UNDERGRADUATE FINAL YEAR PROJECT PROGRESS REPORT

Department of Telecommunication Engineering

NED University of Engineering and Technology

RISC V SoC for Communication

Group Number: 11 **Batch: 2021 – 2025**

Group Member Names:

Rao Muhammad Umer	TC-21073
Muhammad Kashaf Khan	TC-21065
Huzaifa Hassan	TC-21060

Approved by

……...

Dr. Muhammad Fahim-ul-Haque
Assistant Professor of Department of Telecommunications
Project Advisor

Author's Declaration

We declare that we are the sole authors of this project. It is the actual copy of the project that was accepted by our advisor(s) including any necessary revisions. We also grant NED University of Engineering and Technology permission to reproduce and distribute electronic or paper copies of this project.

Signature and Date	Signature and Date	Signature and Date	Signature and Date
.....	.....	.....	.....
.	.	.	.
Huzaifa Hassan	M. Kashaf Khan	Rao M. Umer	-
TC-21060	TC-21065	TC-21073	-
HASSAN4410300@cloud.neduet.edu.pk	KHAN4430173@cloud.neduet.edu.pk	UMER4404443@cloud.neduet.edu.pk	-

Statement of Contributions

- The project includes collective collaboration of all group members in designing and verifying the different modules present in the project.
- A Single Cycle RISC V Processor is designed by Mr. Rao Umer and M. Kashaf on ModelSim and verified by Mr. Huzaifa in terms of correct functionality.
- Mr. Kashaf designed and implemented UART and I2C protocol on ModelSim and verified its functionality.
- The Pipelined RISC V Processor is implemented by Mr. Rao Umer.
- Mr. Rao Umer designed and implemented ROM IP (available in Quartus) to be used in the processor and simulated its testbench to verify its functionality.
- The FIFO modules are designed by M. Kashaf and verified by Mr. Huzaifa and Mr. Rao Umer
- The final placement of all modules into a single block is collectively done by all group members.
- All members contributed in the Report Writing.

Executive Summary

To design a RISC-V processor core that can communicate effectively with other components on a system-on-chip.This includes designing the RISC-V core to be compatible with the, developing interfaces and controllers, ensuring compliance with the protocols, and testing the RISC-V core to ensure it meets the requirements of the protocols. We have to ensure that the RISC-V core can effectively communicate with other components on the SoC, while also maintaining a high level of performance and efficiency. It should meet the requirement of parallel computing because as the demand for more powerful and complex computing systems increases, parallel computing is becoming increasingly important. On-chip networks can help to enable parallel computing by allowing multiple processors or cores to communicate and work together on a single chip.

The history of on-chip networks and processing began with the creation of integrated circuits (ICs). As technology advanced, ICs became more complex and multi-functional, leading to the development of microprocessors. The integration of multiple microprocessors on one chip resulted in multi-core processors, which required efficient communication between cores, leading to the creation of on-chip networks. These networks have evolved over time to become more advanced and efficient through new technologies such as network-on-chip and many-core processors.

This project's methodology is divided into five sections.

The first section covers the literature review, which includes studying research papers, books, and articles to gain a thorough understanding of the project. The second section is about code implementation on software. We used Quartus Prime and ModelSim as the software. The third section is for designing the processor, which includes creating specifications and layout for the processor. The fourth section is for debugging code, which includes fixing errors and warnings. The final section discusses the RTL design on Quartus Prime.

In summary, this project aims to design a RISC-V core and establish communication between protocols in response to the growing need for powerful parallel computing systems. To date, the project has successfully implemented a single cycle processor and conducted literature review on interface I2C. The significance of the project lies in addressing the challenges of chip communication and catering to the demands of the industry. As technology continues to advance

and the number of transistors on a chip increases, on-chip networks are becoming increasingly vital. This project aims to stay ahead of the trend by utilizing advanced communication protocols.

Acknowledgement

We are grateful to start our first acknowledgement with the almighty **Allah al-Rahman**, **al-Rahim**, the one and only we rely on throughout the entire process. His blessings equipped us with the determination and fortitude for which we required to overcome many obstacles and come out catapulted eventually to accomplish our final year design project. On behalf of all of us who had opportunity to participate in this effort we are grateful to everybody involved.

