faculty, Chemistry Department

INDIAN INSTITUTE OF TECHNOLOGY INDORE
December 2017

CANDIDATE'S DECLARATION

We hereby declare that the project entitled **"Performing Mathematical Operations and Parallelizing Them in FPGA"** submitted in partial fulfillment for the award of the degree of Bachelor of Technology in 'ELECTRICAL ENGINEERING' completed under the supervision of **Dr. Srivathsan Vasudevan, Asst. Professor, Electrical Engineering** and **Dr. Satya Bulusu, Asst. Professor, Chemistry,** IIT Indore is an authentic work.

Further, I/we declare that I/we have not submitted this work for the award of any other degree elsewhere.

Signature and name of the student(s) with date

Palash Durugkar **Sunil Kumar Pal**

CERTIFICATE by BTP Guide(s)

It is certified that the above statement made by the students is correct to the best of my/our knowledge.

Signature of BTP Guide(s) with dates and their designation

Dr. S. Vasudevan	**Dr. S. Bulusu**
Asst. Professor, Electrical Engg.	**Asst. Professor, Chemistry**
IIT INDORE	**IIT INDORE**

PREFACE

This report on “PERFORMING MATHEMATICAL OPERATIONS AND PARALLELISING THEM IN FPGA" is prepared under the guidance of Dr. Srivathsan Vasudevan, Professor, Electrical Engineering and Assistant Professor, Dr. Satya Bulusu, Professor, Chemistry Department.

Through this report we have tried to present the system and architecture for solving intensive high-level computation problems for various computational fields. The system is an effort towards making such computational fields realize the potential of application of FPGA based hardware designs and their optimizations for mathematical problem computations and data analysis.

We have tried to the best of our abilities and knowledge to explain the content in a lucid manner. We have also added designs and figures to make it more illustrative.

PALASH DURUGKAR and SUNIL KUMAR PAL
B.Tech. IV Year
Discipline of Electrical Engineering
IIT Indore

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my supervisors Dr. Srivathsan Vasudevan and Dr. Satya Bulusu for providing their invaluable guidance, comments, suggestions and kind support throughout the course of the project.

It is through their help and support, due to which we became able to complete the design and technical report.

Without their support this report would not have been possible.

PALASH DURUGKAR and SUNIL KUMAR PAL
B.Tech. IV Year
Discipline of Electrical Engineering
IIT Indore

ABSTRACT

With the rise of Field Programmable Gate Array (FPGA) Programming, the application of hardware designing and implementation is becoming ubiquitous. FPGA can be used to design a controller in a single chip Integrated Circuit (IC). Most of the research oriented fields today, involve intensive high-level computation problems to be solved within minimum possible time frame and greatest of ease. This project on Performing of Mathematical Operations and parallelizing them in FPGA, is considered to be a key problem solver for such fields. Through this work, we aim to offer an architecture of a setup which could perform a pre-specified mathematical operation or a set of mathematical operations on large chunks of data packets within very low time frames. This paper deals with designing of a high speed UART using Verilog Hardware Description Language .The Universal Asynchronous Receiver Transmitter (UART) is a device used for serial communication between computers and other peripheral devices. Together combining the design structure and parallelizing techniques to reduce latency and optimization on FPGA, would be considered a new innovation idea in intensive high-level computational fields by providing a reprogrammable software that can demonstrate the potential of hardware development and FPGA programming.

TABLE OF CONTENTS

i. Candidate’s Declaration

ii. Supervisor’s Certificate

iii. Preface

iv. Acknowledgements

v. Abstract

1. Introduction
    1.1. FPGAs-A Brief Introduction
    1.2. Basys 3 Board
    1.3. XILINX Vivado Design Suite
    1.4. RealTerm – Serial/TCP Terminal Software
2. Design Implementation Using HDLs(VHDL /Verilog)
3. Design Implementation using HLLs(C/C++)
4. Applications
5. Conclusion and Scope for Future Work
6. References

LIST OF FIGURES

Figure 1 – Island Type View of FPGA Architecture

Figure 2 – FPGA Components

Figure 3 – Logic Cell

Figure 4 – Basys 3 Board Features

Figure 5 - USB-UART Connection on FPGA

Figure 6 - RealTerm Screen Capture

Figure 7 – Structure Overview

Figure 8 – Data Flow in the System

Figure 9 – Steps Followed in using the software

Figure 10 – Data Packet in UART

Figure 11 – Final Synthesised Design of System

Figure 12 – State Machine Representation of Control Module

Figure 13 – RTL level design of system

Figure 14 – I/O Planning

Figure 15 – RTL Implementation

Figure 16 - Directives

Figure 17 - Unrolled Loops with Latency

Figure 18 - Loop Flattening

Figure 19 - Perfect and Semi Perfect Loops

Figure 20 - Before Pipelining and Loop Unrolling

Figure 21 - After Pipelining

Figure 22 - After Loop Unrolling

Figure 23 - Final RTL Design

Figure 24 - timing diagram of operation

Figure 25 - GPU and CPU relationship to accelerate work

LIST OF TABLES

Table 1 - Instruction executions in pipelined processor

Table 2 - Control Dependency

Table 3 - stallings in pipeline

Table 4 - Operand Forwarding

CHAPTER 1: INTRODUCTION

1.1 FPGAs – A Brief Introduction

The word digital has made a dramatic impact on our society. More significant is a continuous trend towards digital solutions in all areas – from electronic instrumentation, control, data manipulation, signals processing, telecommunications etc., to consumer electronics. Development of such solutions has been possible due to good digital system design and modeling techniques. Digital ICs have become universally standardized and have been accepted for use. Whenever a designer has to realize a digital function, he uses a standard set of ICs along with a minimal set of additional discrete circuitry. Field programmable Gate Arrays (FGPAs) are reconfigurable devices that can be electrically programmed to implement a wide variety of logic circuits. An FPGA consists of a uniform array of programmable logic structures that are interconnected by a configurable routing grid. Originally designed to serve as prototyping devices for testing and demonstrating the functionality of digital circuits, FPGAs are now an integral part of high performance systems that include digital, analog and RF components. The revolutionary success of these reconfigurable devices can be attributed to Flexibility in design implementation. The ability to instantly reprogram the FPGA with various circuits at no extra cost promotes reusability of the device, allows rapid design verification and reduces non-recurring expenditure.

- The availability of high performance FPGA based IP cores for popular applications. This allows FPGAs to function as plug and play devices in System on Chip (SoC) platforms.
- Enhanced performance due to the incorporation of specialized hardware like multipliers, high speed memories etc., into the FPGA. Additionally, the specialized blocks are also optimized for area and power which making the FPGA more competitive with custom chips.

Figure 1 shows the island style of FPGA architecture.

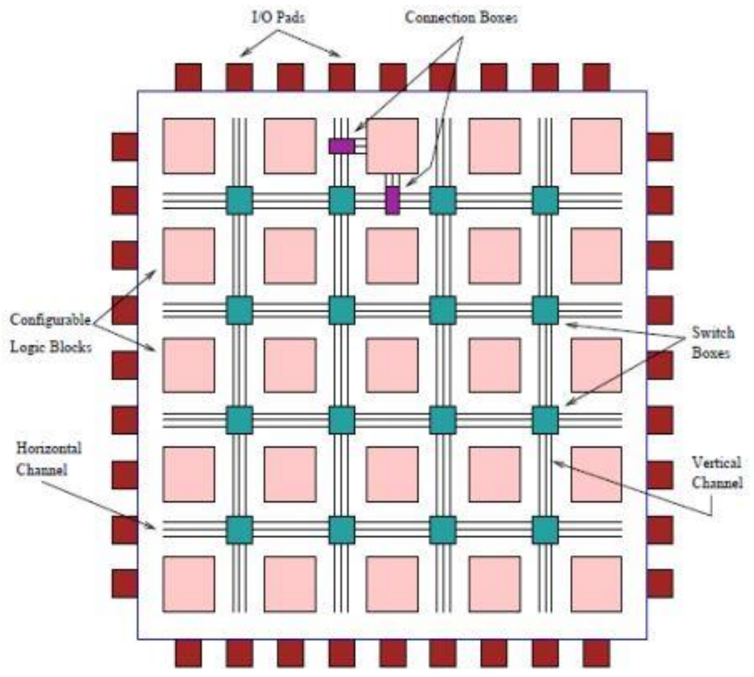


Figure 1. FPGA Architecture [3]

As we mentioned at the beginning, FPGA is an integrated circuit. However the difference from the others is that they can be configurable as we wish. We can explain this as follows: In ordinary or standard ICs which cannot be programmable, there are fixed interconnections between the transistors. Unless they are burned or another unfortunate event does not come, they cannot be changed.

We may consider FPGAs as crude ICs of which their transistors were produced independently. Interconnections between transistors can be done according to the function we defined, then they perform the function we want. So theoretically, any operation that comes to our minds can be done by FPGAs, depending on the transistor capacity.

FPGA is basically consists of Logic Cells, I/O Blocks (Input/Output) and interconnections.

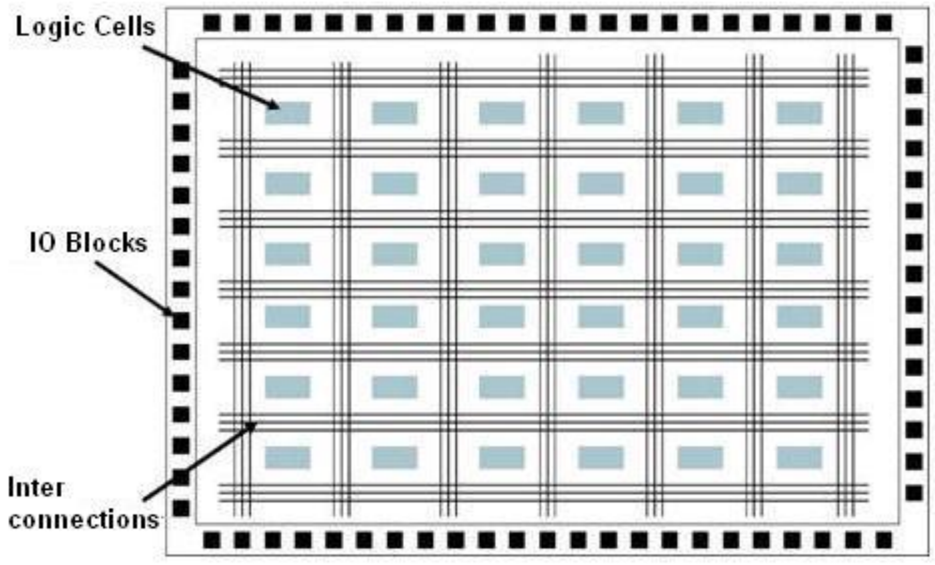


Figure 2 FPGA parts [3]

Logic Cells form the main structure of FPGAs. A Logic-Cell consists of one **Lookup Table (LUT)**, one **D-Flip Flop** and one **2 to 1 Multiplexer**.

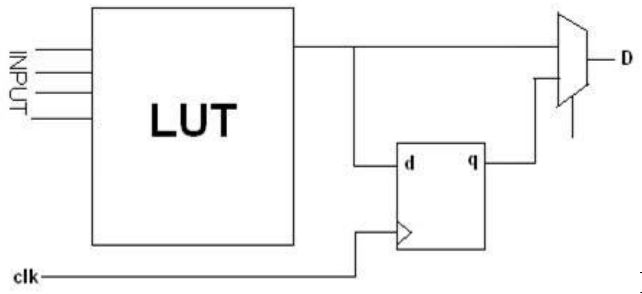


Figure 3. Logic Cell [3]

LUTs are actually small memories (RAM) that fulfil logic operations...
As a result of combination of thousands of Logic Cells, complex and large programs are created.

Interconnections of logic cells are provided by programmable switches and matrix formed data paths (according to the installed program FPGA).

FPGA design defines the set of connections between logic functions, by determining functions of each of the logic cells and status (open /closed) of programmable switches.

RAM Blocks

In almost all of today's FPGAs, memory units called RAMs are allocated. They are used for temporary storage needs which occur during the operation of logic circuits. This RAMs can support single or multiple access. With multiple access, multiple applications can run read /write operations on the RAM. Multiple access is a good solution for transferring data between different process blocks that have different clocks. For great RAM needs, there are Block RAMs in FPGA. However there are small scattered (distributed) RAMs which are interspersed among the logic cells for small data storage needs.

One of the most important feature of FPGAs is the ability to do parallel processing. Ordinary ICs cannot do parallel processing, or they can do it limited. Whereas dozens of or perhaps thousands of parallel processing can be done simultaneously depending on the application with FPGAs. This makes FPGAs unique in applications that require parallel processing. If you use standard ICs like microprocessors, you start to get second frame after processing three operations (get, filter and send) for the first frame. If this process cannot be fast enough, you may miss the next frame. FPGAs do all these operations in parallel. That means while we processing of filtering the first frame, we might begin to take the second frame. And while us sending the first frame to the output, we begin to filter the second frame and get the third frame at the same time.

Furthermore, filtering process requires extensive multiplication process. With standard ICs, we have to do this process sequentially. Whereas this process can be done in parallel with an FPGA so it can be done very quickly.

In summary, FPGAs are hardware-programmable integrated circuits that provide us parallel processing capabilities and opportunity to change the internal structure and its function according to the desired application.

Intellectual Property (IP)

Intellectual Property (IP) refers to the built-in functions that are optimised for speed and performance aspects according to a specific FPGA family. While simple functions are provided free by the manufacturer; more advanced functions are usually required to pay a certain fee.

1.2 BASYS 3 Board

The Basys3 board is a complete, ready-to-use digital circuit development platform based on the latest Artix-7™ Field Programmable Gate Array (FPGA) from Xilinx. With its high-capacity FPGA (Xilinx part number XC7A35T-1CPG236C , low overall cost, and collection of USB, VGA, and other ports, the Basys3 can host designs ranging from introductory combinational circuits to complex sequential circuits like embedded processors and controllers. It includes enough switches, LEDs and other I/O devices to allow a large number designs to be completed without the need for any additional hardware, and enough uncommitted FPGA I/O pins to allow designs to be expanded using Digilent Pmods or other custom boards and circuits.

The Artix-7 FPGA is optimized for high performance logic, and offers more capacity, higher performance, and more resources than earlier designs. Artix-7 35T features include:

- 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 450MHz;
- On-chip analog-to-digital converter (XADC).

The Basys3 also offers an improved collection of ports and peripherals, including:

- 16 user switches
- 16 user LEDs
- 5 user pushbuttons
- 4-digit 7-segment display
- Three Pmod ports
- Pmod for XADC signals
- 12-bit VGA output
- USB-UART Bridge
- Serial Flash
- Digilent USB-JTAG port for FPGA programming and communication
- USB HID Host for mice, keyboards and memory sticks

Walk Around the Board

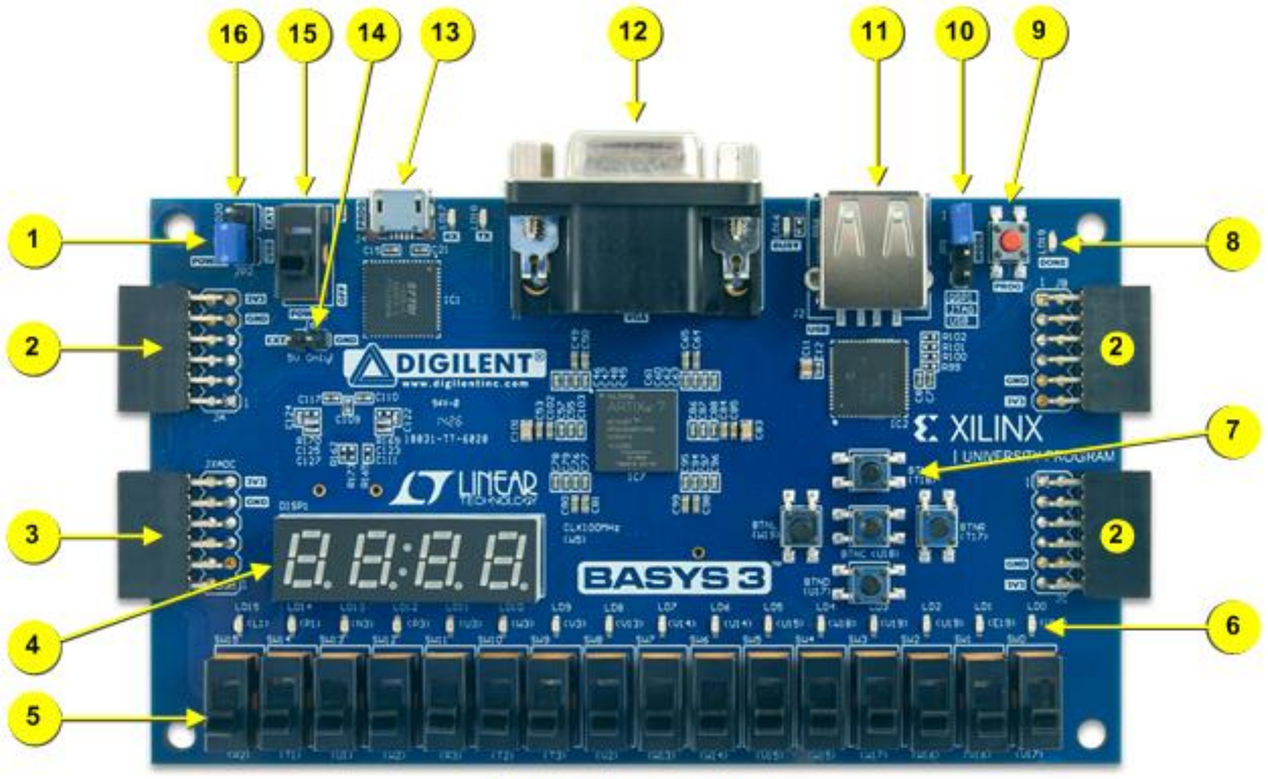
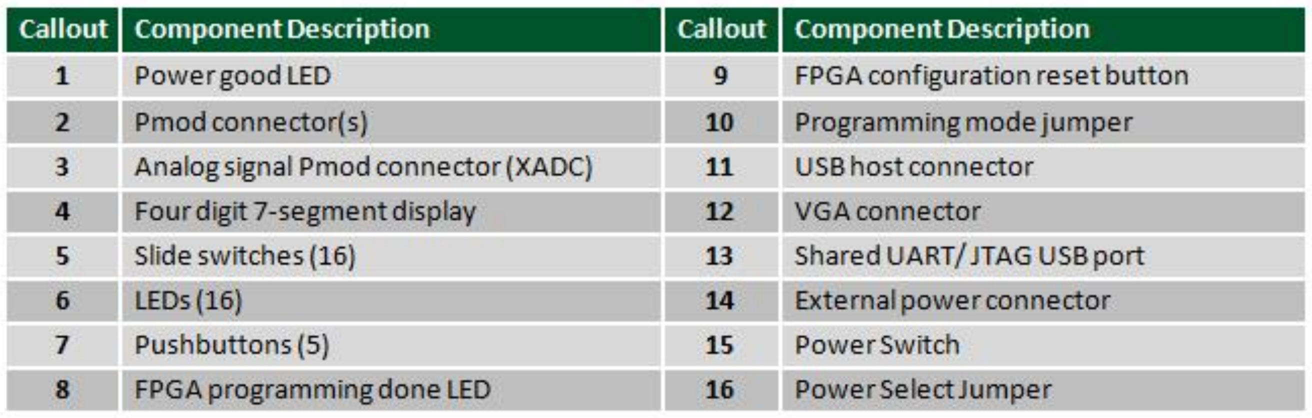


Figure 1. Basys3 board features



Callout	Component Description	Callout	Component Description
1	Power good LED	9	FPGA configuration reset button
2	Pmod connector(s)	10	Programming mode jumper
3	Analog signal Pmod connector (XADC)	11	USB host connector
4	Four digit 7-segment display	12	VGA connector
5	Slide switches (16)	13	Shared UART/ JTAG USB port
6	LEDs (16)	14	External power connector
7	Pushbuttons (5)	15	Power Switch
8	FPGA programming done LED	16	Power Select Jumper

Figure 4. Basys 3 Board Features [4]

1.3 XILINX Vivado Design Suite

The Basys3 works with Xilinx's new high-performance Vivado ® Design Suite. Vivado includes many new tools and design flows that facilitate and enhance the latest design methods. It runs faster, allows better use of FPGA resources, and allows designers to focus their time evaluating design alternatives. The System Edition includes an on-chip logic analyzer, high-level synthesis tool, and other cutting-edge tools, and the free "Webpack" version allows Basys3 designs to be created at no additional cost.

USB-UART Bridge (Serial Port)

The Basys3 includes an FTDI FT2232HQ USB-UART bridge (attached to connector J4) that allows you to use PC applications to communicate with the board using standard Windows COM port commands. Free USB-COM port drivers, convert USB packets to UART/serial port data. Serial port data is exchanged with the FPGA using a two-wire serial port (TXD/RXD). After the drivers are installed, I/O commands can be used from the PC directed to the COM port to produce serial data traffic on the B18 and A18 FPGA pins.

Two on-board status LEDs provide visual feedback on traffic flowing through the port: the transmit LED (LD18) and the receive LED (LD17). Signal names that imply direction are from the point-of-view of the DTE (Data Terminal Equipment), in this case the PC.

The FT2232HQ is also used as the controller for the Digilent USB-JTAG circuitry, but the USB-UART and USB-JTAG functions behave entirely independent of one another. Programmers interested in using the UART functionality of the FT2232 within their design do not need to worry about the JTAG circuitry interfering with the UART data transfers, and vice-versa. The combination of these two features into a single device allows the Basys3 to be programmed, communicated with via UART, and powered from a computer attached with a single Micro USB cable. The connections between the FT2232HQ and the Artix-7 are shown in the below figure.

