DESIGN OF SERDES &
DESIGN OF MULTICORE PROCESSOR FOR MATRIX MULTIPLICATION

A

Dissertation

Submitted in the fulfilment of the requirements

For the award of degree

Of

MASTER OF TECHNOLOGY

In

VLSI Design and Embedded System

By

SIDDHARTH SINGH

(2K20/VLS/19)

Under the Guidance of

Dr. S. Indu

ELECTRONICS AND COMMUNICATION DEPARTMENT
DELHI TECHNOLOGICAL UNIVERSITY
DELHI-110042
SESSION 2020-2022

ELECTRONICS AND COMMUNICATION DEPARTMENT

DELHI TECHNOLOGICAL UNIVERSITY
DELHI-110042
SESSION 2020-2022

CANDIDATE'S DECLARATION

I hereby declare that the work being presented in this dissertation entitled **"Design of Serdes and Design of Multicore Processor Optimized for Matrix Multiplication"** submitted towards the fulfilment of the Major project requirements for the award of degree, Master of Technology in VLSI Design and Embedded System to the Electronics and Communication Dept., Delhi Technological University, is an authentic record of my work carried out from January 2022 to June 2022, under the guidance of Dr. S Indu, Electronics and Communication Dept., Delhi Technological University, Delhi.

I have not submitted the matter embodied in the dissertation for the award of any other degree.

Siddharth Singh
2K20/VLS/19
Electronics and Communication Department
Date: 24th June, 2021

ELECTRONICS AND COMMUNICATION DEPARTMENT

DELHI TECHNOLOGICAL UNIVERSITY
DELHI-110042
SESSION 2020-2022

CERTIFICATE

This is to certify that the dissertation entitled **"Design of Serdes and Design of Multicore Processor Optimized for Matrix Multiplication"** is the authentic record of work done by **Siddharth Singh** under my guidance and supervision. This dissertation is being submitted to the ***Delhi Technological University, Delhi*** towards the fulfilment of the requirements for the award of ***degree of Master of Technology in VLSI Design and Embedded System.***

DATE:26th MAY 2022

DR. S. INDU
Professor
Electronics and Communication Department
Delhi Technological University, Delhi

ACKNOWLEDGEMENT

I would like to express my deep gratitude and appreciation to all the people who have helped and supported me in the process of dissertation. Without their help and support, 1 would not have been able to reach this level of satisfaction with what 1 have learnt and accomplished during my Master's dissertation. First and foremost, I would like to express my deep sense of respect and gratitude towards my supervisor Dr. S. Indu, Professor, Electronics and Communication Dept., DTU, for giving me opportunity to do my Major project of master's dissertation under her guidance. I am very thankful for her for giving me the opportunity to choose such an interesting topic by my own. I would also like to thanks the NPTEL Lectures for their valuable thoughts and knowledge, which motivated me to do better. Finally, none of this would have been possible without incredible support of my friends. They were always supporting me and encouraging me with their best wishes.

Siddharth Singh
Roll No. 2K20/VLS/19
Electronics and Communication Dept.

ABSTRACT

This project covers the analysis, design and simulation of two different work done by me for project which was the design of SerDes system and multiprocessor working.

SerDes is a pair of blocks that is used in high-speed communication and to compensate for limited input/output. Conversion of serial data in parallel interfaces takes place in these blocks. For this project I have simulated the serializer part of the SerDes.

Multicore Processor is designed to have matix multiplication in which every core has its own ALU,control unit and registers.

TABLE OF CONTENTS

DECLARATION ii

CERTIFICATE iii

ACKNOWLEDGEMENT iv

ABSTRACT v

LIST OF FIGURES ix

CHAPTER 1: INTRODUCTION OF SERDES 12

1.1 MOTIVATION 12

1.2 OVERVIEW 13

CHAPTER 2: ARCHITECTURE OF HIGH SPEED SERDES 14

2.1 DATA TRANSMISSION 14

2.1.1 Parallel Transmission

2.1.2 Serial Communication

2.2 HIGH SPEED SERDES 16

2.3 TRANSMITTER 17

2.3.1 Phase Locked Loop

2.3.2 Serializer

2.3.3 Driver

2.4 RECEIEVER 19

2.4.1 Clock And Data Recovery

2.4.2 Deserializer

CHAPTER 3: RESULTS 23

3.1 OVERVIEW 23

3.2 PHASE LOCKED LOOP(PLL) 23

3.3 SERIALIZER 28

3.4 INVERTER BASED TRANSMITTER 29

3.5 RESISTIVE FEEDBACK INVERTER 31

3.6 D FLIP FLOP 32

3.7 CLOCK DATA RECOVERY (CDR) 33

3.8 DESERIALIZER 33

3.9 RECEIVER 34

3.10 CONCLUSION 35

CHAPTER 4 MULTICORE PROCESSOR DESIGN 36

4.1 INTRODUCTION 36

4.1.1 Single-Core Processor

4.1.2 Multi-Core Processor

4.1.3 Design Question

4.1.4 Proposed Solution

4.1.5 Processor Design Flow

CHAPTER 5 MODULES 40

5.1 DATA MEMORY 40

5.2 INSTRUCTION MEMORY 41

5.3 MULTI CORE PROCESSOR 42

5.4 PROCESSOR 42

5.5 CONTROL UNIT 44

5.6 MULTIPLEXER 44

CHAPTER 6: SIMULATION & SUMMARY 46

6.1 SIMULATIONS 46

6.2 DESIGN SUMMARY 48

REFERENCES 51

APPENDIX 52

LIST OF FIGURES

Figure No.	TITLE	PAGE No.
2.1.1	Parallel Transmission	15
2.1.2	Serial Transmission	15
2.2	High Speed Serdes Block Diagram	16
2.3	Transmitter Block Diagram	17
2.3.1	PLL Block Diagram	18
2.4	Receiver Block Diagram	20
2.4.1	CDR Block Diagram	21
3.2.1	Schematic Of Phase Frequency Detector (PFD)	23
3.2.2	Schematic of Charge Pump.	24
3.2.3	Schematic of Current Starved VCO.	25
3.2.4	Schematic of Frequency Divider.	26
3.2.5	Schematic of PLL.	26
3.2.6	Output Graph of PLL	27
3.3	Output Graph of Serializer	28

3.4.1	Schematic of Inverter based Transmitter.	29
3.4.2	Test Circuit of Inverter based Transmitter.	29
3.4.3	Result of Inverter based Transmitter.	30
3.5.1	Schematic of Resistive Feedback Inverter	31
3.5.2	Result of Resistive Feedback Inverter	31
3.6.1	Schematic test circuit of D Flip Flop	32
3.6.2	Result of D Flip Flop	32
3.7	Result of Clock Data Recovery (CDR)	33
3.8	Result of Deserializer	33
3.9.1	Symbol of Receiver	34
3.9.2	Schematic of Receiver	34
3.9.3	Result of Receiver	35
4.1	Flow Graph of Design	39
5.1	Block Diagram of Data Memory	40
5.2	Block Diagram of Instruction Memory	41
5.3	Block Diagram of MultiCore Processor	42
5.5	Block Diagram of Control Unit	44

5.6	Block Diagram of Mux	45
6.1.1	ALU Simulation	46
6.1.2	Register Simulation	46
6.1.3	Inc Register Simulation	46
6.1.4	Control Unit Simulation	47
6.1.5	Receiver Simulation	47
6.1.6	In Memory System Content Editor usage	47
6.2.1	Flow Summary	48
6.2.2	Analysis & Synthesis Summary	48
6.2.3	Fitter Summary	49
6.2.4	Assembler Summary	49
6.2.5	Timing Analyzer Summary	50
6.2.6	EDA Netlist Writer Summary	50

CHAPTER 1: INTRODUCTION

1.1 MOTIVATION

While the number of transistors per square inch in current system on chips (SoCs) grows, more data is processed and transported from and onto chips.

The bandwidth is limited because the pin pitch and number of contacts do not scale at the same rate. To keep up with processing power, the number of pins per pin must be increased. High-speed serialization/deserialization is used to overcome the problem of pin restriction. (SerDes) technology is required to propel innovation forward.

A serializer converts parallel data to a single transmission channel with a higher clock rate. On the receiving side, the serial data is sampled and deserialized to a parallel data channel with a lower clock speed. The core clock of an application-specific integrated circuit (ASIC) Extremely high frequencies of 2 GHz and serialisation factors of usually 16. In these SerDes macros, high-speed circuits are required. 25 Gbps line rates more, where a single bit duration is just approximately 40ps, need enormous efforts to mitigate the consequences of lossy transmission lines.

High-speed serial connections have become an increasingly significant component of today's systems on chips due to their growing relevance for the whole system. Complex protocols, such as PCI Express (PCIe), need a significant amount of interaction between the SerDes macro, which is often implemented as an analogue or mixed signal. The signal block is entirely customised, and the digital logic implementing the real protocol is synthesised from a hardware description language (HDL) using highly automated electronic design automation (EDA) tools.

Different design and verification methodologies used in complete bespoke analogue and partially custom digital design throughout the years might cause major integration issues during SoC implementation. As a result, deadlines are missed, incorrect designs are submitted, and costly chip re-spins occur. While event driven simulators may simulate and verify A whole custom block like a SerDes has a schematic representation and is normally simulated using circuit simulators like SPICE, synthesizable logic expressed in a hardware description language, a whole custom block like a SerDes has a schematic representation and is normally simulated using circuit simulators like SPICE are used because take orders of magnitude longer to compute than event-driven HDL simulations, abstract models for all custom blocks must be constructed in order to simulate the entire SoC. Because SPICE simulation times are orders of magnitude longer than event-driven HDL simulation timings, abstract models for all custom blocks must be constructed in order to simulate the entire SoC in a reasonable amount of time. A fundamental part of the entire SoC verification process is the production of rapid, accurate models that are kept consistent throughout the design cycle.

1.2 OVERVIEW

Current approaches for partial and complete bespoke designs are introduced in the next chapter. Following that, the design and verification process created throughout this thesis, as well as the tools that support it, are provided. The third chapter begins with an overview of contemporary SerDes architectures. Afterwards During the course of this project, the SerDes was conceived and deployed. The thesis is explained as the clock data recovery (CDR) is given special attention in the phase locked loop (PLL) for central clock generation, and the SerDes receiver clocking architecture in general.

CHAPTER 2

ARCHITECTURE OF HIGH SPEED SERDES

2.1 DATA TRANSMISSION

The transfer of information between two chips on a similar board or inter circuit card is known as data transmission. Parallel and serial communication are the two most frequent techniques of knowledge transmission nowadays. Serial communication has been a favoured approach over parallel communication as technology has improved and data rates have climbed to the multi-gigabit range, due to fewer pins, smaller size, and lower power consumption..

2.1.1 Parallel Transmission

Numerous data lines can convey multiple data bits at the same time in parallel communication. The data from the transmitting port is transmitted to the receiving port that corresponds. Parallel communication is simple to set up, but it takes up a lot of space Data skew in parallel communication is caused by ISI (Inter-symbol Interference), noise, and output impedance mismatch. Because of these factors, the delivery time of various data might vary in accordance with specifications (PCI-e, SATA, USB, TBT, DP, HDMI, and M-PHY). Cross talk occurs when the conductors and impedance of one channel cause an unwanted influence on another channel.

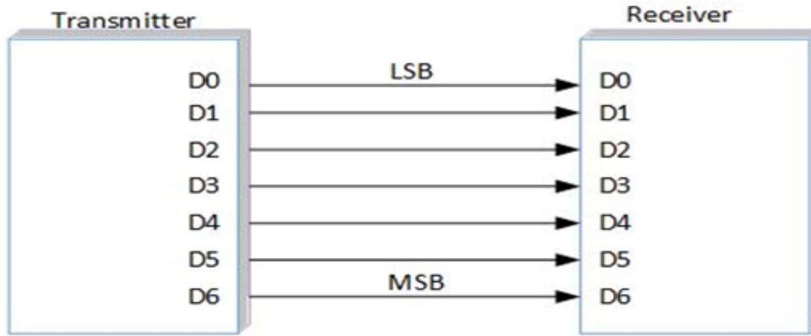


Fig 2.1.1 Parallel Transmission

2.1.2 Serial Communication

Serial communication delivers data across a single channel one bit at a time. There are fewer sending and receiving ports required for serial transmission. Due to its advantages over parallel communication, serial communication is most commonly used in chip-to-chip data transmission. Due to the fact that serial communication requires fewer pins than parallel communication, the values are frequently quite low. Because there is only one channel for data transmission, there is no knowledge skew or cross talk.

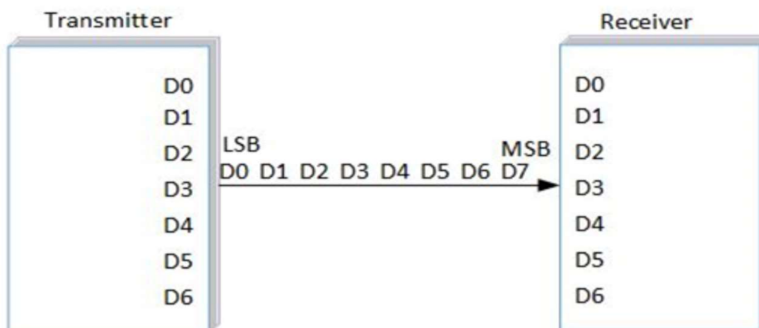


Fig 2.1.2 Serial Transmission

2.2 HIGH SPEED SERDES

SerDes (Serializer/Deserializer) is a device that allows you to transport data serially and is highly useful in high-speed applications.

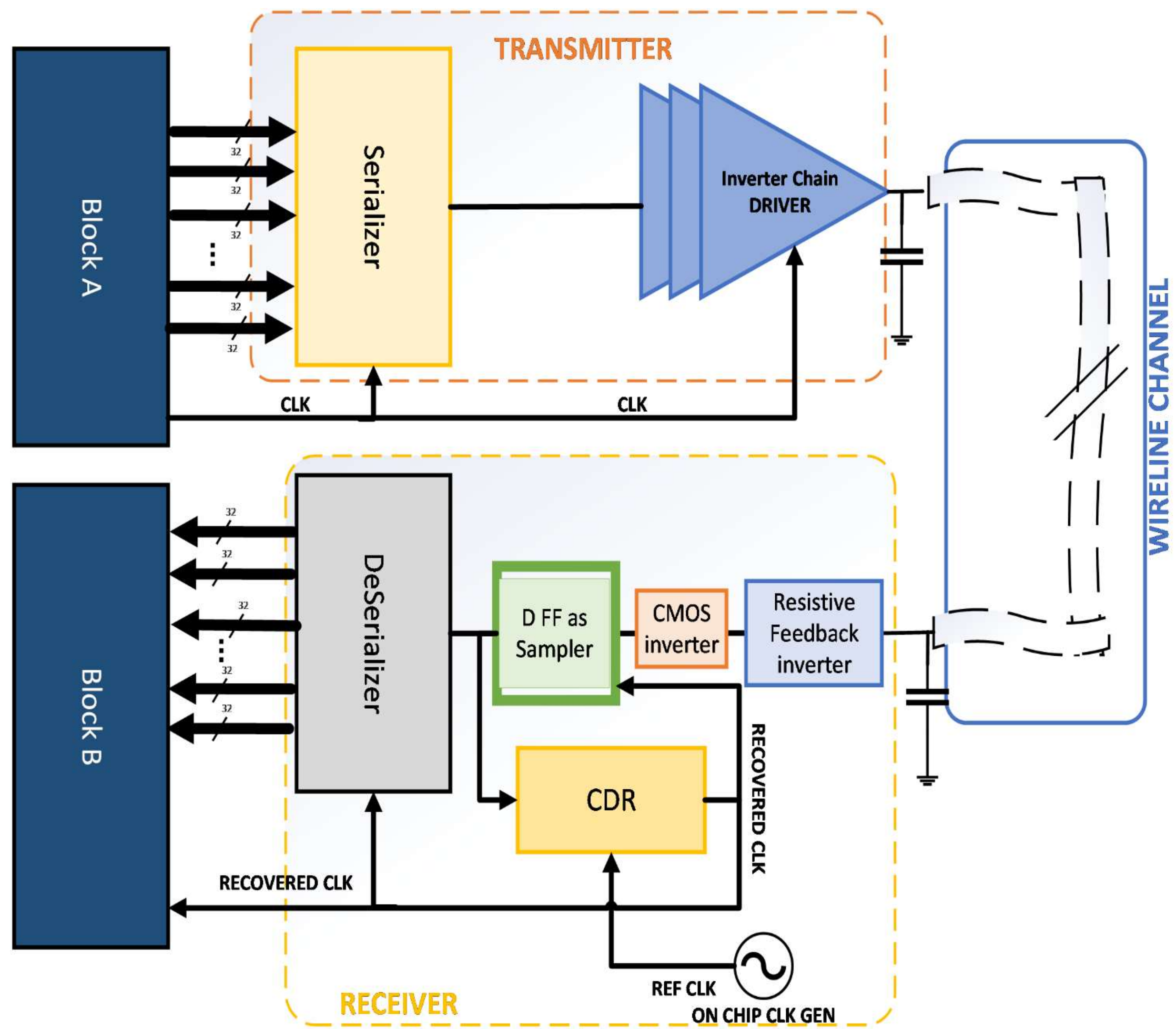


Fig 2.3 High Speed Serdes Block Diagram

2.3 TRANSMITTER

The transmitter is, a bit like the receiver divided into a full custom partition and a semi-custom implementation part as depicted in figure. The functionality implemented within the full custom partition is kept at a minimum in terms of complexity, so as to enhance portability and reduce manual implementation work.

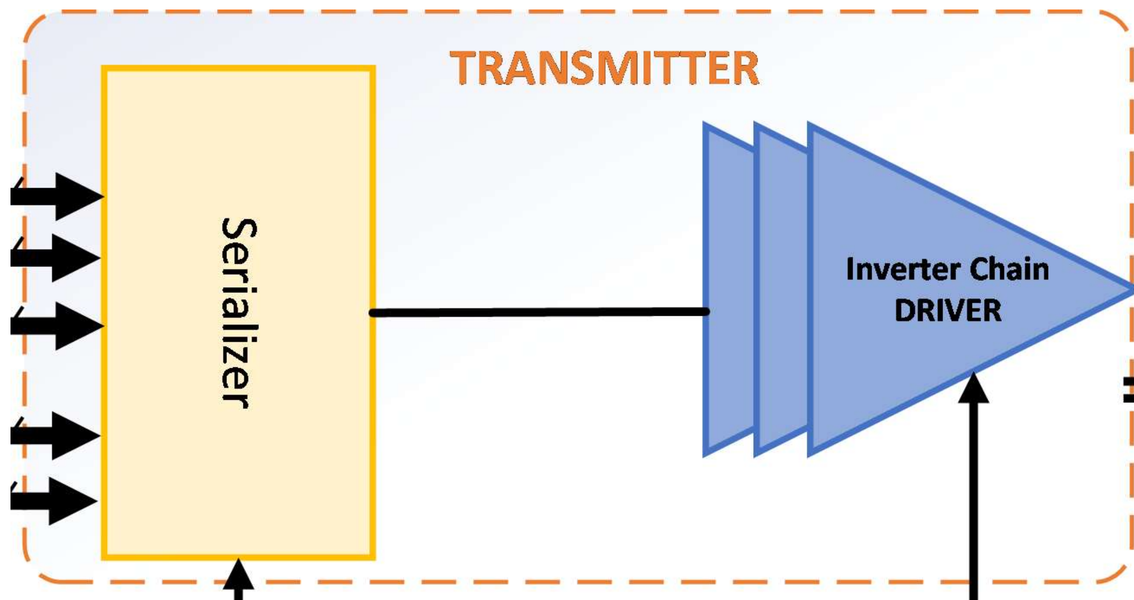


Fig 2.3 Transmitter Block Diagram

2.3.1 PHASE LOCKED LOOP

In a high-speed link system, a phase lock loop (PLL) serves as a timing or clock generator, generating high-frequency clock from a low-frequency reference clock. A PLL is a circuit that synchronises an output (produced by an oscillator) with a reference or input in frequency as well as phase. The synchronised state is commonly referred to as the locked state. The phase error between the oscillator output and the reference signal is zero or constant when the oscillator is locked. When the phase error rises, a control mechanism operates on the oscillator to lower the phase error to a bare minimum. The phase of the output is locked to the phase of

the reference signal as a result of such an impact mechanism. This is why it's commonly referred to as a phase-locked loop. Phase Detectors (PD), Charge Pump (CP), Loop Filter (LPF), and Voltage Controlled Oscillator (VCO) are the most common PLL components, as illustrated in the diagram.