We extend our gratitude to **Mr. Fasahat Hussain** *Technical Program Manager* of **DreamBig Semiconductor Inc.** for serving as the project's mentor and providing valuable guidance and insights. His experience and suggestions made the project much more different as he linked the theoretical knowledge that we have accumulated during the study with practical experience.

On behalf of our group members, we would like to thank **Dr. Fahim ul Haq** for his valued supervision in the project. He has been a great source of guidance, advice, support and more importantly patience when it comes to mentoring the project. The same goes with our co-supervisor **Miss Hafsa Amanullah** for whose suggestions and encouragement our journey was so effective. Your advice, close monitoring, and incredibly polite tolerance during the work on the project has been invaluable.

Special acknowledgment is also accorded to the faculty members of the Department of Telecommunications Engineering for their critiques and valuable input that enriched the output of the study. We would like to thank coordinator of the final year project **Sir Muneeb Ahmed** for his coordination and support throughout the project.

At the same time, it is also important to mention that our success is based on the support of the **Department of Telecommunications Engineering**, which has offered all necessary tools for our cooperation. We have highly appreciated their support that has been shown for quite some time now. We acknowledge the contribution of everyone that has been involved in the process. Their valuable and significant contributions have been immeasurable, and for your leadership, mentorship, and support we thank you.

Table of Contents

List of Figures

List of Abbreviations

ISA Instruction Set Architecture

ICs Integrated Circuits

SoC System-on-Chip

FPGA Field Programmable Gate Array

I/O Input/Output

ASIC Application Specific Integrated Circuit

FIFO First-In-First-Out UART Universal Asynchronous Receiver/Transmitter

UART Universal Asynchronous Receiver/Transmitter

Tx Transmitter

Rx Receive

I2C Inter-Integrated Circuit

SCL Serial Clock

SDA Serial Data

RISC-V Reduced Instruction Set Computer - Five

United Nations Sustainable Development Goals

The Sustainable Development Goals (SDGs) are the blueprint to achieve a better and more sustainable future for all. They address the global challenges we face, including poverty, inequality, climate change, environmental degradation, peace and justice. There are a total of 17 SDGs as mentioned below. Check the appropriate SDGs related to the project.

☐No Poverty

☐Zero Hunger

☐Good Health and Well-being

☐Quality Education

☐Gender Equality

☐Clean Water and Sanitation

☐Affordable and Clean Energy

☑Decent Work and Economic Growth

☑Industry, Innovation, and Infrastructure

☐Reduced Inequalities

☐Sustainable Cities and Communities

☑Responsible Consumption and Production

☐Climate Action

☐Life Below Water

☐Life on Land

☐Peace, Justice, and Strong Institutions

☐Partnerships to Achieve the Goals

F/SOP/FYDP 02/06/00

Similarity Index Report

Following students have compiled the final year report on the topic given below for partial fulfillment of the requirement for Bachelor's degree in Telecommunications.

Project Title RISC V Soc for Communication

S. No.	Student Name	Seat Number
1.	Huzaifa Hassan	TC-21060
2.	Muhammad Kashaf Khan	TC-21065
3.	Rao Muhammad Umer	TC-21073

This is to certify that the Plagiarism test was conducted on complete report, and overall similarity index was found to be less than 20%, with maximum 5% from single source, as required.

Signature and Date

.....

Dr. Fahim ul Haque

Chapter 1
Introduction

On-chip networks, also referred to as on-chip interconnects are the communication channels in a computer chip or integrated circuits. They enable the various section of the chip — processor, memory, and the interfaces for input/output to 'speak' to each other. The on-chip distributed digital networks.

In this project we perform the link between the protocols. RISC-V is our main processor it is an ISA from which tailored processors can be designed. It has been used more often used in the industry especially in the area of embedded and IoT devices. These processors can be used to interface on-chip networks that will make a system-on-chip (SoC) design [1]. This makes its architecture to be flexible and extensible so that it can accommodate different company peripheral devices and communication interfaces. This makes it well suited for use in on-chip networks, as mentioned below in the introduction to this article.