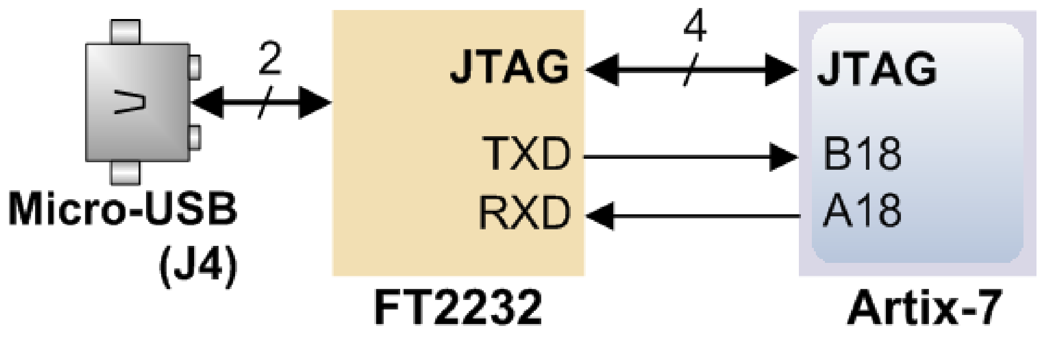
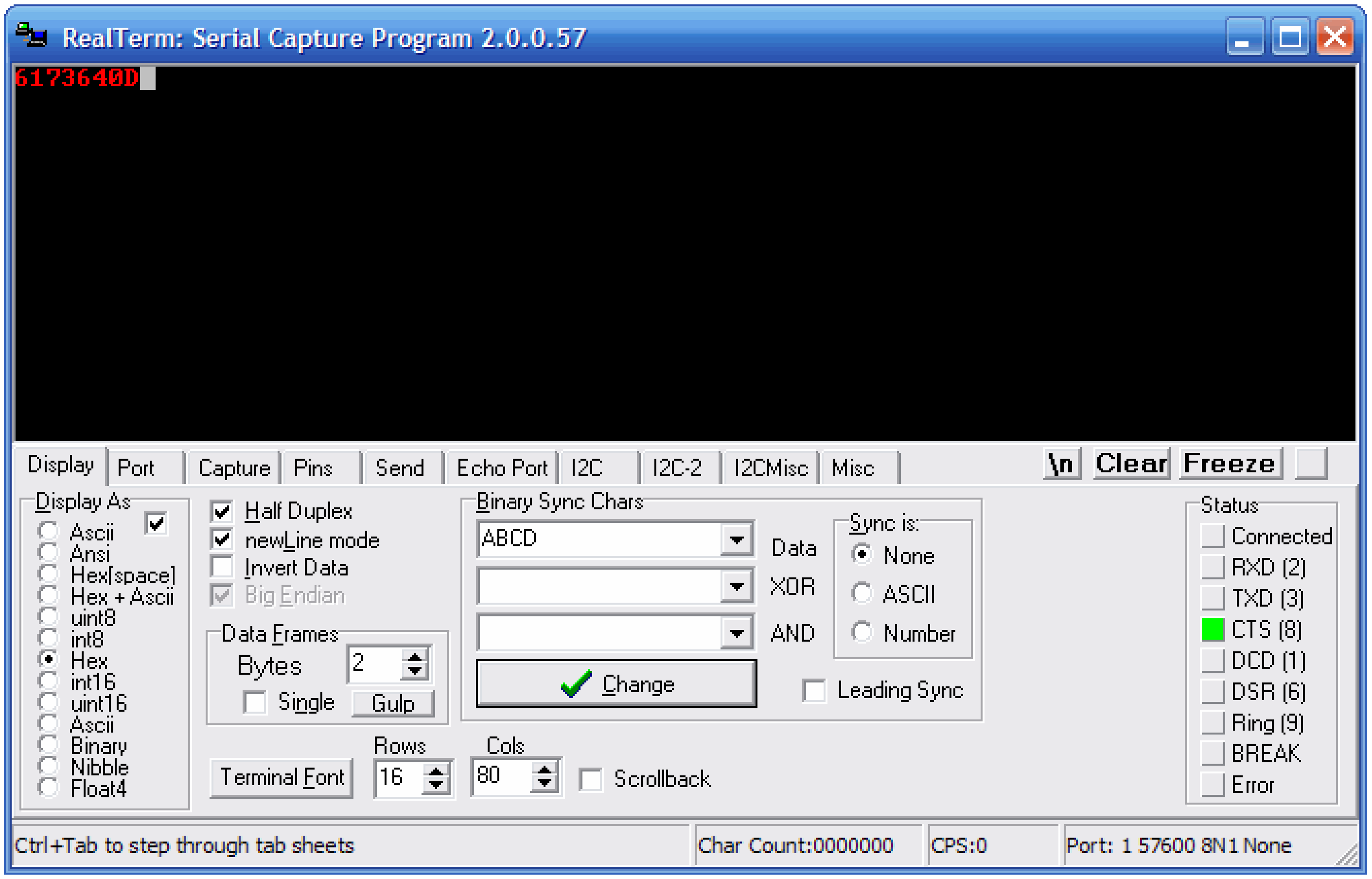


Figure 5. USB-UART Connection on FPGA [4]

1.4 RealTerm – Serial/TCP Terminal Software

Realterm is an engineer's terminal program specially designed for capturing, controlling and debugging binary and other difficult data streams. It is the best tool for debugging comms.



```
6173640D
```

Display As: Hex; Half Duplex ✓; newLine mode ✓; Bytes: 2; Binary Sync Chars: ABCD; Sync is: None; Rows: 16; Cols: 80; CTS (8) 🟩

Ctrl+Tab to step through tab sheets | Char Count:0000000 | CPS:0 | Port: 1 57600 8N1 None

Figure 6. RealTerm Screen Capture [5]

RealTerm is a terminal software solution that enables network administrators to capture, manage or debug binary and other data streams. If you are looking for a tool that can debug dialling modems or BBS, you might need to keep on looking as RealTerm does not support such devices.

Its interface is meant to be as intuitive as possible, as each main function is detailed in its own tab

We can also modify the port, the parity and the data bits, so to come up with a configuration that best suits your needs.

One of the benefits of RealTerm is that it comes with support for hotkeys, which can help you save a lot of time when handling difficult data streams, thus allowing you to focus more on the debugging process than on operating the application.

All in all, RealTerm provides users with numerous functions that enable them to debug binary or other types of data streams, yet advanced skills are required in order to make the most of its features.

CHAPTER 2: HARDWARE DESIGN IMPLEMENTATION THROUGH HDLs

The main overview of the architecture could be explained as followed in the flow chart below:

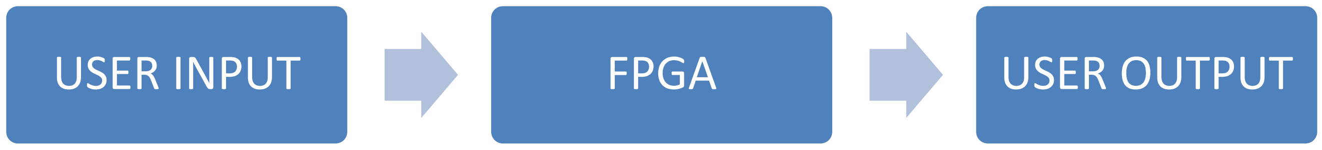


Figure 7. Structure Overview

Assuming that the user has decided a fixed number of input data/numbers (very large quantity of data) to be sent from a host PC and he/she wants that some mathematical operations or a set of mathematical operations to be performed on the numbers/data in a pre-defined fashion and the resulting data to be shown back to the user on the host PC.

Firstly, we need to be sure about the architecture of the process, how the data should be flowed from the PC, through the hardware FPGA, perform mathematical operations and the resultants to go through the hardware back to the host PC.

Secondly, we need to devise a way so that the hardware FPGA and the host PC could interact with each other, or communicate with each other for data flow.

The user inputs the data through the keyboard also enabling the hardware through switches and making it ready for incoming data and setting up the terminal software. Since, the main motive is to design a digital logic system, the data flowing will be binary (8-bit). We can use any of, serial or parallel communication for the data flow, but it is more feasible to use serial communication for the same because it is easier to use and understand, easier to implement on chip, low cost and placing space. Hence, Serial Communication is more preferable over the Parallel Communication Techniques.

The data is received by the receiver (first serial communication block) which then transmits the data further serially to be stored in two separate memory locations. These memory locations are designed to store large quantity of data at a place, and keep them stored until an instruction is given to remove data stored one at a time. We can set the depth of the memory according to the user preferences and needs and the fashion in which the data has to be stored (no. of bits).

Now that we have all the data stored, we can give instructions to remove data one at a time from the two memory locations and perform mathematical operations on the corresponding pairs of data/numbers and make the resultants to move further in another memory location. This final memory location would be containing all the resultant datas, which need to be sent back to the user.

The resultants move on to the transmission block (second serial communication block) one at a time, which converts these serial data back to parallel (back in the fashion in which they were input) and through some digital logic could be made available for the user to be seen back on the terminal software.

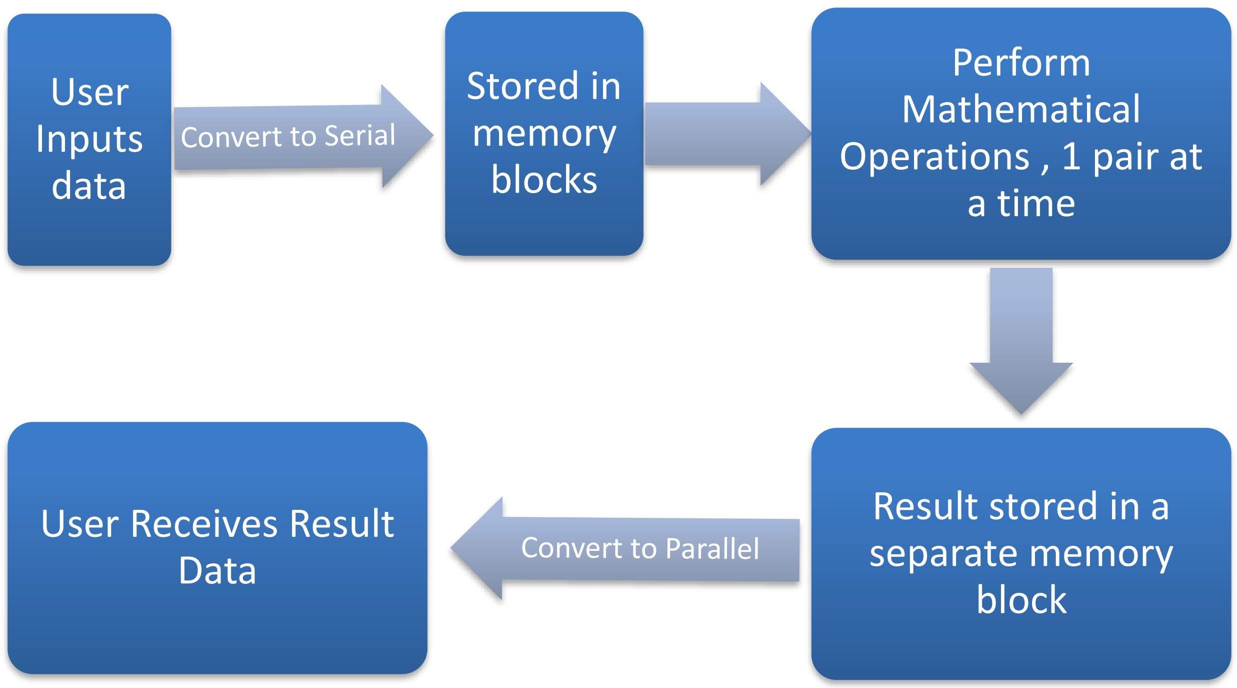

User Inputs data
Convert to Serial
Stored in memory blocks
Perform Mathematical Operations , 1 pair at a time
Result stored in a separate memory block
Convert to Parallel
User Receives Result Data

Figure 8. Data Flow in System

This whole process could be very well designed in XILINX Vivado Design Suite, given the knowledge of HDLs (Hardware Description Languages) like, Verilog or VHDL. These languages are specially designed for hardware programming. Since, FPGA is reprogrammable and we can possibly design the system for any type of mathematical operations using HDLs given its ease of understanding and structural flow, it gives us an advantage.

Also, the understanding of the serial communication protocol, the whole process where the conversion of parallel data from a controlling device like a CPU into serial form, its transmission in serial to the receiving block, which then converts the serial data back into parallel data for the receiving device, has been explained in detail in the later stages of the report.

We had certain source files available to us, mainly the transmission and reception files for data communication. The only problem lies in interconnecting all the modules and components through hardware programming without any errors and successful synthesis and implementation designs as well as the placing the I/O pins (switches and LEDs for the input and output ports respectively), to finally get a bitstream file for the entire project. We can easily design a VHDL/Verilog code for the same, which has been shown in the later stages of the report.

Below mentioned are the various stages that we need to go through while designing in XILINX Vivado Design Suite:

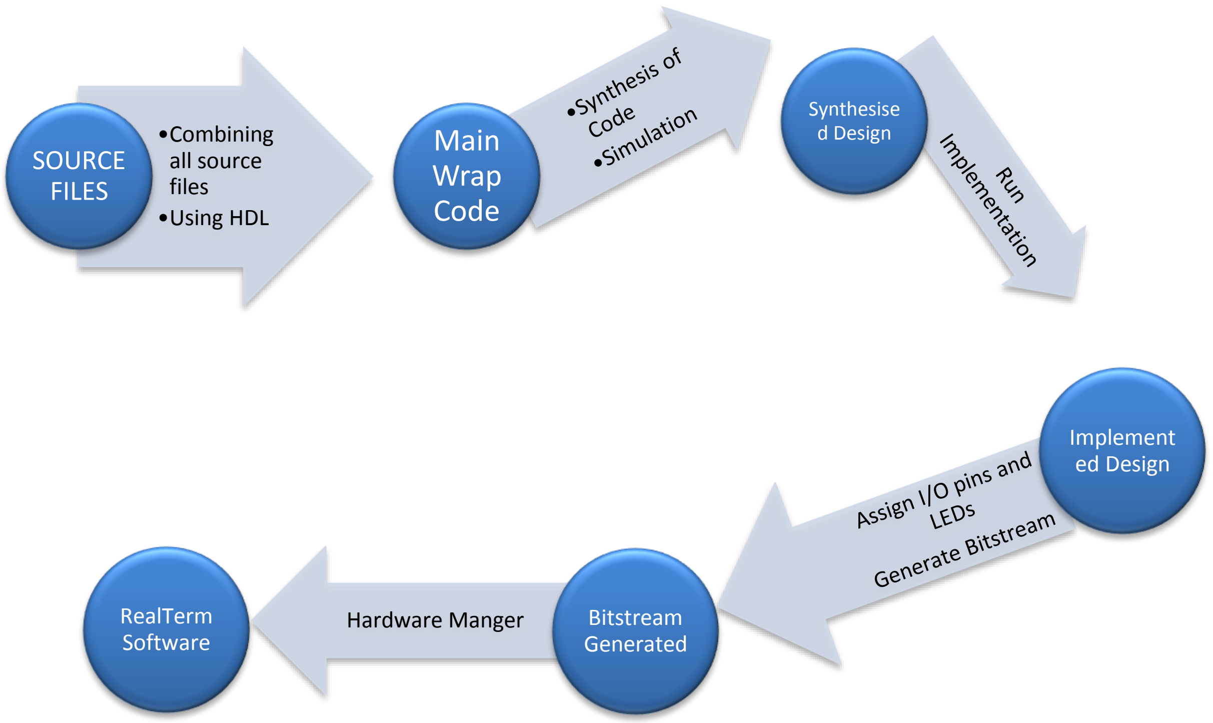


Figure 9. Steps using the softwares

SERIAL COMMUNICATION PROTOCOL → UART COMMUNICATION

Parallel vs. Serial

Parallel interfaces transfer multiple bits at the same time. They usually require **buses** of data - transmitting across eight, sixteen, or more wires. Data is transferred in huge, crashing waves of 1's and 0's.Serial interfaces stream their data, one single bit at a time. These interfaces can operate on as little as one wire, usually never more than four. Think of the two interfaces as a stream of cars: a parallel interface would be the 8+ lane mega-highway, while a serial interface is more like a two-lane rural country road. Over a set amount of time, the mega-highway potentially gets more people to their destinations, but that rural two-laner serves its purpose and costs a fraction of the funds to build.

Parallel communication certainly has its benefits. It's fast, straightforward, and relatively easy to implement. But it requires many more input/output (I/O) lines.So, we often opt for serial communication, sacrificing potential speed for pin real estate.

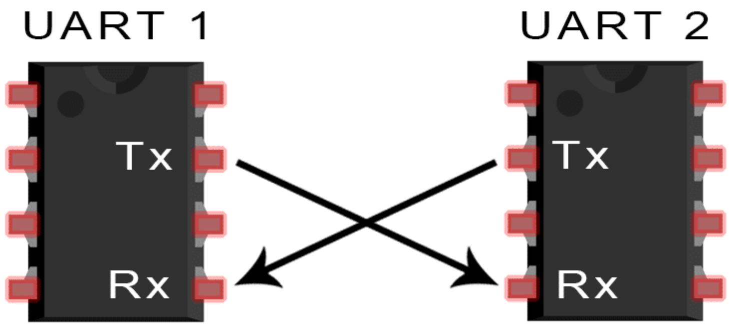
Asynchronous serial communication means that data is transferred **without support from an external clock signal**. This transmission method is perfect for minimizing the required wires and I/O pins, but it does mean we need to put some extra effort into reliably transferring and receiving data.

SERIAL COMMUNICATION PROTOCOL (UART COMMUNICATION)

While USB has almost completely replaced those old cables and connectors, UARTs are definitely not a thing of the past. You'll find UARTs being used in many DIY electronics projects to connect GPS modules, Bluetooth modules, and RFID card reader modules to your Raspberry Pi, Arduino, or other microcontrollers.
UART stands for Universal Asynchronous Receiver/Transmitter. It's not a communication protocol like SPI and I2C, but a physical circuit in a microcontroller, or a stand-alone IC. A UART's main purpose is to transmit and receive serial data.

One of the best things about UART is that it only uses two wires to transmit data between devices.

In UART communication, two UARTs communicate directly with each other. The transmitting UART converts parallel data from a controlling device like a CPU into serial form, transmits it in serial to the receiving UART, which then converts the serial data back into parallel data for the receiving device. Only two wires are needed to transmit data between two UARTs. Data flows from the Tx pin of the transmitting UART to the Rx pin of the receiving UART.



UARTs transmit data *asynchronously*, which means there is no clock signal to synchronize the output of bits from the transmitting UART to the sampling of bits by the receiving UART. Instead of a clock signal, the transmitting UART adds start and stop bits to the data packet being transferred. These bits define the beginning and end of the data packet so the receiving UART knows when to start reading the bits.
When the receiving UART detects a start bit, it starts to read the incoming bits at a specific frequency known as the *baud rate*. Baud rate is a measure of the speed of data transfer, expressed in bits per second (bps). Both UARTs must operate at about the same baud rate. The baud rate between the transmitting and receiving UARTs can only differ by about 10% before the timing of bits gets too far off.
Both UARTs must also must be configured to transmit and receive the same data packet structure.

UART transmitted data is organized into *packets*. Each packet contains 1 start bit, 5 to 9 data bits (depending on the UART), an optional *parity* bit, and 1 or 2 stop bits:

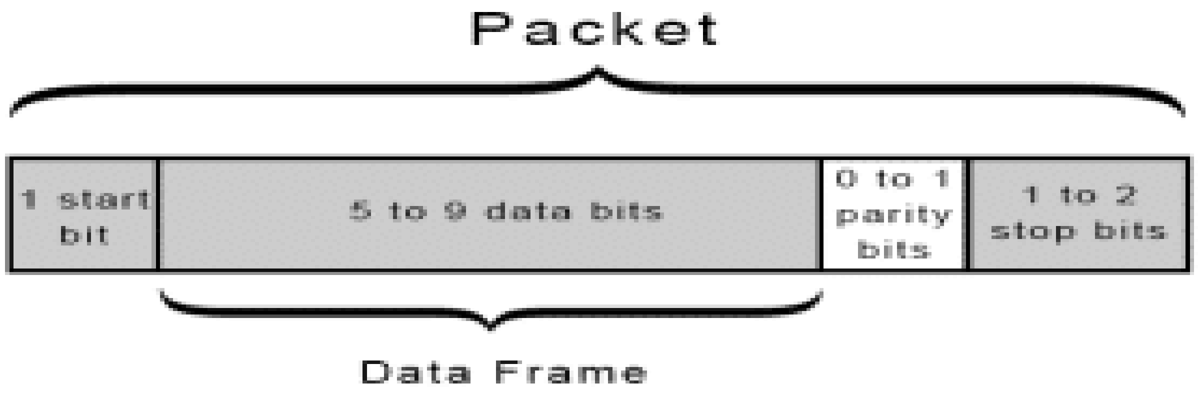


Figure 10. Data Packet in UART [6]

START BIT

The UART data transmission line is normally held at a high voltage level when it's not transmitting data. To start the transfer of data, the transmitting UART pulls the transmission line from high to low for one clock cycle. When the receiving UART detects the high to low voltage transition, it begins reading the bits in the data frame at the frequency of the baud rate.

DATA FRAME

The data frame contains the actual data being transferred. It can be 5 bits up to 8 bits long if a parity bit is used. If no parity bit is used, the data frame can be 9 bits long. In most cases, the data is sent with the least significant bit first.

PARITY

Parity describes the evenness or oddness of a number. The parity bit is a way for the receiving UART to tell if any data has changed during transmission. Bits can be changed by electromagnetic radiation, mismatched baud rates, or long distance data transfers. After the receiving UART reads the data frame, it counts the number of bits with a value of 1 and checks if the total is an even or odd number. If the parity bit is a 0 (even parity), the 1 bits in the data frame should total to an even number. If the parity bit is a 1 (odd parity), the 1 bits in the data frame should total to an odd number. When the parity bit matches the data, the UART knows that the transmission was free of errors. But if the parity bit is a 0, and the total is odd; or the parity bit is a 1, and the total is even, the UART knows that bits in the data frame have changed.

STOP BITS

To signal the end of the data packet, the sending UART drives the data transmission line from a low voltage to a high voltage for at least two bit durations.

ADVANTAGES AND DISADVANTAGES OF UARTS

No communication protocol is perfect, but UARTs are pretty good at what they do. Here are some pros and cons to help you decide whether or not they fit the needs of your project:

ADVANTAGES

- Only uses two wires
- No clock signal is necessary
- Has a parity bit to allow for error checking
- The structure of the data packet can be changed as long as both sides are set up for it
- Well documented and widely used method

DISADVANTAGES

- The size of the data frame is limited to a maximum of 9 bits
- Doesn’t support multiple slave or multiple master systems
- The baud rates of each UART must be within 10% of each other

WORKING AND FINAL DESIGN

COMPONENTS INVOLVED IN THE DESIGN

1. **First In First Out (FIFO) Memory Block**
2. **UART Serial Communication Blocks – Receiver and Transmission Block**
3. **Clock Divider Module**
4. **Selection Module**
5. **Adder Block**

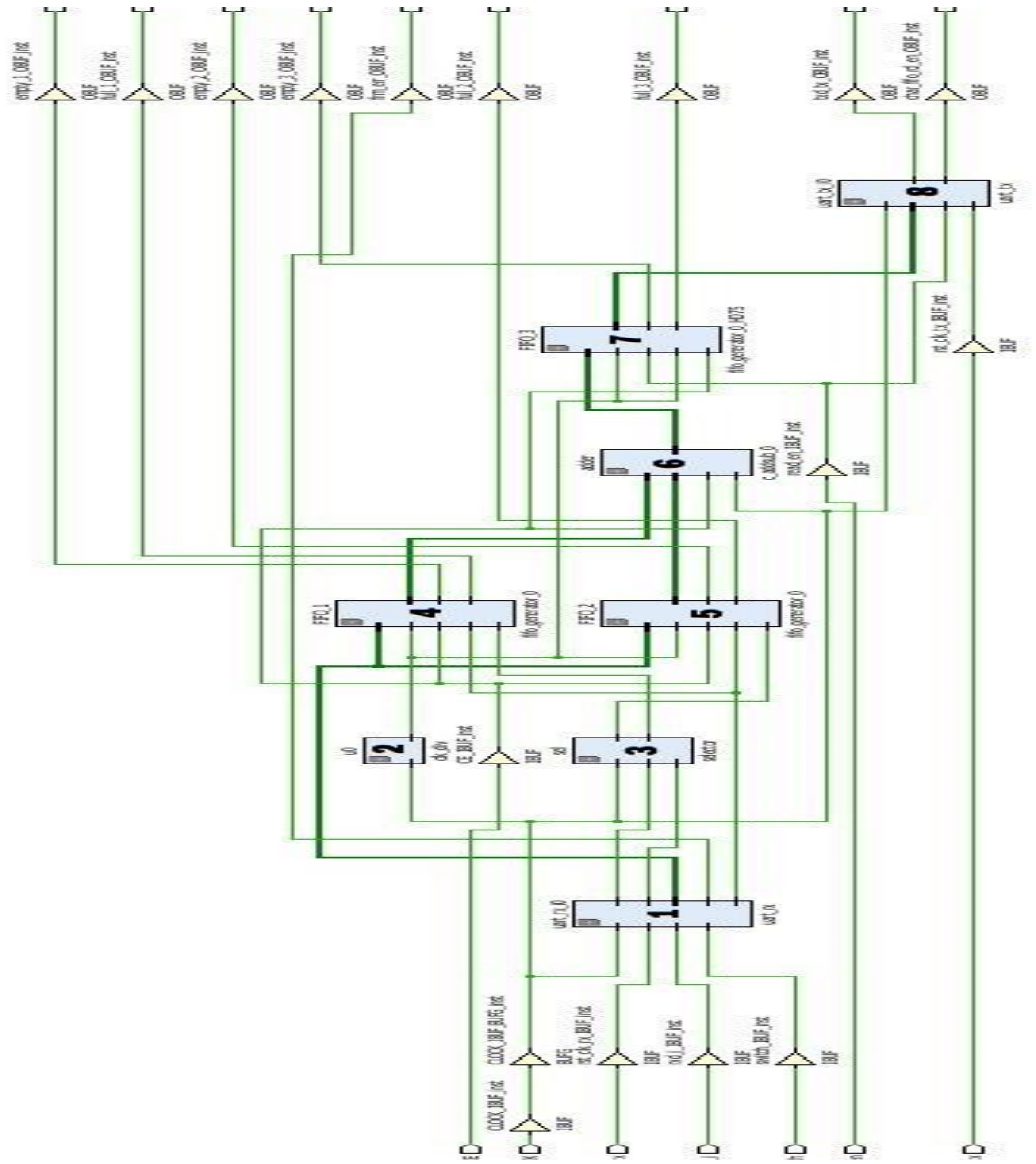


Figure 11. Final Synthesized Design of System

- **FIRST IN FIRST OUT (FIFO) MEMORY BLOCK**

As the name suggests, FIFOs are a special kind of memory blocks for any digital logic system, whatever be the depth of this memory block (the no. of data entries that it can store) , whichever number was the first to be entered will be the first one to come out of the block when given certain instruction.
The FIFO Generator core is a fully verified first-in first-out memory queue for use in any application requiring ordered storage and retrieval, enabling high-performance and area-optimized designs. The core provides an optimized solution for all FIFO configurations and delivers maximum performance (up to 500 MHz) while using minimum resources. This core can be customized using the Vivado IP customizers in the IP catalog as a complete solution with control logic already implemented, including management of the read and write pointers and the generation of status flags.