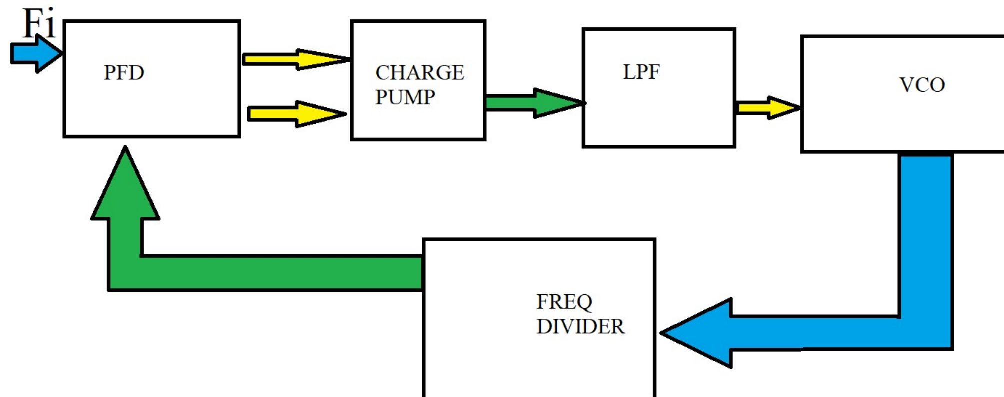


Fig. 2.3.1 PLL Block Diagram

2.3.2 DRIVER

The transmitter must create precise voltage swing on the channel in accordance with protocol specifications. A current or voltage mode driver is employed to achieve correct voltage swing and impedance matching. Currently selected mode to achieve a voltage swing of 500 mV, drivers generally guide current close to 20 mA. Driver resistor in parallel with the high-speed output impedance is used to match the output impedance. switch to current The Norton current mode driver is mostly employed in high voltage swings. equivalent parallel termination, making output impedance management simple. Whereas Thevenin-equivalent series termination is used by a voltage mode driver. Mode of voltage Because the driver uses less

current to achieve the same voltage swing, it uses less powerto current mode driver but it is difficult to generate impedance matching with voltage mode driver.

2.4 RECEIEVER

The fully customised section is broken down into various sections. The datapath starts with the analogue frontend and continues with all of the samplers. A clocking module also exists, which generates all sampler clock phases using the lane clock from the lane PLL, with the option to modify phases based on the CDR control vectors. Furthermore, the Rx core is where all entire custom blocks' central bias is generated, as well as the DFE's reference level creation. The digital toplevel processes all control and status signals that are delivered to the Rx core module border.

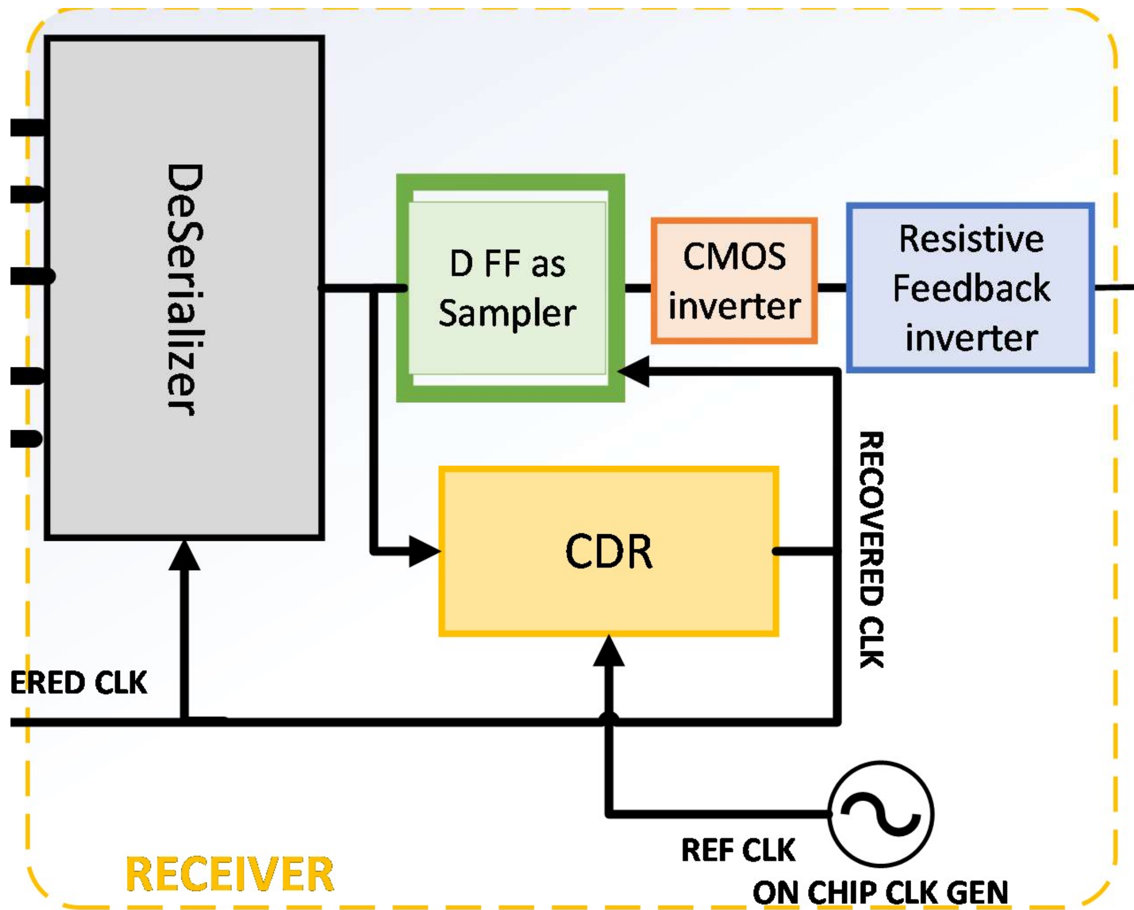


Fig. 2.4 Receiver Block Diagram

2.4.1 CLOCK AND DATA RECOVERY

The clock data recovery is a critical component of the receiver that influences overall performance. Its job is to extract phase information from the incoming data stream and modify the sampling phase of the receiver accordingly.

In fact, it is extremely similar to that of a PLL. A phase detector (PD) checks the incoming data phase with the current sample phase from a high-level perspective. To establish the CDR control loop parameters, the phase detector creates a control voltage proportionate to the phase difference, which is filltered by the loop fillter (LF). A voltage-controlled oscillator (VCO) that

integrates into the sampling clock phase uses the filltered control voltage to set the instantaneous frequency. Because no frequency multiplication occurs, no feedback divider is necessary, unlike with a PLL.

As with PLLs, this resulted in the development of digital clock data recovery circuits. Higher noise immunity and resilience against power supply variability arise from the digital approach. Because there are no passive components in the loop, it has excellent scaling properties when it comes to process shrinks. It also gives greater flexibility due to digitally configurable fillter coeffcients.

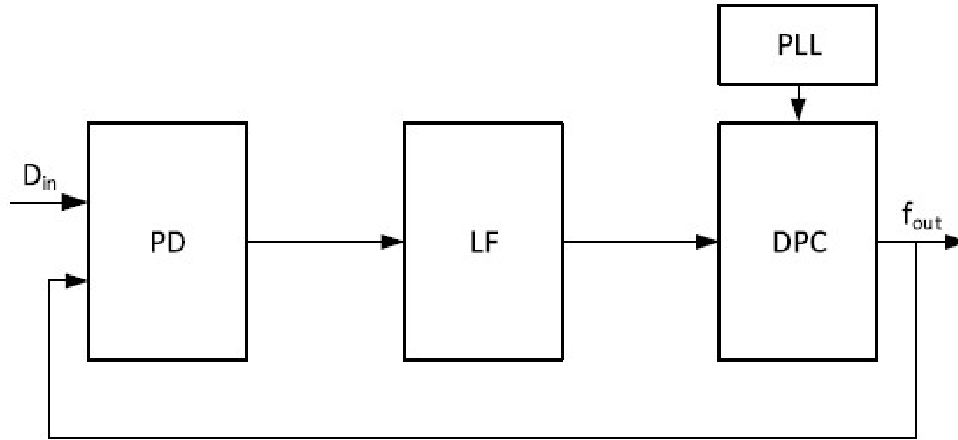


Fig 2.4.1 CDR Block Diagram

2.4.2 DESERIALIZER

The serial-to-parallel conversion is performed via the deserializer process, as indicated in the figure. The serial data will first be sent to two D ip operations, which will collect the data on both the rising and falling edges of the clock. We now have two distinct data streams, even and odd, but even data arrives first. As a result, retiming even data with odd data will be further delayed by the clock's dropping edge. So we have two data streams running in parallel.

CHAPTER 3: RESULTS

3.1 OVERVIEW

While the preceding chapters concentrated on the design methods and the SerDes architecture itself, this chapter shows how these concerns were put into practice.

The whole SerDes architecture was previously provided as a top-down paradigm. Verilog models are used in each cell. The schematics are designed in ModelSim and TSMC 130nm technology model libraries are used.

3.2 Phase Locked Loop (PLL)

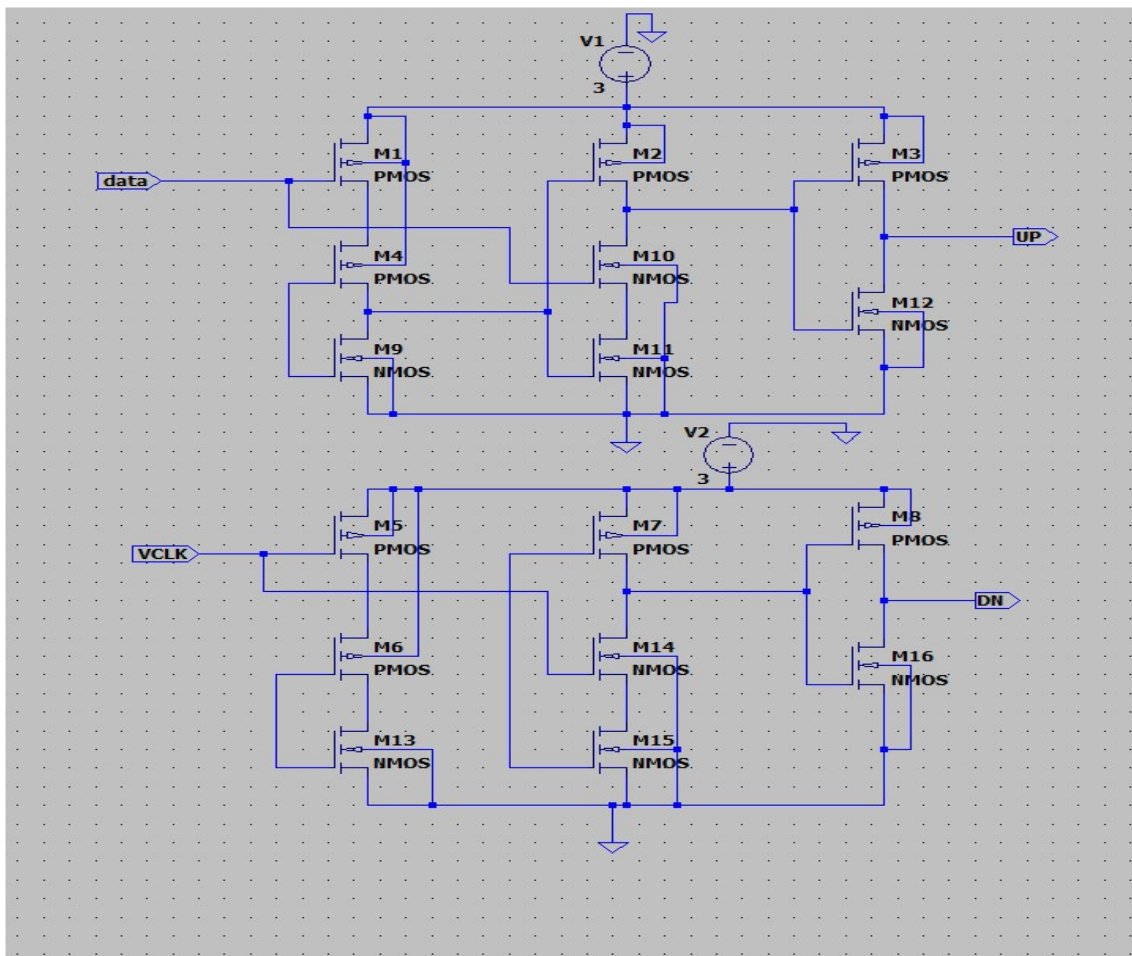


Fig 3.2.1 Schematic of Phase Frequency Detector (PFD)

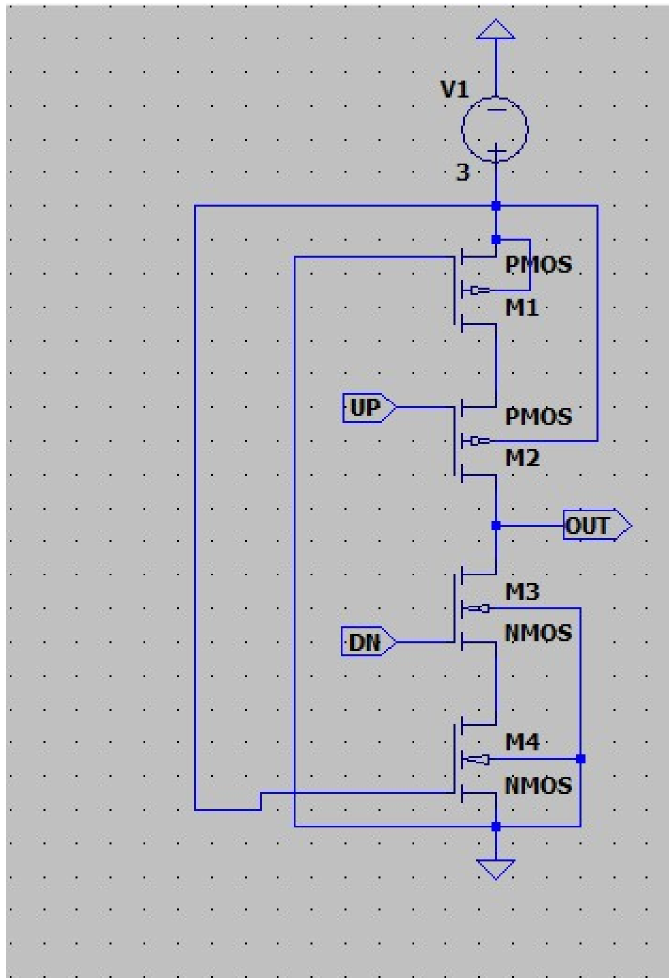


Fig 3.2.2 Schematic of Charge Pump.

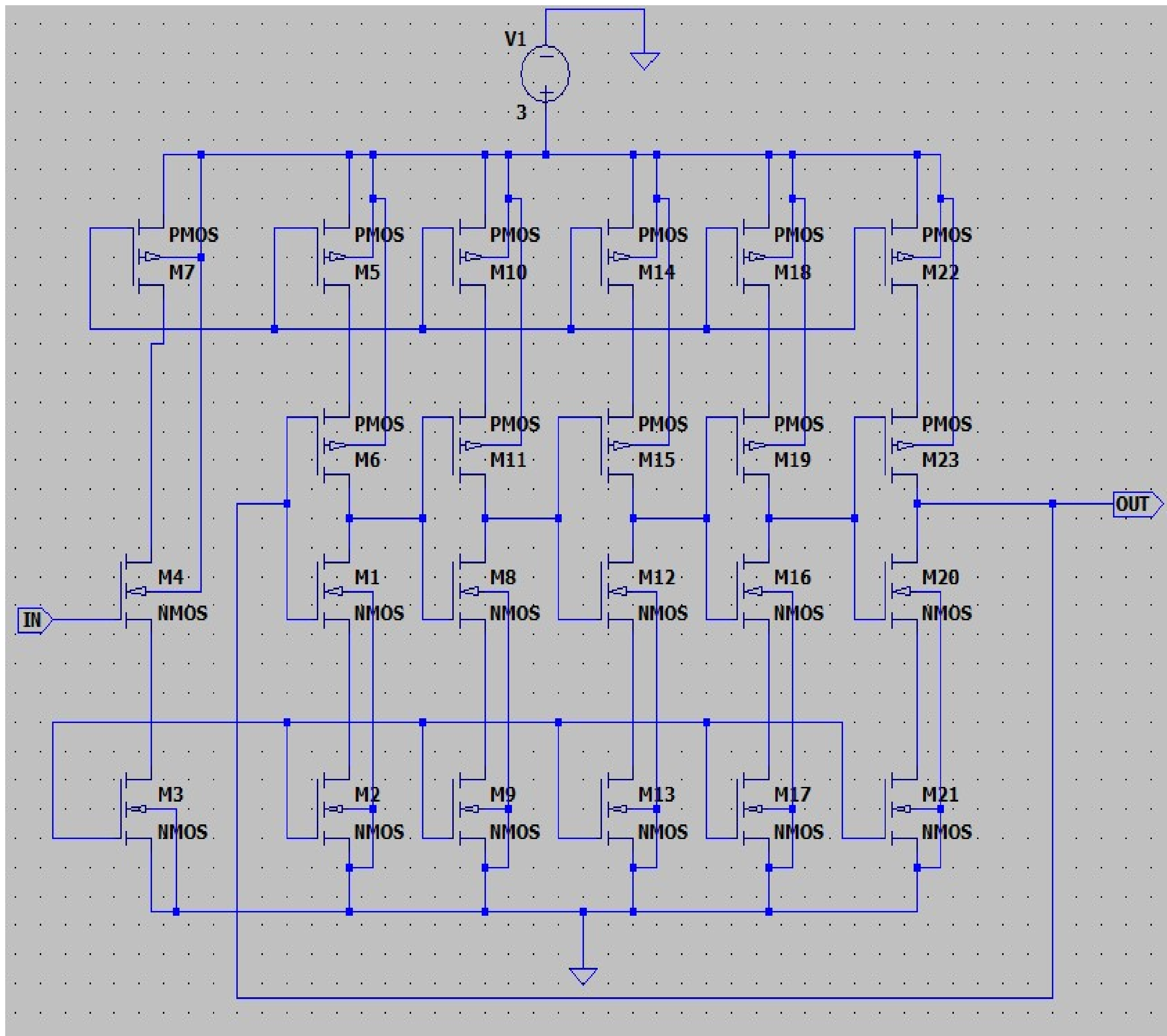


Fig 3.2.3 Schematic of Current Starved VCO.

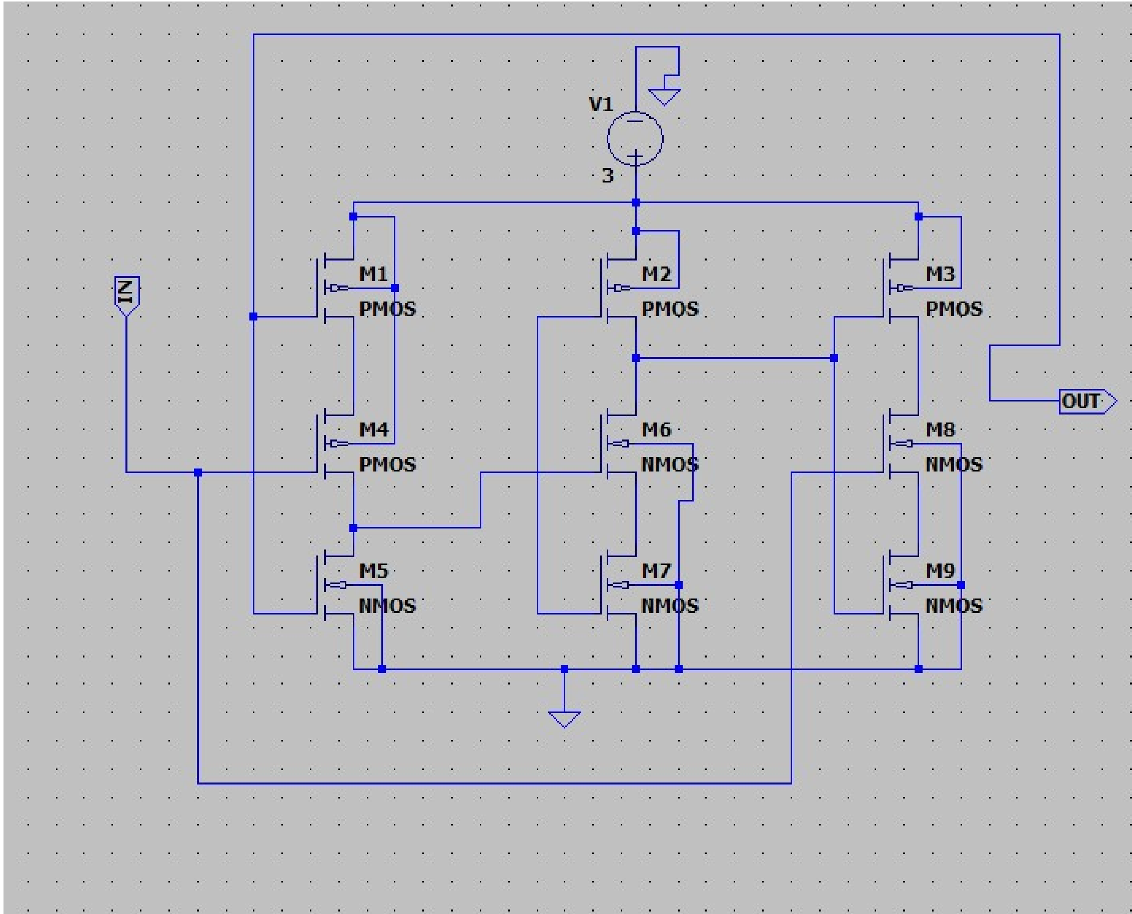


Fig 3.2.4 Schematic of Frequency Divider.