In other words, RISC-V processors can be easily incorporated to compose SoC system, because RISC-V has the remarkable feature of flexibility and extensibility and low power consumption in chips, and also RISC-V is absolutely an open source architecture and many companies will participate in the RISC-V development, and then the on-chip network can be easily integrated with the other technologies [1].

1.1 Background Information

The on-chip network and processing began from the beginning of computer engineering with the creation of ICs. These are the microchips or Integrated Circuits which are small semiconductor materials for many photonic devices inclusive of transistors, diodes and some other electric devices. They form circuits able to do the operations of logic, to store data, and process them as well.

ICs are small, but developing and they started to incorporate more functionality into a single chip as the technology was enhancing. This gave rise to microprocessors; which are ICs that include a central processing unit in combination with such other units as memory and interfaces for inputs and outputs. Microprocessors act as heart of computers and they provide control solution to numerous products such as, personal computers, web-based handsets,

automobiles and home appliances.

As microprocessors became more powerful, it became increasingly possible at the system and software levels to link many of these microprocessors, eventually leading to an evolution to multi-core processors [1]. These processors have multiple CPU's linked to perform specific tasks including computation. When multi-core processes started becoming popular, inter-core communication became a critical problem and on-chip networks were introduced [11]. Such networks facilitate interaction of core systems to the extent that they can share information and materials readily.

Contemporary on chip communications have escalated to be more proficient and complex with concepts such as no-network on chip and many core processor [13]. While these improvements have created a way for more cores to be incorporated into a single chip, performance, as well as energy consumption, have also been enhanced.

This evolution of on-chip networks and processing has been fundamental in the formation of current computing systems and has uniquely contributed to the progress of other disciplines such as artificial intelligence and similarly other fields such as big data and IoT.

1.2 Significance and Motivation

1.2.1 Significance

In embedded systems, the processor has to communicate with peripheral like memory units in order to execute instruction and manage data. Various techniques in memory interfacing can be restrictive since they are bound by factors like the number of pins, power consumption and are not easily expandable hence not fit for some applications.
One of the most known is the I2C protocol, or Inter-Integrated Circuit, used in low-speed short-range communication lines [12]. It works with just two lines; SCL and SDA for its functioning. So incorporation of I2C inside the RISC-V processor facilitates the best means towards the memory mapping. This approach reduces the complexity of definition of the hardware, eliminates additional wiring and cables where they are not needed, and enables the creation of optimized and easily expanded and reconfigured systems of a reasonable scale, which makes it ideal for use in an environment with limited resources.

Therefore, the need to connect different processors bears different considerations which include parallelism, implementation of loads, expansion, and other means of communication [23].

1.2.2 Motivation

As single computing units become complex with multi-core processors and other features, on-chip communication has to be effective for higher performance and low power consumption. There are several reasons to do this project which include:

Chip communication is a rather important component of contemporary electronics and its applications are numerous and varied. At university, we can equip ourselves with useful knowledge and skills that will be highly demanded in the labor market.

Communication using chips is still a developing innovation area, where new technologies and applications are constantly being created. The immediate reason motivating us to work on a given project in this area is the desire to engage ourselves with state-of-the-art technology to proactively create new products.

Searching new paradigms and methodologies to ensure effective and efficient communication on-chip which overcomes the challenges include high-speed signaling and power management.

Drawing another benefit – to increase the knowledge about the principles of on-chip communication and their relation to other fields of computer engineering and electronics.

Addressing the specific need of the industry for efficient, high-performing and low-power chips for applications such as mobility, servers, and the Internet of Things.

1.3 Aims and Objectives

The main goal behind having an on-chip network for a chip that is designed using RISC-V is to enhance communication among various parts of the chip. The objectives of such a design may include:

It mainly deals with the enhancement of the generic hardware interfaces to enhance the bitrates used in transferring data between the various parts of the chip faster data processing .Open source which means that the platform can be adapted and changed easily wherever needed.

Again, since it is open source, the cost of developing and paying a license fee as it is for commercial cores is even lower.