For this project, we have used FIFO as an in-built IP (Intellectual Property) from the IP Catalog available in the software, which is customizable according to user needs.

Here, the FIFO memory block is being used to store large data in groups, for example, a FIFO with a write depth of 16 and write width of 8, would be able to store 16 eight- bit numbers in it. This can be changed in the customization settings while designing the IP.

Also, another variation that has been used is the Independent Clocks Block RAM (which enables us to choose different clocks for both read and write operations simultaneously). This is necessary to improve the performance of the system on a whole. We cannot have slower clocks for writing the data into the FIFO if the depth is very much.

The key components/ports of the FIFO memory block being:

- read clock - clock followed when reading data from the FIFO
- write clock – clock followed when writing data into the FIFO
- read enable – tells the FIFO when to read
- write enable – tells the FIFO when to write
- data input port – data input through this port (our case 8-bit binary numbers)
- data output port - data output through this port (our case 8-bit binary numbers)
- full flag – indicates when the FIFO memory is complete and cannot take any more entries
- empty flag – indicates when the FIFO memory is completely empty

- **UART SERIAL COMMUNICATION BLOCKS**
  - **UART TRANSMISSION BLOCK**
    - **Metastability Hardener**
    - **Baud Generator**
    - **Control Module**
  - **UART RECIVER MODULE**
    - **Baud Generator**
    - **Control Module**

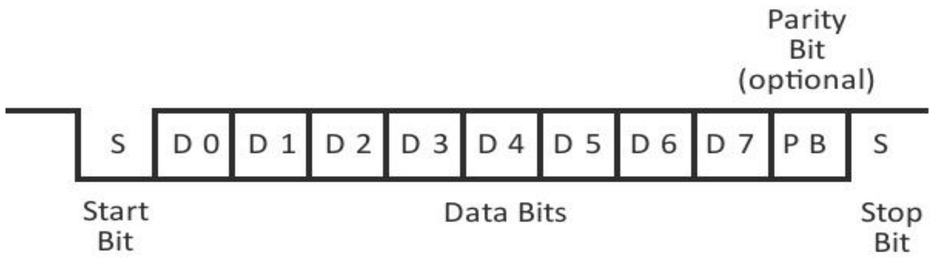
Metastability Hardener -- This is a basic meta-stability hardener; it double synchronizes an asynchronous signal onto a new clock domain.

Metastability is a phenomenon that can cause system failure in digital devices, including FPGAs, when a signal is transferred between circuitry in unrelated or asynchronous clock domains. If the data output signal resolves to a valid state before the next register captures the data, then the metastable signal does not negatively impact the system operation. But if the metastable signal does not resolve to a low or high state before it reaches the next design register, it can cause the system to fail. Continuing the ball and hill analogy, failure can occur when the time it takes for the ball to reach the bottom of the hill (a stable logic value 0 or 1) exceeds the allotted time, which is the register's tCO plus any timing slack in the path from the register. When a metastable signal does not resolve in the allotted time, a logic failure can result if the destination logic observes inconsistent logic states, that is, different destination registers capture different values for the metastable signal. [Refer appendix (1)]

Baud Generator -- Generates a 16x Baud enable. This signal is generated 16 times per bit at the correct baud rate as determined by the parameters for the system clock frequency and the Baud rate. This is essential for the sampling of bits incoming into both UART blocks for communication.

The Baud Generator generates a signal which ticks 16 times per bit at the baud rate of 9600 bps. This signal is essential for the sampling of data bits and to decide whether the data flowing is valid or not. [Refer appendix (2)]

CONTROL MODULE for Input -- Implements the state machines for doing RS232 reception. Based on the detection of the falling edge of the synchronized rxd input, this module waits 1/2 of a bit period (8 periods of baud_x16_en) to find the middle of the start bit, and resamples it. If rxd is still low it accepts it as a valid START bit, and captures the rest of the character, otherwise it rejects the start bit and returns to idle. After detecting the START bit, it advances 1 full bit period at a time (16 periods of baud_x16_en) to end up in the middle of the 8 data bits, where it samples the 8 data bits. After the last bit is sampled (the MSbit, since the Lsbit is sent first), it waits one additional bit period to check for the STOP bit. If the rxd line is not high (the value of a STOP bit), a framing error is signaled. Regardless of the value of the rxd, though, the module returns to the IDLE state and immediately begins looking for the start of the next character. The total cycle time through the state machine is 9 1/2 bit periods (not 10) – this allows for a mismatch between the transmit and receive clock rates by as much as 5%.



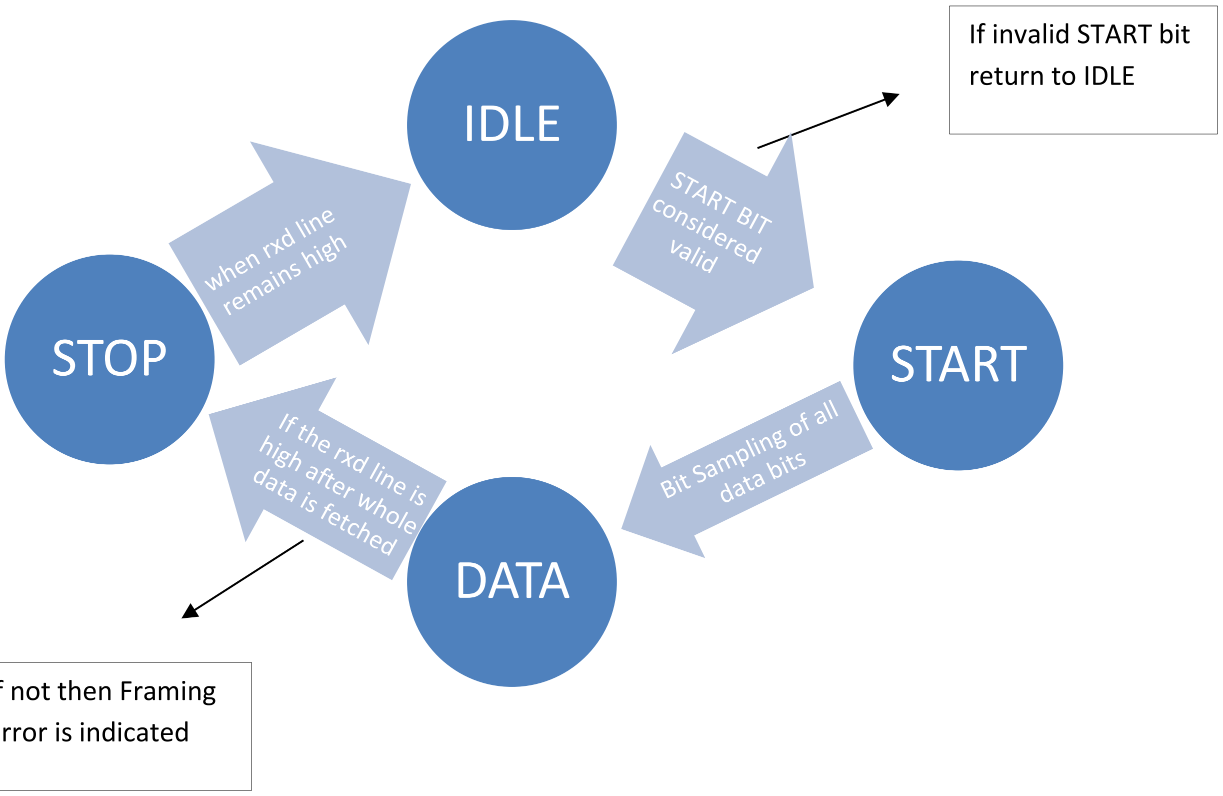

IDLE
START BIT considered valid
If invalid START bit return to IDLE
START
Bit Sampling of all data bits
DATA
If the rxd line is high after whole data is fetched
If not then Framing Error is indicated
STOP
when rxd line remains high

Figure 12. State Machine Representation of Control Module

CONTROL MODULE for Output-- Implements the state machines for doing RS232 transmission. Whenever a character is ready for transmission (as indicated by the empty signal from the character FIFO), this module will transmit the character. The basis of this design is a simple state machine. When in IDLE, it waits for the character FIFO to indicate that a character is available, at which time, it immediately starts transmission. It spends 16 baud_x16_en periods in the START state, transmitting the START condition (1'b0), then transitions to the DATA state, where it sends the 8 data bits (Lsbit first), each lasting 16 baud_x16_en periods, and finally going to the STOP state for 16 periods, where it transmits the STOP value (1'b1). On the last baud_x16_en period of the last data bit (in the DATA state), it issues the POP signal to the character FIFO. Since the SM is only enabled when baud_x16_en is asserted, the resulting pop signal must then be ANDed with baud_x16_en to ensure that only one character is popped at a time. On the last baud_x16_en period of the STOP state, the empty indication from the character FIFO is inspected; if asserted, the SM returns to the IDLE state, otherwise it transitions directly to the START state to start the transmission of the next character. There are two internal counters – one which counts off the 16 pulses of baud_x16_en, and a second which counts the 8 bits of data. The generation of the output (txd_tx) follows one complete baud_x16_en period after the state machine and other internal counters.

```
// Assert the rd_en to the FIFO for only ONE clock period

  assign char_fifo_rd_en = char_fifo_pop && baud_x16_en;
```

It is a similar finite state machine as previous one, the only addition is the logic through which every character is popped out to be seen on the terminal screen.

- **CLOCK DIVIDER MODULE**
  This module is essential because the read operations for the FIFO require a slower clock and since, on the Basys 3 Board there is a single clock of 100 MHz available to us, we need to reduce the clock frequency to much lower values for successful read operations.
  For our case, we have reduced the available clock frequency of 100MHz to 1Hz.
  [Refer appendix (3)]

- **SELECTION MODULE**
  This module is necessary because it enables us to decide about the latest input data entry, that, where among the 2 FIFOs it should be stored, because the mathematical operations have to be performed between a pair of data entries. On the FPGA, the input is given through switch, if low, then FIFO1, if high, then FIFO2.
  [Refer appendix (4)]

- **INVERSION MODULE**

  This module is necessary because through this we can decide when the transmission of the result data starts. On the FPGA, it is incorporated through a switch.
  [Refer appendix (5)]

- **ADDER BLOCK**

  This is the module where the main mathematical operation is taking place – the addition of 8-bit binary numbers. It has the 2 inputs connected to the data output ports of both FIFOs and an enable port which decides when the operation takes place. It passes on the 8-bit binary sum ahead to the result storing FIFO, after performing each successive operation.
  This block is being implemented as an in-built Intellectual Property (IP) in the XILINX Vivado software. It is always customizable according to user needs.

Combining all these components we obtained the final design of the complete architecture of the system to Perform Mathematical Operations with Interaction between the Hardware FPGA and host PC. Next, we will look at the main design wrapper (the main code which connects all these modules together).

MAIN PROGRAM CODE AND WORKING

Given below is the main structural code for the whole system which involves all of the above mentioned modules to obtain the final design of the system.

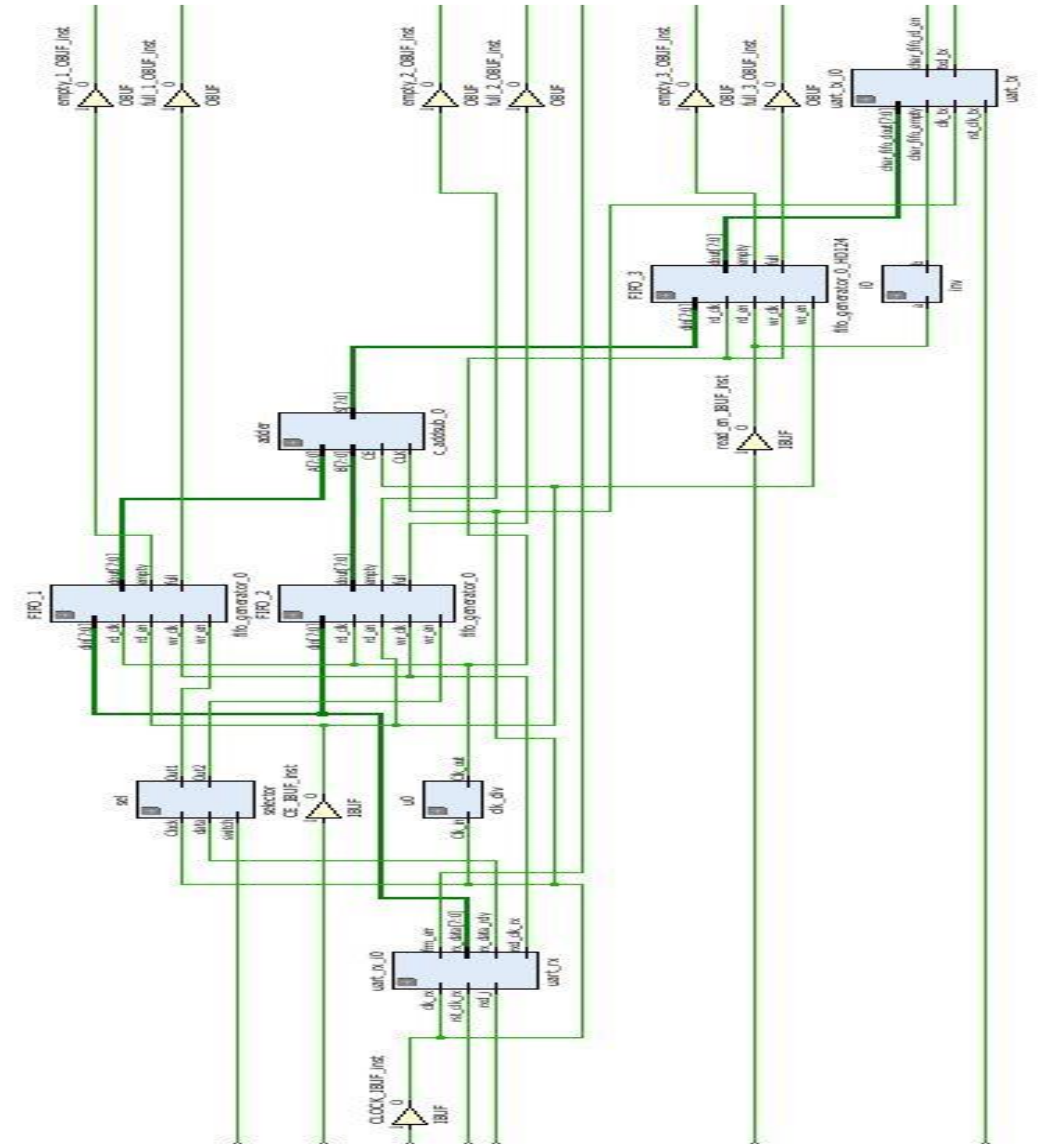
```
`timescale 1ns / 1ps


module main(
input             CLOCK,
input           rst_clk_rx,   // Active HIGH reset - synchronous to clk_rx
input           rxd_i,        // RS232 RXD pin - Directly from pad
output           frm_err,     // The STOP bit was not detected
output          full_1,
output          empty_1,
output          full_2,
output          empty_2,
output          full_3,
output          empty_3,
input        rst_clk_tx,      // Active HIGH reset - synchronous to clk_tx
output       char_fifo_rd_en, // Pop signal to the char FIFO
output       txd_tx,          // The transmit serial signal
input         switch,
input         CE,
input         read_en
);

  //***************************************************************************
  // Parameter definitions
  //***************************************************************************

    parameter BAUD_RATE    = 9600;              // Baud rate
    parameter CLOCK_RATE   = 100000000;
    parameter sys_clk = 100000000;          // 100 MHz system clock
    parameter clk_out = 1;    // 1 Hz clock output
    parameter max = sys_clk / (2*clk_out); // max-counter size

    //***************************************************************************
    // Wire Declarations
    //***************************************************************************
      wire c;
      wire write_clk;
      wire w1;
      wire w2;
      wire write_en;
      wire [7:0] data_out1;
      wire [7:0] data_out2;
      wire [7:0] sum;
      wire [7:0] data_in;
      wire [7:0] data_out3;
      wire fifo3_empty;
      wire trans_start;
   //***************************************************************************
    // Code
    //***************************************************************************

inv  i0(.a(read_en) , .b(trans_start));

clk_div #( .sys_clk(100000000) , .clk_out(1) , .max(50000000)) u0 (.Clk_in(CLOCK) ,
.Clk_out(c));

uart_rx uart_rx_i0(
.clk_rx (CLOCK),
.rst_clk_rx (rst_clk_rx),
.rxd_i (rxd_i),
.rxd_clk_rx (write_clk),
.rx_data (data_in),
.rx_data_rdy (write_en),
.frm_err (frm_err)
);
```

```
selector sel(
.switch(switch),
.data(write_en),
.Clock(CLOCK),
.Out1(w2),
.Out2(w1)
);

fifo_generator_0 FIFO_1 (
  .wr_clk(write_clk),  // input wire wr_clk
  .rd_clk(c),  // input wire rd_clk
  .din(data_in),        // input wire [7 : 0] din
  .wr_en(w2),    // input wire wr_en
  .rd_en(CE),    // input wire rd_en
  .dout(data_out1),      // output wire [7 : 0] dout
  .full(full_1),      // output wire full
  .empty(empty_1)    // output wire empty
);

fifo_generator_0 FIFO_2 (
  .wr_clk(write_clk),  // input wire wr_clk
  .rd_clk(c),  // input wire rd_clk
  .din(data_in),        // input wire [7 : 0] din
  .wr_en(w1),    // input wire wr_en
  .rd_en(CE),    // input wire rd_en
  .dout(data_out2),      // output wire [7 : 0] dout
  .full(full_2),      // output wire full
  .empty(empty_2)    // output wire empty
);

c_addsub_0 adder (
  .A(data_out1),      // input wire [7 : 0] A
  .B(data_out2),      // input wire [7 : 0] B
  .CLK(CLOCK),  // input wire CLK
  .CE(CE),    // input wire CE
  .S(sum)      // output wire [7 : 0] S
);

fifo_generator_0 FIFO_3 (
  .wr_clk(c),  // input wire wr_clk
  .rd_clk(c),  // input wire rd_clk
  .din(sum),        // input wire [7 : 0] din
  .wr_en(CE),    // input wire wr_en
  .rd_en(read_en),    // input wire rd_en
  .dout(data_out3),      // output wire [7 : 0] dout
  .full(full_3),      // output wire full
  .empty(empty_3)    // output wire empty
);



uart_tx uart_tx_i0(
.clk_tx(CLOCK),
.rst_clk_tx(rst_clk_tx),
//.baud_x16_en(baud_x16_en),
.char_fifo_empty(trans_start),
.char_fifo_dout(data_out3),
.char_fifo_rd_en(char_fifo_rd_en),
.txd_tx(txd_tx)
);

endmodule
```

We have taken a simple exam of 8-bit binary addition mathematical operation, even more complex mathematical operations can be achieved, and we just have to design a module in HDL for that specific operation or a set of operations.

Shown next is the elaborated RTL level design of the system after executing the main wrapper (main code)

Figure 13. RTL level design of system



The user inputs the data through the keyboard also enabling the hardware through switches and making it ready for incoming data and setting up the terminal software. It flows through the UART Block 1 as it receives parallel data and changes it to serial fashion. The input can now go towards any of the 2 FIFOs storing input datas, the path would be decided by the state of the selection module. The process continues until both the FIFOs are full.

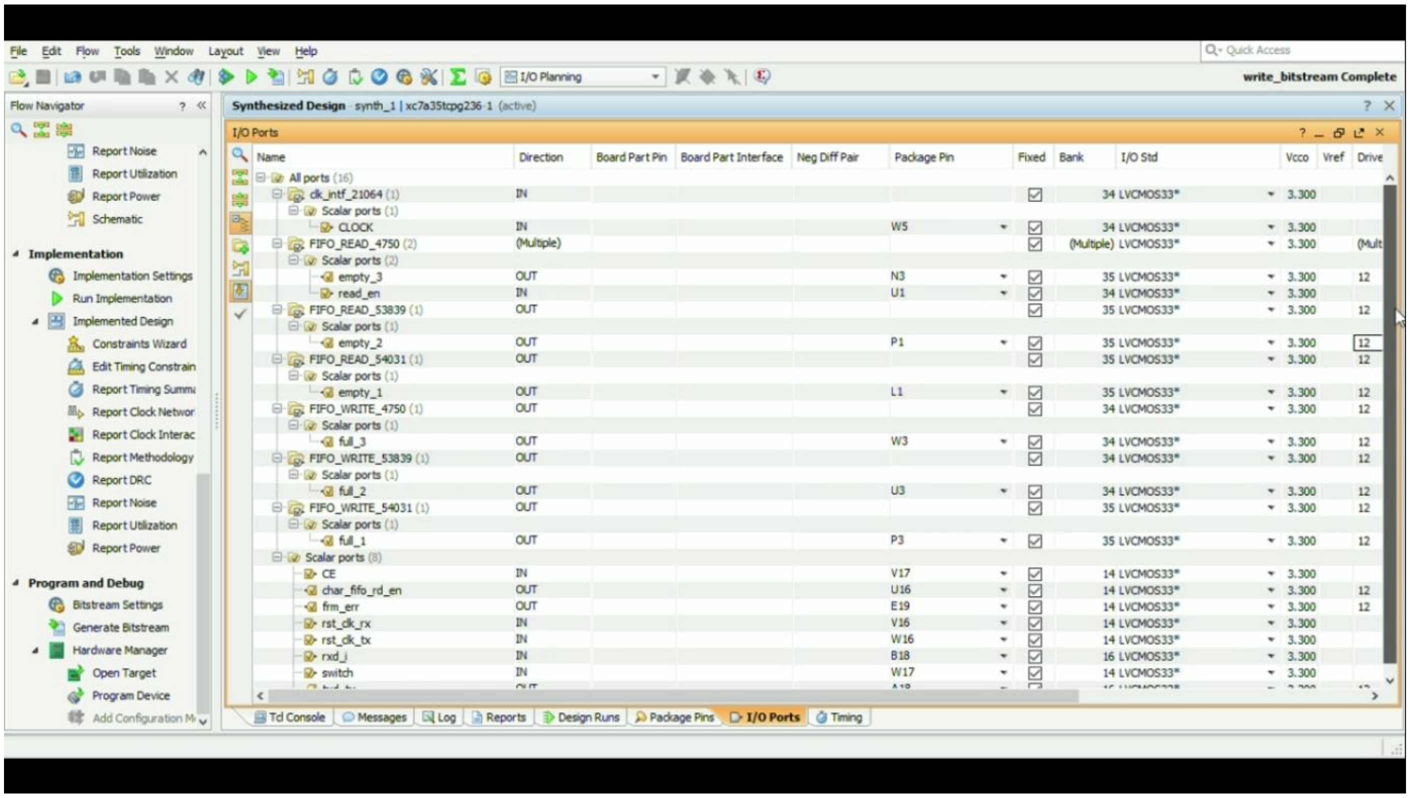
We have a common enable signals for the first 2 FIFOs and the adder enable. All of them go high together.

Now that we have all the data stored, we can give read enable signals to remove data one at a time from both the memory locations and perform mathematical operations on the corresponding pairs of data/numbers.

First the operation between a pair of numbers takes place and the result goes on to the result storing FIFO. Only after that the next operation on the next pair of numbers takes place. This process goes on until both the input FIFOs get empty, and the result storing FIFO gets full.

This final memory location would be containing all the resultant datas, which need to be sent back to the user. The read enable ensures that the data moves out one by one, through the UART Block 2 for transmission, the serial data now is converted back to parallel fashion and popped one character at a time for the result to be shown on the terminal software.

After synthesis of the main Verilog code, we have to designate the I/O ports with appropriate switches and LEDs , a screen capture of the same is also shown below, which is obtained by using the I/O Planning option in the software.