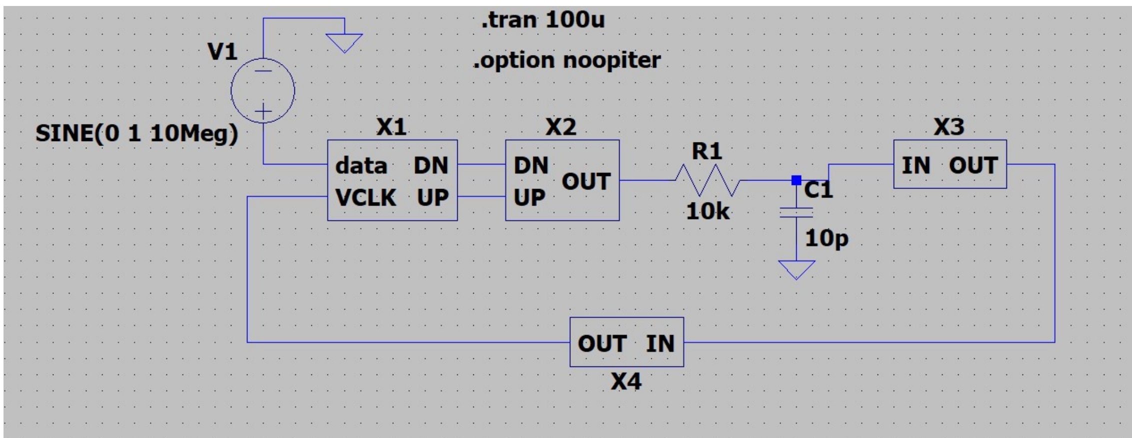


Fig 3.2.5 Schematic of PLL.

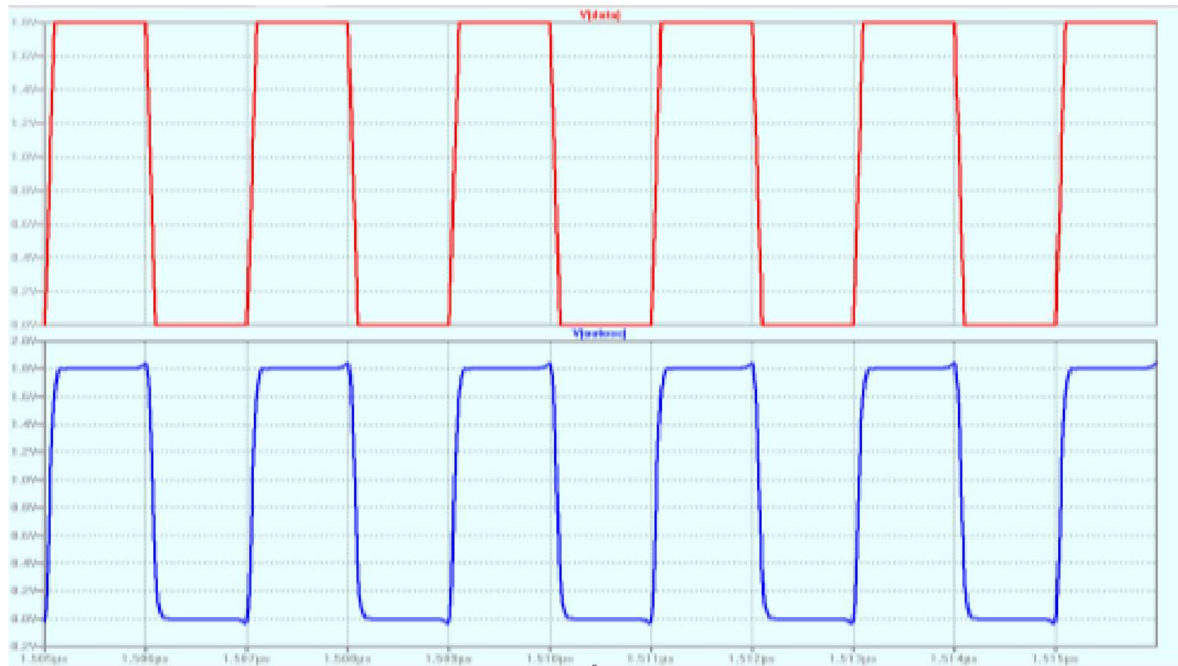


Fig 3.2.6 Output Graph of PLL

3.3 Serializer

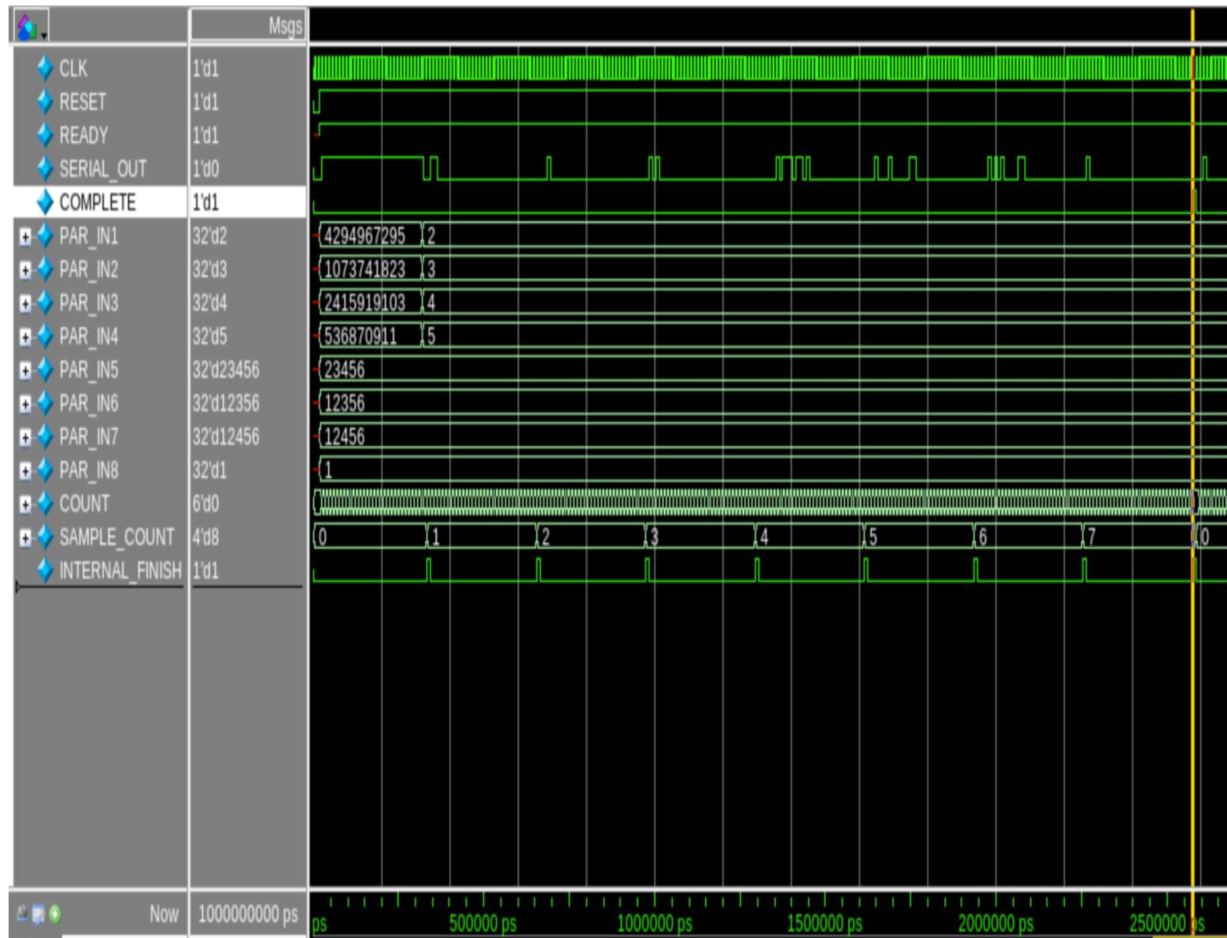


Fig 3.3 Output Graph of Serializer

3.4 Inverter based Transmitter

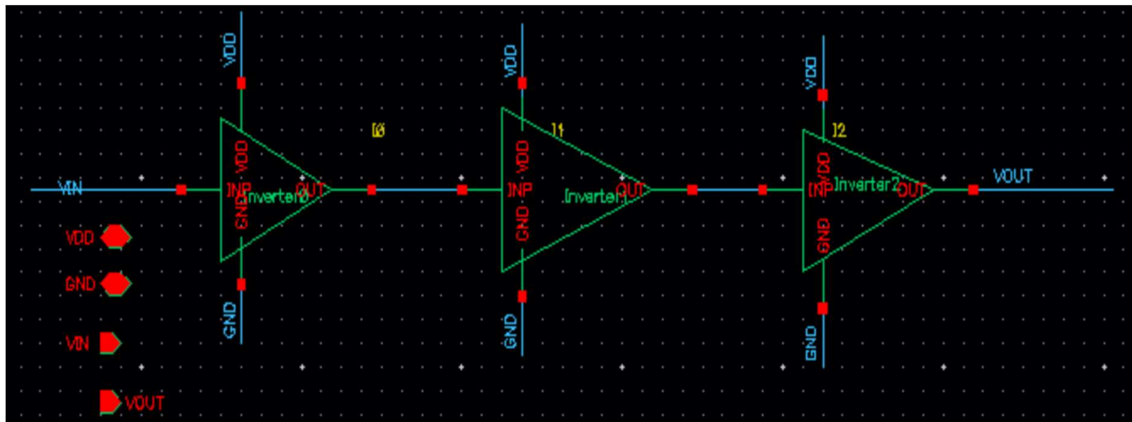


Fig 3.4.1 Schematic of Inverter based Transmitter.

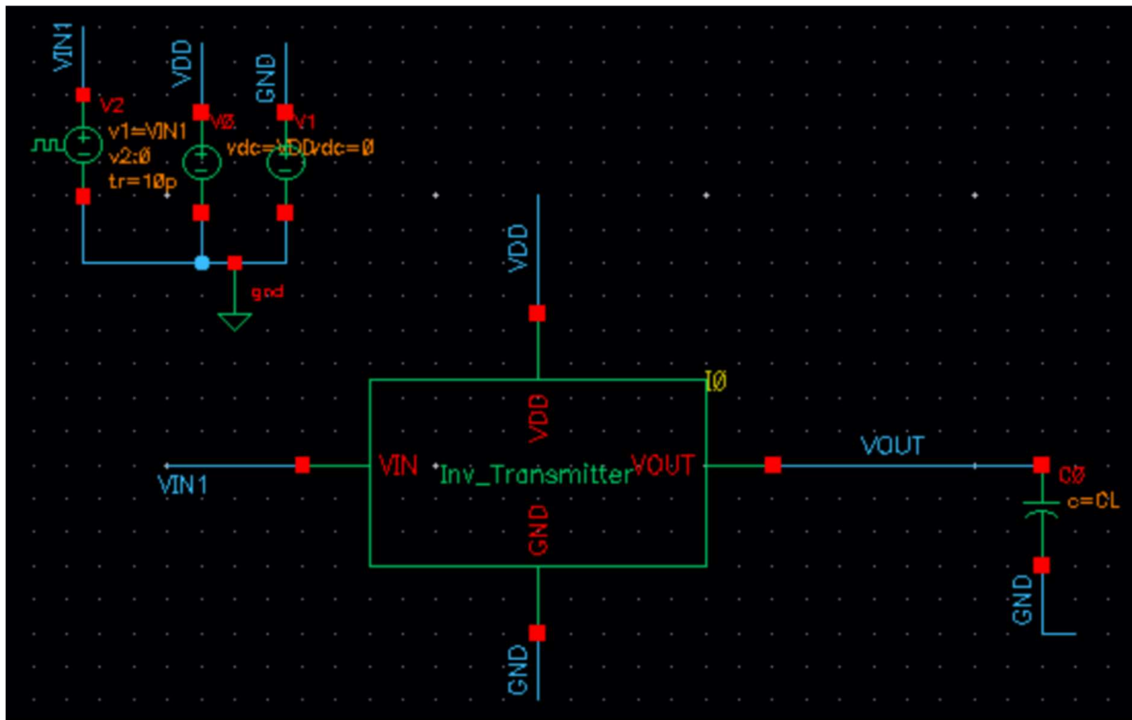


Fig 3.4.2 Test Circuit of Inverter based Transmitter.

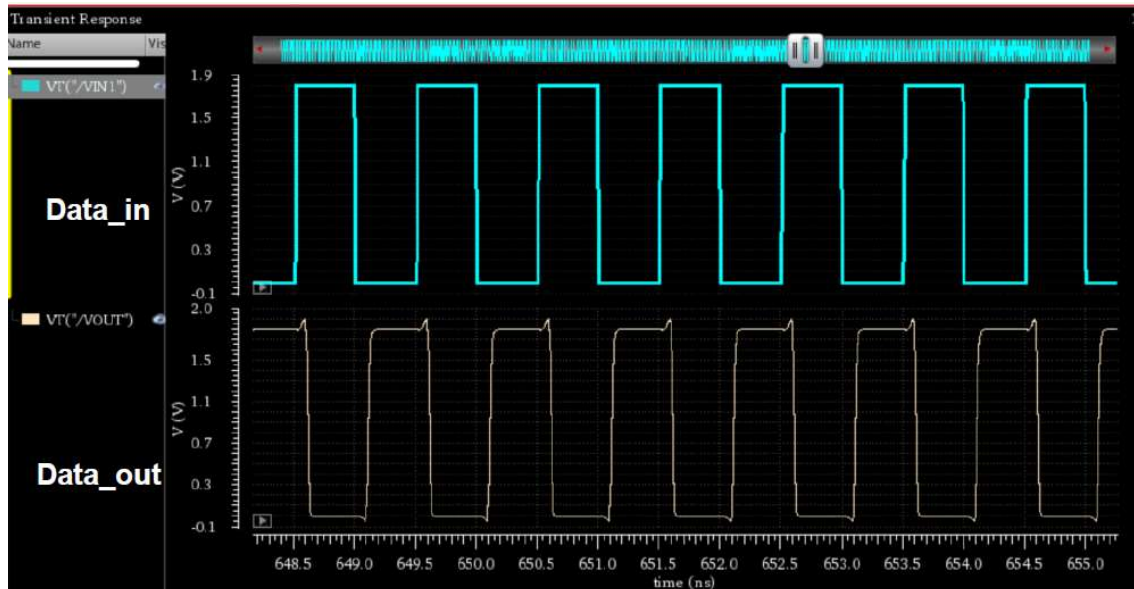


Fig 3.4.3 Result of Inverter based Transmitter.