1.4 Methodology

1.4.1 Literature Review

Literature review means retrospective critical analysis of research on a certain theme. Moreover, this review is concerned with RISC-V architecture, UART protocol, I2C protocol, interfacing strategies, and System Verilog. We first started looking at what RISC-V architectures are, which gave us a basis of the single-cycle processor and its initial construction, which proved to be essential to this project. After that, we searched for books in order to become more acquainted with the language syntax and basics of System Verilog which allowed to script the modules for the single-cycle processor [5]. As for the last aspect, we looked at the interfacing. All of these topics are discussed in detail in Chapter 2 of this dissertation.

1.4.2 Software Implementation

Software implementation refers to the process of coding on software for a specific system or application.

1.4.2.1 Designing the Processor

To design a processor, the architectural description and physical structure of the processor have to be created, identifying the instruction set, the number of registers required and many more. Another

form of design is when a high level model of the processor is created with tools for simulations and verification purposes.

1.4.2.2 Coding of the Single Cycle Processor

It may concern encoding of the single-cycle processor so as to provide a precise control to the processor such that it can perform a single instruction in one clock cycle.

1.4.2.3 Synthesis and Debugging

After coding the next process is to combine code which means to provide a format that implements the code on physical hardware [1]. The process is generally referred to as the translation of high level language to the gate-level in order to facilitate the design of the Processor Hardware Organization. Stakeholders can choose to debug their applications since it is a phase that is dedicated to the identification of the errors in code.

1.4.2.4 RTL Simulation

RTL (Register Transfer Level) simulation is the process of simulating a digital circuit's behavior at the RTL level [10]. It involves testing how the processor would function in a real-world scenario using its RTL description. This step is vital in the design process as it allows designers to ensure the processor operates correctly before moving to physical hardware implementation.

1.4.2.5 Summary

All the major activities in using software simulation have been described above right from the design phase to the testing phase. The step by step coding of the single cycle processor has been explained elaborately in Chapter 3 with reference to how the various modules are designed, as well as realized and combined to form the entire processor system. In this chapter the overall coding methodology is described as well as the function of each module within the processor and how they intersect as well as the specific functions they carry out. Further, the chapter also covers the tools and language used during the construction one, of which is System Verilog language for the hardware description while Model SIM for the simulation.

They are defined by code translation, where the processor's architectural details are described using hardware description languages, followed by simulation and debugging phases to validate the correctness of the implementation. This simulation enhances the confidence of designers on the

intended, expected and optimal function of the processor including the ALU, control unit, registers, and memory [20]. Thus the simulation process provides a protocol through error checking and comparison to ensure that the single cycle processor is executing instructions as planned, and that the individual elements are functioning well within the system. Last is the exercise of the specified modules into what constitutes an entire working processor, tests and effectiveness of the processor are determined by test runs before going to the hardware.

1.5 Report Outline

1.5.1 Scope

Real-time embedded systems are increasingly using heterogeneous architectures that incorporate various processing cores and hardware accelerators. These platforms can be implemented on silicon or deployed on FPGA boards using industry-standard protocols in a project of designing and realizing RISC-V processor's central processing, UART and I2C communication protocol based on FPGA and SoC [25]. It includes designing, synthesizing and optimizing of the RISC-V core for implementation on FPGA and the proper utilization of resources, performance and power. Most of the tests of the hardware implementation will be performed at the FPGA side to ensure that the RISC-V core is working as expected, both the functionality of the core and the timing must be validated, as well as the proper I2C communication with external memory modules. In the context of SoC integration, the project is to integrate the RISC-V processor core into a SoC system architecture and, besides that, implement the peripheral interfaces, on-chip and external memory controllers, and the design of SoC specific IPs [24]. This integration will mainly emphasize the development of a flexible SoC which will use RISC-V core for regular computations, UART and I2C for interfacing the peripherals [18-21]. This power and performance optimization will also be highlighted in the project so as to propose the SoC that can meet the demands of embedded systems or IoT applications but at the same time need to be small, energy efficient and having high performance. Commercially available IP blocks that use the interface that can be easily integrated into a larger design for an FPGA or ASIC to meet specific functional requirements.