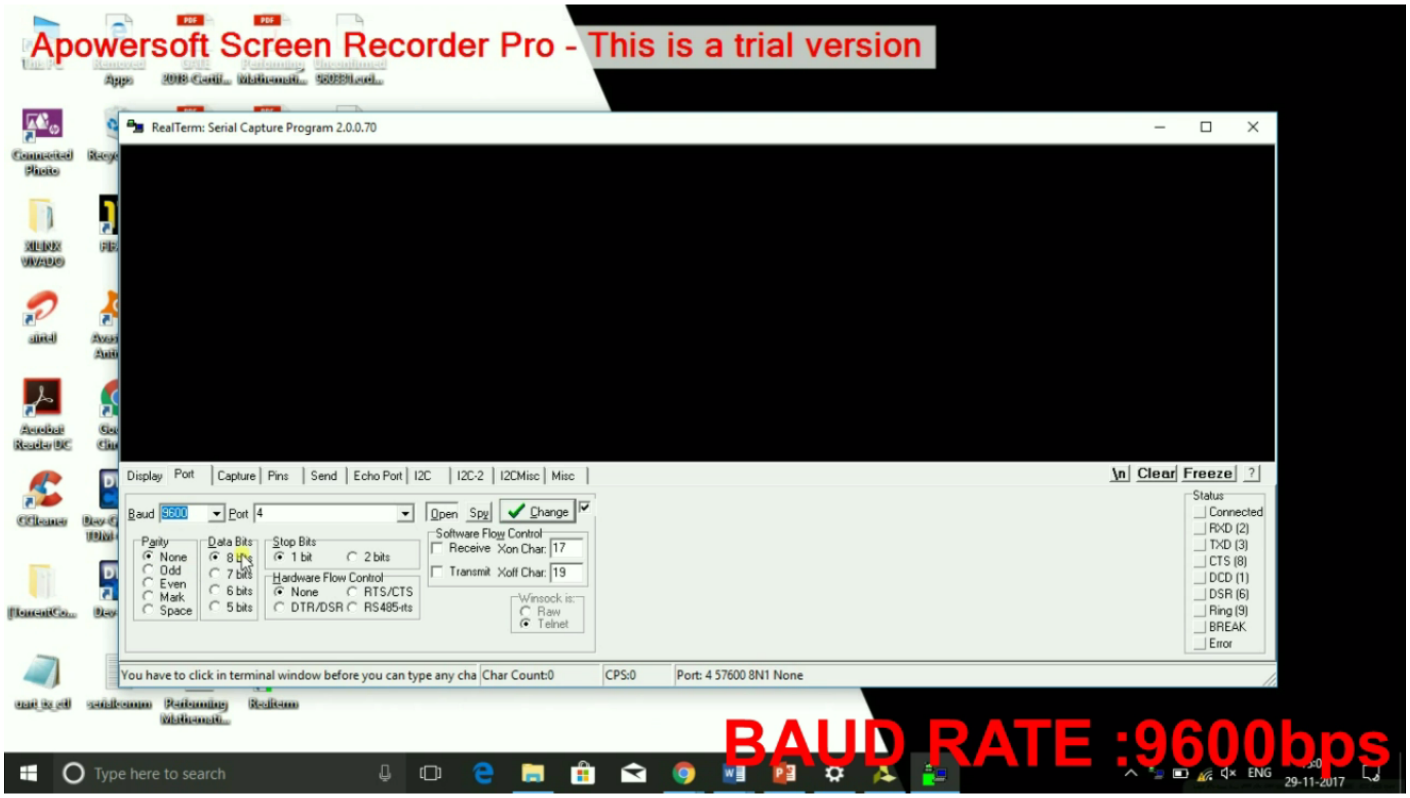
Synthesized Design - synth_1 | xc7a35tcpg236-1 (active)

I/O Ports

Name	Direction	Board Part Pin	Board Part Interface	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std	Vcco	Vref	Drive
All ports (16)											
clk_intf_21064 (1)	IN					✓	34	LVCMOS33*	3.300		
Scalar ports (1)											
CLOCK	IN				W5	✓	34	LVCMOS33*	3.300		
FIFO_READ_4750 (2)	(Multiple)					✓	(Multiple)	LVCMOS33*	3.300		(Mult
Scalar ports (2)											
empty_3	OUT				N3	✓	35	LVCMOS33*	3.300		12
read_en	IN				U1	✓	34	LVCMOS33*	3.300		
FIFO_READ_53839 (1)	OUT					✓	35	LVCMOS33*	3.300		12
Scalar ports (1)											
empty_2	OUT				P1	✓	35	LVCMOS33*	3.300		12
FIFO_READ_54031 (1)	OUT					✓	35	LVCMOS33*	3.300		12
Scalar ports (1)											
empty_1	OUT				L1	✓	35	LVCMOS33*	3.300		12
FIFO_WRITE_4750 (1)	OUT					✓	34	LVCMOS33*	3.300		12
Scalar ports (1)											
full_3	OUT				W3	✓	34	LVCMOS33*	3.300		12
FIFO_WRITE_53839 (1)	OUT					✓	34	LVCMOS33*	3.300		12
Scalar ports (1)											
full_2	OUT				U3	✓	34	LVCMOS33*	3.300		12
FIFO_WRITE_54031 (1)	OUT					✓	35	LVCMOS33*	3.300		12
Scalar ports (1)											
full_1	OUT				P3	✓	35	LVCMOS33*	3.300		12
Scalar ports (8)											
CE	IN				V17	✓	14	LVCMOS33*	3.300		
char_fifo_rd_en	OUT				U16	✓	14	LVCMOS33*	3.300		12
frm_err	OUT				E19	✓	14	LVCMOS33*	3.300		12
rst_clk_rx	IN				V16	✓	14	LVCMOS33*	3.300		
rst_clk_tx	IN				W16	✓	14	LVCMOS33*	3.300		
rxd_i	IN				B18	✓	16	LVCMOS33*	3.300		
switch	IN				W17	✓	14	LVCMOS33*	3.300		

Figure 14. I/O Planning

The Baud Rate is set at 9600 bps, both UARTs must operate at the same Baud Rate for the communication to be successful.



RealTerm: Serial Capture Program 2.0.0.70

Baud 9600 Port 4

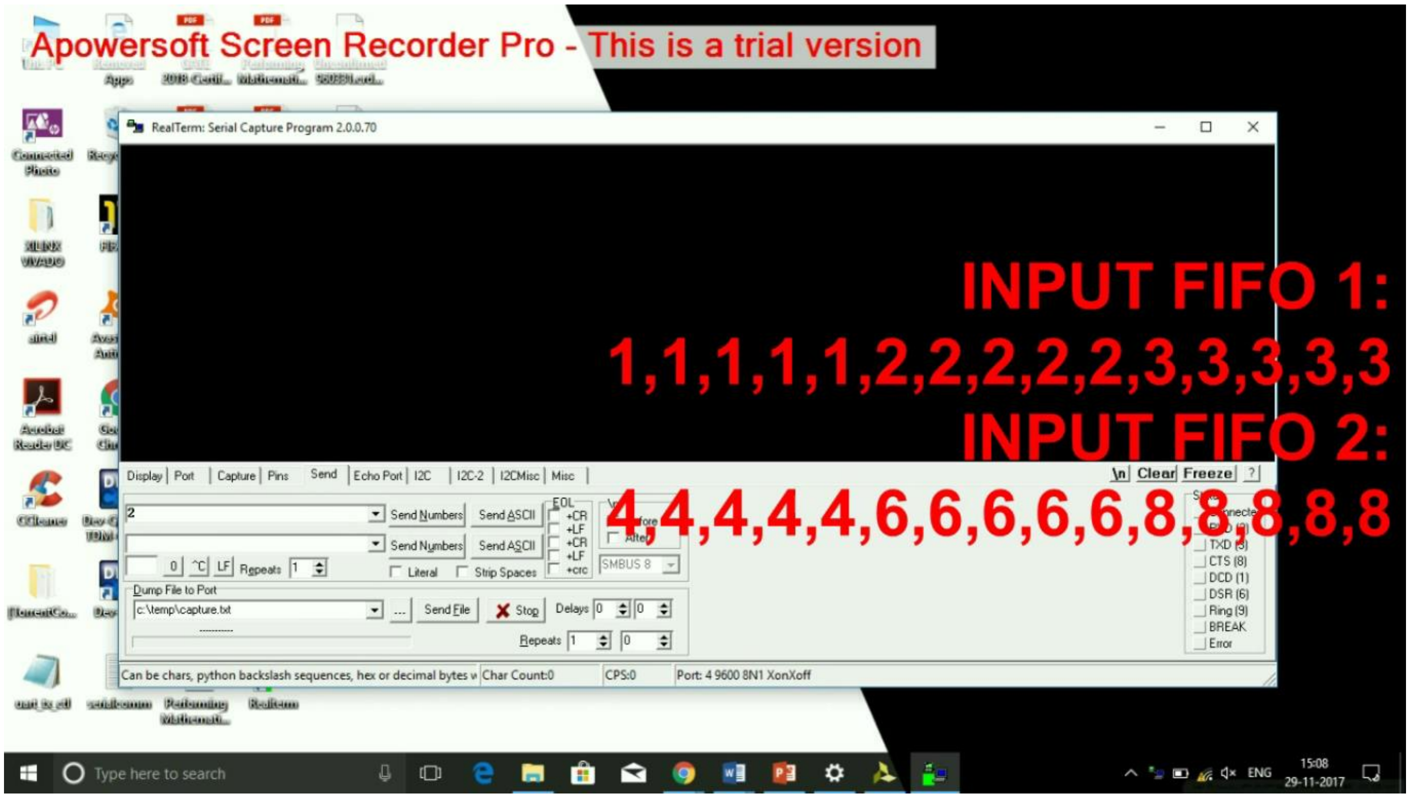
Parity: None · Data Bits: 8 bits · Stop Bits: 1 bit · Hardware Flow Control: None

You have to click in terminal window before you can type any cha Char Count:0 CPS:0 Port: 4 57600 8N1 None

BAUD RATE :9600bps

User Input in FIFO 1 (write depth-16, write width – 8bits) - 1,1,1,1,1,2,2,2,2,2,3,3,3,3,3

User Input in FIFO 2 (write depth-16, write width – 8bits) – 4,4,4,4,4,6,6,6,6,6,8,8,8,8,8



RealTerm: Serial Capture Program 2.0.0.70

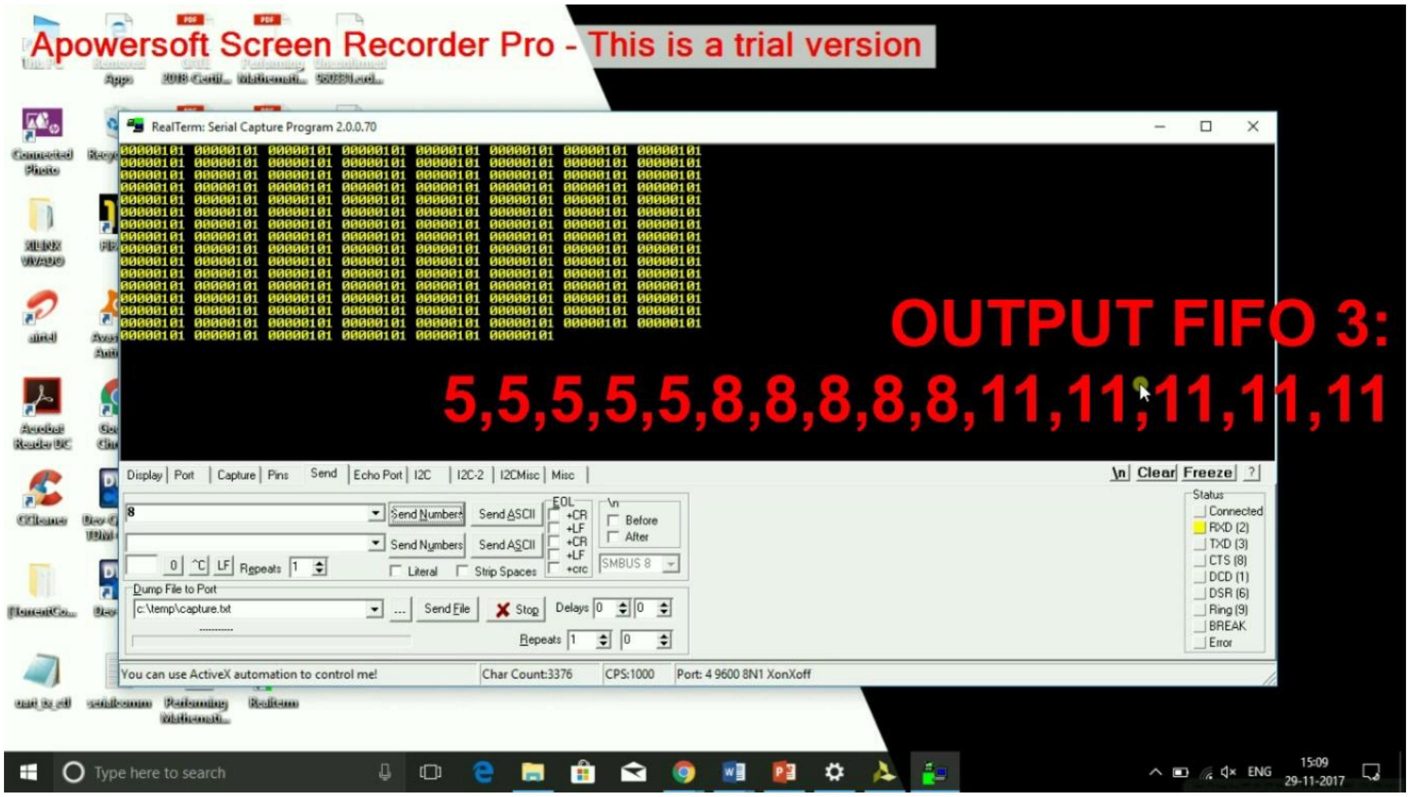
INPUT FIFO 1:
1,1,1,1,1,2,2,2,2,2,3,3,3,3,3
INPUT FIFO 2:
4,4,4,4,4,6,6,6,6,6,8,8,8,8,8

2

Dump File to Port c:\temp\capture.txt

Can be chars, python backslash sequences, hex or decimal bytes v Char Count:0 CPS:0 Port: 4 9600 8N1 XonXoff

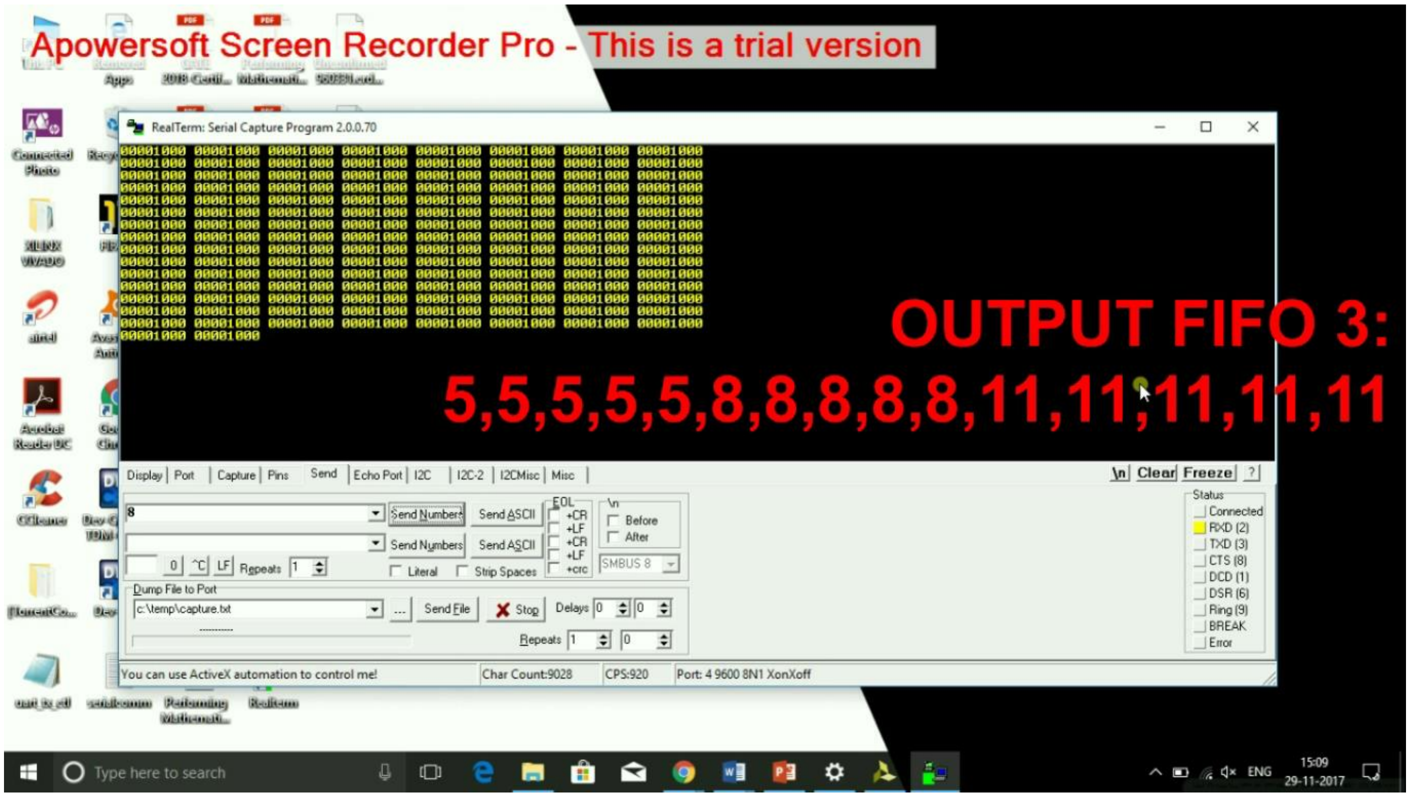
Output to User from FIFO3 (write depth -16 , write width- bits) – 5,5,5,5,5,8,8,8,8,8,11,11,11,11,11



```
00000101 00000101 00000101 00000101 00000101 00000101 00000101 00000101
00000101 00000101 00000101 00000101 00000101 00000101 00000101 00000101
00000101 00000101 00000101 00000101 00000101 00000101 00000101 00000101
00000101 00000101 00000101 00000101 00000101 00000101 00000101 00000101
00000101 00000101 00000101 00000101 00000101 00000101 00000101 00000101
00000101 00000101 00000101 00000101 00000101 00000101 00000101 00000101
00000101 00000101 00000101 00000101 00000101 00000101 00000101 00000101
00000101 00000101 00000101 00000101 00000101 00000101 00000101 00000101
00000101 00000101 00000101 00000101 00000101 00000101 00000101 00000101
00000101 00000101 00000101 00000101 00000101 00000101 00000101 00000101
00000101 00000101 00000101 00000101 00000101 00000101 00000101 00000101
00000101 00000101 00000101 00000101 00000101 00000101 00000101 00000101
00000101 00000101 00000101 00000101 00000101 00000101 00000101 00000101
00000101 00000101 00000101 00000101 00000101 00000101 00000101 00000101
00000101 00000101 00000101 00000101 00000101 00000101 00000101 00000101
00000101 00000101 00000101 00000101 00000101 00000101
```

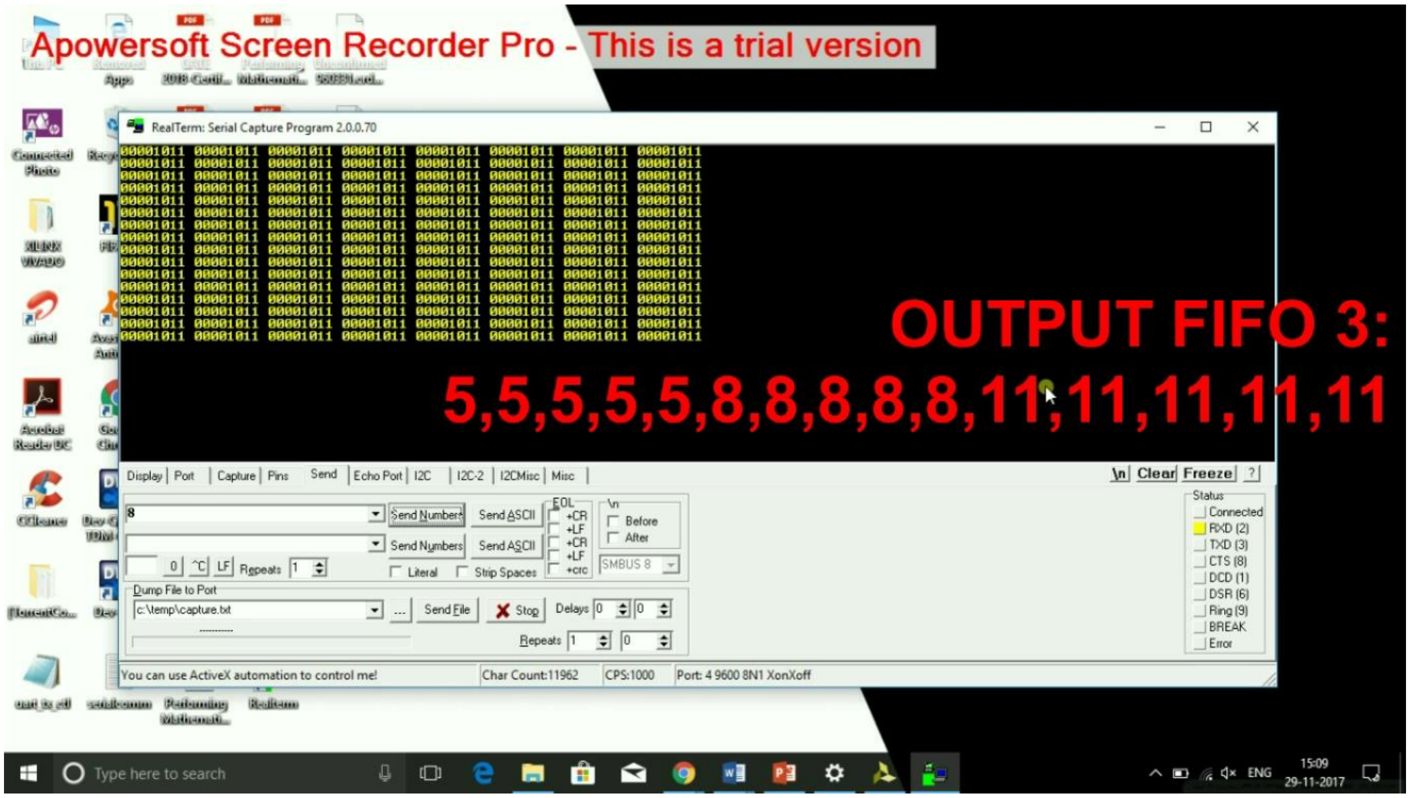
Send: 8 — Repeats 1
EOL: +CR +LF +CR +LF +crc — \n: Before, After — SMBUS 8
Dump File to Port: c:\temp\capture.txt — Delays 0 0 — Repeats 1 0

You can use ActiveX automation to control me! | Char Count:3376 | CPS:1000 | Port: 4 9600 8N1 XonXoff


Apowersoft Screen Recorder Pro - This is a trial version
RealTerm: Serial Capture Program 2.0.0.70
OUTPUT FIFO 3:
5,5,5,5,5,8,8,8,8,8,11,11,11,11,11
Display | Port | Capture | Pins | Send | Echo Port | I2C | I2C-2 | I2CMisc | Misc
\n Clear Freeze ?
Send Numbers | Send ASCII
Dump File to Port
Send File | Stop
Status: Connected, RXD (2), TXD (3), CTS (8), DCD (1), DSR (6), Ring (9), BREAK, Error

```
00001000 00001000 00001000 00001000 00001000 00001000 00001000 00001000
00001000 00001000 00001000 00001000 00001000 00001000 00001000 00001000
00001000 00001000 00001000 00001000 00001000 00001000 00001000 00001000
00001000 00001000 00001000 00001000 00001000 00001000 00001000 00001000
00001000 00001000 00001000 00001000 00001000 00001000 00001000 00001000
00001000 00001000 00001000 00001000 00001000 00001000 00001000 00001000
00001000 00001000 00001000 00001000 00001000 00001000 00001000 00001000
00001000 00001000 00001000 00001000 00001000 00001000 00001000 00001000
00001000 00001000 00001000 00001000 00001000 00001000 00001000 00001000
00001000 00001000 00001000 00001000 00001000 00001000 00001000 00001000
00001000 00001000 00001000 00001000 00001000 00001000 00001000 00001000
00001000 00001000 00001000 00001000 00001000 00001000 00001000 00001000
00001000 00001000 00001000 00001000 00001000 00001000 00001000 00001000
00001000 00001000 00001000 00001000 00001000 00001000 00001000 00001000
00001000 00001000 00001000 00001000 00001000 00001000 00001000 00001000
00001000 00001000
```

```
You can use ActiveX automation to control me!   Char Count:9028   CPS:920   Port: 4 9600 8N1 XonXoff
```


Apowersoft Screen Recorder Pro - This is a trial version
RealTerm: Serial Capture Program 2.0.0.70
OUTPUT FIFO 3:
5,5,5,5,5,8,8,8,8,8,11,11,11,11,11
Display | Port | Capture | Pins | Send | Echo Port | I2C | I2C-2 | I2CMisc | Misc
\n Clear Freeze ?
Send Numbers | Send ASCII
Dump File to Port
Send File | Stop
Status: Connected, RXD (2), TXD (3), CTS (8), DCD (1), DSR (6), Ring (9), BREAK, Error

```
00001011 00001011 00001011 00001011 00001011 00001011 00001011 00001011
00001011 00001011 00001011 00001011 00001011 00001011 00001011 00001011
00001011 00001011 00001011 00001011 00001011 00001011 00001011 00001011
00001011 00001011 00001011 00001011 00001011 00001011 00001011 00001011
00001011 00001011 00001011 00001011 00001011 00001011 00001011 00001011
00001011 00001011 00001011 00001011 00001011 00001011 00001011 00001011
00001011 00001011 00001011 00001011 00001011 00001011 00001011 00001011
00001011 00001011 00001011 00001011 00001011 00001011 00001011 00001011
00001011 00001011 00001011 00001011 00001011 00001011 00001011 00001011
00001011 00001011 00001011 00001011 00001011 00001011 00001011 00001011
00001011 00001011 00001011 00001011 00001011 00001011 00001011 00001011
00001011 00001011 00001011 00001011 00001011 00001011 00001011 00001011
00001011 00001011 00001011 00001011 00001011 00001011 00001011 00001011
00001011 00001011 00001011 00001011 00001011 00001011 00001011 00001011
00001011 00001011 00001011 00001011 00001011 00001011 00001011 00001011
00001011 00001011 00001011 00001011 00001011 00001011 00001011 00001011
```

```
You can use ActiveX automation to control me!   Char Count:11962   CPS:1000   Port: 4 9600 8N1 XonXoff
```

As observed the mathematical operations that were performed came out be correct, but there is on big problem that needs to be tackled. The time taken for this process to complete was very much, which is not desirable for the industry/computational fields, we need to devise a way to reduce this time and still obtain proper results. The architecture is achieved but the time constraint needs to be taken care of.