3.5 Resistive Feedback Inverter

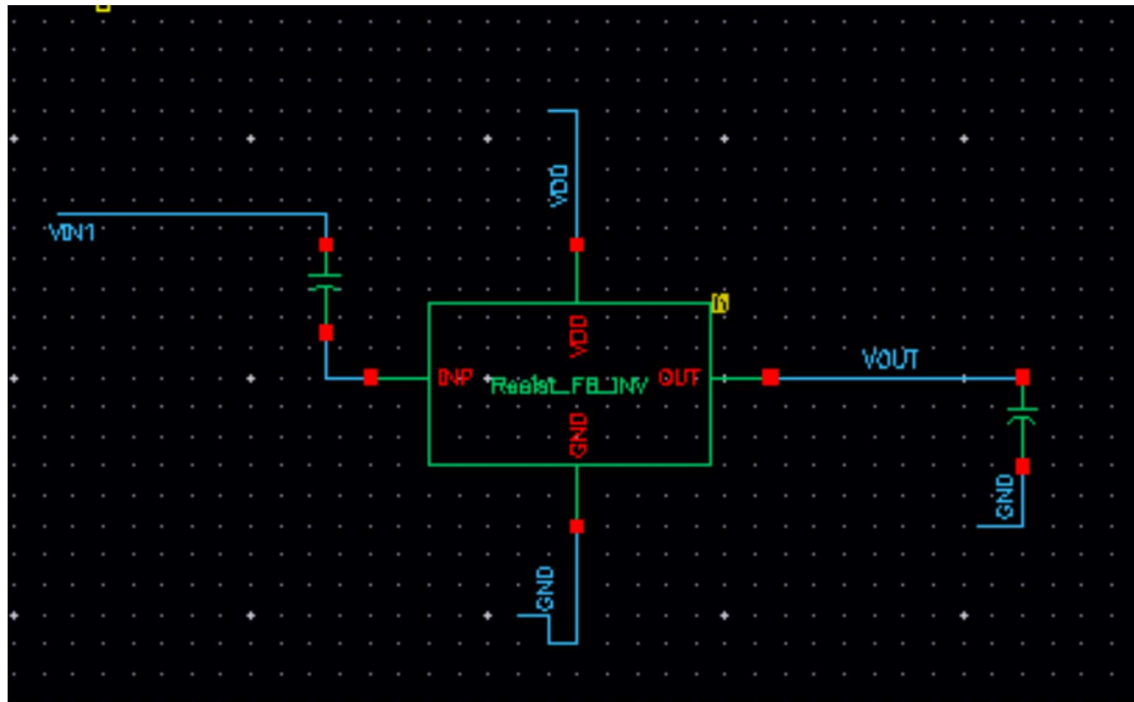


Fig 3.5.1 Schematic of Resistive Feedback Inverter

Fig 3.5.2 Result of Resistive Feedback Inverter

3.6 D Flip Flop

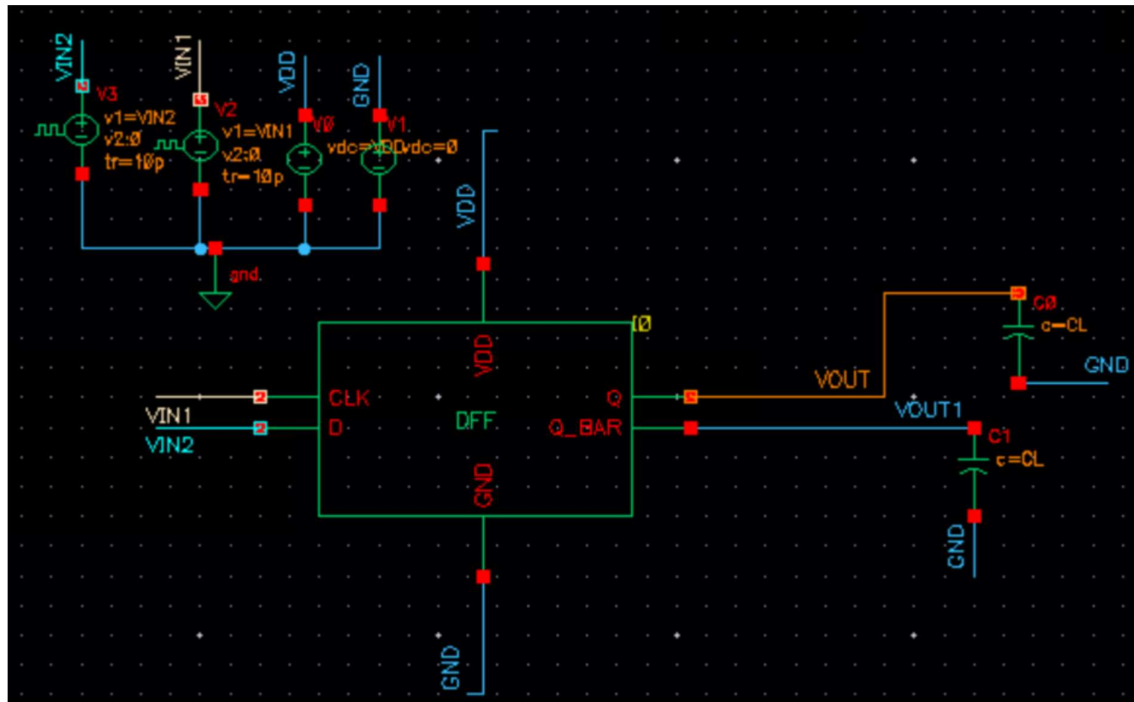


Fig 3.6.1 Schematic test circuit

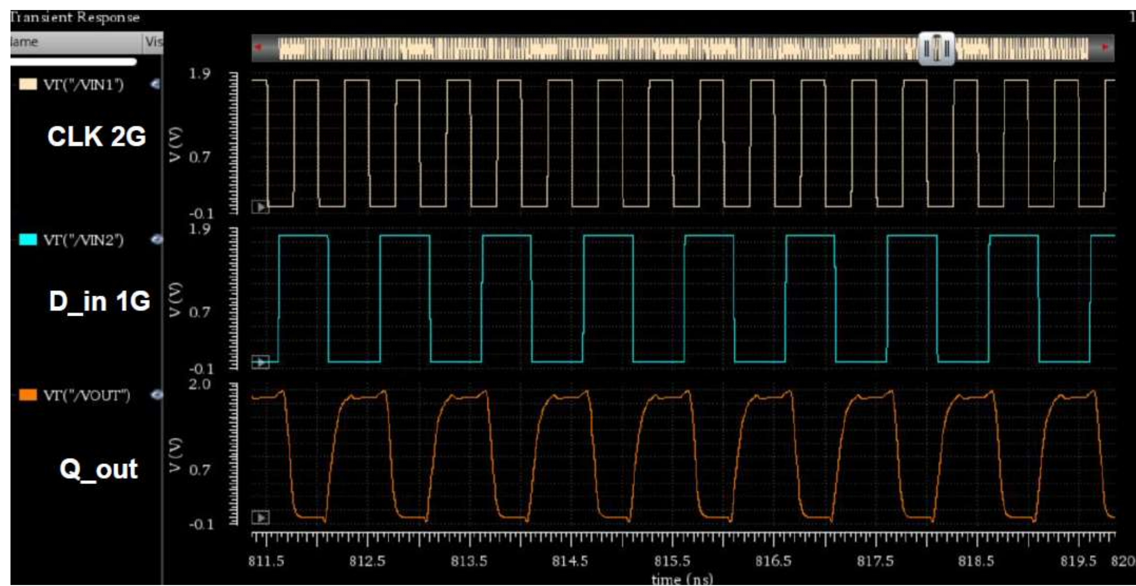


Fig 3.6.2 Result

3.7 Clock Data Recovery (CDR)

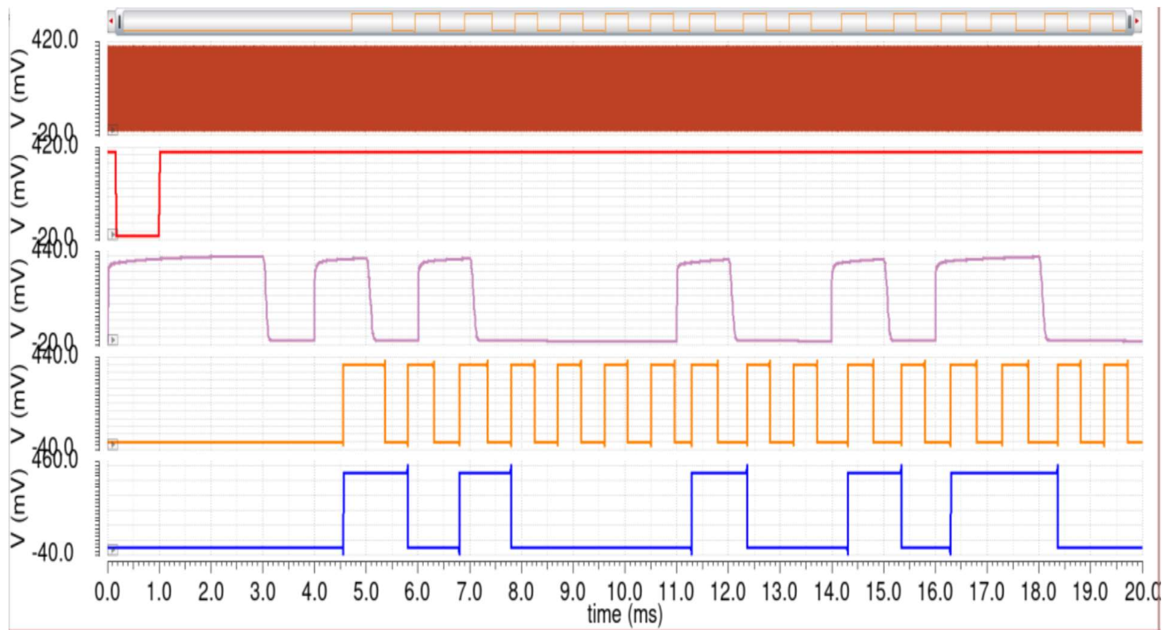


Fig 3.7 Result of Clock Data Recovery (CDR)

3.8 Deserializer

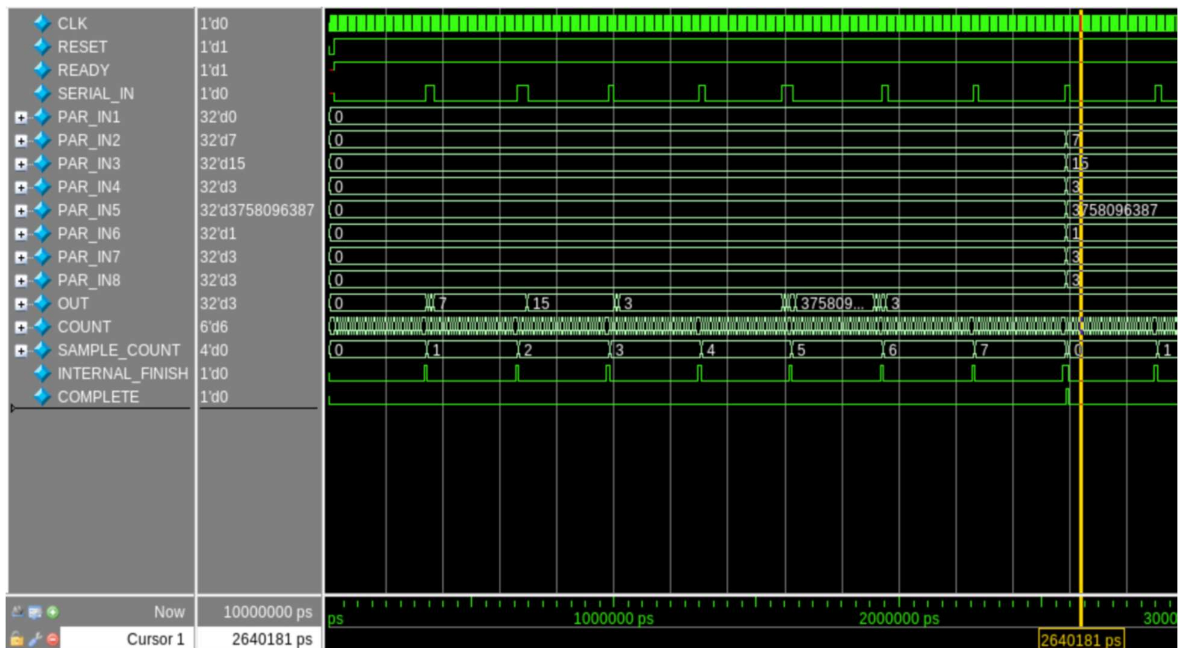


Fig 3.8 Result of Deserializer

3.9 Receiver

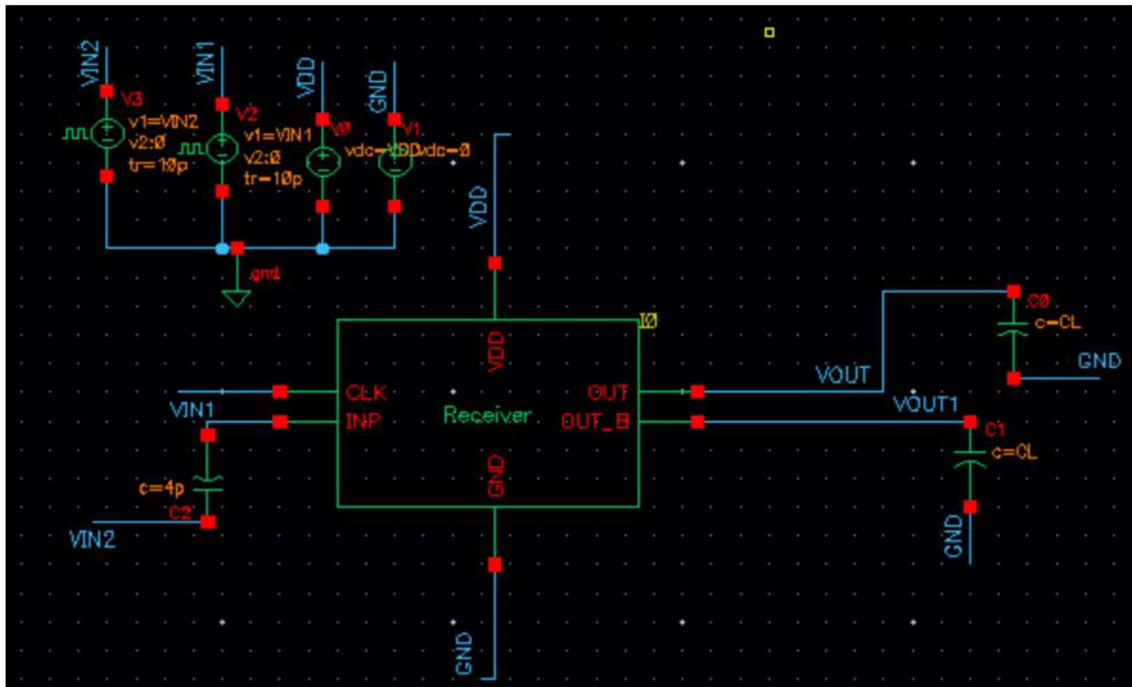


Fig 3.9.1 Symbol of Receiver

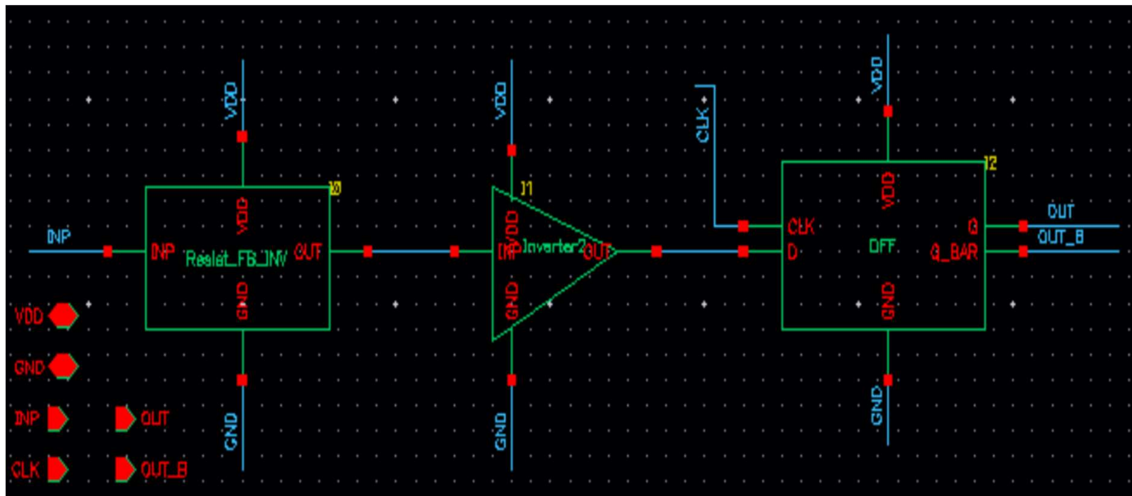


Fig 3.9.2 Schematic

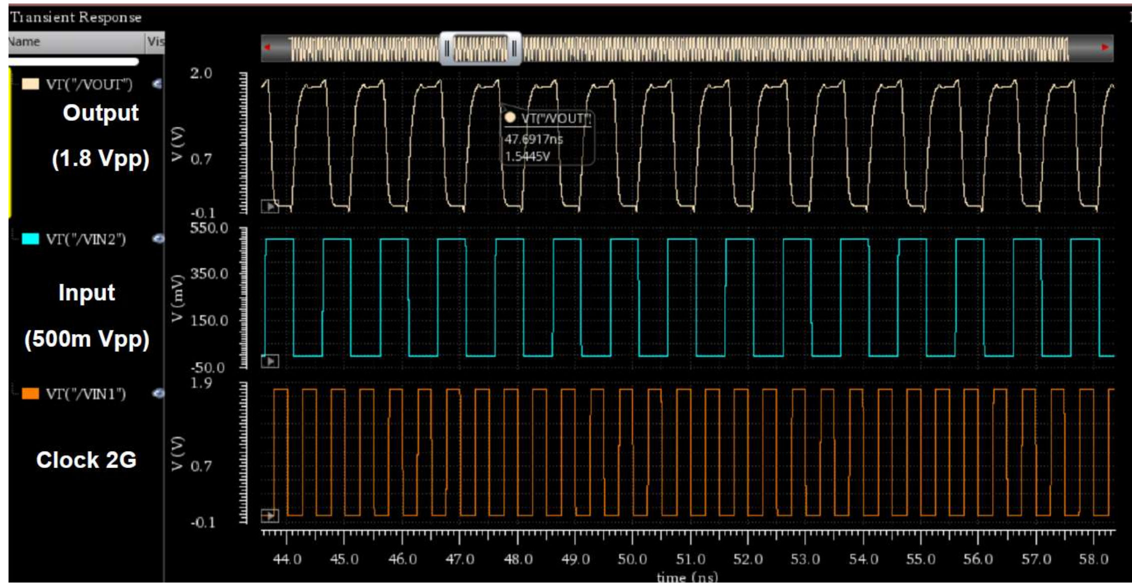


Fig 3.9.3 Result of Receiver

3.10 Conclusion

Achieved decoding of 500mV signal at the input of Receiver

CHAPTER 4

MULTICORE PROCESSOR DESIGN

4.1 INTRODUCTION

The objective of this project is to design a Multi-Core Processor which is optimized for matrix multiplication.

I have implemented the design using Verilog HDL. For the simulation I have designed test benches for each module. Quartus Prime 18.1 Lite edition with Cyclone IV softcore processor is used for the implementations.

4.1.1 SINGLE-CORE PROCESSOR

The Processing unit is the electronic circuitry within a computer that executes instructions that set up a computer program. The processor performs basic arithmetic, logic, controlling and input/output (I/O) operations determined by the instructions of the program. This consists of a control unit to control every single task within the processor, an Arithmetic and Logic Unit (ALU) to do arithmetic and logical operations and registers to keep data temporally. The processor needs external components such as data memory, instruction memory for data storage and I/O circuitry for user interface.

4.1.2 MULTI-CORE PROCESSOR

A multi-core processor is a processor which contains more than one processor core which have similar capabilities and characteristics as mentioned in the previous topic. Each core has its own control unit, registers and ALU. Therefore, they can work independently. Time consumption for the process can be drastically reduced by dividing the process among each core in an intelligent manner. Single Instruction - Multiple Data (SIMD) is a one method which can be used to divide a large process into similar smaller sub-processes and use each individual core for one sub-process. In this document we have shown how to do a matrix multiplication process using SIMD method, with the help of a multi-core processor.

4.1.3 DESIGN QUESTION

The offered project requires you to develop a single instruction multiple data multi-core processor capable of multiplying two matrices. There are no restrictions or limitations, such as matrices' form, size, signed or unsigned, or variable types such as floating-point integers.

4.1.4 PROPOSED SOLUTION

The primary goal is to create a multi-core processor that is optimised for matrix multiplication. The assignment may be broken down into the following sections.

1. Make random matrices with the appropriate dimensions.
2. Calculate the number of cores required to convert matrices into binary data streams.
3. Convert assembly code instructions into a binary data stream.
4. From the PC, send the instructions to the FPGA board's Instruction memory module.

5. Transfer data from the PC to the data memory module FPGA board (input matrices and initialization data).

6. Multiply two matrices in the FPGA using the defined multi-core processor and save the result matrix in the data memory.

The proposed solution is implemented on an ALTERA DE2-115 FPGA board which has a EP4CE115F29C7 Cyclone IV E soft-core processor. The designing process can be described as.

4.1.5 PROCESSOR DESIGN FLOW

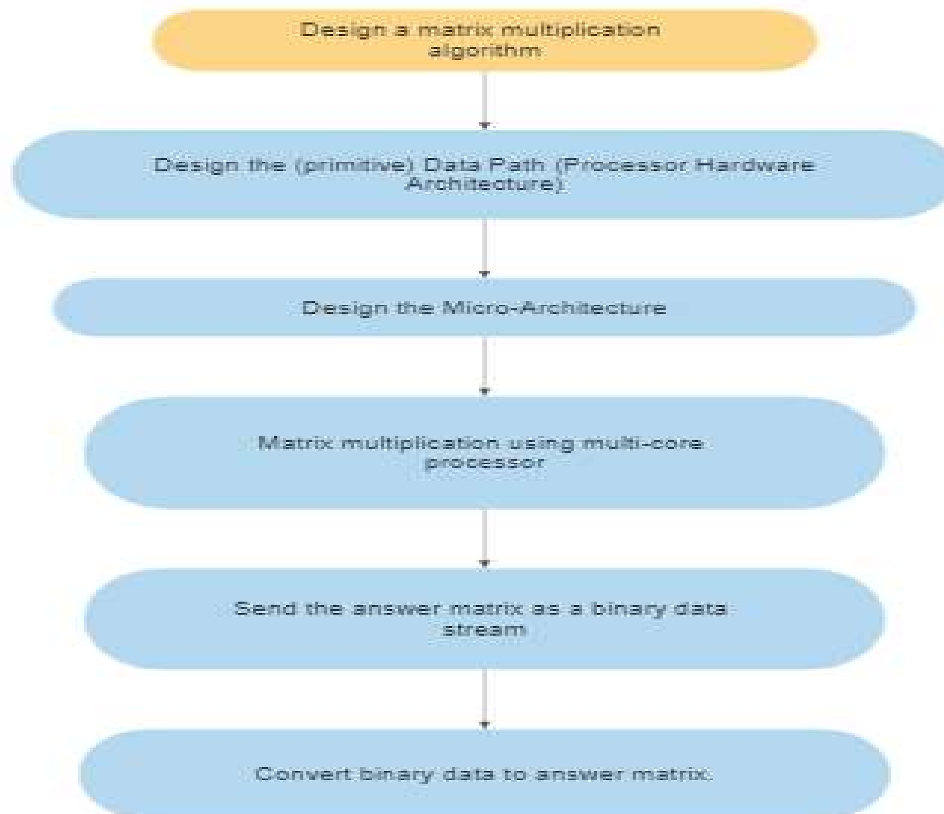


Fig 4.1 Flow Graph of Design

CHAPTER 5 MODULES

5.1 DATA MEMORY

This module contains the primary memory for storing data for processing. The data after reading data from UART, it is transmitted to data memory for storage. Module for storing data contains 4096 places with a 12 bit width. There are four inputs and one output on this module. Clock, write enable, a 12-bit address, and a 12-bit data input are the inputs. The result is a data output of 12 bits (q). Data input and output over 12 bits are limited to a single core.

When the number of cores is increased, the bit length is proportionally increased.

As described in the third way, count the cores. For instance, data width

When the core count is two, the data width becomes 24 bits, and when the core count is three, the data width becomes 36 bits is 3.

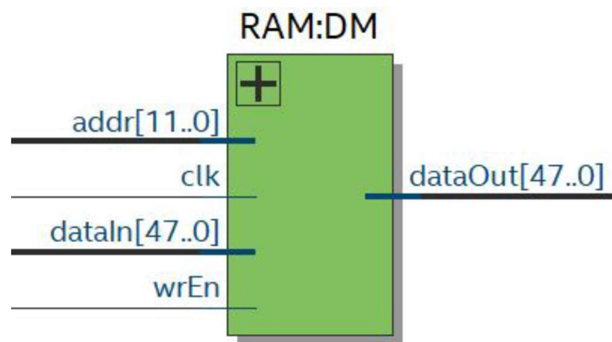


Fig 5.1 Block Diagram of Data Memory

5.2 INSTRUCTION MEMORY

This is where instructions for the procedure are stored. In the instruction memory, all instructions written in assembly language are saved as machine code. When the CPU wants to execute instructions, it pulls them from this module.

This memory holds instructions in the sequence that the assembly code specifies. This module has 256 locations with an 8-bit width. There are four inputs to this. Clock, wrEn, 8-bit address, and 8-bit data are the four variables. The output of this module is 8 bits (q).