1.5.2 Outline

This report has been divided into 6 chapters that highlights all the progress that has been made in the project up till now. The first chapter covers the basic introduction about what SoC is, what is

our core processor, what is the background of all the main technologies related to our project and what are the aims, objectives and significance of this project. Then the second chapter is all about the base of this project that is Literature Review. We did all the necessary research on RISC-V processors, Processor Pipelining, UART protocol, interfacing of devices with processors. Then chapter 3 covers everything about our core processor. It discusses the architecture of a single cycle processor and its simulation process. And the last chapter concludes our report work. Chapter 4 explores the design and implementation of a pipelined RISC-V processor, beginning with an introduction to pipelining and its benefits. It details the modeling process, including the Control Unit and Hazard Unit, which manage pipeline flow and resolve data and control hazards. Finally, it presents testbench creation and RTL simulation results to validate functionality and performance. Chapter 5 focuses on designing and implementing a ROM IP core, including configuring the ROM module and using the Altsyncram component for efficient memory synthesis. It covers memory initialization with initialization files and the proper connection of address and clock signals. The chapter concludes with the creation of a testbench and RTL simulation to verify the ROM's functionality. Chapter 6 covers the design and functionality of UART (Universal Asynchronous Receiver/Transmitter) for serial communication. It explains the key components, such as the transmitter and receiver, and how data is formatted into frames for transmission. The chapter also includes simulations to validate the operation of both the transmitter and receiver, ensuring proper communication and data integrity.

Chapter 2
Literature Review

2.1 Introduction

This chapter presents a review of the studies performed to gain further insight into RISC-V processors, communication protocols and integration of components with processors. Apart from that we also took secondary inputs of external and internal consultants but the main part of the data and information was collected from the academic research papers, some trustworthy website and then the whole data was analyzed.

2.2 RISC-V Processors

RISC-V is an open source instruction set architecture conceived from reduced instruction set computing architecture. It was designed as a versatile, fast and economical way of equipping the human society with productive computing structures. One of RISC-V's key benefits is that it is an open instruction set, meaning that its use does not cost any money and it can be changed with no permission needed [20].

The RISC-V instruction set includes foundational base instruction sets, making it versatile and easy to implement in various ways while maintaining compatibility across implementations. This flexibility allows it to be used effectively across a wide range of microarchitectures, from compact aeronautic systems to large, high-performance systems designed for speed [21]. Moreover, RISC-V allows further creation and design of further sub extensions, including both present and potential following advanced hardware enhancements. This flexibility allows that it may be relevant and an ability to change throughout the course of the company's evolution.

RISC-V ISA has multiple base instruction sets and instructions as its building blocks for designing many diverse processors and systems. There are actually a couple of ways to implement this RISC-V processor and one of which is single-cycle architecture. In this approach, each instruction is run in a single clock cycle with the help of direct connection to control unit, IR, the ALU, and DM [7]. The full description of the architecture of single-cycle processor is provided in chapter 3.

2.3 Communication Protocols

Since SoC is a structure comprising of many functional parts such as CPUs, memory units and

peripheral devices, established communication protocols are very important in SoC architectures. These protocols are supposed to be highly flexible and complex, which can be used in possibly multiprocessor systems with application of one or multiple cores and further configurations. In multi-processor systems, communications protocol are useful in the synchronized transfer of data and control messages between the processors [12]. This is done through connecting the processors by an interface so that there is synchronization between all of them.

Moreover, the same interface can be proposed for other functional units such as peripherals and memory, contributing to the resultant system having a well-structured form. The specifics of securing communication with the help of standardized protocols are numerous, but one of the most obvious benefits is the fact that they allow the system to offer the required level of flexibility to interact between various parts of the system [11],[14]. Because such concerns are isolated, these protocols simplify the inter-component relationships and reduce the possibility of having wrong designs or compatibility problems [18]. Furthermore, the programmable nature permitting their implementation on different technologies means that the change in FPGAs and ASICs, for instance, will afford system designers even more creativity and flexibility in regard to system design.