CHAPTER 3: PARALLELISING AND LATENCY REDUCTION

PARALLELISING OR PARALLEL COMPUTING:

Parallelizing or parallel computing is a type of computing architecture in which several processors execute or process an application or computation simultaneously. Parallel computing helps in performing large computations by dividing the workload between more than one processor, all of which work through the computation at the same time. Most supercomputers employ parallel computing principles to operate. Parallel computing is also known as parallel processing.

Suppose you have a lot of work to be done, and want to get it done much faster, so you hire 100 workers. If the work is 100 separate jobs that don't depend on each other, and they all take the same amount of time and can be easily parceled out to the workers, then you'll get it done about 100 times faster. A different option would be to parallelize each job, as discussed below, and then run these parallelization's one after another. However, as will be shown, this is probably a less efficient way of doing the work. Occasionally this isn't true because on computers, doing all of the job on one processor may require storing many results on disk, while the parallel job may spread the intermediate results in the RAM of the different processors. RAM is much faster than disks. However, if the program isn't spending a lot of time using the disk then embarrassingly parallel is the smart way to go? Assume this is what you should do unless you analyze the situation and determine that it isn't. Embarrassingly parallel is simple, and if you can get the workers do it for free then it is the cheapest solution as well.

Design Latency

Latency is the number of clock cycles it takes to complete an instruction or set of instructions to generate an application result value

Design Throughput

Throughput is another metric used to determine overall performance of an implementation. It is the number of clock cycles it takes for the processing logic to accept the next input data sample. With this value, it is important to remember that the clock frequency of the circuit changes the meaning of the throughput number.

CHAPTER 5: RTL IMPLEMENTATION ON FPGA USING VIVADO HLS

BASIC INTRODUCTION

Vivado HLS is a software that can be used to perform mathematical or any operation on FPGA using HLL like C / C++ . Vivado Hls gives you a platform where you can easily make code to perform any operation using c /c++ languages and you can use latency reduction techniques like on vivado hls to low latency highly efficient output.

The flow chart below will give you a brief introduction on how we will be working on Vivado hls to perform any RTL implementation

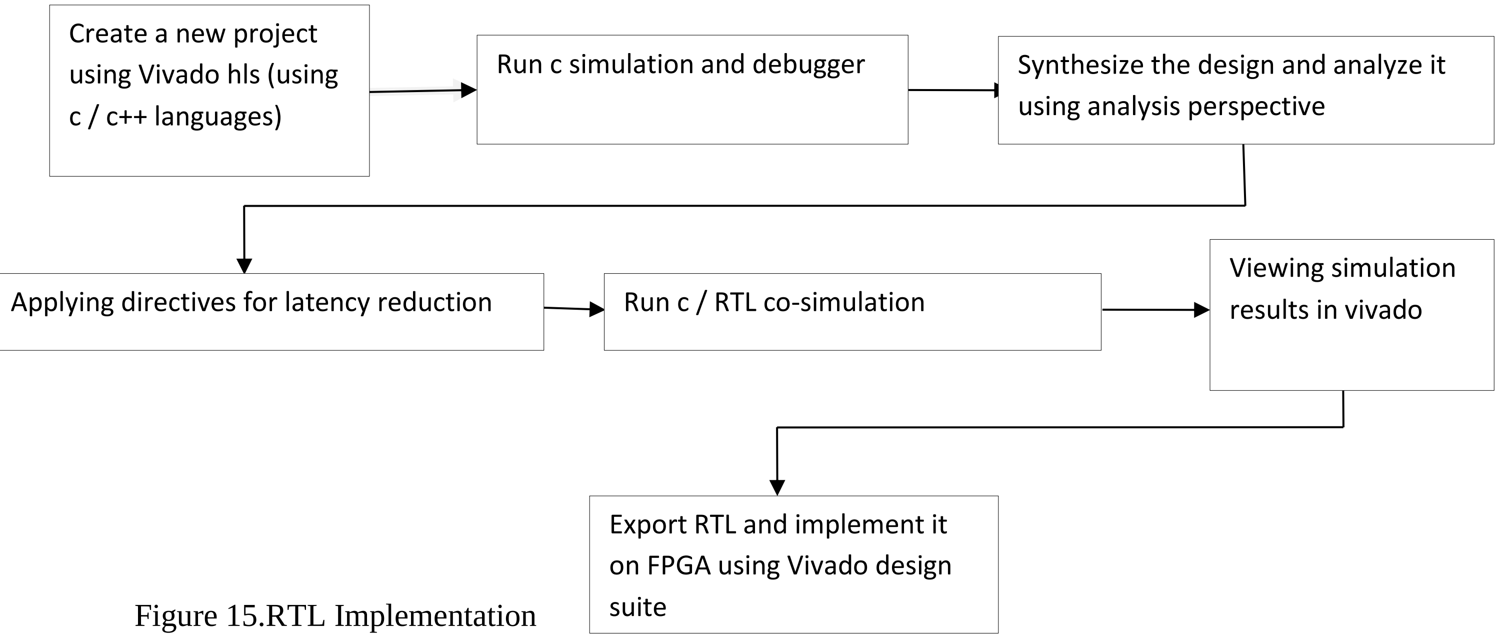

Create a new project using Vivado hls (using c / c++ languages)
Run c simulation and debugger
Synthesize the design and analyze it using analysis perspective
Applying directives for latency reduction
Run c / RTL co-simulation
Viewing simulation results in vivado
Export RTL and implement it on FPGA using Vivado design suite

Figure 15.RTL Implementation

In our project we would be multiplying two matrices A [ ] and B [ ] and we would be storing the result in a matrix C [ ]. We would be using c language to make the code for this matrix multiplication and after that we are going to export the RTL design and will try perform it on FPGA board.

(For simplicity we are assuming all matrices as 3*3 matrix)

C CODE FOR MATRIX MULTIPLICATION THAT WE ARE GOING TO USE

TEST BENCH FILE:

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

#define M 3
#define N 3

void mul_mat(char A[M][N],char B[M][N],char C[M][N]);


int main()
{
char a[M][N],b[M][N];
char c[M][N];


printf("\n the first matrix:");
for (int x=0;x<M;x++)
            {
                  printf("\n");
                  main_label0:for(int y=0;y<N;y++)
                  {
                        a[x][y]=rand()%5;
                        printf("%d",a[x][y]);
                  printf(     " \t");
             }
                 }


printf("\n the second matrix:\n");
for ( int w=0;w<M;w++)
            {
                  printf("\n");
                  main_label1:for( int z=0;z<N;z++)
                  {
                        b[w][z]=rand()%5;
                        printf("%d",b[w][z]);
                  printf(     " \t");
                  }
            }


mul_mat(a,b,c);

return 0;

}
```

LATENCY REDUCTION TECHNIQUES :

To optimize the result produced in the above code we are going to use certain directives present in directive panel of vivado Hls and in the end we would be comparing the results produced by each directive and choosing one with the lowest latency to implement on the FPGA board.

Directive panel :

Figure below contains all the directives which we can use for optimizations in vivado Hls.

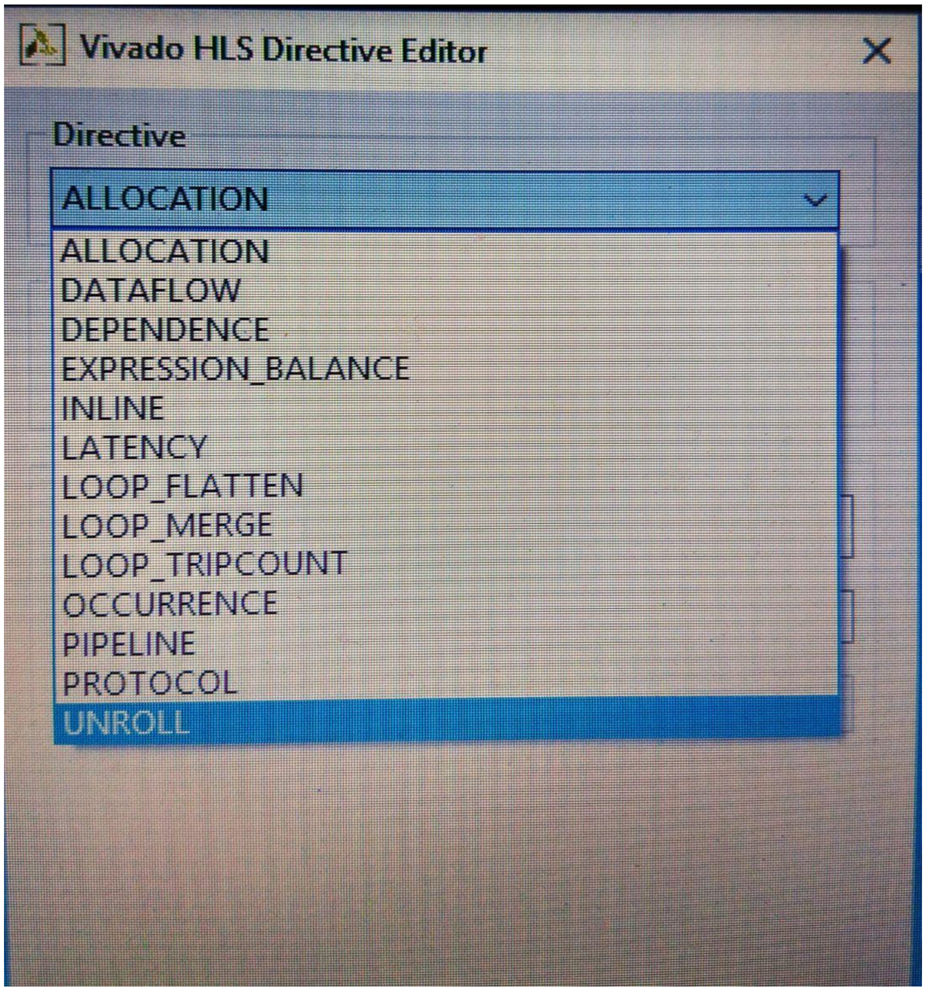


Figure 16. Directives

Directives which we are going to use are pipelining and loop unrolling to reduce the latency of our code.

To improve the performance of a CPU we have two options:
1) Improve the hardware by introducing faster circuits.
2) Arrange the hardware such that more than one operation can be performed at the same time.

Since, there is a limit on the speed of hardware and the cost of faster circuits is quite high, we have to adopt the 2nd option.

PIPELINING:

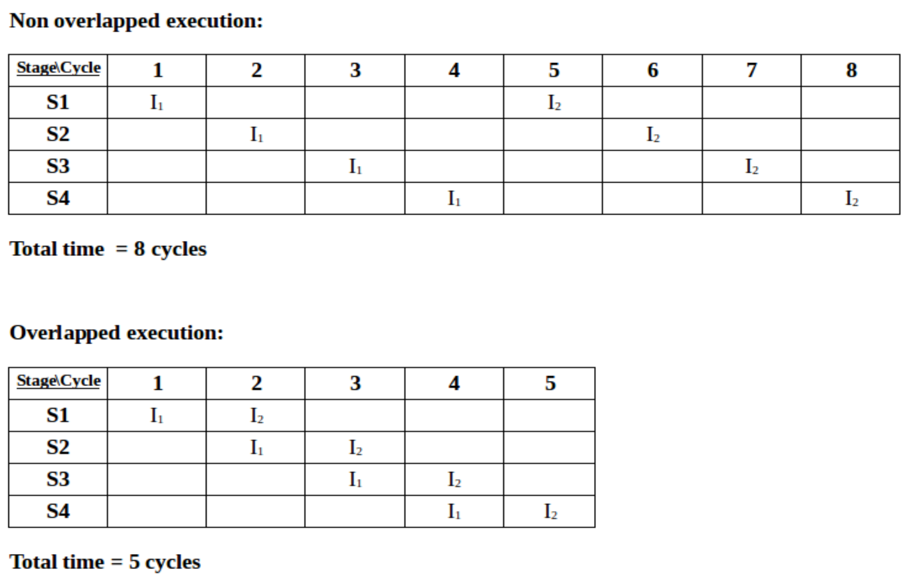
Pipelining is a process of arrangement of hardware elements of the CPU such that its overall performance is increased. Simultaneous execution of more than one instruction takes place in a pipelined processor.

Design of a basic pipeline

- In a pipelined processor, a pipeline has two ends, the input end and the output end. Between these ends, there are multiple stages/segments such that output of one stage is connected to input of next stage and each stage performs a specific operation.
- Interface registers are used to hold the intermediate output between two stages. These interface registers are also called latch or buffer.
- All the stages in the pipeline along with the interface registers are controlled by a common clock.

Execution in a pipelined processor:

Execution sequence of instructions in a pipelined processor can be visualized using a space-time diagram. For example, consider a processor having 4 stages and let there be 2 instructions to be executed. We can visualize the execution sequence through the following space-time diagrams:



Non overlapped execution:

Stage\Cycle	1	2	3	4	5	6	7	8
S1	I_1				I_2			
S2		I_1				I_2		
S3			I_1				I_2	
S4				I_1				I_2

Total time = 8 cycles

Overlapped execution:

Stage\Cycle	1	2	3	4	5
S1	I_1	I_2			
S2		I_1	I_2		
S3			I_1	I_2	
S4				I_1	I_2

Total time = 5 cycles

Table 1. Instruction executions in pipelined processor [10]

Pipeline Stages

RISC processor has 5 stage instruction pipeline to execute all the instructions in the RISC instruction set. Following are the 5 stages of RISC pipeline with their respective operations:

- **Stage 1** (instruction fetch)

In this stage the CPU reads instructions from the address in the memory whose value is present in the program counter

- **Stage 2** (instruction decode)

In this stage, instruction is decoded and the register file is accessed to get the values from the registers used in the instruction

- **Stage 3** (instruction execute)

In this stage, ALU operations are performed

- **Stage 4** (memory access)

In this stage, memory operands are read and written from/to the memory that is present in the instruction

- **Stage 5**(write back)

In this stage, computed/fetched value is written back to the register present in the instruction.

Dependencies in a pipelined processor

There are mainly three types of dependencies possible in a pipelined processor. These are:

1) Structural Dependency

2) Control Dependency

3) Data Dependency

These dependencies may introduce stalls in the pipeline.

Stall: A stall is a cycle in pipeline without new input.

Structural dependency

This dependency arises due to the resource conflict in the pipeline. A resource conflict is a situation when more than one instruction tries to access the same resource in the same cycle. A resource can be a register, memory, or ALU.

Example

Instruction\ Cycle	**1**	**2**	**3**	**4**	**5**
I_1	IF(Mem)	ID	EX	Mem	
I_2		IF(Mem)	ID	EX	
I_3			IF(Mem)	ID	EX
I_4				IF(Mem)	ID

Resource conflict

Control dependency (Branch Hazards)

This type of dependency occurs during the transfer of control instructions such as BRANCH, CALL, JMP, etc. On many instruction architectures, the processor will not know the target address of these instructions when it needs to insert the new instruction into the pipeline. Due to this, unwanted instructions are fed to the pipeline.

Consider the following sequence of instructions in the program:

100:I_1

101:I_2 (JMP250)

102:I_3

..

..

250: BI_1

Expected output: I_1 -> I_2 -> BI_1

NOTE: Generally, the target address of the JMP instruction is known after ID stage only.

Instruction \ Cycle	**1**	**2**	**3**	**4**	**5**	**6**
I_1	IF	ID	EX	Mem	WB	
I_2		IF	ID (PC: 250)	EX	Mem	WB
I_3			IF	ID	EX	Mem
BI_1				IF	ID	EX

Unwanted instruction

Table 2. Control Dependency [10]

Output Sequence: I_1 -> I_2 -> I_3 -> BI_1

So, the output sequence is not equal to the expected output that means the pipeline is not implemented correctly.

DATA HAZARDS

Data hazards occur when instructions that exhibit data dependence, modify data in different stages of a pipeline. Hazard cause delays in the pipeline. There are mainly three types of data hazards:

1) RAW (Read after Write) [Flow dependency]

2) WAR (Write after Read) [Anti-Data dependency]

3) WAW (Write after Write) [Output dependency]

Example: Let there be two instructions I1 and I2 such that:
I1:ADDR1,R2,R3
I2:SUB R4, R1, R2

When the above instructions are executed in a pipelined processor, then data dependency condition will occur, which means that I_2 tries to read the data before I_1 writes it, therefore, I_2 incorrectly gets the old value from I_1.

Instruction\Cycle	1	2	3	4
I_1	IF	ID	EX	DM
I_2		IF	ID (old value)	EX

Table 3. stalls in pipeline [10]

To minimize data dependency stalls in the pipeline, **operand forwarding** is used.

Operand Forwarding : In operand forwarding, we use the interface registers present between the stages to hold intermediate output so that dependent instruction can access new value from the interface register directly.

Considering the same example:
I1 : ADD R1, R2, R3
I2 : SUB R4, R1, R2

Instruction\Cycle	1	2	3	4
I_1	IF	ID	EX	DM
I_2		IF	ID	EX

Operand Forwarding

Table 4. Operand Forwarding [10]

Keeping all the limitations in mind we applied pipelining in the third or say the outermost loop of our main code of matrix multiplication.

LOOP UNROLLING:

Loop unrolling is a loop transformation technique that helps to optimize the execution time of a program. Loops with small number of iterations can be unrolled for higher performance. Loop unrolling aims to increase the program's speed by eliminating loop control instruction and loop test instructions.

Program 1:

```
// This program does not uses loop unrolling.
#include<stdio.h>

int main(void)
{
    for (int i=0; i<5; i++)
        printf("Hello\n"); //print hello 5 times

    return 0;
}
```

Program 2:

```
// This program uses loop unrolling.
#include<stdio.h>

int main(void)
{
    // unrolled the for loop in program 1
    printf("Hello\n");
    printf("Hello\n");
    printf("Hello\n");
    printf("Hello\n");
    printf("Hello\n");

    return 0;
}
```

Output:

```
Hello
Hello
Hello
Hello
Hello
```

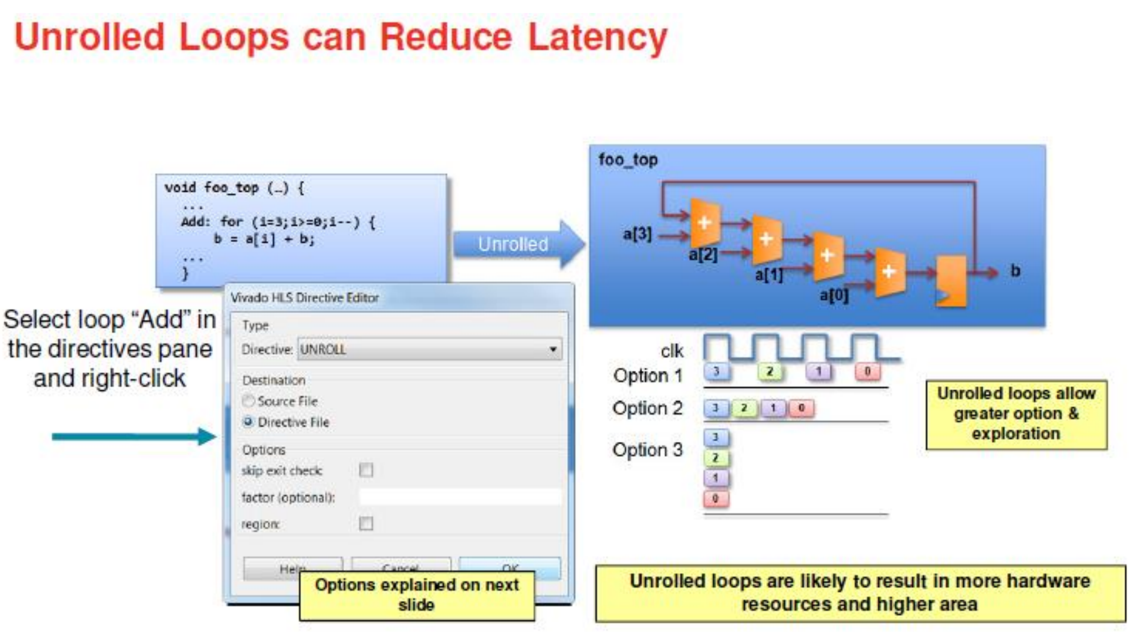
Illustration:

Program 2 is more efficient than program 1 because in program 1 there is a need to check the value of i and increment the value of i every time round the loop. So small loops like this or loops where there is fixed number of iterations are involved can be unrolled completely to reduce the loop overhead.

Advantages:

- Increases program efficiency.
- Reduces loop overhead.
- If statements in loop are not dependent on each other, they can be executed in parallel.

Figure 17. Unrolled Loops with Latency [15]



Unrolled Loops can Reduce Latency