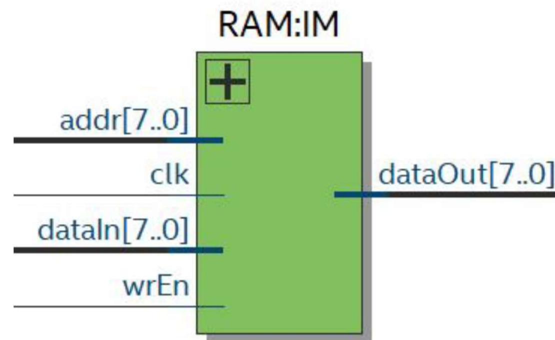


Fig 5.3 Block Diagram of Instruction Memory

5.3 MULTI CORE PROCESSOR

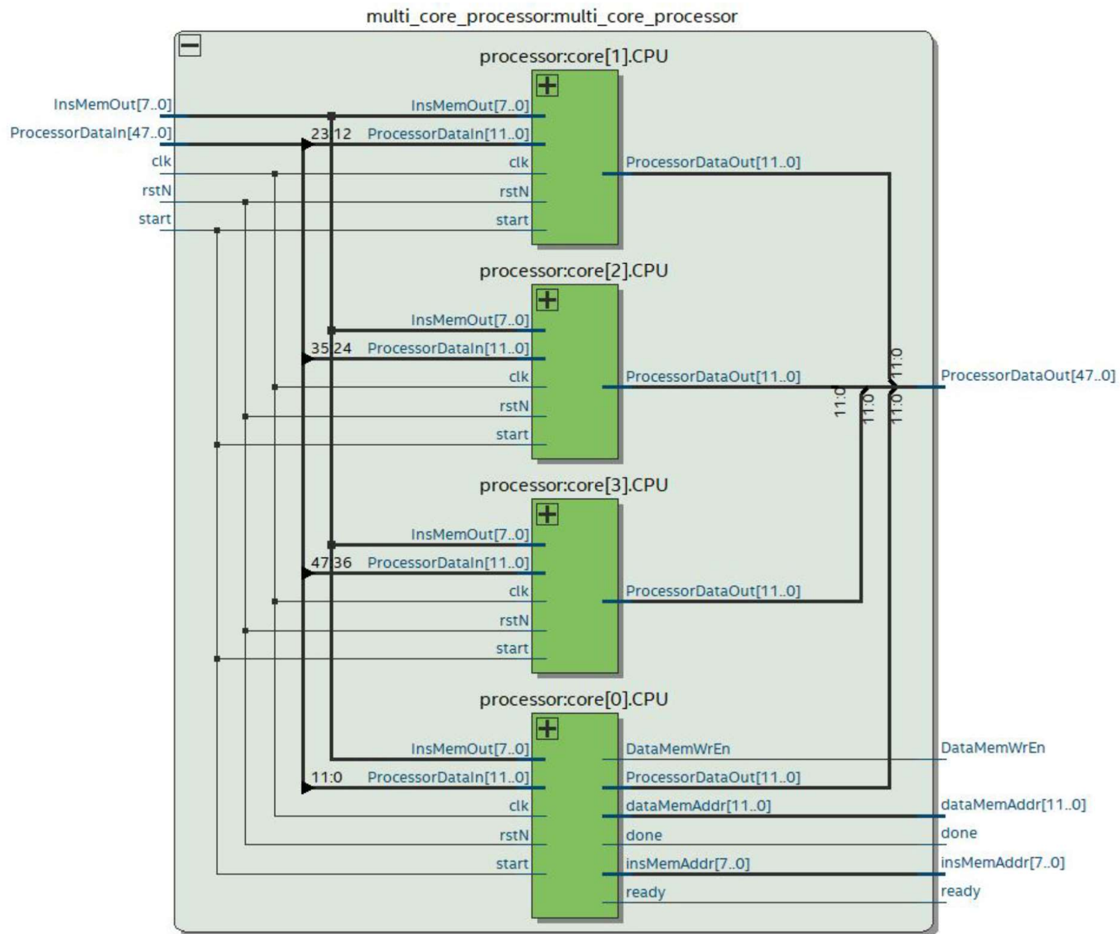


Fig 5.3 Block Diagram of MultiCore Processor

5.4 PROCESSOR

This is the module that holds all the instances of other modules that are used to regulate processing.

There are 5 inputs of the processor

1.clk: gives clock pulse.

2.fromdatamem: gives data from data memory to processor.

3.fromInsmem: gives instructions from the instruction-memory to the processor.

4.rst: resets processor to initial state.

5.start: gives command to start process.

It generates 6 outputs

1. dMemWrEn: This command allows you to write to the data memory.

2. dataMemAddr: This specifies the address of a data-memory location. It's a 12-bit image.

3. insMemAddr: This specifies the location of instruction memory in memory. This is an 8-bit image.

4. toDataMem: This specifies the data value to be written to the data memory. This has a 12-bit width.

5. ready: indicates that the processor is ready to begin the procedure.

6. done: This signifies the completion of a task.

5.5 CONTROL UNIT

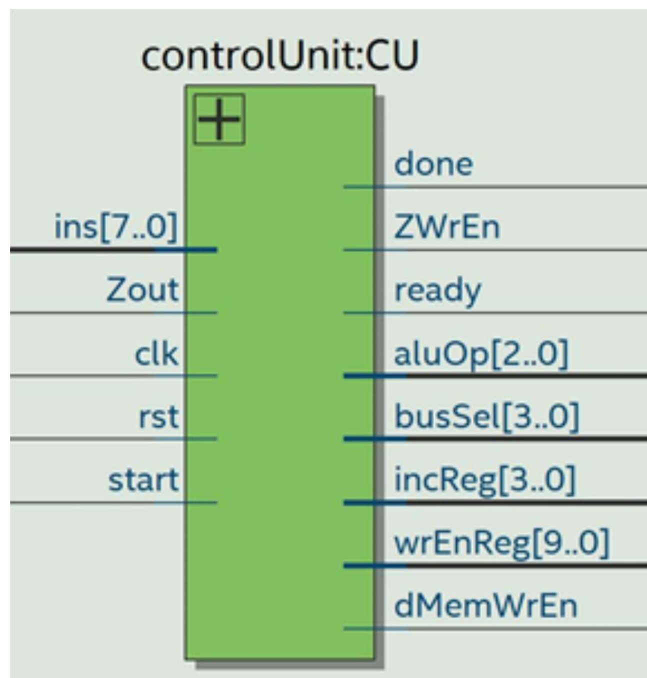


Fig 5.5 Block Diagram of Control Unit

5.6 MULTIPLEXER

The DATA-BUS is a series of 12 wires that transfer data between CPU registers. The bus is coupled to the inputs and outputs of registers such as R, RL, RC, RP, RQ, R1 and AC in the design. As a result, the bus can read and write data to those registers. This data-bus also connects to Data Memory.

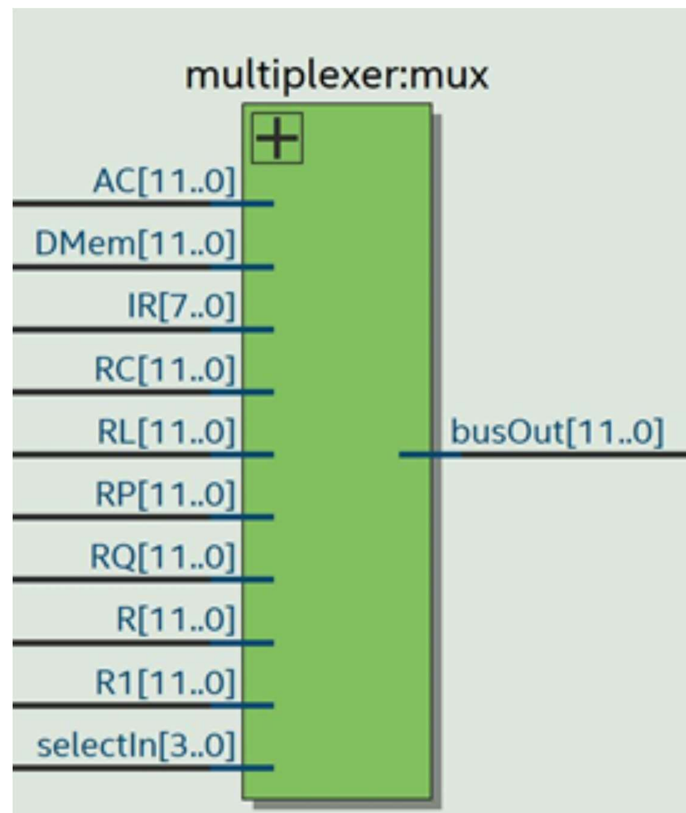


Fig 5.6 Block Diagram of Mux

CHAPTER 6: SIMULATION & SUMMARY

6.1 SIMULATIONS

Simulation was done in questasim 10.6 software.

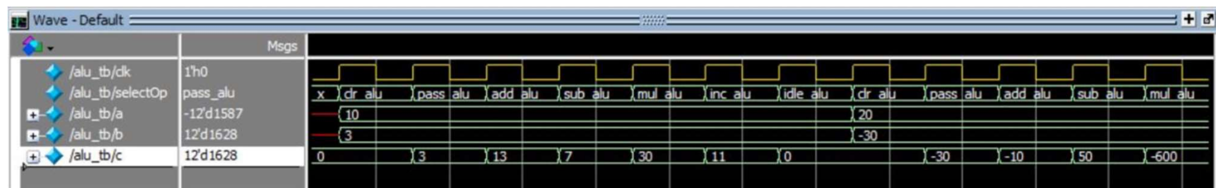


Fig 6.1.1 ALU Simulation

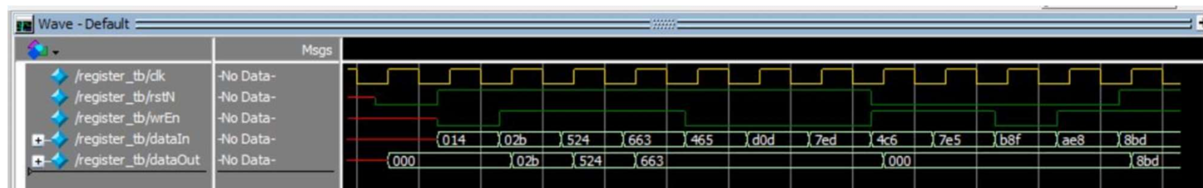


Fig 6.1.2 Register Simulation

Fig 6.1.3 Inc Register Simulation

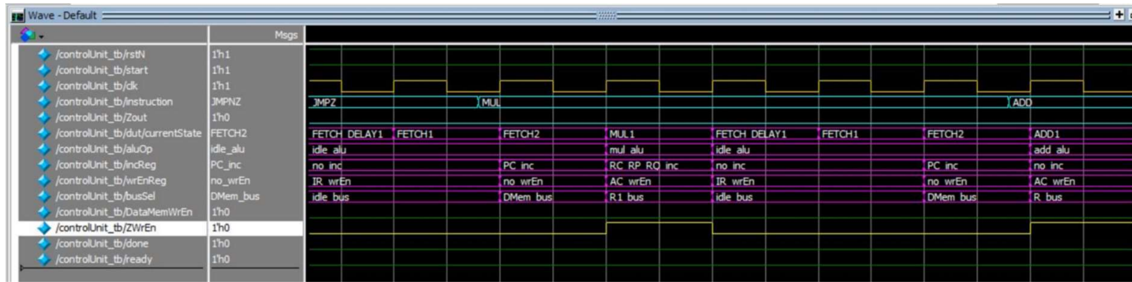


Fig 6.1.4 Control Unit Simulation

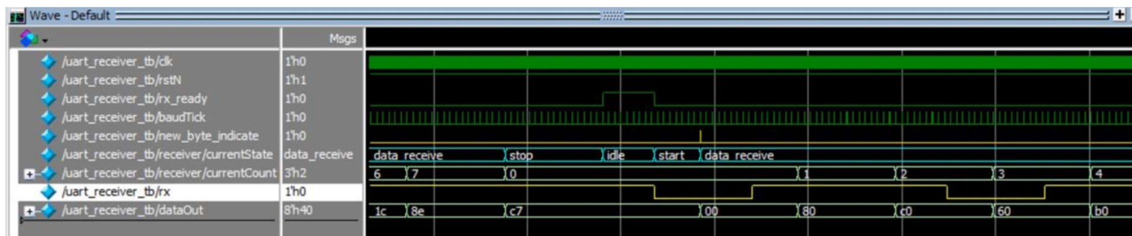


Fig 6.1.5 Receiver Simulation

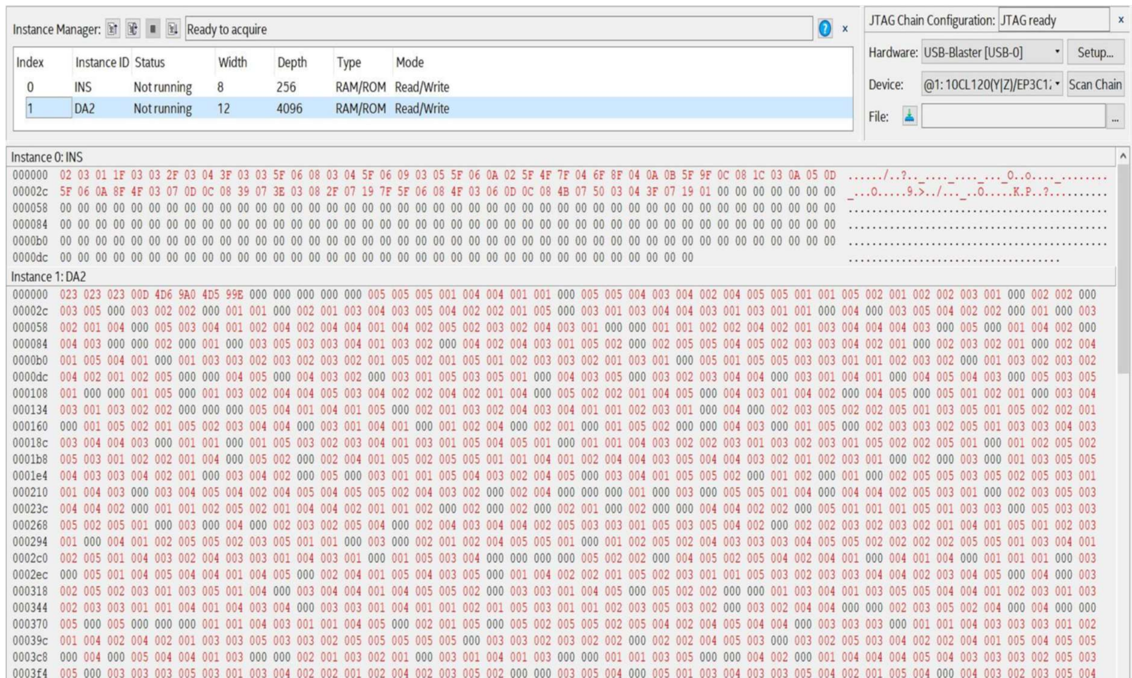


Fig 6.1.6 In Memory System Content Editor usage

6.2 DESIGN SUMMARY

Fig 6.2.1 Flow Summary

toFpga.v | Compilation Report - matrix_multiply

Table of Contents
- Flow Summary
- Flow Settings
- Flow Non-Default Global Setti
- Flow Elapsed Time
- Flow OS Summary
- Flow Log
- Analysis & Synthesis
- Fitter
- Assembler
- Timing Analyzer
- EDA Netlist Writer
- Flow Messages
- Flow Suppressed Messages

Flow Summary

Flow Summary	
Flow Status	Successful - Mon Dec 27 00:26:25 2021
Quartus Prime Version	21.1.0 Build 842 10/21/2021 SJ Lite Edition
Revision Name	matrix_multiply
Top-level Entity Name	toFpga
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	1,065 / 114,480 (< 1 %)
Total registers	426
Total pins	63 / 529 (12 %)
Total virtual pins	0
Total memory bits	51,200 / 3,981,312 (1 %)
Embedded Multiplier 9-bit elements	2 / 532 (< 1 %)
Total PLLs	0 / 4 (0 %)

Fig 6.2.2 Analysis & Synthesis Summary

toFpga.v | Compilation Report - matrix_multiply

Table of Contents
- Flow Summary
- Flow Settings
- Flow Non-Default Global Se
- Flow Elapsed Time
- Flow OS Summary
- Flow Log
- Analysis & Synthesis
  - Summary
  - Settings
  - Parallel Compilation
  - Source Files Read
  - Resource Usage Summa
  - Resource Utilization by
  - RAM Summary
  - DSP Block Usage Summ
  - State Machines
  - Optimization Results
  - Source Assignments
  - Parameter Settings by E
  - LPM Parameter Settings
  - Connectivity Checks

Analysis & Synthesis Summary

Analysis & Synthesis Summary	
Analysis & Synthesis Status	Successful - Mon Dec 27 00:25:49 2021
Quartus Prime Version	21.1.0 Build 842 10/21/2021 SJ Lite Edition
Revision Name	matrix_multiply
Top-level Entity Name	toFpga
Family	Cyclone IV E
Total logic elements	1,120
Total registers	426
Total pins	63
Total virtual pins	0
Total memory bits	51,200
Embedded Multiplier 9-bit elements	2
Total PLLs	0

Fig 6.2.3 Fitter Summary

toFpga.v | Compilation Report - matrix_multiply

Table of Contents
- Flow Summary
- Flow Settings
- Flow Non-Default Global Se
- Flow Elapsed Time
- Flow OS Summary
- Flow Log
- Analysis & Synthesis
- Fitter
  - Summary
  - Settings
  - Parallel Compilation
  - Netlist Optimizations
  - Ignored Assignments
  - Incremental Compilatio
  - Pin-Out File
  - Resource Section
  - I/O Rules Section
  - Device Options
  - Operating Settings and
  - Messages
  - Suppressed Messages

Fitter Summary

<<Filter>>

Fitter Status	Successful - Mon Dec 27 00:26:12 2021
Quartus Prime Version	21.1.0 Build 842 10/21/2021 SJ Lite Edition
Revision Name	matrix_multiply
Top-level Entity Name	toFpga
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	1,065 / 114,480 (< 1 %)
Total registers	426
Total pins	63 / 529 (12 %)
Total virtual pins	0
Total memory bits	51,200 / 3,981,312 (1 %)
Embedded Multiplier 9-bit elements	2 / 532 (< 1 %)
Total PLLs	0 / 4 (0 %)

Fig 6.2.4 Assembler Summary

toFpga.v | Compilation Report - matrix_multiply

Table of Contents
- Flow Summary
- Flow Settings
- Flow Non-Default Global Setti
- Flow Elapsed Time
- Flow OS Summary
- Flow Log
- Analysis & Synthesis
- Fitter
- Assembler
  - Summary
  - Settings
  - Generated Files
  - Device Options: D:/VLSI/M
  - Messages
- Timing Analyzer
- EDA Netlist Writer
- Flow Messages
- Flow Suppressed Messages

Assembler Summary

<<Filter>>

Assembler Status	Successful - Mon Dec 27 00:26:19 2021
Revision Name	matrix_multiply
Top-level Entity Name	toFpga
Family	Cyclone IV E
Device	EP4CE115F29C7

Fig 6.2.5 Timing Analyzer Summary

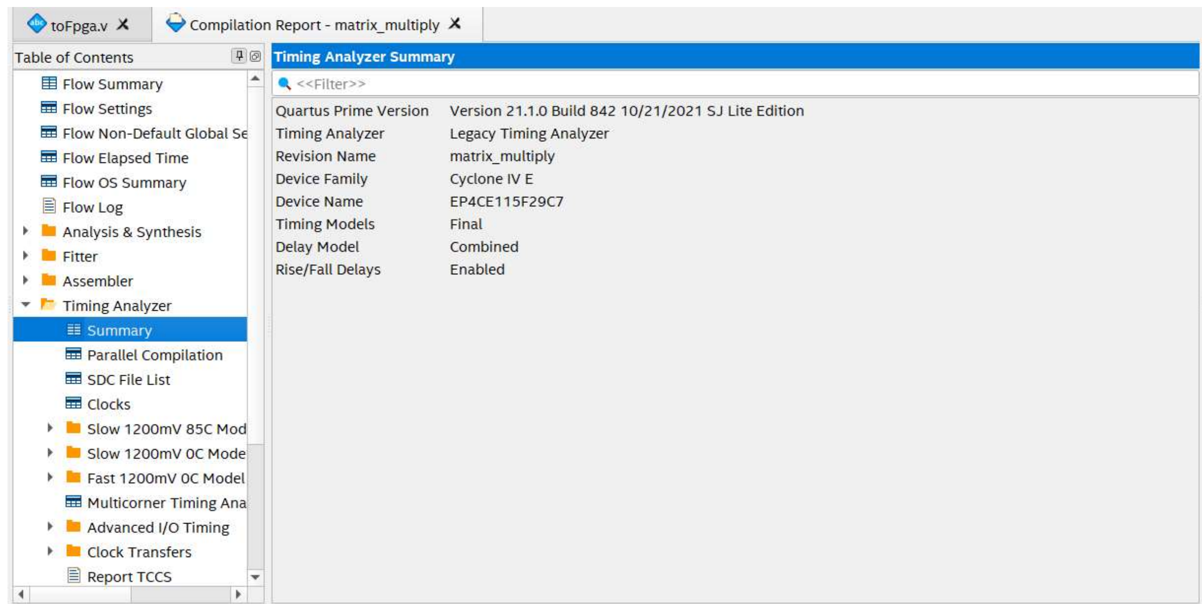
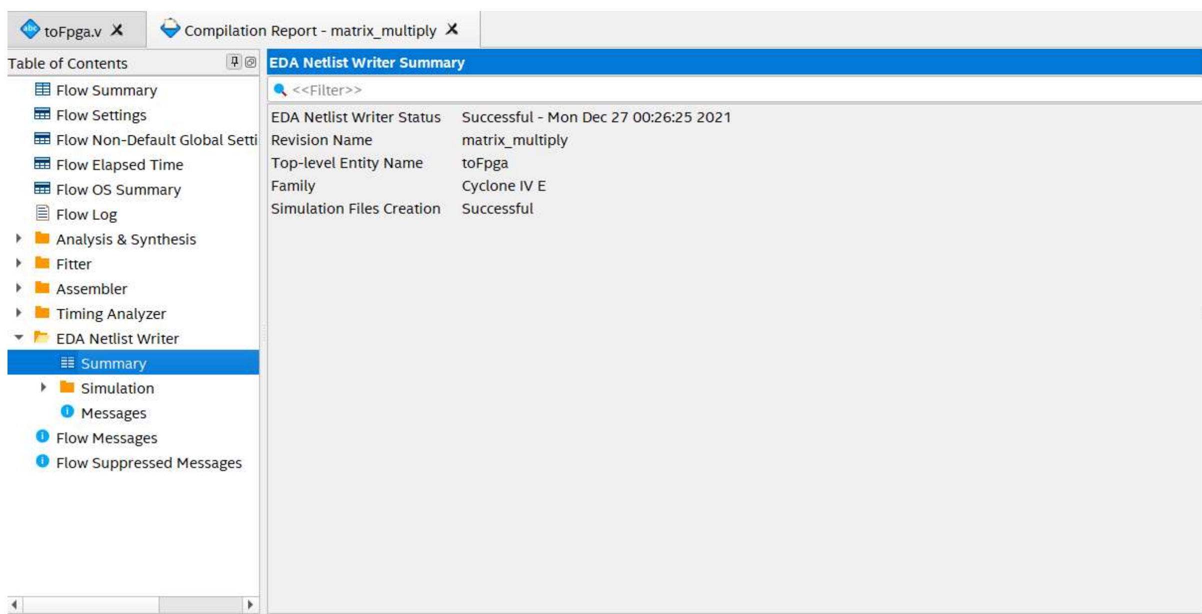


Fig 6.2.6 EDA Netlist Writer Summary



REFERENCES

[1] H. Ahn, A. Dong, A. Wong, S. L. Chaitanya Ambatipudi, X. Luo and G. Zhang, "56Gbps PAM4 SerDes Link Parameter Optimization for Improved Post-FEC BER," *2019 IEEE 28th Conference on Electrical Performance of Electronic Packaging and Systems (EPEPS)*, 2019, pp. 1-3, doi: 10.1109/EPEPS47316.2019.193195

[2] M. Harwood *et al*., "A 12.5Gb/s SerDes in 65nm CMOS Using a Baud-Rate ADC with Digital Receiver Equalization and Clock Recovery," *2007 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, 2007, pp. 436-591, doi: 10.1109/ISSCC.2007.373481.

[3] T. Beukema *et al*., "A 6.4-Gb/s CMOS SerDes core with feed-forward and decision-feedback equalization," in *IEEE Journal of Solid-State Circuits*, vol. 40, no. 12, pp. 2633-2645, Dec. 2005, doi: 10.1109/JSSC.2005.856584.

[4] H. Kimura *et al*., "A 28 Gb/s 560 mW Multi-Standard SerDes With Single-Stage Analog Front-End and 14-Tap Decision Feedback Equalizer in 28 nm CMOS," in *IEEE Journal of Solid-State Circuits*, vol. 49, no. 12, pp. 3091-3103, Dec. 2014, doi: 10.1109/JSSC.2014.2349974.

[5] A. Roshan-Zamir, O. Elhadidy, H. Yang and S. Palermo, "A Reconfigurable 16/32 Gb/s Dual-Mode NRZ/PAM4 SerDes in 65-nm CMOS," in *IEEE Journal of Solid-State Circuits*, vol. 52, no. 9, pp. 2430-2447, Sept. 2017, doi: 10.1109/JSSC.2017.2705070.

[6] J. Lee, P. -C. Chiang, P. -J. Peng, L. -Y. Chen and C. -C. Weng, "Design of 56 Gb/s NRZ and PAM4 SerDes Transceivers in CMOS Technologies," in *IEEE Journal of Solid-State Circuits*, vol. 50, no. 9, pp. 2061-2073, Sept. 2015, doi: 10.1109/JSSC.2015.2433269.

[7] Gupta, Manjari & Bhargava, Lava & Sreedevi, Indu. (2021). Mapping techniques in multicore processors: current and future trends. The Journal of Supercomputing. 77. 10.1007/s11227-021-03650-6.

[8] Rashdan, Mostafa & El-Sayed, Fahmi & Salman, Mohammad. (2020). Performance Comparison between SerDes and Time-Based Serial Links. 37-41. 10.1109/ICEEE49618.2020.9102626.

[9] Y. Hu, Y. Wang, L. Wang, Q. Cao and J. Kuang, "Performance Evaluation of Time Distribution over SerDes-based Interconnections for PET System," *2018 IEEE Nuclear Science Symposium and Medical Imaging Conference Proceedings (NSS/MIC)*, 2018, pp. 1-2, doi: 10.1109/NSSMIC.2018.8824573.

CHAPTER 7 APPENDIX

7.1 SerDes related Verilog modules

7.1.1 Serializer.v

```
1.  module serializer_unit_cell_1 (CLK,
    RESET,SERIAL_OUT,READY,INTERNAL_FINISH,COMPLETE,PAR_IN1,PAR_IN2,PAR_IN3,PAR_IN4,PAR_IN5,
    PAR_IN6,PAR_IN7,PAR_IN8,COUNT,SAMPLE_COUNT);
2.
3.   input CLK,RESET,READY;
4.   output reg SERIAL_OUT,INTERNAL_FINISH,COMPLETE;
5.   input [31:0] PAR_IN1,PAR_IN2,PAR_IN3,PAR_IN4,PAR_IN5,PAR_IN6,PAR_IN7,PAR_IN8;
6.
7.   reg [31:0] int_PAR1,int_PAR2,int_PAR3,int_PAR4,int_PAR5,int_PAR6,int_PAR7,int_PAR8;
8.  output reg [5:0] COUNT;
9.  output reg [3:0] SAMPLE_COUNT;
10. always @(posedge CLK or negedge RESET)
11. begin
12.   if(RESET == 0)
13.   begin
14.       SERIAL_OUT <= 0;
15.       INTERNAL_FINISH <= 0;
16.       COUNT <= 6'd0;
17.       COMPLETE <= 0;
18.       SAMPLE_COUNT <= 4'd0;
19.   end
20.   else if(READY == 1)
21.   begin
22.          if(SAMPLE_COUNT >= 4'd8 && INTERNAL_FINISH == 1)
23.      begin
24.             COMPLETE <= 0;
25.             SAMPLE_COUNT <= 4'd0;
26.             SERIAL_OUT <= 0;
27.             INTERNAL_FINISH <= 0;
28.             COUNT <= 6'd0;
29.      end
30.   else if (COMPLETE ==0)
31.      begin
32.
33.               /*if(COUNT >= 6'd32)
34.               begin
35.                 INTERNAL_FINISH <= 1;
36.                 SERIAL_OUT <= 0;
37.                 SAMPLE_COUNT <= SAMPLE_COUNT + 1;
38.                 COUNT <= 6'd0;
39.               end
40.               else
41.               begin*/
42.                       case(SAMPLE_COUNT)
43.                4'd0: begin
44.                 SERIAL_OUT <= PAR_IN1[COUNT];
45.                 if(COUNT >= 6'd31)
46.                 begin
47.                       INTERNAL_FINISH <= 1;
48.                       SAMPLE_COUNT <= SAMPLE_COUNT + 1;
49.                     COUNT <= 6'd0;
50.                 end
51.                 else
52.                 begin
53.                       COUNT <= COUNT+6'd1;
54.                       INTERNAL_FINISH <= 0;
```

```
55.                 end
56.                      end
57.             4'd1: begin
58.                 SERIAL_OUT <= PAR_IN2[COUNT];
59.                 //INTERNAL_FINISH <= 0;
60.                 if(COUNT >= 6'd31)
61.                 begin
62.                       INTERNAL_FINISH <= 1;
63.                       SAMPLE_COUNT <= SAMPLE_COUNT + 1;
64.                     COUNT <= 6'd0;
65.                 end
66.                 else
67.                 begin
68.                       COUNT <= COUNT+6'd1;
69.                       INTERNAL_FINISH <= 0;
70.                 end
71.                      end
72.                4'd2: begin
73.                 SERIAL_OUT <= PAR_IN3[COUNT];
74.                 //INTERNAL_FINISH <= 0;
75.                 if(COUNT >= 6'd31)
76.                 begin
77.                       INTERNAL_FINISH <= 1;
78.                       SAMPLE_COUNT <= SAMPLE_COUNT + 1;
79.                     COUNT <= 6'd0;
80.                 end
81.                 else
82.                 begin
83.                       COUNT <= COUNT+6'd1;
84.                       INTERNAL_FINISH <= 0;
85.                 end
86.                      end
87.             4'd3: begin
88.                 SERIAL_OUT <= PAR_IN4[COUNT];
89.                 //INTERNAL_FINISH <= 0;
90.                 if(COUNT >= 6'd31)
91.                 begin
92.                       INTERNAL_FINISH <= 1;
93.                       SAMPLE_COUNT <= SAMPLE_COUNT + 1;
94.                     COUNT <= 6'd0;
95.                 end
96.                 else
97.                 begin
98.                       COUNT <= COUNT+6'd1;
99.                       INTERNAL_FINISH <= 0;
100.                      end
101.                           end
102.                  4'd4: begin
103.                      SERIAL_OUT <= PAR_IN5[COUNT];
104.                      //INTERNAL_FINISH <= 0;
105.                      if(COUNT >= 6'd31)
106.                      begin
107.                            INTERNAL_FINISH <= 1;
108.                            SAMPLE_COUNT <= SAMPLE_COUNT + 1;
109.                          COUNT <= 6'd0;
110.                      end
111.                      else
112.                      begin
113.                            COUNT <= COUNT+6'd1;
114.                            INTERNAL_FINISH <= 0;
115.                      end
116.                           end
117.                  4'd5: begin
118.                      SERIAL_OUT <= PAR_IN6[COUNT];
119.                      //INTERNAL_FINISH <= 0;
120.                      if(COUNT >= 6'd31)
121.                      begin
122.                            INTERNAL_FINISH <= 1;
123.                            SAMPLE_COUNT <= SAMPLE_COUNT + 1;
124.                          COUNT <= 6'd0;
```

```
125.                        end
126.                        else
127.                        begin
128.                              COUNT <= COUNT+6'd1;
129.                              INTERNAL_FINISH <= 0;
130.                        end
131.                             end
132.                       4'd6: begin
133.                        SERIAL_OUT <= PAR_IN7[COUNT];
134.                        //INTERNAL_FINISH <= 0;
135.                        if(COUNT >= 6'd31)
136.                        begin
137.                              INTERNAL_FINISH <= 1;
138.                              SAMPLE_COUNT <= SAMPLE_COUNT + 1;
139.                            COUNT <= 6'd0;
140.                        end
141.                        else
142.                        begin
143.                              COUNT <= COUNT+6'd1;
144.                              INTERNAL_FINISH <= 0;
145.                        end
146.                             end
147.                    4'd7: begin
148.                        SERIAL_OUT <= PAR_IN8[COUNT];
149.                        //INTERNAL_FINISH <= 0;
150.                        if(COUNT >= 6'd31)
151.                        begin
152.                              INTERNAL_FINISH <= 1;
153.                              SAMPLE_COUNT <= SAMPLE_COUNT + 1;
154.                            COUNT <= 6'd0;
155.                              COMPLETE <= 1;
156.                        end
157.                        else
158.                        begin
159.                              COUNT <= COUNT+6'd1;
160.                              INTERNAL_FINISH <= 0;
161.                        end
162.                             end
163.                    default: begin
164.                              SAMPLE_COUNT <= 4'd9;
165.                            SERIAL_OUT <= 0;
166.                            INTERNAL_FINISH <= 1;
167.                    end
168.                    endcase
169.         end
170.            else
171.      begin
172.            COUNT <= 6'd0;
173.            COMPLETE <=0;
174.      end
175.     end
176.      else
177.      begin
178.            COUNT <= 6'd0;
179.            SERIAL_OUT <= 0;
180.            INTERNAL_FINISH <= 0;
181.            COMPLETE <= 0;
182.            SAMPLE_COUNT <= 4'd0;
183.      end
184.
185.  end
186.
187.  always @(posedge READY or negedge RESET)
188.  begin
189.      if(RESET == 0)
190.            begin
191.                int_PAR1 <= 32'd0;
192.                int_PAR2 <= 32'd0;
193.                int_PAR3 <= 32'd0;
194.                int_PAR4 <= 32'd0;
```

```
195.                int_PAR5 <= 32'd0;
196.                int_PAR6 <= 32'd0;
197.                int_PAR7 <= 32'd0;
198.                int_PAR8 <= 32'd0;
199.            end
200.        else
201.            begin
202.        int_PAR1 <= PAR_IN1;
203.        int_PAR2 <= PAR_IN2;
204.        int_PAR3 <= PAR_IN3;
205.        int_PAR4 <= PAR_IN4;
206.        int_PAR5 <= PAR_IN5;
207.        int_PAR6 <= PAR_IN6;
208.        int_PAR7 <= PAR_IN7;
209.        int_PAR8 <= PAR_IN8;
210.            end
211.    end
212.
213.    always @(posedge COMPLETE or negedge RESET)
214.    begin
215.        if(RESET == 0)
216.            begin
217.                int_PAR1 <= 32'd0;
218.                int_PAR2 <= 32'd0;
219.                int_PAR3 <= 32'd0;
220.                int_PAR4 <= 32'd0;
221.                int_PAR5 <= 32'd0;
222.                int_PAR6 <= 32'd0;
223.                int_PAR7 <= 32'd0;
224.                int_PAR8 <= 32'd0;
225.            end
226.        else
227.            begin
228.        int_PAR1 <= PAR_IN1;
229.        int_PAR2 <= PAR_IN2;
230.        int_PAR3 <= PAR_IN3;
231.        int_PAR4 <= PAR_IN4;
232.        int_PAR5 <= PAR_IN5;
233.        int_PAR6 <= PAR_IN6;
234.        int_PAR7 <= PAR_IN7;
235.        int_PAR8 <= PAR_IN8;
236.            end
237.    end
238.    endmodule
239.
```