```
void foo_top (…) {
  ...
  Add: for (i=3;i>=0;i--) {
      b = a[i] + b;
  ...
  }
```

Select loop “Add” in the directives pane and right-click

Vivado HLS Directive Editor

Type
Directive: UNROLL

Destination
Source File
Directive File [✓]

Options
skip exit check
factor (optional):
region:

Options explained on next slide

Unrolled loops allow greater option & exploration

Unrolled loops are likely to result in more hardware resources and higher area

Disadvantages:

- Increased program code size, which can be undesirable.
- Possible increased usage of register in a single iteration to store temporary variables which may reduce performance.
- Apart from very small and simple codes, unrolled loops that contain branches are even slower than recursions.

PARTIAL UNROLLING

Fully unrolling loops can create a lot of hardware

Loops can be partially unrolled

– Provides the type of exploration shown in the previous slide

Partial Unrolling

– A standard loop of N iterations can be unrolled to by a factor

– For example unroll by a factor 2, to have N/2 iterations

LOOP FLATTENING

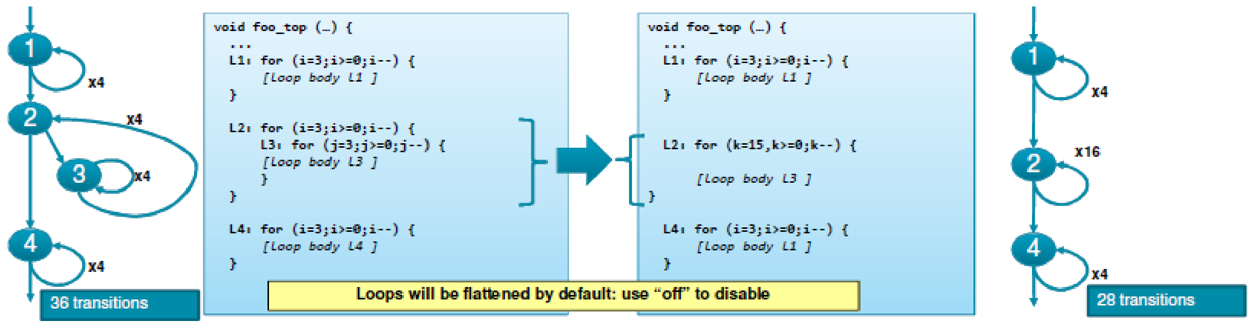
Vivado HLS can automatically flatten nested loops

– A faster approach than manually changing the code

Flattening should be specified on the inner most loop

– It will be flattened into the loop above

– The “off” option can prevent loops in the hierarchy from being flattened



```
void foo_top (…) {
  ...
  L1: for (i=3;i>=0;i--) {
      [Loop body L1 ]
  }

  L2: for (i=3;i>=0;i--) {
      L3: for (j=3;j>=0;j--) {
      [Loop body L3 ]
      }
  }

  L4: for (i=3;i>=0;i--) {
      [Loop body L4 ]
  }
```

```
void foo_top (…) {
  ...
  L1: for (i=3;i>=0;i--) {
      [Loop body L1 ]
  }

  L2: for (k=15,k>=0;k--) {

      [Loop body L3 ]
}

  L4: for (i=3;i>=0;i--) {
      [Loop body L1 ]
  }
```

Figure 18. Loop Flattening [15]

PERFECT AND SEMI PERFECT LOOPS

Only perfect and semi-perfect loops can be flattened

– The loop should be labeled or directives cannot be applied

– Perfect Loops

– Only the inner most loop has body (contents)

– There is no logic specified between the loop statements

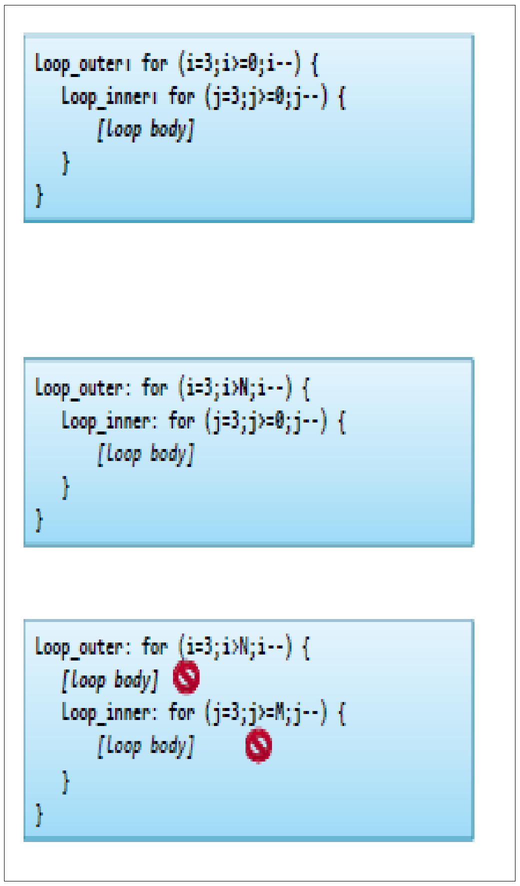
– The loop bounds are constant

– Semi-perfect Loops

– Only the inner most loop has body (contents)

– There is no logic specified between the loop statements

– The outer most loop bound can be variable



```
Loop_outer: for (i=3;i>=0;i--) {
   Loop_inner: for (j=3;j>=0;j--) {
       [Loop body]
   }
}
```

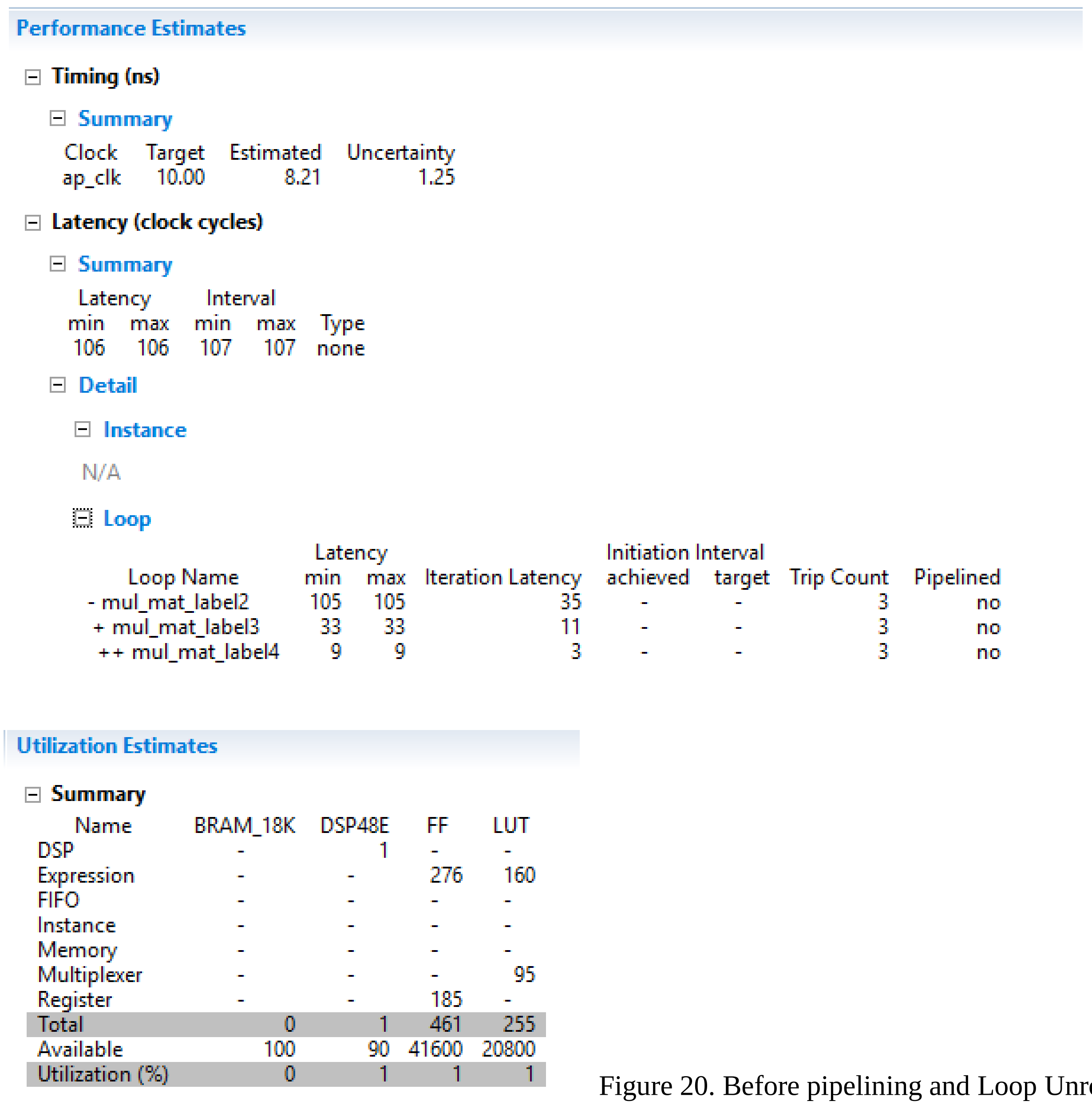
```
Loop_outer: for (i=3;i>N;i--) {
   Loop_inner: for (j=3;j>=0;j--) {
       [Loop body]
   }
}
```

```
Loop_outer: for (i=3;i>N;i--) {
   [Loop body]
   Loop_inner: for (j=3;j>=M;j--) {
       [Loop body]
   }
}
```

Figure 19.
Perfect and Semi –perfect
loop[15]

RESULTS AND THEIR COMPARISON

Before pipelining and loop unrolling:



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.21	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
106	106	107	107	none

Detail

Instance

N/A

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- mul_mat_label2	105	105	35	-	-	3	no
+ mul_mat_label3	33	33	11	-	-	3	no
++ mul_mat_label4	9	9	3	-	-	3	no

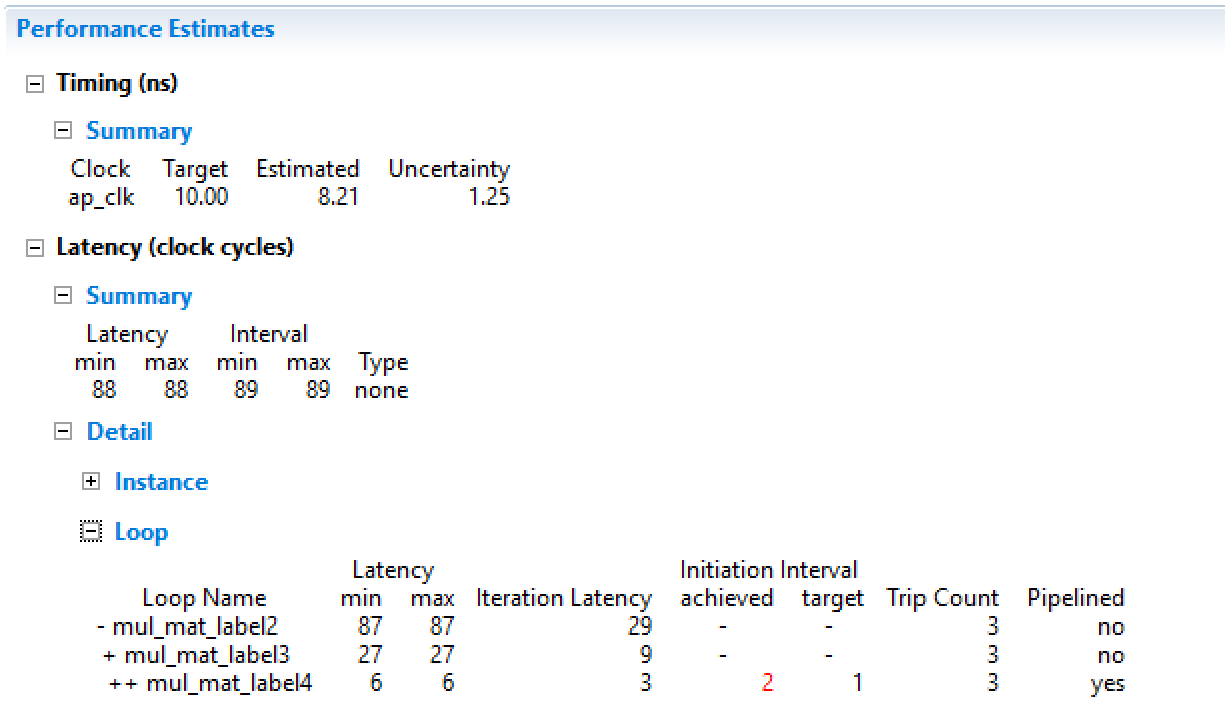
Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	1	-	-
Expression	-	-	276	160
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	95
Register	-	-	185	-
Total	0	1	461	255
Available	100	90	41600	20800
Utilization (%)	0	1	1	1

Figure 20. Before pipelining and Loop Unrolling

After pipelining:



Performance Estimates

⊟ Timing (ns)

⊟ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.21	1.25

⊟ Latency (clock cycles)

⊟ Summary

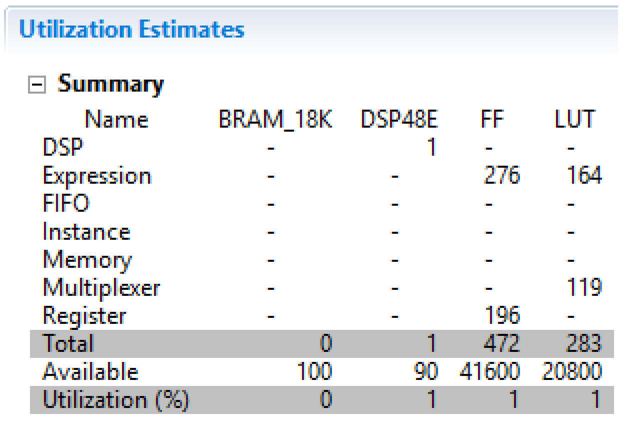
Latency		Interval		
min	max	min	max	Type
88	88	89	89	none

⊟ Detail

⊞ Instance

⊟ Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- mul_mat_label2	87	87	29	-	-	3	no
+ mul_mat_label3	27	27	9	-	-	3	no
++ mul_mat_label4	6	6	3	2	1	3	yes



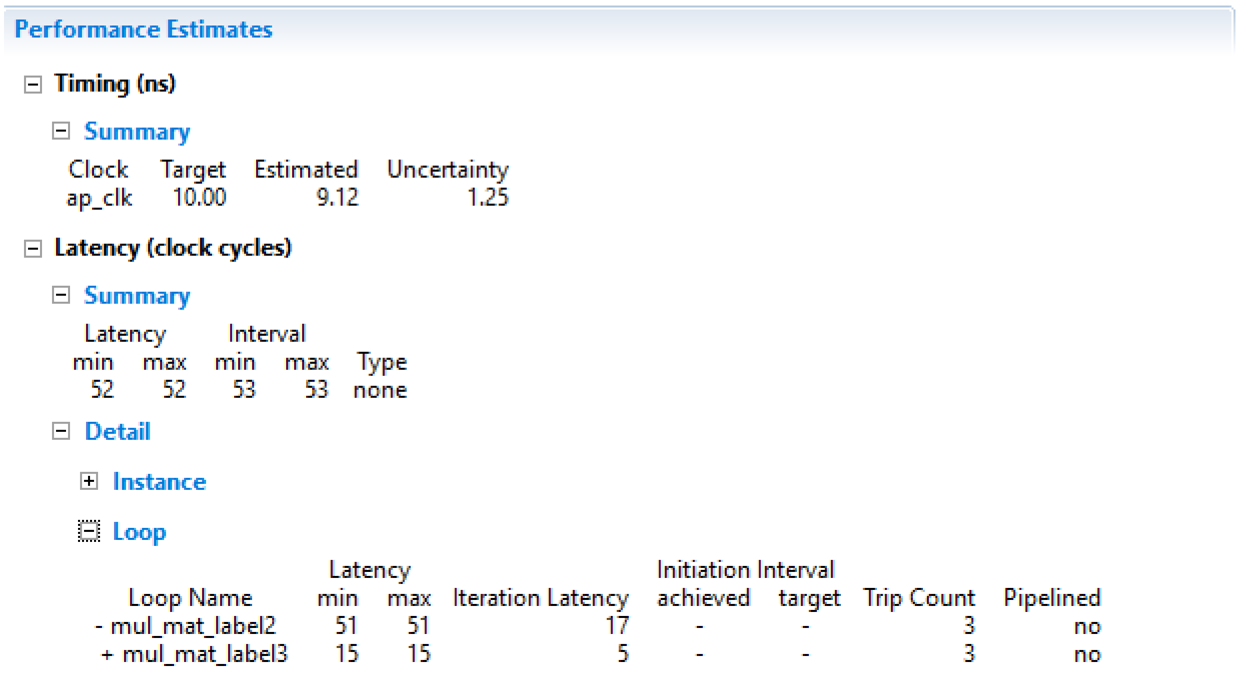
Utilization Estimates

⊟ Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	1	-	-
Expression	-	-	276	164
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	119
Register	-	-	196	-
Total	0	1	472	283
Available	100	90	41600	20800
Utilization (%)	0	1	1	1

Figure 21. After Pipelining

After loop unrolling:



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.12	1.25

Latency (clock cycles)

Summary

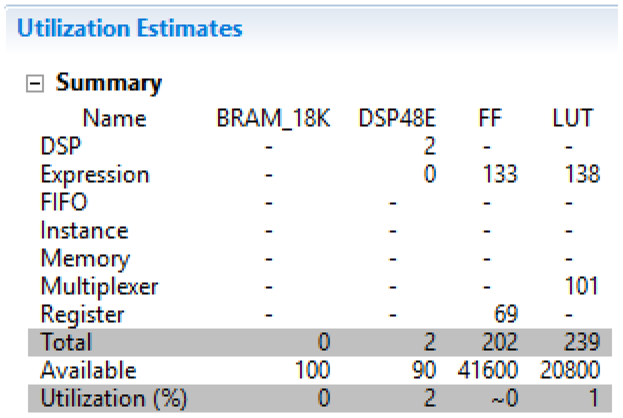
Latency		Interval		
min	max	min	max	Type
52	52	53	53	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- mul_mat_label2	51	51	17	-	-	3	no
+ mul_mat_label3	15	15	5	-	-	3	no



Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	2	-	-
Expression	-	0	133	138
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	101
Register	-	-	69	-
Total	0	2	202	239
Available	100	90	41600	20800
Utilization (%)	0	2	~0	1

Figure 22. After Loop Unrolling

After comparing all three results we have come to conclusion that the latency is minimum in case of loop unrolling also the utilization of hardware is also maximum in case of loop unrolling. Since the latency in the case of loop unrolling is minimum so we would be using that code for further operations.

FINAL RTL CODE (VHDL)

```
==============================================================
-- RTL generated by Vivado(TM) HLS - High-Level Synthesis from C, C++ and SystemC
-- Version: 2017.2
-- Copyright (C) 1986-2017 Xilinx, Inc. All Rights Reserved.
--
-- ==============================================================

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity mul_mat is
port (
    ap_clk : IN STD_LOGIC;
    ap_rst : IN STD_LOGIC;
    ap_start : IN STD_LOGIC;
    ap_done : OUT STD_LOGIC;
    ap_idle : OUT STD_LOGIC;
    ap_ready : OUT STD_LOGIC;
    A_address0 : OUT STD_LOGIC_VECTOR (3 downto 0);
    A_ce0 : OUT STD_LOGIC;
    A_q0 : IN STD_LOGIC_VECTOR (7 downto 0);
    B_address0 : OUT STD_LOGIC_VECTOR (3 downto 0);
    B_ce0 : OUT STD_LOGIC;
    B_q0 : IN STD_LOGIC_VECTOR (7 downto 0);
    C_address0 : OUT STD_LOGIC_VECTOR (3 downto 0);
    C_ce0 : OUT STD_LOGIC;
    C_we0 : OUT STD_LOGIC;
    C_d0 : OUT STD_LOGIC_VECTOR (7 downto 0) );
end;


architecture behav of mul_mat is
    attribute CORE_GENERATION_INFO : STRING;
    attribute CORE_GENERATION_INFO of behav : architecture is

"mul_mat,hls_ip_2017_2,{HLS_INPUT_TYPE=c,HLS_INPUT_FLOAT=0,HLS_INPUT_FIXED=0,HLS_INPUT_
PART=xc7a35tcpg236-
3,HLS_INPUT_CLOCK=10.000000,HLS_INPUT_ARCH=others,HLS_SYN_CLOCK=9.120000,HLS_SYN_LAT=52
,HLS_SYN_TPT=none,HLS_SYN_MEM=0,HLS_SYN_DSP=2,HLS_SYN_FF=202,HLS_SYN_LUT=239}";
    constant ap_const_logic_1 : STD_LOGIC := '1';
    constant ap_const_logic_0 : STD_LOGIC := '0';
    constant ap_ST_fsm_state1 : STD_LOGIC_VECTOR (6 downto 0) := "0000001";
    constant ap_ST_fsm_state2 : STD_LOGIC_VECTOR (6 downto 0) := "0000010";
    constant ap_ST_fsm_state3 : STD_LOGIC_VECTOR (6 downto 0) := "0000100";
    constant ap_ST_fsm_state4 : STD_LOGIC_VECTOR (6 downto 0) := "0001000";
    constant ap_ST_fsm_state5 : STD_LOGIC_VECTOR (6 downto 0) := "0010000";
    constant ap_ST_fsm_state6 : STD_LOGIC_VECTOR (6 downto 0) := "0100000";
    constant ap_ST_fsm_state7 : STD_LOGIC_VECTOR (6 downto 0) := "1000000";
    constant ap_const_lv32_0 : STD_LOGIC_VECTOR (31 downto 0) :=
"00000000000000000000000000000000";
    constant ap_const_lv32_1 : STD_LOGIC_VECTOR (31 downto 0) :=
"00000000000000000000000000000001";
    constant ap_const_lv1_0 : STD_LOGIC_VECTOR (0 downto 0) := "0";
    constant ap_const_lv32_2 : STD_LOGIC_VECTOR (31 downto 0) :=
"00000000000000000000000000000010";
```