7.1.2 Deserializer.v

```
1.  module deserialiser_unit_cell_1 (CLK,RESET,READY, SERIAL_IN,
    PAR_IN1,PAR_IN2,PAR_IN3,PAR_IN4,PAR_IN5,PAR_IN6,PAR_IN7,PAR_IN8,
2.  COMPLETE,INTERNAL_FINISH,COUNT,SAMPLE_COUNT,OUT);
3.  input CLK,RESET,READY,SERIAL_IN;
4.  output reg [31:0] PAR_IN1,PAR_IN2,PAR_IN3,PAR_IN4,PAR_IN5,PAR_IN6,PAR_IN7,PAR_IN8,OUT;
5.  reg [31:0]
    int_PAR_IN1,int_PAR_IN2,int_PAR_IN3,int_PAR_IN4,int_PAR_IN5,int_PAR_IN6,int_PAR_IN7,int_
    PAR_IN8;
6.  output reg [5:0] COUNT;
7.  output reg [3:0] SAMPLE_COUNT;
8.  output reg INTERNAL_FINISH,COMPLETE;
9.
10. always @(posedge CLK or negedge RESET)
11. begin
12.   if(RESET == 0)
13.   begin
```

```
14.       int_PAR_IN1 <= 32'd0;int_PAR_IN2 <= 32'd0;int_PAR_IN3 <= 32'd0;int_PAR_IN4 <=
    32'd0;int_PAR_IN5 <= 32'd0;int_PAR_IN6 <= 32'd0;int_PAR_IN7 <= 32'd0;int_PAR_IN8 <=
    32'd0;
15.       INTERNAL_FINISH <= 0;
16.       OUT <= 32'd0;
17.       COUNT <= 6'd0;
18.       COMPLETE <= 0;
19.       SAMPLE_COUNT <= 4'd0;
20.   end
21.   else if(READY == 1)
22.   begin
23.          if(SAMPLE_COUNT > 4'd7 && INTERNAL_FINISH == 1)
24.      begin
25.             COMPLETE <= 0;
26.             SAMPLE_COUNT <= 4'd0;
27.             int_PAR_IN1 <= 32'd0;int_PAR_IN2 <= 32'd0;int_PAR_IN3 <= 32'd0;int_PAR_IN4
    <= 32'd0;int_PAR_IN5 <= 32'd0;int_PAR_IN6 <= 32'd0;int_PAR_IN7 <= 32'd0;int_PAR_IN8 <=
    32'd0;
28.           //OUT <= 32'd0;
29.             INTERNAL_FINISH <= 0;
30.             COUNT <= 6'd2;
31.             OUT[1] <= SERIAL_IN;
32.      end
33.   else if (COMPLETE ==0 && INTERNAL_FINISH == 0)
34.      begin
35.                     case(SAMPLE_COUNT)
36.               4'd0: begin
37.                OUT[COUNT] <= SERIAL_IN;
38.                if(COUNT >= 6'd31)
39.                begin
40.                      INTERNAL_FINISH <= 1;
41.                      //SAMPLE_COUNT <= SAMPLE_COUNT + 1;
42.                    //COUNT <= 6'd0;
43.                    //int_PAR_IN1 <= OUT;
44.                end
45.                else
46.                begin
47.                      COUNT <= COUNT+6'd1;
48.                      INTERNAL_FINISH <= 0;
49.                end
50.                     end
51.            4'd1: begin
52.                OUT[COUNT] <= SERIAL_IN;
53.                //INTERNAL_FINISH <= 0;
54.                if(COUNT >= 6'd31)
55.                begin
56.                      INTERNAL_FINISH <= 1;
57.                      //SAMPLE_COUNT <= SAMPLE_COUNT + 1;
58.                    //COUNT <= 6'd0;
59.                      //int_PAR_IN2 <= OUT;
60.                end
61.                else
62.                begin
63.                      COUNT <= COUNT+6'd1;
64.                      INTERNAL_FINISH <= 0;
65.                end
66.                     end
67.               4'd2: begin
68.                OUT[COUNT] <= SERIAL_IN;
69.                //INTERNAL_FINISH <= 0;
70.                if(COUNT >= 6'd31)
71.                begin
72.                      INTERNAL_FINISH <= 1;
73.                      //SAMPLE_COUNT <= SAMPLE_COUNT + 1;
74.                    //COUNT <= 6'd0;
75.                      //int_PAR_IN3 <= OUT;
76.                end
77.                else
78.                begin
79.                      COUNT <= COUNT+6'd1;
```

```
80.                         INTERNAL_FINISH <= 0;
81.                   end
82.                        end
83.               4'd3: begin
84.                   OUT[COUNT] <= SERIAL_IN;
85.                   //INTERNAL_FINISH <= 0;
86.                   if(COUNT >= 6'd31)
87.                   begin
88.                         INTERNAL_FINISH <= 1;
89.                         //SAMPLE_COUNT <= SAMPLE_COUNT + 1;
90.                       //COUNT <= 6'd0;
91.                         //int_PAR_IN4 <= OUT;
92.                   end
93.                   else
94.                   begin
95.                         COUNT <= COUNT+6'd1;
96.                         INTERNAL_FINISH <= 0;
97.                   end
98.                        end
99.               4'd4: begin
100.                            OUT[COUNT] <= SERIAL_IN;
101.                            //INTERNAL_FINISH <= 0;
102.                            if(COUNT >= 6'd31)
103.                            begin
104.                                  INTERNAL_FINISH <= 1;
105.                                  //SAMPLE_COUNT <= SAMPLE_COUNT + 1;
106.                                //COUNT <= 6'd0;
107.                                  //int_PAR_IN5 <= OUT;
108.                            end
109.                            else
110.                            begin
111.                                  COUNT <= COUNT+6'd1;
112.                                  INTERNAL_FINISH <= 0;
113.                            end
114.                                 end
115.                        4'd5: begin
116.                            OUT[COUNT] <= SERIAL_IN;
117.                            //INTERNAL_FINISH <= 0;
118.                            if(COUNT >= 6'd31)
119.                            begin
120.                                  INTERNAL_FINISH <= 1;
121.                                  //SAMPLE_COUNT <= SAMPLE_COUNT + 1;
122.                                //COUNT <= 6'd0;
123.                                  //int_PAR_IN6 <= OUT;
124.                            end
125.                            else
126.                            begin
127.                                  COUNT <= COUNT+6'd1;
128.                                  INTERNAL_FINISH <= 0;
129.                            end
130.                                 end
131.                           4'd6: begin
132.                            OUT[COUNT] <= SERIAL_IN;
133.                            //INTERNAL_FINISH <= 0;
134.                            if(COUNT >= 6'd31)
135.                            begin
136.                                  INTERNAL_FINISH <= 1;
137.                                  //SAMPLE_COUNT <= SAMPLE_COUNT + 1;
138.                                //COUNT <= 6'd0;
139.                                  //int_PAR_IN7 <= OUT;
140.                            end
141.                            else
142.                            begin
143.                                  COUNT <= COUNT+6'd1;
144.                                  INTERNAL_FINISH <= 0;
145.                            end
146.                                 end
147.                        4'd7: begin
148.                            OUT[COUNT] <= SERIAL_IN;
149.                            //INTERNAL_FINISH <= 0;
```

```
150.                            if(COUNT >= 6'd31)
151.                            begin
152.                                  INTERNAL_FINISH <= 1;
153.                                  //SAMPLE_COUNT <= SAMPLE_COUNT + 1;
154.                                //COUNT <= 6'd0;
155.                                  //int_PAR_IN8 <= OUT;
156.                            end
157.                            else
158.                            begin
159.                                  COUNT <= COUNT+6'd1;
160.                                  INTERNAL_FINISH <= 0;
161.                            end
162.                                 end
163.                        default: begin
164.                                  SAMPLE_COUNT <= 4'd9;
165.                                int_PAR_IN1 <= 32'd0;int_PAR_IN2 <= 32'd0;int_PAR_IN3 <=
    32'd0;int_PAR_IN4 <= 32'd0;int_PAR_IN5 <= 32'd0;int_PAR_IN6 <= 32'd0;int_PAR_IN7 <=
    32'd0;int_PAR_IN8 <= 32'd0;
166.                                  OUT <= 32'd0;
167.                                INTERNAL_FINISH <= 1;
168.                        end
169.                        endcase
170.         end
171.            else if (COMPLETE ==0 && INTERNAL_FINISH == 1)
172.               begin
173.
174.                        case(SAMPLE_COUNT)
175.                           4'd0: begin
176.                            if(COUNT >= 6'd31)
177.                            begin
178.                                  OUT[0] <= SERIAL_IN;
179.                                  INTERNAL_FINISH <= 0;
180.                                  SAMPLE_COUNT <= SAMPLE_COUNT + 1;
181.                                COUNT <= 6'd1;
182.                                int_PAR_IN1 <= OUT;
183.                            end
184.                            end
185.                        4'd1: begin
186.                            if(COUNT >= 6'd31)
187.                            begin
188.                                  OUT[0] <= SERIAL_IN;
189.                                  INTERNAL_FINISH <= 0;
190.                                  SAMPLE_COUNT <= SAMPLE_COUNT + 1;
191.                                COUNT <= 6'd1;
192.                                  int_PAR_IN2 <= OUT;
193.                            end
194.                            end
195.                           4'd2: begin
196.                            if(COUNT >= 6'd31)
197.                            begin
198.                                  OUT[0] <= SERIAL_IN;
199.                                  INTERNAL_FINISH <= 0;
200.                                  SAMPLE_COUNT <= SAMPLE_COUNT + 1;
201.                                COUNT <= 6'd1;
202.                                  int_PAR_IN3 <= OUT;
203.                            end
204.                                 end
205.                        4'd3: begin
206.                            if(COUNT >= 6'd31)
207.                            begin
208.                                  INTERNAL_FINISH <= 0;
209.                                  SAMPLE_COUNT <= SAMPLE_COUNT + 1;
210.                                COUNT <= 6'd1;
211.                                  int_PAR_IN4 <= OUT;
212.                            end
213.                                 end
214.                        4'd4: begin
215.                            if(COUNT >= 6'd31)
216.                            begin
217.                                  OUT[0] <= SERIAL_IN;
```

```
218.                                 INTERNAL_FINISH <= 0;
219.                                 SAMPLE_COUNT <= SAMPLE_COUNT + 1;
220.                               COUNT <= 6'd1;
221.                                 int_PAR_IN5 <= OUT;
222.                           end
223.                                end
224.                       4'd5: begin
225.                           if(COUNT >= 6'd31)
226.                           begin
227.                                 OUT[0] <= SERIAL_IN;
228.                                 INTERNAL_FINISH <= 0;
229.                                 SAMPLE_COUNT <= SAMPLE_COUNT + 1;
230.                               COUNT <= 6'd1;
231.                                 int_PAR_IN6 <= OUT;
232.                           end
233.                                end
234.                          4'd6: begin
235.                           if(COUNT >= 6'd31)
236.                           begin
237.                                 OUT[0] <= SERIAL_IN;
238.                                 INTERNAL_FINISH <= 0;
239.                                 SAMPLE_COUNT <= SAMPLE_COUNT + 1;
240.                               COUNT <= 6'd1;
241.                                 int_PAR_IN7 <= OUT;
242.                           end
243.                                end
244.                       4'd7: begin
245.                           if(COUNT >= 6'd31)
246.                           begin
247.                                 OUT[0] <= SERIAL_IN;
248.                                 INTERNAL_FINISH <= 1;
249.                                 SAMPLE_COUNT <= SAMPLE_COUNT + 1;
250.                               COUNT <= 6'd1;
251.                                 COMPLETE <= 1;
252.                                 int_PAR_IN8 <= OUT;
253.                           end
254.                                end
255.                       default: begin
256.                                 SAMPLE_COUNT <= 4'd9;
257.                               int_PAR_IN1 <= 32'd0;int_PAR_IN2 <= 32'd0;int_PAR_IN3 <=
    32'd0;int_PAR_IN4 <= 32'd0;int_PAR_IN5 <= 32'd0;int_PAR_IN6 <= 32'd0;int_PAR_IN7 <=
    32'd0;int_PAR_IN8 <= 32'd0;
258.                                 OUT <= 32'd0;
259.                               INTERNAL_FINISH <= 0;
260.                                 COMPLETE <= 1;
261.                                 OUT[0] <= SERIAL_IN;
262.                       end
263.                       endcase
264.
265.                end
266.             else
267.       begin
268.             COUNT <= 6'd0;
269.             COMPLETE <=0;
270.       end
271.      end
272.       else
273.       begin
274.             COUNT <= 6'd0;
275.             int_PAR_IN1 <= 32'd0;int_PAR_IN2 <= 32'd0;int_PAR_IN3 <= 32'd0;int_PAR_IN4
    <= 32'd0;int_PAR_IN5 <= 32'd0;int_PAR_IN6 <= 32'd0;int_PAR_IN7 <= 32'd0;int_PAR_IN8 <=
    32'd0;
276.             INTERNAL_FINISH <= 0;
277.             OUT <= 32'd0;
278.             COMPLETE <= 0;
279.             SAMPLE_COUNT <= 4'd0;
280.       end
281.
282.   end
283.
```

```
284.  always @(posedge COMPLETE or negedge RESET)
285.  begin
286.       if(RESET == 0)
287.           begin
288.               PAR_IN1 <= 32'd0;PAR_IN2 <= 32'd0;PAR_IN3 <= 32'd0;PAR_IN4 <=
    32'd0;PAR_IN5 <= 32'd0;PAR_IN6 <= 32'd0;PAR_IN7 <= 32'd0;PAR_IN8 <= 32'd0;
289.           end
290.       else
291.           begin
292.       PAR_IN1 <= int_PAR_IN1;
293.       PAR_IN2 <= int_PAR_IN2;
294.       PAR_IN3 <= int_PAR_IN3;
295.       PAR_IN4 <= int_PAR_IN4;
296.       PAR_IN5 <= int_PAR_IN5;
297.       PAR_IN6 <= int_PAR_IN6;
298.       PAR_IN7 <= int_PAR_IN7;
299.       PAR_IN8 <= int_PAR_IN8;
300.           end
301.  end
302.
303.  endmodule
304.
```

7.2 Multicore Processor related Verilog modules

7.2.1 Control Unit.v

```
1.  module controlUnit
2.  #(
3.      parameter INS_WIDTH = 8
4.  )
5.  (
6.      input clk,rstN,startN,Zout,
7.      input [INS_WIDTH-1:0]ins,
8.      output reg [2:0]aluOp,
9.      output reg [3:0]incReg,    // {PC, RC, RP, RQ}
10.     output reg [9:0]wrEnReg,   // {AR, R, PC, IR, RL, RC, RP, RQ, R1, AC}
11.     output reg [3:0]busSel,
12.     output reg DataMemWrEn,ZWrEn,
13.     output done,ready
14. );
15. localparam  IDLE = 6'd0,  //states
16.
17.             NOP1 = 6'd1,
18.
19.             ENDOP1 = 6'd2,
20.
21.             CLAC1 = 6'd3,
22.
23.             FETCH_DELAY1 = 6'd37,
24.             FETCH1 = 6'd4,
25.             FETCH2 = 6'd5,
26.
27.             LDIAC_DELAY1 =  6'd38,
28.             LDIAC1 = 6'd6,
29.             LDIAC2 = 6'd7,
30.             LDIAC_DELAY2 = 6'd39,
31.             LDIAC3 = 6'd8,
```

```
32.
33.                LDAC1 = 6'd9,
34.                LDAC_DELAY1 = 6'd40,
35.                LDAC2 = 6'd10,
36.
37.                STIR_DELAY1 = 6'd41,
38.                STIR1 = 6'd11,
39.                STIR2 = 6'd12,
40.                STIR_DELAY2 = 6'd42,
41.                STIR3 = 6'd13,
42.
43.                STR1 = 6'd14,
44.                STR_DELAY1 = 6'd43,
45.                STR2 = 6'd15,
46.
47.                JUMP_DELAY1 = 6'd44,
48.                JUMP1 = 6'd16,
49.                JUMP2 = 6'd17,
50.
51.                JMPNZY_DILAY1 = 6'd45,
52.                JMPNZY1 = 6'd18,
53.                JMPNZY2 = 6'd19,
54.                JMPNZN1 = 6'd20,
55.
56.                JMPZY_DELAY1 = 6'd46,
57.                JMPZY1 = 6'd21,
58.                JMPZY2 = 6'd22,
59.                JMPZN1 = 6'd23,
60.
61.                MUL1 = 6'd24,
62.
63.                ADD1 = 6'd25,
64.
65.                SUB1 = 6'd26,
66.
67.                INCAC1 = 6'd27,
68.
69.                MV_RL_AC1 = 6'd28,
70.
71.                MV_RP_AC1 = 6'd29,
72.
73.                MV_RQ_AC1 = 6'd30,
74.
75.                MV_RC_AC1 = 6'd31,
76.
77.                MV_R_AC1 = 6'd32,
78.
79.                MV_R1_AC1 = 6'd33,
80.
81.                MV_AC_RP1 = 6'd34,
82.
83.                MV_AC_RQ1 = 6'd35,
84.
85.                MV_AC_RL1 = 6'd36;
86.
87. localparam  clr_alu   =  3'd0,
88.             pass_alu =  3'd1,
89.             add_alu   =  3'd2,
90.             sub_alu   =  3'd3,
91.             mul_alu   =  3'd4,
92.             inc_alu   =  3'd5,
93.             idle_alu =  3'd6;
94.
95. localparam  DMem_bus = 4'b0,
96.             R_bus     = 4'd1,
97.             IR_bus    = 4'd2,
98.             RL_bus    = 4'd3,
99.             RC_bus    = 4'd4,
100.              RP_bus    = 4'd5,
101.              RQ_bus    = 4'd6,
```

```
102.                R1_bus   = 4'd7,
103.                AC_bus   = 4'd8,
104.                idle_bus = 4'd9;
105.
106.  localparam
107.                NOP          =  8'd0,
108.                ENDOP    =       8'd1,
109.                CLAC     =       8'd2,
110.                LDIAC    =       8'd3,
111.                LDAC     =       8'd4,
112.                STR      =       8'd5,
113.                STIR     =       8'd6,
114.                JUMP     =       8'd7,
115.                JMPNZ    =       8'd8,
116.                JMPZ     =       8'd9,
117.                MUL          =  8'd10,
118.                ADD          =  8'd11,
119.                SUB          =  8'd12,
120.                INCAC    =  8'd13,
121.                MV_RL_AC =       {4'd1,4'd15},
122.                MV_RP_AC =       {4'd2,4'd15},
123.                MV_RQ_AC =       {4'd3,4'd15},
124.                MV_RC_AC =       {4'd4,4'd15},
125.                MV_R_AC  =       {4'd5,4'd15},
126.                MV_R1_AC =       {4'd6,4'd15},
127.                MV_AC_RP =       {4'd7,4'd15},
128.                MV_AC_RQ =       {4'd8,4'd15},
129.                MV_AC_RL =       {4'd9,4'd15};
130.
131.  localparam [9:0]
132.        no_wrEn = 10'b0000000000,
133.        AR_wrEn = 10'b1000000000,
134.        R_wrEn  = 10'b0100000000,
135.        PC_wrEn = 10'b0010000000,
136.        IR_wrEn = 10'b0001000000,
137.        RL_wrEn = 10'b0000100000,
138.        RC_wrEn = 10'b0000010000,
139.        RP_wrEn = 10'b0000001000,
140.        RQ_wrEn = 10'b0000000100,
141.        R1_wrEn = 10'b0000000010,
142.        AC_wrEn = 10'b0000000001;
143.
144.  localparam [3:0]
145.        no_inc = 4'b0000,
146.        PC_inc = 4'b1000,
147.        RC_inc = 4'b0100,
148.        RP_inc = 4'b0010,
149.        RQ_inc = 4'b0001,
150.        RC_RP_RQ_inc = 4'b0111;
151.
152.  reg [5:0]currentState, nextState;
153.
154.  always @(posedge clk) begin
155.        if (~rstN) begin
156.            currentState <= IDLE;
157.        end
158.        else begin
159.            currentState <= nextState;
160.        end
161.  end
162.  always @(startN, Zout, ins, currentState) begin
163.        case (currentState)
164.            IDLE: begin
165.                aluOp <= idle_alu;
166.                incReg <= 4'd0;
167.                wrEnReg <= 10'd0;
168.                busSel <= idle_bus;
169.                DataMemWrEn <= 1'b0;
170.                ZWrEn <= 1'b0;
171.
```

```
172.                if (~startN)
173.                    nextState <= FETCH1;
174.                else
175.                    nextState <= IDLE;
176.            end
177.
178.            NOP1: begin
179.                aluOp <= idle_alu;
180.                incReg <= 4'd0;
181.                wrEnReg <= 10'd0;
182.                busSel <= idle_bus;
183.                DataMemWrEn <= 1'b0;
184.                ZWrEn <= 1'b0;
185.                nextState <= FETCH1;
186.            end
187.
188.            ENDOP1: begin
189.                aluOp <= idle_alu;
190.                incReg <= 4'd0;
191.                wrEnReg <= 10'd0;
192.                busSel <= DMem_bus;
193.                DataMemWrEn <= 1'b0;
194.                ZWrEn <= 1'b0;
195.                nextState <= ENDOP1;
196.            end
197.
198.            CLAC1: begin
199.                aluOp <= clr_alu;
200.                incReg <= 4'd0;
201.                wrEnReg <= 10'd1;
202.                busSel <= idle_bus;
203.                DataMemWrEn <= 1'b0;
204.                ZWrEn <= 1'b1;
205.                nextState <= FETCH1;
206.            end
207.
208.            FETCH_DELAY1: begin
209.                aluOp <= idle_alu;
210.                incReg <= 4'd0;
211.                wrEnReg <= IR_wrEn;
212.                busSel <= idle_bus;
213.                DataMemWrEn <= 1'b0;
214.                ZWrEn <= 1'b0;
215.                nextState <= FETCH1;
216.            end
217.
218.            FETCH1: begin
219.                aluOp <= idle_alu;
220.                incReg <= 4'd0;
221.                wrEnReg <= IR_wrEn;
222.                busSel <= idle_bus;
223.                DataMemWrEn <= 1'b0;
224.                ZWrEn <= 1'b0;
225.                nextState <= FETCH2;
226.            end
227.
228.            FETCH2: begin
229.                aluOp <= idle_alu;
230.                incReg <= PC_inc;
231.                wrEnReg <= 10'd0;
232.                busSel <= 4'd0;
233.                DataMemWrEn <= 1'b0;
234.                ZWrEn <= 1'b0;
235.                case (ins)   //has to deside what is the next state
236.
237.                    NOP: nextState <= NOP1;
238.                    ENDOP: nextState <= ENDOP1;
239.                    CLAC: nextState <= CLAC1;
240.                    LDIAC: nextState <= LDIAC_DELAY1;
241.                    LDAC: nextState <= LDAC1;
```

```
242.                STR: nextState <= STR1;
243.                STIR: nextState <= STIR_DELAY1;
244.                JUMP: nextState <= JUMP_DELAY1;
245.                JMPNZ: nextState <= (Zout == 0)? JMPNZY_DILAY1 : JMPNZN1;
246.                JMPZ: nextState <= (Zout == 1)? JMPZY_DELAY1 : JMPZN1;
247.                MUL: nextState <= MUL1;
248.                ADD: nextState <= ADD1;
249.                SUB: nextState <= SUB1;
250.                INCAC: nextState <= INCAC1;
251.                MV_RL_AC: nextState <= MV_RL_AC1;
252.                MV_RP_AC: nextState <= MV_RP_AC1;
253.                MV_RQ_AC: nextState <= MV_RQ_AC1;
254.                MV_RC_AC: nextState <= MV_RC_AC1;
255.                MV_R_AC: nextState <= MV_R_AC1;
256.                MV_R1_AC: nextState <= MV_R1_AC1;
257.                MV_AC_RP: nextState <= MV_AC_RP1;
258.                MV_AC_RQ: nextState <= MV_AC_RQ1;
259.                MV_AC_RL: nextState <= MV_AC_RL1;
260.                default : nextState <= IDLE;
261.
262.            endcase
263.        end
264.
265.        LDIAC_DELAY1: begin
266.            aluOp <= idle_alu;
267.            incReg <= 4'd0;
268.            wrEnReg <= IR_wrEn;
269.            busSel <= idle_bus;
270.            DataMemWrEn <= 1'b0;
271.            ZWrEn <= 1'b0;
272.            nextState <= LDIAC1;
273.        end
274.
275.        LDIAC1: begin
276.            aluOp <= idle_alu;
277.            incReg <= 4'd0;
278.            wrEnReg <= IR_wrEn;
279.            busSel <= idle_bus;
280.            DataMemWrEn <= 1'b0;
281.            ZWrEn <= 1'b0;
282.            nextState <= LDIAC2;
283.        end
284.
285.        LDIAC2: begin
286.            aluOp <= idle_alu;
287.            incReg <= PC_inc;
288.            wrEnReg <= AR_wrEn;
289.            busSel <= IR_bus;
290.            DataMemWrEn <= 1'b0;
291.            ZWrEn <= 1'b0;
292.            nextState <= LDIAC_DELAY2;
293.        end
294.
295.        LDIAC_DELAY2: begin
296.            aluOp <= pass_alu;
297.            incReg <= no_inc;
298.            wrEnReg <= AC_wrEn;
299.            busSel <= DMem_bus;
300.            DataMemWrEn <= 1'b0;
301.            ZWrEn <= 1'b1;
302.            nextState <= LDIAC3;
303.        end
304.
305.        LDIAC3: begin
306.            aluOp <= pass_alu;
307.            incReg <= no_inc;
308.            wrEnReg <= AC_wrEn;
309.            busSel <= DMem_bus;
310.            DataMemWrEn <= 1'b0;
311.            ZWrEn <= 1'b1;
```

```
312.                nextState <= FETCH_DELAY1;
313.            end
314.
315.            LDAC1: begin
316.                aluOp <= idle_alu;
317.                incReg <= no_inc;
318.                wrEnReg <= AR_wrEn;
319.                busSel <= AC_bus;
320.                DataMemWrEn <= 1'b0;
321.                ZWrEn <= 1'b0;
322.                nextState <= LDAC_DELAY1;
323.            end
324.
325.            LDAC_DELAY1: begin
326.                aluOp <= pass_alu;
327.                incReg <= no_inc;
328.                wrEnReg <= AC_wrEn;
329.                busSel <= DMem_bus;
330.                DataMemWrEn <= 1'b0;
331.                ZWrEn <= 1'b1;
332.                nextState <= LDAC2;
333.            end
334.
335.            LDAC2: begin
336.                aluOp <= pass_alu;
337.                incReg <= no_inc;
338.                wrEnReg <= AC_wrEn;
339.                busSel <= DMem_bus;
340.                DataMemWrEn <= 1'b0;
341.                ZWrEn <= 1'b1;
342.                nextState <= FETCH_DELAY1;
343.            end
344.
345.            STR1: begin
346.                aluOp <= idle_alu;
347.                incReg <= no_inc;
348.                wrEnReg <= AR_wrEn;
349.                busSel <= AC_bus;
350.                DataMemWrEn <= 1'b0;
351.                ZWrEn <= 1'b0;
352.                nextState <= STR_DELAY1;
353.            end
354.
355.            STR_DELAY1: begin
356.                aluOp <= idle_alu;
357.                incReg <= no_inc;
358.                wrEnReg <= no_wrEn;
359.                busSel <= idle_bus;
360.                DataMemWrEn <= 1'b1;
361.                ZWrEn <= 1'b0;
362.                nextState <= STR2;
363.            end
364.
365.            STR2: begin
366.                aluOp <= idle_alu;
367.                incReg <= no_inc;
368.                wrEnReg <= no_wrEn;
369.                busSel <= idle_bus;
370.                DataMemWrEn <= 1'b1;
371.                ZWrEn <= 1'b0;
372.                nextState <= FETCH1;
373.            end
374.
375.            STIR_DELAY1: begin
376.                aluOp <= idle_alu;
377.                incReg <= no_inc;
378.                wrEnReg <= IR_wrEn;
379.                busSel <= idle_bus;
380.                DataMemWrEn <= 1'b0;
381.                ZWrEn <= 1'b0;
```

```
382.                nextState <= STIR1;
383.            end
384.
385.            STIR1: begin
386.                aluOp <= idle_alu;
387.                incReg <= no_inc;
388.                wrEnReg <= IR_wrEn;
389.                busSel <= idle_bus;
390.                DataMemWrEn <= 1'b0;
391.                ZWrEn <= 1'b0;
392.                nextState <= STIR2;
393.            end
394.
395.            STIR2: begin
396.                aluOp <= idle_alu;
397.                incReg <= PC_inc;
398.                wrEnReg <= AR_wrEn;
399.                busSel <= IR_bus;
400.                DataMemWrEn <= 1'b0;
401.                ZWrEn <= 1'b0;
402.                nextState <= STIR_DELAY2;
403.            end
404.
405.            STIR_DELAY2: begin
406.                aluOp <= idle_alu;
407.                incReg <= no_inc;
408.                wrEnReg <= no_wrEn;
409.                busSel <= idle_bus;
410.                DataMemWrEn <= 1'b1;
411.                ZWrEn <= 1'b0;
412.                nextState <= STIR3;
413.            end
414.
415.            STIR3: begin
416.                aluOp <= idle_alu;
417.                incReg <= no_inc;
418.                wrEnReg <= no_wrEn;
419.                busSel <= idle_bus;
420.                DataMemWrEn <= 1'b1;
421.                ZWrEn <= 1'b0;
422.                nextState <= FETCH_DELAY1;
423.            end
424.
425.            JUMP_DELAY1: begin
426.                aluOp <= idle_alu;
427.                incReg <= 4'd0;
428.                wrEnReg <= IR_wrEn;
429.                busSel <= 4'd0;
430.                DataMemWrEn <= 1'b0;
431.                ZWrEn <= 1'b0;
432.                nextState <= JUMP1;
433.            end
434.
435.            JUMP1: begin
436.                aluOp <= idle_alu;
437.                incReg <= 4'd0;
438.                wrEnReg <= IR_wrEn;
439.                busSel <= 4'd0;
440.                DataMemWrEn <= 1'b0;
441.                ZWrEn <= 1'b0;
442.                nextState <= JUMP2;
443.            end
444.
445.            JUMP2: begin
446.                aluOp <= idle_alu;
447.                incReg <= 4'd0;
448.                wrEnReg <= PC_wrEn;
449.                busSel <= IR_bus;
450.                DataMemWrEn <= 1'b0;
451.                ZWrEn <= 1'b0;
```

```
452.                nextState <= FETCH_DELAY1;
453.            end
454.
455.            JMPNZY_DILAY1: begin
456.                aluOp <= idle_alu;
457.                incReg <= 4'd0;
458.                wrEnReg <= IR_wrEn;
459.                busSel <= 4'd0;
460.                DataMemWrEn <= 1'b0;
461.                ZWrEn <= 1'b0;
462.                nextState <= JMPNZY1;
463.            end
464.
465.            JMPNZY1: begin
466.                aluOp <= idle_alu;
467.                incReg <= 4'd0;
468.                wrEnReg <= IR_wrEn;
469.                busSel <= 4'd0;
470.                DataMemWrEn <= 1'b0;
471.                ZWrEn <= 1'b0;
472.                nextState <= JMPNZY2;
473.            end
474.
475.            JMPNZY2: begin
476.                aluOp <= idle_alu;
477.                incReg <= 4'd0;
478.                wrEnReg <= PC_wrEn;
479.                busSel <= IR_bus;
480.                DataMemWrEn <= 1'b0;
481.                ZWrEn <= 1'b0;
482.                nextState <= FETCH_DELAY1;
483.            end
484.
485.            JMPNZN1: begin
486.                aluOp <= idle_alu;
487.                incReg <= PC_inc;
488.                wrEnReg <= no_wrEn;
489.                busSel <= idle_bus;
490.                DataMemWrEn <= 1'b0;
491.                ZWrEn <= 1'b0;
492.                nextState <= FETCH_DELAY1;
493.            end
494.
495.            JMPZY_DELAY1: begin
496.                aluOp <= idle_alu;
497.                incReg <= 4'd0;
498.                wrEnReg <= IR_wrEn;
499.                busSel <= 4'd0;
500.                DataMemWrEn <= 1'b0;
501.                ZWrEn <= 1'b0;
502.                nextState <= JMPZY1;
503.            end
504.
505.            JMPZY1: begin
506.                aluOp <= idle_alu;
507.                incReg <= 4'd0;
508.                wrEnReg <= IR_wrEn;
509.                busSel <= 4'd0;
510.                DataMemWrEn <= 1'b0;
511.                ZWrEn <= 1'b0;
512.                nextState <= JMPZY2;
513.            end
514.
515.            JMPZY2: begin
516.                aluOp <= idle_alu;
517.                incReg <= 4'd0;
518.                wrEnReg <= PC_wrEn;
519.                busSel <= IR_bus;
520.                DataMemWrEn <= 1'b0;
521.                ZWrEn <= 1'b0;
```

```
522.                nextState <= FETCH_DELAY1;
523.            end
524.
525.            JMPZN1: begin
526.                aluOp <= idle_alu;
527.                incReg <= PC_inc;
528.                wrEnReg <= no_wrEn;
529.                busSel <= idle_bus;
530.                DataMemWrEn <= 1'b0;
531.                ZWrEn <= 1'b0;
532.                nextState <= FETCH_DELAY1;
533.            end
534.
535.            MUL1: begin
536.                aluOp <= mul_alu;
537.                incReg <= RC_RP_RQ_inc;
538.                wrEnReg <= AC_wrEn;
539.                busSel <= R1_bus;
540.                DataMemWrEn <= 1'b0;
541.                ZWrEn <= 1'b1;
542.                nextState <= FETCH_DELAY1;
543.            end
544.
545.            ADD1: begin
546.                aluOp <= add_alu;
547.                incReg <= no_inc;
548.                wrEnReg <= AC_wrEn;
549.                busSel <= R_bus;
550.                DataMemWrEn <= 1'b0;
551.                ZWrEn <= 1'b1;
552.                nextState <= FETCH_DELAY1;
553.            end
554.
555.            SUB1: begin
556.                aluOp <= sub_alu;
557.                incReg <= no_inc;
558.                wrEnReg <= AC_wrEn;
559.                busSel <= RC_bus;
560.                DataMemWrEn <= 1'b0;
561.                ZWrEn <= 1'b1;
562.                nextState <= FETCH_DELAY1;
563.            end
564.
565.            INCAC1: begin
566.                aluOp <= inc_alu;
567.                incReg <= no_inc;
568.                wrEnReg <= AC_wrEn;
569.                busSel <= idle_bus;
570.                DataMemWrEn <= 1'b0;
571.                ZWrEn <= 1'b1;
572.                nextState <= FETCH_DELAY1;
573.            end
574.
575.            MV_RL_AC1: begin
576.                aluOp <= idle_alu;
577.                incReg <= no_inc;
578.                wrEnReg <= RL_wrEn;
579.                busSel <= AC_bus;
580.                DataMemWrEn <= 1'b0;
581.                ZWrEn <= 1'b0;
582.                nextState <= FETCH_DELAY1;
583.            end
584.
585.            MV_RP_AC1: begin
586.                aluOp <= idle_alu;
587.                incReg <= no_inc;
588.                wrEnReg <= RP_wrEn;
589.                busSel <= AC_bus;
590.                DataMemWrEn <= 1'b0;
591.                ZWrEn <= 1'b0;
```

```
592.                nextState <= FETCH_DELAY1;
593.            end
594.
595.            MV_RQ_AC1: begin
596.                aluOp <= idle_alu;
597.                incReg <= no_inc;
598.                wrEnReg <= RQ_wrEn;
599.                busSel <= AC_bus;
600.                DataMemWrEn <= 1'b0;
601.                ZWrEn <= 1'b0;
602.                nextState <= FETCH_DELAY1;
603.            end
604.
605.            MV_RC_AC1: begin
606.                aluOp <= idle_alu;
607.                incReg <= no_inc;
608.                wrEnReg <= RC_wrEn;
609.                busSel <= AC_bus;
610.                DataMemWrEn <= 1'b0;
611.                ZWrEn <= 1'b0;
612.                nextState <= FETCH_DELAY1;
613.            end
614.
615.            MV_R_AC1: begin
616.                aluOp <= idle_alu;
617.                incReg <= no_inc;
618.                wrEnReg <= R_wrEn;
619.                busSel <= AC_bus;
620.                DataMemWrEn <= 1'b0;
621.                ZWrEn <= 1'b0;
622.                nextState <= FETCH_DELAY1;
623.            end
624.
625.            MV_R1_AC1: begin
626.                aluOp <= idle_alu;
627.                incReg <= no_inc;
628.                wrEnReg <= R1_wrEn;
629.                busSel <= AC_bus;
630.                DataMemWrEn <= 1'b0;
631.                ZWrEn <= 1'b0;
632.                nextState <= FETCH_DELAY1;
633.            end
634.
635.            MV_AC_RP1: begin
636.                aluOp <= pass_alu;
637.                incReg <= no_inc;
638.                wrEnReg <= AC_wrEn;
639.                busSel <= RP_bus;
640.                DataMemWrEn <= 1'b0;
641.                ZWrEn <= 1'b1;
642.                nextState <= FETCH_DELAY1;
643.            end
644.
645.            MV_AC_RQ1: begin
646.                aluOp <= pass_alu;
647.                incReg <= no_inc;
648.                wrEnReg <= AC_wrEn;
649.                busSel <= RQ_bus;
650.                DataMemWrEn <= 1'b0;
651.                ZWrEn <= 1'b1;
652.                nextState <= FETCH_DELAY1;
653.            end
654.
655.            MV_AC_RL1: begin
656.                aluOp <= pass_alu;
657.                incReg <= no_inc;
658.                wrEnReg <= AC_wrEn;
659.                busSel <= RL_bus;
660.                DataMemWrEn <= 1'b0;
661.                ZWrEn <= 1'b1;
```

```
662.                nextState <= FETCH_DELAY1;
663.            end
664.
665.            default : begin
666.                aluOp <= idle_alu;
667.                incReg <= 4'd0;
668.                wrEnReg <= 10'd0;
669.                busSel <= 4'd0;
670.                DataMemWrEn <= 1'b0;
671.                ZWrEn <= 1'b0;
672.                nextState <= IDLE;
673.            end
674.
675.        endcase
676.
677.    end
678.
```

7.2.2 ALU.v

```
1.  module ALU
2.  #(parameter WIDTH = 12)
3.  (
4.      input signed [WIDTH-1:0]a,b,
5.      input [2:0]selectOp,
6.      output signed [WIDTH-1:0]dataOut
7.  );
8.
9.  localparam [2:0]
10.             clr  =  3'd0,
11.             pass =  3'd1,
12.             add  =  3'd2,
13.             sub  =  3'd3,
14.             mul  =  3'd4,
15.             inc  =  3'd5,
16.             idle =  3'd6;
17. assign dataOut =    (selectOp == clr)? {WIDTH{1'b0}}:
18.                     (selectOp == pass)? b:
19.                     (selectOp == add)? a+b:
20.                     (selectOp == sub)? a-b:
21.                     (selectOp == mul)? a*b:
22.                     (selectOp == inc)? a+1'b1:
23.                     (selectOp == idle)? {WIDTH{1'b0}}:
24.                     {WIDTH{1'b0}};
25.
26. endmodule
27.
```

7.2.3 Multiplexer.v

```
1.  module multiplexer
2.  #(
3.      parameter REG_WIDTH = 12,
4.      parameter INS_WIDTH = 8
5.  )
6.  (
7.      input [3:0]selectIn,
8.      input [REG_WIDTH-1:0]DMem, R, RL, RC, RP, RQ, R1, AC,
9.      input [INS_WIDTH-1:0]IR,
10.     output [REG_WIDTH-1:0]busOut
11. );
12. localparam  [3:0]
13.     DMem_sel = 4'b0,
14.     R_sel    = 4'd1,
```

```
15.     IR_sel   = 4'd2,
16.     RL_sel   = 4'd3,
17.     RC_sel   = 4'd4,
18.     RP_sel   = 4'd5,
19.     RQ_sel   = 4'd6,
20.     R1_sel   = 4'd7,
21.     AC_sel   = 4'd8,
22.     idle     = 4'd9;
23. assign busOut = (selectIn == DMem_sel)? DMem:
24.                 (selectIn == R_sel) ? R:
25.                 (selectIn == IR_sel)? {{(REG_WIDTH-INS_WIDTH){1'b0}},IR}:
26.                 (selectIn == RL_sel)? RL:
27.                 (selectIn == RC_sel)? RC:
28.                 (selectIn == RP_sel)? RP:
29.                 (selectIn == RQ_sel)? RQ:
30.                 (selectIn == R1_sel)? R1:
31.                 (selectIn == AC_sel)? AC:
32.                 {REG_WIDTH{1'b0}};
33.
34. endmodule
35.
```

7.2.4 Processor.v

```
1.  module processor
2.  #(
3.      parameter REG_WIDTH = 12,
4.      parameter INS_WIDTH = 8,
5.      parameter DATA_MEM_ADDR_WIDTH = 12,
6.      parameter INS_MEM_ADDR_WIDTH = 8
7.  )
8.  (
9.      input clk,rstN,startN,
10.     input [REG_WIDTH-1:0]ProcessorDataIn,
11.     input [INS_WIDTH-1:0]InsMemOut,
12.     output [REG_WIDTH-1:0]ProcessorDataOut,
13.     output [DATA_MEM_ADDR_WIDTH-1:0]dataMemAddr,
14.     output [INS_MEM_ADDR_WIDTH-1:0]insMemAddr,
15.     output DataMemWrEn,
16.     output done,ready
17. );
18.
19. wire [REG_WIDTH-1:0]alu_a, alu_b, alu_out;
20. wire [2:0]select_alu_op;
21. wire [3:0]busSel;
22. wire [REG_WIDTH-1:0]busOut;
23. wire [INS_WIDTH-1:0]IRout;
24. wire [REG_WIDTH-1:0] Rout, RLout, RCout, RPout, RQout, R1out, ACout;
25. wire Zout, ZWrEn;
26. wire [3:0]incReg;
27. wire [9:0]wrEnReg;
28.
29. //instantiation of each module within the processor are as follows.
30.
31. controlUnit #(.INS_WIDTH(INS_WIDTH)) CU(.clk(clk), .rstN(rstN), .startN(startN),
    .Zout(Zout), .ins(IRout),
32.                 .aluOp(select_alu_op), .incReg(incReg), .wrEnReg(wrEnReg),
    .busSel(busSel),
33.                 .DataMemWrEn(DataMemWrEn), .ZWrEn(ZWrEn), .done(done), .ready(ready));
34.
35. ALU #(.WIDTH(REG_WIDTH)) alu(.a(alu_a), .b(alu_b), .selectOp(select_alu_op),
    .dataOut(alu_out));
36.
37. multiplexer #(.REG_WIDTH(REG_WIDTH), .INS_WIDTH(INS_WIDTH)) mux(.selectIn(busSel),
    .DMem(ProcessorDataIn),
```

```
38.                 .R(Rout), .RL(RLout), .RC(RCout), .RP(RPout), .RQ(RQout), .R1(R1out),
    .AC(ACout),
39.                 .IR(IRout), .busOut(busOut));
40.
41. register #(.WIDTH(REG_WIDTH)) AR(.dataIn(busOut), .wrEn(wrEnReg[9]), .rstN(rstN),
    .clk(clk),
42.                 .dataOut(dataMemAddr));
43.
44. register #(.WIDTH(REG_WIDTH)) R(.dataIn(busOut), .wrEn(wrEnReg[8]), .rstN(rstN),
    .clk(clk), .dataOut(Rout));
45.
46. incRegister #(.WIDTH(INS_WIDTH)) PC(.dataIn(IRout), .wrEn(wrEnReg[7]), .rstN(rstN),
    .clk(clk),
47.                 .incEn(incReg[3]), .dataOut(insMemAddr));
48.
49. register #(.WIDTH(INS_WIDTH)) IR(.dataIn(InsMemOut), .wrEn(wrEnReg[6]), .rstN(rstN),
50.                 .clk(clk), .dataOut(IRout));
51.
52. register #(.WIDTH(REG_WIDTH)) RL(.dataIn(busOut), .wrEn(wrEnReg[5]), .rstN(rstN),
    .clk(clk), .dataOut(RLout));
53.
54. incRegister #(.WIDTH(REG_WIDTH)) RC(.dataIn(busOut), .wrEn(wrEnReg[4]), .rstN(rstN),
55.                 .clk(clk), .incEn(incReg[2]), .dataOut(RCout));
56.
57. incRegister #(.WIDTH(REG_WIDTH)) RP(.dataIn(busOut), .wrEn(wrEnReg[3]), .rstN(rstN),
58.                 .clk(clk), .incEn(incReg[1]), .dataOut(RPout));
59.
60. incRegister #(.WIDTH(REG_WIDTH)) RQ(.dataIn(busOut), .wrEn(wrEnReg[2]), .rstN(rstN),
61.                 .clk(clk), .incEn(incReg[0]), .dataOut(RQout));
62.
63. register #(.WIDTH(REG_WIDTH)) R1(.dataIn(busOut), .wrEn(wrEnReg[1]), .rstN(rstN),
    .clk(clk), .dataOut(R1out));
64.
65. register #(.WIDTH(REG_WIDTH)) AC(.dataIn(alu_out), .wrEn(wrEnReg[0]), .rstN(rstN),
    .clk(clk), .dataOut(ACout));
66.
67. zReg #(.WIDTH(REG_WIDTH)) Z(.dataIn(alu_out), .clk(clk), .rstN(rstN), .wrEn(ZWrEn),
    .Zout(Zout));
68.
69. assign ProcessorDataOut = Rout;
70. assign alu_a = ACout;
71. assign alu_b = busOut;
72.
73. endmodule
74.
```

17% Overall Similarity

Top sources found in the following databases:

- 17% Internet database
- Crossref database
- 5% Publications database
- Crossref Posted Content database

TOP SOURCES

The sources with the highest number of matches within the submission. Overlapping sources will not be displayed.

#	Source	Type	Similarity
1	repository.iiitd.edu.in	Internet	4%
2	archiv.ub.uni-heidelberg.de	Internet	3%
3	coursehero.com	Internet	2%
4	digital.lib.usu.edu	Internet	1%
5	bmas.designers-guide.org	Internet	<1%
6	en.wikipedia.org	Internet	<1%
7	"General Review of Phase-Locked Loops", Digital Phase Lock Loops, ...	Crossref	<1%
8	Agrawal, Mohit, and Kaushalendra Trivedi. "All Optical Frequency Enco...	Crossref	<1%
9	Bowen Li, Brandon Jiao, Chih-Hsun Chou, Romi Mayder, Paul Franzon. ...	Crossref	<1%

#	Source	Type	Similarity
10	Jameel Ahmed, Mohammed Yakoob Siyal, Shaheryar Najam, Zohaib N...	Crossref	<1%
11	ipcommunications.tmcnet.com	Internet	<1%
12	portal.research.lu.se	Internet	<1%
13	De Michele, Luca Antonio <1975>(Rovatti, Riccardo and Setti, Gianluca...	Publication	<1%
14	web.mit.edu	Internet	<1%
15	documents.mx	Internet	<1%
16	trace.tennessee.edu	Internet	<1%
17	d-nb.info	Internet	<1%
18	epic.awi.de	Internet	<1%
19	M. Hidaka, N. Ando, T. Satoh, S. Tahara. "High-resolution current meas...	Crossref	<1%
20	M DOMEIKA. "Multi-core Processors and Embedded", Software Develo...	Crossref	<1%
21	ebin.pub	Internet	<1%

22	scholarworks.rit.edu Internet	<1%
23	slideshare.net Internet	<1%