```
    constant ap_const_lv32_3 : STD_LOGIC_VECTOR (31 downto 0) :=
"00000000000000000000000000000011";
    constant ap_const_lv32_4 : STD_LOGIC_VECTOR (31 downto 0) :=
"00000000000000000000000000000100";
    constant ap_const_lv32_5 : STD_LOGIC_VECTOR (31 downto 0) :=
"00000000000000000000000000000101";
    constant ap_const_lv2_0 : STD_LOGIC_VECTOR (1 downto 0) := "00";
    constant ap_const_lv1_1 : STD_LOGIC_VECTOR (0 downto 0) := "1";
    constant ap_const_lv32_6 : STD_LOGIC_VECTOR (31 downto 0) :=
"00000000000000000000000000000110";
    constant ap_const_lv2_3 : STD_LOGIC_VECTOR (1 downto 0) := "11";
    constant ap_const_lv2_1 : STD_LOGIC_VECTOR (1 downto 0) := "01";
    constant ap_const_lv5_1 : STD_LOGIC_VECTOR (4 downto 0) := "00001";
    constant ap_const_lv5_2 : STD_LOGIC_VECTOR (4 downto 0) := "00010";
    constant ap_const_lv4_6 : STD_LOGIC_VECTOR (3 downto 0) := "0110";
    constant ap_const_lv3_3 : STD_LOGIC_VECTOR (2 downto 0) := "011";
    constant ap_const_boolean_1 : BOOLEAN := true;

    signal ap_CS_fsm : STD_LOGIC_VECTOR (6 downto 0) := "0000001";
    attribute fsm_encoding : string;
    attribute fsm_encoding of ap_CS_fsm : signal is "none";
    signal ap_CS_fsm_state1 : STD_LOGIC;
    attribute fsm_encoding of ap_CS_fsm_state1 : signal is "none";
    signal i_1_fu_138_p2 : STD_LOGIC_VECTOR (1 downto 0);
    signal i_1_reg_275 : STD_LOGIC_VECTOR (1 downto 0);
    signal ap_CS_fsm_state2 : STD_LOGIC;
    attribute fsm_encoding of ap_CS_fsm_state2 : signal is "none";
    signal tmp_4_fu_160_p2 : STD_LOGIC_VECTOR (4 downto 0);
    signal tmp_4_reg_280 : STD_LOGIC_VECTOR (4 downto 0);
    signal exitcond4_fu_132_p2 : STD_LOGIC_VECTOR (0 downto 0);
    signal A_addr_reg_285 : STD_LOGIC_VECTOR (3 downto 0);
    signal A_addr_1_reg_290 : STD_LOGIC_VECTOR (3 downto 0);
    signal A_addr_2_reg_295 : STD_LOGIC_VECTOR (3 downto 0);
    signal j_1_fu_199_p2 : STD_LOGIC_VECTOR (1 downto 0);
    signal j_1_reg_303 : STD_LOGIC_VECTOR (1 downto 0);
    signal ap_CS_fsm_state3 : STD_LOGIC;
    attribute fsm_encoding of ap_CS_fsm_state3 : signal is "none";
    signal exitcond3_fu_193_p2 : STD_LOGIC_VECTOR (0 downto 0);
    signal ap_CS_fsm_state4 : STD_LOGIC;
    attribute fsm_encoding of ap_CS_fsm_state4 : signal is "none";
    signal A_load_reg_318 : STD_LOGIC_VECTOR (7 downto 0);
    signal B_load_reg_323 : STD_LOGIC_VECTOR (7 downto 0);
    signal ap_CS_fsm_state5 : STD_LOGIC;
    attribute fsm_encoding of ap_CS_fsm_state5 : signal is "none";
    signal tmp_s_fu_244_p2 : STD_LOGIC_VECTOR (4 downto 0);
    signal tmp_s_reg_333 : STD_LOGIC_VECTOR (4 downto 0);
    signal tmp_9_2_fu_249_p2 : STD_LOGIC_VECTOR (7 downto 0);
    signal tmp_9_2_reg_338 : STD_LOGIC_VECTOR (7 downto 0);
    signal grp_fu_264_p3 : STD_LOGIC_VECTOR (7 downto 0);
    signal tmp_2_2_reg_343 : STD_LOGIC_VECTOR (7 downto 0);
    signal ap_CS_fsm_state6 : STD_LOGIC;
    attribute fsm_encoding of ap_CS_fsm_state6 : signal is "none";
    signal i_reg_109 : STD_LOGIC_VECTOR (1 downto 0);
    signal j_reg_120 : STD_LOGIC_VECTOR (1 downto 0);
    signal ap_CS_fsm_state7 : STD_LOGIC;
    attribute fsm_encoding of ap_CS_fsm_state7 : signal is "none";
    signal tmp_4_cast_fu_166_p1 : STD_LOGIC_VECTOR (31 downto 0);
    signal tmp_5_cast_fu_177_p1 : STD_LOGIC_VECTOR (31 downto 0);
    signal tmp_6_cast_fu_188_p1 : STD_LOGIC_VECTOR (31 downto 0);
    signal tmp_3_fu_205_p1 : STD_LOGIC_VECTOR (31 downto 0);
    signal tmp_8_cast_fu_220_p1 : STD_LOGIC_VECTOR (31 downto 0);
    signal tmp_7_cast_fu_239_p1 : STD_LOGIC_VECTOR (31 downto 0);
    signal tmp_10_cast_fu_255_p1 : STD_LOGIC_VECTOR (31 downto 0);
    signal tmp_1_fu_148_p3 : STD_LOGIC_VECTOR (3 downto 0);
    signal p_shl_cast_fu_156_p1 : STD_LOGIC_VECTOR (4 downto 0);
    signal tmp_cast_fu_144_p1 : STD_LOGIC_VECTOR (4 downto 0);
    signal tmp_5_fu_171_p2 : STD_LOGIC_VECTOR (4 downto 0);
    signal tmp_6_fu_182_p2 : STD_LOGIC_VECTOR (4 downto 0);
    signal tmp_3_cast1_fu_210_p1 : STD_LOGIC_VECTOR (3 downto 0);
    signal tmp_8_fu_214_p2 : STD_LOGIC_VECTOR (3 downto 0);
    Signal tmp_3_cast_fu_229_p1: STD_LOGIC_VECTOR (2 downto 0);
    signal tmp_7_fu_233_p2 : STD_LOGIC_VECTOR (2 downto 0);
    signal tmp_3_cast2_fu_225_p1 : STD_LOGIC_VECTOR (4 downto 0);
    signal tmp_9_2_fu_249_p0 : STD_LOGIC_VECTOR (7 downto 0);
    signal tmp_9_2_fu_249_p1 : STD_LOGIC_VECTOR (7 downto 0);
    signal grp_fu_259_p3 : STD_LOGIC_VECTOR (7 downto 0);
    signal ap_NS_fsm : STD_LOGIC_VECTOR (6 downto 0);

    component mul_mat_mac_muladbkb IS
    generic (
        ID : INTEGER;
        NUM_STAGE : INTEGER;
        din0_WIDTH : INTEGER;
        din1_WIDTH : INTEGER;
        din2_WIDTH : INTEGER;
        dout_WIDTH : INTEGER );
    port (
        din0 : IN STD_LOGIC_VECTOR (7 downto 0);
        din1 : IN STD_LOGIC_VECTOR (7 downto 0);
        din2 : IN STD_LOGIC_VECTOR (7 downto 0);
        dout : OUT STD_LOGIC_VECTOR (7 downto 0) );
    end component;


    component mul_mat_mac_muladcud IS
    generic (
        ID : INTEGER;
        NUM_STAGE : INTEGER;
        din0_WIDTH : INTEGER;
        din1_WIDTH : INTEGER;
        din2_WIDTH : INTEGER;
        dout_WIDTH : INTEGER );
    port (
        din0 : IN STD_LOGIC_VECTOR (7 downto 0);
        din1 : IN STD_LOGIC_VECTOR (7 downto 0);
        din2 : IN STD_LOGIC_VECTOR (7 downto 0);
        dout : OUT STD_LOGIC_VECTOR (7 downto 0) );
    end component;



begin
    mul_mat_mac_muladbkb_U0 : component mul_mat_mac_muladbkb
    generic map (
        ID => 1,
        NUM_STAGE => 1,
        din0_WIDTH => 8,
        din1_WIDTH => 8,
        din2_WIDTH => 8,
        dout_WIDTH => 8)
    port map (
        din0 => B_load_reg_323,
        din1 => A_load_reg_318,
        din2 => tmp_9_2_reg_338,
        dout => grp_fu_259_p3);
    mul_mat_mac_muladcud_U1 : component mul_mat_mac_muladcud
    generic map (
        ID => 1,
        NUM_STAGE => 1,
        din0_WIDTH => 8,
        din1_WIDTH => 8,
        din2_WIDTH => 8,
        dout_WIDTH => 8)
    port map (
        din0 => B_q0,
        din1 => A_q0,
        din2 => grp_fu_259_p3,
        dout => grp_fu_264_p3);





    ap_CS_fsm_assign_proc : process(ap_clk)
    begin
        if (ap_clk'event and ap_clk =  '1') then
            if (ap_rst = '1') then
                ap_CS_fsm <= ap_ST_fsm_state1;
            else
                ap_CS_fsm <= ap_NS_fsm;
            end if;
        end if;
    end process;


    i_reg_109_assign_proc : process (ap_clk)
    begin
        if (ap_clk'event and ap_clk = '1') then
            if (((ap_const_logic_1 = ap_CS_fsm_state3) and (exitcond3_fu_193_p2 =
ap_const_lv1_1))) then
                i_reg_109 <= i_1_reg_275;
            elsif (((ap_const_logic_1 = ap_CS_fsm_state1) and (ap_start =
ap_const_logic_1))) then
                i_reg_109 <= ap_const_lv2_0;
            end if;
        end if;
    end process;

    j_reg_120_assign_proc : process (ap_clk)
    begin
        if (ap_clk'event and ap_clk = '1') then
            if ((ap_const_logic_1 = ap_CS_fsm_state7)) then
                j_reg_120 <= j_1_reg_303;
            elsif (((ap_const_logic_1 = ap_CS_fsm_state2) and (exitcond4_fu_132_p2 =
ap_const_lv1_0))) then
                j_reg_120 <= ap_const_lv2_0;
            end if;
        end if;
    end process;
    process (ap_clk)
    begin
        if (ap_clk'event and ap_clk = '1') then
            if (((ap_const_logic_1 = ap_CS_fsm_state2) and (exitcond4_fu_132_p2 =
ap_const_lv1_0))) then
                A_addr_1_reg_290 <= tmp_5_cast_fu_177_p1(4 - 1 downto 0);
                A_addr_2_reg_295 <= tmp_6_cast_fu_188_p1(4 - 1 downto 0);
                A_addr_reg_285 <= tmp_4_cast_fu_166_p1(4 - 1 downto 0);
                tmp_4_reg_280 <= tmp_4_fu_160_p2;
            end if;
        end if;
    end process;
    process (ap_clk)
    begin
        if (ap_clk'event and ap_clk = '1') then
            if ((ap_const_logic_1 = ap_CS_fsm_state4)) then
                A_load_reg_318 <= A_q0;
                B_load_reg_323 <= B_q0;
            end if;
        end if;
    end process;
    process (ap_clk)
    begin
        if (ap_clk'event and ap_clk = '1') then
            if ((ap_const_logic_1 = ap_CS_fsm_state2)) then
                i_1_reg_275 <= i_1_fu_138_p2;
            end if;
        end if;
    end process;
    process (ap_clk)
    begin
        if (ap_clk'event and ap_clk = '1') then
            if ((ap_const_logic_1 = ap_CS_fsm_state3)) then
                j_1_reg_303 <= j_1_fu_199_p2;
            end if;
        end if;
    end process;
    process (ap_clk)
    begin
        if (ap_clk'event and ap_clk = '1') then
            if ((ap_const_logic_1 = ap_CS_fsm_state6)) then
                tmp_2_2_reg_343 <= grp_fu_264_p3;
            end if;
        end if;
    end process;
    process (ap_clk)
    begin
        if (ap_clk'event and ap_clk = '1') then
            if ((ap_const_logic_1 = ap_CS_fsm_state5)) then
                tmp_9_2_reg_338 <= tmp_9_2_fu_249_p2;
                tmp_s_reg_333 <= tmp_s_fu_244_p2;
            end if;
        end if;
    end process;

    ap_NS_fsm_assign_proc : process (ap_start, ap_CS_fsm, ap_CS_fsm_state1, 
ap_CS_fsm_state2, exitcond4_fu_132_p2, ap_CS_fsm_state3, exitcond3_fu_193_p2)
    begin
        case ap_CS_fsm is
            when ap_ST_fsm_state1 => 
                if (((ap_const_logic_1 = ap_CS_fsm_state1) and (ap_start = 
ap_const_logic_1))) then
                    ap_NS_fsm <= ap_ST_fsm_state2;
                else
                    ap_NS_fsm <= ap_ST_fsm_state1;
                end if;
            when ap_ST_fsm_state2 => 
                if (((ap_const_logic_1 = ap_CS_fsm_state2) and (exitcond4_fu_132_p2 = 
ap_const_lv1_1))) then
                    ap_NS_fsm <= ap_ST_fsm_state1;
                else
                    ap_NS_fsm <= ap_ST_fsm_state3;
                end if;
            when ap_ST_fsm_state3 =>
                if (((ap_const_logic_1 = ap_CS_fsm_state3) and (exitcond3_fu_193_p2 =
ap_const_lv1_1))) then
                    ap_NS_fsm <= ap_ST_fsm_state2;
                else
                    ap_NS_fsm <= ap_ST_fsm_state4;
                end if;
            when ap_ST_fsm_state4 =>
                ap_NS_fsm <= ap_ST_fsm_state5;
            when ap_ST_fsm_state5 =>
                ap_NS_fsm <= ap_ST_fsm_state6;
            when ap_ST_fsm_state6 =>
                ap_NS_fsm <= ap_ST_fsm_state7;
            when ap_ST_fsm_state7 =>
                ap_NS_fsm <= ap_ST_fsm_state3;
            when others =>
                ap_NS_fsm <= "XXXXXXX";
        end case;
    end process;

    A_address0_assign_proc : process(A_addr_reg_285, A_addr_1_reg_290,
A_addr_2_reg_295, ap_CS_fsm_state3, ap_CS_fsm_state4, ap_CS_fsm_state5)
    begin
        if ((ap_const_logic_1 = ap_CS_fsm_state5)) then
            A_address0 <= A_addr_1_reg_290;
        elsif ((ap_const_logic_1 = ap_CS_fsm_state4)) then
            A_address0 <= A_addr_2_reg_295;
        elsif ((ap_const_logic_1 = ap_CS_fsm_state3)) then
            A_address0 <= A_addr_reg_285;
        else
            A_address0 <= "XXXX";
        end if;
    end process;


    A_ce0_assign_proc : process(ap_CS_fsm_state3, ap_CS_fsm_state4, ap_CS_fsm_state5)
    begin
        if (((ap_const_logic_1 = ap_CS_fsm_state3) or (ap_const_logic_1 =
ap_CS_fsm_state4) or (ap_const_logic_1 = ap_CS_fsm_state5))) then
            A_ce0 <= ap_const_logic_1;
        else
            A_ce0 <= ap_const_logic_0;
        end if;
    end process;


    B_address0_assign_proc : process(ap_CS_fsm_state3, ap_CS_fsm_state4,
ap_CS_fsm_state5, tmp_3_fu_205_p1, tmp_8_cast_fu_220_p1, tmp_7_cast_fu_239_p1)
    begin
        if ((ap_const_logic_1 = ap_CS_fsm_state5)) then
            B_address0 <= tmp_7_cast_fu_239_p1(4 - 1 downto 0);
        elsif ((ap_const_logic_1 = ap_CS_fsm_state4)) then
            B_address0 <= tmp_8_cast_fu_220_p1(4 - 1 downto 0);
        elsif ((ap_const_logic_1 = ap_CS_fsm_state3)) then
            B_address0 <= tmp_3_fu_205_p1(4 - 1 downto 0);
        else
            B_address0 <= "XXXX";
        end if;
    end process;


    B_ce0_assign_proc : process(ap_CS_fsm_state3, ap_CS_fsm_state4, ap_CS_fsm_state5)
    begin
        if (((ap_const_logic_1 = ap_CS_fsm_state3) or (ap_const_logic_1 = 
ap_CS_fsm_state4) or (ap_const_logic_1 = ap_CS_fsm_state5))) then 
            B_ce0 <= ap_const_logic_1;
        else 
            B_ce0 <= ap_const_logic_0;
        end if; 
    end process;

    C_address0 <= tmp_10_cast_fu_255_p1(4 - 1 downto 0);

    C_ce0_assign_proc : process(ap_CS_fsm_state7)
    begin
        if ((ap_const_logic_1 = ap_CS_fsm_state7)) then 
            C_ce0 <= ap_const_logic_1;
        else 
            C_ce0 <= ap_const_logic_0;
        end if; 
    end process;

    C_d0 <= tmp_2_2_reg_343;

    C_we0_assign_proc : process(ap_CS_fsm_state7)
    begin
        if ((ap_const_logic_1 = ap_CS_fsm_state7)) then 
            C_we0 <= ap_const_logic_1;
        else 
            C_we0 <= ap_const_logic_0;
        end if; 
    end process;

    ap_CS_fsm_state1 <= ap_CS_fsm(0);
    ap_CS_fsm_state2 <= ap_CS_fsm(1);
    ap_CS_fsm_state3 <= ap_CS_fsm(2);
    ap_CS_fsm_state4 <= ap_CS_fsm(3);
    ap_CS_fsm_state5 <= ap_CS_fsm(4);
    ap_CS_fsm_state6 <= ap_CS_fsm(5);
    ap_CS_fsm_state7 <= ap_CS_fsm(6);

    ap_done_assign_proc : process(ap_CS_fsm_state2, exitcond4_fu_132_p2)
    begin
        if (((ap_const_logic_1 = ap_CS_fsm_state2) and (exitcond4_fu_132_p2 = 
ap_const_lv1_1))) then 
            ap_done <= ap_const_logic_1;
        else 
            ap_done <= ap_const_logic_0;
        end if; 
    end process;


    ap_idle_assign_proc : process(ap_start, ap_CS_fsm_state1)
    begin
        if (((ap_const_logic_0 = ap_start) and (ap_const_logic_1 = ap_CS_fsm_state1))) 
then 
            ap_idle <= ap_const_logic_1;
        else 
            ap_idle <= ap_const_logic_0;
        end if; 
    end process;


    ap_ready_assign_proc : process(ap_CS_fsm_state2, exitcond4_fu_132_p2)
    begin
```

```
        if (((ap_const_logic_1 = ap_CS_fsm_state2) and (exitcond4_fu_132_p2 = 
ap_const_lv1_1))) then 
            ap_ready <= ap_const_logic_1;
        else
            ap_ready <= ap_const_logic_0;
        end if; 
    end process;

    exitcond3_fu_193_p2 <= "1" when (j_reg_120 = ap_const_lv2_3) else "0";
    exitcond4_fu_132_p2 <= "1" when (i_reg_109 = ap_const_lv2_3) else "0";
    i_1_fu_138_p2 <= std_logic_vector(unsigned(i_reg_109) + unsigned(ap_const_lv2_1));
    j_1_fu_199_p2 <= std_logic_vector(unsigned(j_reg_120) + unsigned(ap_const_lv2_1));
    p_shl_cast_fu_156_p1 <= 
std_logic_vector(IEEE.numeric_std.resize(unsigned(tmp_1_fu_148_p3),5));
        tmp_10_cast_fu_255_p1 <= 
std_logic_vector(IEEE.numeric_std.resize(signed(tmp_s_reg_333),32));

    tmp_1_fu_148_p3 <= (i_reg_109 & ap_const_lv2_0);
    tmp_3_cast1_fu_210_p1 <= 
std_logic_vector(IEEE.numeric_std.resize(unsigned(j_reg_120),4));
    tmp_3_cast2_fu_225_p1 <= 
std_logic_vector(IEEE.numeric_std.resize(unsigned(j_reg_120),5));
    tmp_3_cast_fu_229_p1 <= 
std_logic_vector(IEEE.numeric_std.resize(unsigned(j_reg_120),3));
    tmp_3_fu_205_p1 <= 
std_logic_vector(IEEE.numeric_std.resize(unsigned(j_reg_120),32));
        tmp_4_cast_fu_166_p1 <= 
std_logic_vector(IEEE.numeric_std.resize(signed(tmp_4_fu_160_p2),32));

    tmp_4_fu_160_p2 <= std_logic_vector(unsigned(p_shl_cast_fu_156_p1) - 
unsigned(tmp_cast_fu_144_p1));
        tmp_5_cast_fu_177_p1 <= 
std_logic_vector(IEEE.numeric_std.resize(signed(tmp_5_fu_171_p2),32));

    tmp_5_fu_171_p2 <= std_logic_vector(unsigned(tmp_4_fu_160_p2) + 
unsigned(ap_const_lv5_1));
        tmp_6_cast_fu_188_p1 <= 
std_logic_vector(IEEE.numeric_std.resize(signed(tmp_6_fu_182_p2),32));

    tmp_6_fu_182_p2 <= std_logic_vector(unsigned(tmp_4_fu_160_p2) + 
unsigned(ap_const_lv5_2));
    tmp_7_cast_fu_239_p1 <= 
std_logic_vector(IEEE.numeric_std.resize(unsigned(tmp_7_fu_233_p2),32));
    tmp_7_fu_233_p2 <= std_logic_vector(unsigned(tmp_3_cast_fu_229_p1) + 
unsigned(ap_const_lv3_3));
    tmp_8_cast_fu_220_p1 <= 
std_logic_vector(IEEE.numeric_std.resize(unsigned(tmp_8_fu_214_p2),32));
    tmp_8_fu_214_p2 <= std_logic_vector(unsigned(tmp_3_cast1_fu_210_p1) + 
unsigned(ap_const_lv4_6));
    tmp_9_2_fu_249_p0 <= B_q0;
    tmp_9_2_fu_249_p1 <= A_q0;
    tmp_9_2_fu_249_p2 <= 
std_logic_vector(IEEE.numeric_std.resize(unsigned(std_logic_vector(signed(tmp_9_2_fu_24
9_p0) * signed(tmp_9_2_fu_249_p1))), 8));
    tmp_cast_fu_144_p1 <= 
std_logic_vector(IEEE.numeric_std.resize(unsigned(i_reg_109),5));
    tmp_s_fu_244_p2 <= std_logic_vector(unsigned(tmp_4_reg_280) + 
unsigned(tmp_3_cast2_fu_225_p1));

end behav;
```

FINAL DESIGN :

The design of matrix multiplication of a 3*3 matrices produced by the code in previous chapter is as follows

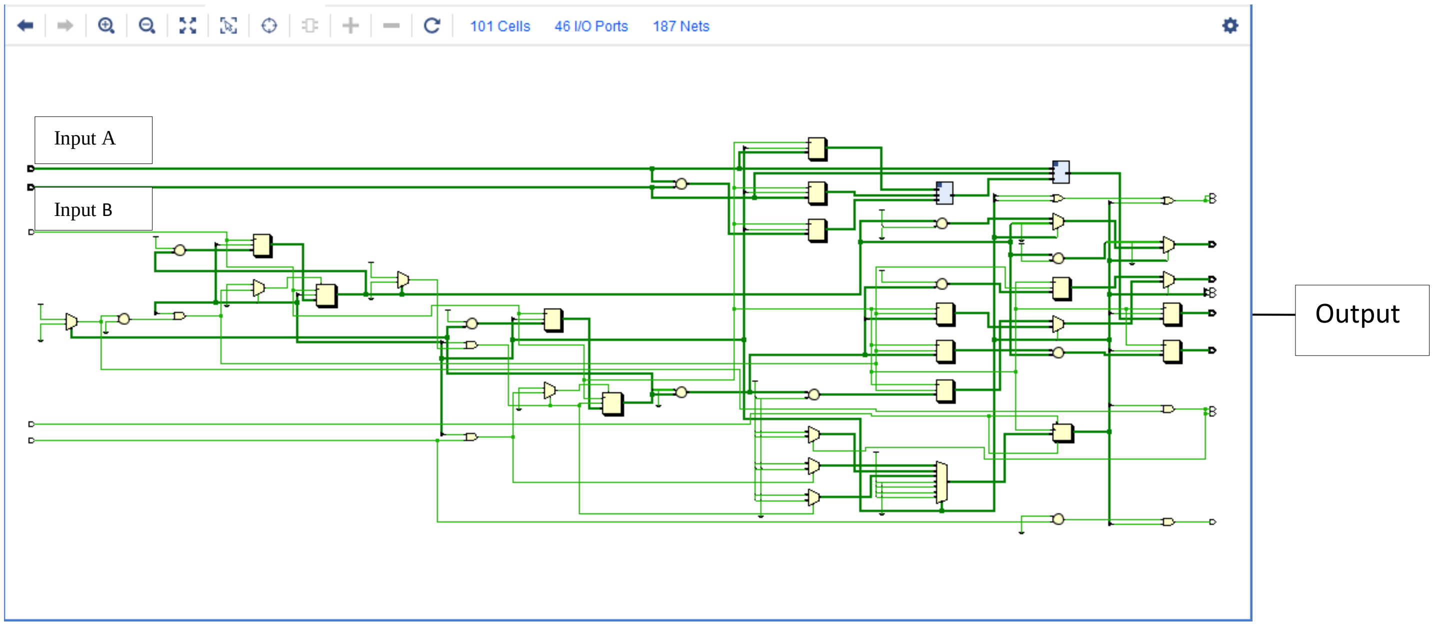
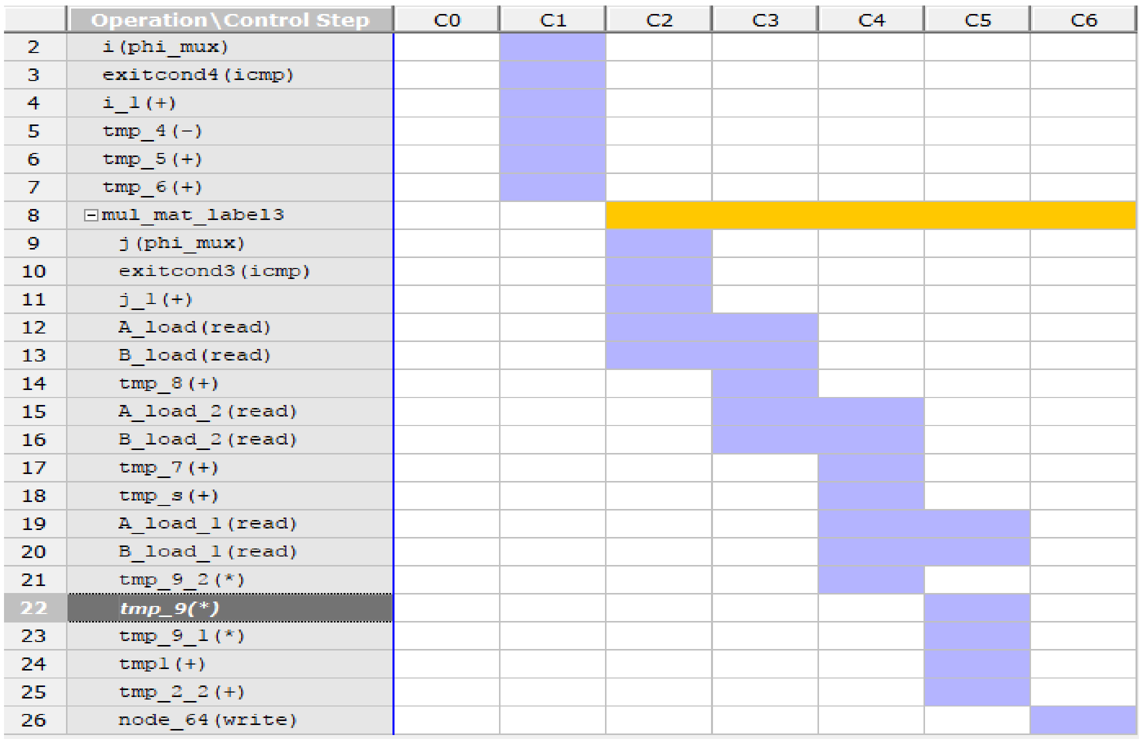


Figure 23. Final RTL Design

And operation / control steps for this design are as follows:



	Operation\Control Step	C0	C1	C2	C3	C4	C5	C6
2	i(phi_mux)		🟦					
3	exitcond4(icmp)		🟦					
4	i_1(+)		🟦					
5	tmp_4(-)		🟦					
6	tmp_5(+)		🟦					
7	tmp_6(+)		🟦					
8	⊟mul_mat_label3			🟨	🟨	🟨	🟨	🟨
9	j(phi_mux)			🟦				
10	exitcond3(icmp)			🟦				
11	j_1(+)			🟦				
12	A_load(read)			🟦	🟦			
13	B_load(read)			🟦	🟦			
14	tmp_8(+)				🟦			
15	A_load_2(read)				🟦	🟦		
16	B_load_2(read)				🟦	🟦		
17	tmp_7(+)					🟦		
18	tmp_s(+)					🟦		
19	A_load_1(read)					🟦	🟦	
20	B_load_1(read)					🟦	🟦	
21	tmp_9_2(*)					🟦		
22	tmp_9(*)						🟦	
23	tmp_9_1(*)						🟦	
24	tmp1(+)						🟦	
25	tmp_2_2(+)						🟦	
26	node_64(write)							🟦

Figure 24. timing diagram of operation

CHAPTER 4: APPLICATIONS BASED ON THE PROJECT

Computation-intensive algorithms require a high level of parallelism and programmability, which make them good candidate for hardware acceleration using fine-grained processor arrays. Using Hardware Description Language (HDL), it is very difficult to design and manage fine-grained processing units and therefore High-Level Language (HLL) is a preferred alternative.

Data-intensive computing is a class of parallel computing applications which use a data parallel approach to process large volumes of data typically terabytes or petabytes in size and typically referred to as big data. Computing applications which devote most of their execution time to computational requirements are deemed compute-intensive, whereas computing applications which require large volumes of data and devote most of their processing time to I/O and manipulation of data are deemed data-intensive.

The concept of this project can be used to solve the major problems faced by many research oriented fields which involve intensive high-level computational problems.

- The algorithms involved in **climate modeling and weather predictions** are very complex and computationally demanding as they operate over large domains requiring high resolutions or very long integration times thus exacerbating the need for high-performance resources so that accurate estimates can be found in useful time (a weather prediction for tomorrow is hardly useful if it requires more than 24 hours to compute). Improving prediction quality and accuracy requires higher resolution models found by exploring a wide range of its parameters. This is a daunting and error-prone process, demanding even more computing power.

- **Astronomy and Astrophysics** are two fields that often require optimizing problems of high complexity or analyzing a huge amount of data and the so-called complete optimization methods are inherently limited by the size of the problem/data. For instance, reliable analysis of large amounts of data is central to modern astrophysics and astronomical sciences in general. Some of these problems, such as the estimation of non-lineal parameters, the development of automatic learning techniques, the implementation of control systems, or the resolution of multi-objective optimization problems, have had (and have) a special repercussion in the fields.

- **Intensive Mathematical Fields**

- Statistical distributions - the Binomial, Poisson and Gaussian
- Least squares fitting to data. The extension to multi-variate least squares is made using matrix methods

- Numerical solution to differential equations. Euler Method to Runge-Kutta method along with Boundary Value problems
- Find periods in unequally sampled data. It is shown how a period can be extracted from noisy data. This leads to the concept of the periodogram.

- **Computational physics**, to solve problems in quantum physics, electromagnetism, biophysics, mechanics, chaos, nonlinear dynamics, and other areas.
- **Computational Chemistry,** for computing the interatomic distances between the particles present in a crystal lattice. The study can be further be taken ahead to understand the potential energy behavior for each particle in the lattice as potential energy is proportional to the interatomic distance.

The growth of FPGA Programming and hardware implementation is unimaginable, they can be used almost for all possible digital logic setup purposes.

CHAPTER 5: CONCLUSION AND SCOPE FOR FUTURE WORK

As the FPGA offers us a dual advantage of the flexibility in hardware designing and parallelizing the design process, the concept of the project can be explained as; we are looking to incorporate a method using FPGA and hardware programming to combine the two aspects being, designing a complex system to solve high-level intensive computations and to optimize the system performance to give a reduced latency (delay).

Through this work in our project we have achieved the level where we can say that the FPGA is completely user friendly and is reprogrammable in any way so as to achieve almost any possible digital logic system.

We have designed the Structures for two separate kinds of mathematical operations, basic mathematical operations (addition, etc.), which we achieved using HDL programming (VHDL/Verilog), and matrix multiplication, which we achieved using HLL programming(C/C++).

We have also achieved a possible way of interaction between the hardware FPGA and the host PC, using UART Serial Communication Protocol, and have also verified its reliability and design ease on the FPGA.

We have also studied and implemented certain parallelizing directives (pipelining, loop unrolling, etc.) which could be used to reduce latencies for the basic architecture designs. The optimization is a key component in today’s world and is essential for any hardware development.

Since, the project is still in developmental stage, the ultimate objective is to combine both the works that have been done by us, one, the architecture of system and interaction of FPGA and PC, and second, optimizing/reducing time cycles required for the operations to be completed, to create one final software that could do both the works efficiently and provide improved and enhanced results in faster time periods.

Obviously, the architecture can be reprogrammed according to user needs and the parallelizing techniques can always be improved according to the increasing demand in development models of hardware implemented computation modules.

REFERENCES

- Shouqian Yu ,Lili Yi, Weihai Chen, Zhaojin Wen - Implementation of a Multi-channel UART Controller Based on FIFO Technique and FPGA
- Sonali Dhage, Manali Patil,Navnath Temgire,Pushkar Vaity, Sangeeta Parshionikar - Design and FPGA Implementation of a High Speed UART
- FPGA Center - http://fpgacenter.com/fpga/index.php
- BASYS 3 Reference Manual - https://reference.digilentinc.com/basys3/refmanual
- RealTerm - Serial/TCP Terminal Software - http://www.softpedia.com/get/System/SystemMiscellaneous/RealTerm.shtml
- Basics of UART Communication - http://www.circuitbasics.com/basics-uart-communication/
- Evolutionary Computation in Astronomy and Astrophysics: A Review -- José A. García Gutiérrez, Carlos Cotta, Antonio J. Fernández-Leiva
- CLIMA2016 Computational Challenges for Climate Modeling and Weather Prediction
- Computer Organization and Architecture and Pipelining - http://www.geeksforgeeks.org/computer-organization-and-architecture-pipelining-set-2-dependencies-and-data-hazard/
- Parallel Computing - https://en.wikipedia.org/wiki/Parallel_computing
- Ruediger Willenberg, Paul Chow Electrical & Computer Engineering, University of Toronto -- A software parallel programming approach to FPGA-accelerated computing
- Parallel Computing (EECS) - http://web.eecs.umich.edu/~qstout/parallel.html
- Loop Unrolling - http://www.geeksforgeeks.org/loop-unrolling/
- Introduction to FPGA Design with Vivado High-Level Synthesis https://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf

APPENDIX

1) Meta- Stability Hardener Source Code:

```
`timescale 1ns/1ps
module meta_harden (
  input          clk_dst,     // Destination clock
  input          rst_dst,     // Reset - synchronous to destination clock
  input          signal_src,  // Asynchronous signal to be synchronized
  output reg     signal_dst   // Synchronized signal
);
//***************************************************************************
// Register declarations
//***************************************************************************
  reg        signal_meta;    // After sampling the async signal, this has a high probability of being metastable.
                             The second sampling (signal_dst) has a much lower probability of being metastable.

//***************************************************************************
// Code
//***************************************************************************
  always @(posedge clk_dst)
  begin
    if (rst_dst)
    begin
      signal_meta <= 1'b0;
      signal_dst  <= 1'b0;
    end
    else // if !rst_dst
    begin
      signal_meta <= signal_src;
      signal_dst  <= signal_meta;
    end // if rst
  end // always
endmodule
```

2. Baud Generator Source Code:

```
//  Note: Divider must be at least 2 (thus CLOCK_RATE must be at least 32x  BAUD_RATE)
`timescale 1ns/1ps
module uart_baud_gen (
  // Write side inputs
  input       clk,         // Clock input
  input       rst,         // Active HIGH reset - synchronous to clk
  output      baud_x16_en  // Oversampled Baud rate enable
);
//***************************************************************************
// Constant Functions
//***************************************************************************
```

```
  // Generate the ceiling of the log base 2 - i.e. the number of bits required to hold N values. A vector of size
clogb2(N) will hold the values 0 to N-1
  function integer clogb2;
    input [31:0] value;
    reg   [31:0] my_value;
    begin
      my_value = value - 1;
      for (clogb2 = 0; my_value > 0; clogb2 = clogb2 + 1)
        my_value = my_value >> 1;
    end
  endfunction
//***************************************************************************
// Parameter definitions
//***************************************************************************
  parameter BAUD_RATE    = 9_600;             // Baud rate
  parameter CLOCK_RATE   = 100_000_000;
  // The OVERSAMPLE_RATE is the BAUD_RATE times 16
  localparam OVERSAMPLE_RATE = BAUD_RATE * 16;
  // The divider is the CLOCK_RATE / OVERSAMPLE_RATE - rounded up (so add 1/2 of the
OVERSAMPLE_RATE before the integer division)
  localparam DIVIDER = (CLOCK_RATE+OVERSAMPLE_RATE/2) / OVERSAMPLE_RATE;
  // The value to reload the counter is DIVIDER-1;
  localparam OVERSAMPLE_VALUE = DIVIDER - 1;
// The required width of the counter is the ceiling of the base 2 logarithm of the DIVIDER
  localparam CNT_WID = clogb2(DIVIDER);
//***************************************************************************
// Reg declarations
//***************************************************************************
  reg [CNT_WID-1:0] internal_count;
  reg               baud_x16_en_reg;
//***************************************************************************
// Wire declarations
//***************************************************************************
  wire [CNT_WID-1:0] internal_count_m_1; // Count minus 1
//***************************************************************************
// Code
//***************************************************************************
  assign internal_count_m_1 = internal_count - 1'b1;
// Count from DIVIDER-1 to 0, setting baud_x16_en_reg when internal_count=0. The signal
baud_x16_en_reg must come from a flop (since it is a module output) so schedule it to be set when the next
count is 1 (i.e. when internal_count_m_1 is 0).
  always @(posedge clk)
  begin
    if (rst)
    begin
      internal_count  <= OVERSAMPLE_VALUE;
      baud_x16_en_reg <= 1'b0;
    end
    else
    begin
      // Assert baud_x16_en_reg in the next clock when internal_count will be zero in that clock (thus when
internal_count_m_1 is 0).
```

```
      baud_x16_en_reg   <= (internal_count_m_1 == {CNT_WID{1'b0}});

      // Count from OVERSAMPLE_VALUE down to 0 repeatedly
      if (internal_count == {CNT_WID{1'b0}})
      begin
        internal_count    <= OVERSAMPLE_VALUE;
      end
      else // internal_count is not 0
      begin
        internal_count    <= internal_count_m_1;
      end
    end // if rst
  end // always
  assign baud_x16_en = baud_x16_en_reg;
        endmodule
```

3. <u>CLOCK DIVIDER SOURCE CODE:</u>

```
        // Clock divider circuit
        // From 100 MHz to 1 Hz with %50 duty cycle

        module clk_div(Clk_in, Clk_out);

        input Clk_in;  // input ports
        output reg Clk_out ; // output ports
        reg Clk_out_1 ;//= 1'b0; // provide initial condition for this register.

        // counter size calculation according to input and output frequencies
        parameter sys_clk = 100000000;        // 100 MHz system clock
        parameter clk_out = 1;            // 1 Hz clock output
        parameter max = sys_clk / (2*clk_out); // max-counter size

        reg [31:0]counter = 0; // 32-bit counter size
          initial Clk_out_1 = 1'b0;

        always@(posedge Clk_in) begin
                if (counter == max-1)
                        begin
                        counter <= 0;
                        Clk_out_1 <= ~Clk_out_1;
                        end
                else
                        begin
                        counter <= counter + 1'd1;
                        end

                 Clk_out = Clk_out_1;

                end
        endmodule
```

4. SELECTION MODULE CODE

```
module selector (switch,data, Clock,Out1, Out2);
input switch;
input data;
input Clock;
output reg Out1;
output reg Out2;

always@(posedge Clock)
begin
case (switch)

1'b0:  begin
       Out1 = data;
       Out2 = "0";
       end
1'b1:  begin
       Out1 = "0";
       Out2 = data;
       end
 endcase
 end

endmodule
```

5. INVERSION MODULE CODE

```
module inv(
    input a,
    output b
    );

 not(b,a);

endmodule
```

6. RECEIVER UART CONTROL MODULE

```
`timescale 1ns/1ps
module uart_rx_ctl (
  // Write side inputs
  input           clk_rx,       // Clock input
  input           rst_clk_rx,   // Active HIGH reset - synchronous to clk_rx
  input           baud_x16_en,  // 16x oversampling enable

  input           rxd_clk_rx,   // RS232 RXD pin - after sync to clk_rx

  output reg [7:0] rx_data,      // 8 bit data output
                               //  - valid when rx_data_rdy is asserted
  output reg       rx_data_rdy,  // Ready signal for rx_data
  output reg       frm_err       // The STOP bit was not detected
);
//***************************************************************************
// Parameter definitions
//***************************************************************************
  // State encoding for main FSM
  localparam
    IDLE  = 2'b00,
    START = 2'b01,
    DATA  = 2'b10,
    STOP  = 2'b11;
//***************************************************************************
// Reg declarations
//***************************************************************************
  reg [1:0]    state;            // Main state machine
  reg [3:0]    over_sample_cnt;   // Oversample counter - 16 per bit
  reg [2:0]    bit_cnt;           // Bit counter - which bit are we RXing
//***************************************************************************
// Wire declarations
//***************************************************************************
  wire         over_sample_cnt_done; // We are in the middle of a bit
  wire         bit_cnt_done;         // This is the last data bit
//***************************************************************************
// Code
//***************************************************************************
  // Main state machine
  always @(posedge clk_rx)
  begin
    if (rst_clk_rx)
    begin
      state       <= IDLE;
    end
    else
    begin
      if (baud_x16_en)
      begin
        case (state)
          IDLE: begin
            // On detection of rxd_clk_rx being low, transition to the START
            // state
            if (!rxd_clk_rx)
            begin
              state <= START;
            end
          end // IDLE state
          START: begin
            // After 1/2 bit period, re-confirm the start state
            if (over_sample_cnt_done)
            begin
              if (!rxd_clk_rx)
              begin
                // Was a legitimate start bit (not a glitch)
                state <= DATA;
              end
              else
              begin
                // Was a glitch - reject
                state <= IDLE;
              end
            end // if over_sample_cnt_done
          end // START state
          DATA: begin
            // Once the last bit has been received, check for the stop bit
            if (over_sample_cnt_done && bit_cnt_done)
            begin
              state <= STOP;
            end
          end // DATA state
          STOP: begin
            // Return to idle
            if (over_sample_cnt_done)
            begin
              state <= IDLE;
            end
          end // STOP state
        endcase
      end // if baud_x16_en
    end // if rst_clk_rx
  end // always
  // Oversample counter
  // Pre-load to 7 when a start condition is detected (rxd_clk_rx is 0 while in IDLE) - this will get us to the
middle of the first bit. Pre-load to 15 after the START is confirmed and between all data bits.
  always @(posedge clk_rx)
  begin
    if (rst_clk_rx)
    begin
      over_sample_cnt    <= 4'd0;
    end
    else
    begin
      if (baud_x16_en)
      begin
        if (!over_sample_cnt_done)
        begin
          over_sample_cnt <= over_sample_cnt - 1'b1;
        end
        else
        begin
          if ((state == IDLE) && !rxd_clk_rx)
          begin
            over_sample_cnt <= 4'd7;
          end
          else if ( ((state == START) && !rxd_clk_rx) || (state == DATA)  )
          begin
            over_sample_cnt <= 4'd15;
          end
        end
      end // if baud_x16_en
    end // if rst_clk_rx
  end // always
  assign over_sample_cnt_done = (over_sample_cnt == 4'd0);
  // Track which bit we are about to receive
  // Set to 0 when we confirm the start condition
  // Increment in all DATA states
  always @(posedge clk_rx)
  begin
    if (rst_clk_rx)
    begin
      bit_cnt    <= 3'b0;
    end
    else
    begin
      if (baud_x16_en)
      begin
        if (over_sample_cnt_done)
        begin
          if (state == START)
          begin
            bit_cnt <= 3'd0;
          end
          else if (state == DATA)
          begin
            bit_cnt <= bit_cnt + 1'b1;
          end
        end // if over_sample_cnt_done
      end // if baud_x16_en
    end // if rst_clk_rx
  end // always
  assign bit_cnt_done = (bit_cnt == 3'd7);
  // Capture the data and generate the rdy signal. The rdy signal will be generated as soon as the last bit of
data is captured - even though the STOP bit hasn't been confirmed. It will remain asserted for one BIT period
(16 baud_x16_en periods)
  always @(posedge clk_rx)
  begin
    if (rst_clk_rx)
    begin
      rx_data     <= 8'b0000_0000;
      rx_data_rdy <= 1'b0;
    end
    else
    begin
      if (baud_x16_en && over_sample_cnt_done)
      begin
        if (state == DATA)
        begin
          rx_data[bit_cnt] <= rxd_clk_rx;
          rx_data_rdy      <= (bit_cnt == 3'd7);
        end
        else
        begin
          rx_data_rdy      <= 1'b0;
        end
      end
    end // if rst_clk_rx
  end // always
  // Framing error generation Generate for one baud_x16_en period as soon as the framing bit is supposed to
be sampled
  always @(posedge clk_rx)
  begin
    if (rst_clk_rx)
    begin
      frm_err     <= 1'b0;
    end
    else
    begin
      if (baud_x16_en)
      begin
        if ((state == STOP) && over_sample_cnt_done && !rxd_clk_rx)
        begin
          frm_err <= 1'b1;
        end
        else
        begin
          frm_err <= 1'b0;
        end
      end // if baud_x16_en
    end // if rst_clk_rx
  end // always
```

7. TRANSMISSION UART CONTROL MODULE

```
`timescale 1ns/1ps
 module uart_tx_ctl (
  input           clk_tx,          // Clock input
  input           rst_clk_tx,      // Active HIGH reset - synchronous to clk_tx
  input           baud_x16_en,     // 16x bit oversampling pulse
  input           char_fifo_empty, // Empty signal from char FIFO (FWFT)
  input      [7:0] char_fifo_dout,  // Data from the char FIFO
  output           char_fifo_rd_en, // Pop signal to the char FIFO
  output reg       txd_tx           // The transmit serial signal
);
//***************************************************************************
// Parameter definitions
//***************************************************************************
  // State encoding for main FSM
  localparam
    IDLE  = 2'b00,
    START = 2'b01,
    DATA  = 2'b10,
    STOP  = 2'b11;
//*****************************************************************************
// Reg declarations
//*****************************************************************************
  reg [1:0]    state;             // Main state machine
  reg [3:0]    over_sample_cnt;   // Oversample counter - 16 per bit
  reg [2:0]    bit_cnt;           // Bit counter - which bit are we RXing
  reg          char_fifo_pop;     // POP indication to FIFO
                                  // ANDed with baud_x16_en before module
                                  // output
//*****************************************************************************
// Wire declarations
//*****************************************************************************
  wire         over_sample_cnt_done; // We are in the middle of a bit
  wire         bit_cnt_done;         // This is the last data bit
//*****************************************************************************
// Code
//*****************************************************************************
  // Main state machine
  always @(posedge clk_tx)
  begin
    if (rst_clk_tx)
    begin
      state         <= IDLE;
      char_fifo_pop <= 1'b0;
    end
    else
    begin
      if (baud_x16_en)
      begin
        char_fifo_pop <= 1'b0;
        case (state)
          IDLE: begin
            // When the character FIFO is not empty, transition to the START state
            if (!char_fifo_empty)
            begin
              state <= START;
            end
          end // IDLE state

          START: begin
            if (over_sample_cnt_done)
            begin
              state <= DATA;
            end // if over_sample_cnt_done
          end // START state
          DATA: begin
            // Once the last bit has been transmitted, send the stop bit
            // Also, we need to POP the FIFO
            if (over_sample_cnt_done && bit_cnt_done)
            begin
```

```
              char_fifo_pop <= 1'b1;
              state         <= STOP;
            end
          end // DATA state
          STOP: begin
            if (over_sample_cnt_done)
            begin
              // If there is no new character to start, return to IDLE, else
              // start it right away
              if (char_fifo_empty)
              begin
                state <= IDLE;
              end
              else
              begin
                state <= START;
              end
            end
          end // STOP state
        endcase
      end // if baud_x16_en
    end // if rst_clk_tx
  end // always
  // Assert the rd_en to the FIFO for only ONE clock period
  assign char_fifo_rd_en = char_fifo_pop && baud_x16_en;
  // Oversample counter-- Pre-load whenever we are starting a new character (in IDLE or in STOP), or
whenever we are within a character (when we are in START or DATA).
  always @(posedge clk_tx)
  begin
    if (rst_clk_tx)
    begin
      over_sample_cnt    <= 4'd0;
    end
    else
    begin
      if (baud_x16_en)
      begin
        if (!over_sample_cnt_done)
        begin
          over_sample_cnt <= over_sample_cnt - 1'b1;
        end
        else
        begin
          if (((state == IDLE) && !char_fifo_empty) ||
              (state == START) ||
              (state == DATA)  ||
              ((state == STOP) && !char_fifo_empty))
          begin
            over_sample_cnt <= 4'd15;
          end
        end
      end // if baud_x16_en
    end // if rst_clk_tx
  end // always
  assign over_sample_cnt_done = (over_sample_cnt == 4'd0);
  // Track which bit we are about to transmit
  // Set to 0 in the START state
  // Increment in all DATA states
  always @(posedge clk_tx)
  begin
    if (rst_clk_tx)
    begin
      bit_cnt    <= 3'b0;
    end
    else
    begin
      if (baud_x16_en)
      begin
        if (over_sample_cnt_done)
        begin
          if (state == START)
          begin
            bit_cnt <= 3'd0;
          end
          else if (state == DATA)
          begin
            bit_cnt <= bit_cnt + 1'b1;
          end
        end // if over_sample_cnt_done
      end // if baud_x16_en
    end // if rst_clk_tx
  end // always
  assign bit_cnt_done = (bit_cnt == 3'd7);
  // Generate the output
  always @(posedge clk_tx)
  begin
    if (rst_clk_tx)
    begin
      txd_tx    <= 1'b1;
    end
    else
    begin
      if (baud_x16_en)
      begin
        if ((state == STOP) || (state == IDLE))
        begin
          txd_tx <= 1'b1;
        end
        else if (state == START)
        begin
          txd_tx <= 1'b0;
        end
        else // we are in DATA
        begin
```

```
        txd_tx <= char_fifo_dout[bit_cnt];
      end
    end // if baud_x16_en
  end // if rst
end // always
endmodule
```

8. Matrix Multiplication Source Code:

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

#define M 3
#define N 3
#define t 3
#define u 3
#define v 3

void mul_mat(char A[M][N],char B[M][N],char C[M][N])

{
char i,j,k;
if(t!=u)
{
printf("\nNot possible");
exit(0);
}
else
{
mul_mat_label2:for(i=0;i<t;i++)
{
mul_mat_label3:for(j=0;j<u;j++)
{
C[i][j]=0;
mul_mat_label4:for(k=0;k<v;k++)

{

      C[i][j] +=  A[i][k]*B[k][j];


}

}

}

}
printf("\nThe resulting matrix:\n");
mul_mat_label1:for(i=0;i<t;i++)
{
mul_mat_label0:for(j=0;j<u;j++)
{

printf("%d\t",C[i][j]);

}

printf("\n");

}
}
```