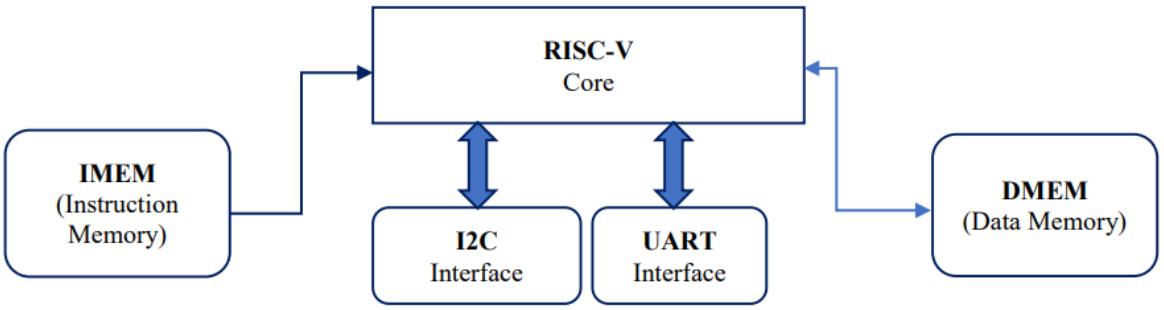

Figure 1 RISC-V Core Interface using I2C and UART Protocol

2.4 Interfacing

Embedded systems are manufactured to accomplish particular objectives, involving processors communicating with them. Such systems usually involve microcomputing in the form of a microcontroller and subordinate features such as digital and analog programmable input/output, serial interface, timers as well as any other requisite to facilitate implementation. UART and I2C

protocols are effectively used to connect processors required as the interfaces which act as a bridge between two or more systems and facilitate data exchange as shown in Figure 1.

2.4.1 UART: Universal Asynchronous Receiver/Transmitter

UART is a serial data transfer protocol that works on start bit, data bits, optionally the parity bit and a stop bit. It is often used for connection between a microcontroller and another microcontroller or a computer. UART is easy to implement and uses only two signal wires (Transmit and Receive) while it can run in full-duplex mode.

2.4.1.1 Implementation of UART

For UART implementation a controller is needed which is a special chip for serial communication. One can use any FPGA starter boards for making interfacing for serial communication. This implies that the most basic function of the FPGA will effectively be repurposed as a UART controller and is connected to an added system like computers or another microcontroller most often through given pins [20].

To configure the FPGA as a UART controller, you will need to implement the following functionalities:

Baud rate generator: This is used to set the communication speed between the FPGA and the serial device.

Shift register: This is used to convert parallel data to serial data.

2.4.2 I2C: Inter-Integrated Circuit

I2C interface uses a unique hardware component called I2C controller for the instance of I2C communication. This controller interfaces with the microcontroller through two specific pins a list of signals that are available on the SDA (Serial Data) and SCL (Serial Clock). A normal controller includes state machine, a clock generator and data buffer [21].

The state machine also controls the flow of communication between the devices, clock generator generates clock that is needed to synchronize data transfer rate in communication. On the other hand, the data buffer stores in-coming or out-going data to enhance flow and co-ordination during communication.

2.4.2.1 Implementation of I2C

In the case of hardware requirement, most of time in I2C (Inter-Integrated Circuit) needs I2C controller which is a separate Functional Module used to works on the two wired serial communicating system. I2C usage with an FPGA starter board is fairly adoptable way of integrating I2C communication into an FPGA design [23]. The FPGA must be configured with the following features in order to serve as an I2C controller:

Clock Generator: This gives the clock signal to be set as SCL for synchronizing between the peripheral devices/FPGA and the I2C device.

Data Shift Register: This is used in serial to parallel and parallel to serial most of the data can be transmitted one bit at a given time over the SDA line.

Address Decoder: This translates the 7 bit or 10 bit address of the I2C device so that communication is targeted towards the right peripheral.

When the I2C controller is implemented it is necessary for the SCL of the FPGA to be linked to the SCL of the I2C device and the SDA of the FPGA to the SDA of the I2C device. In order to obtain the correct signal levels on SCL and SDA channels, the pull-up resistors are required for these lines [24]. The other factor is the synchronization of the speed of the I2C controller to the need of the I2C device for communication to take place efficiently. In general, incorporating I2C interfaces on an FPGA starter board improves your FPGA layout by adding the ability and facility to interface with a multitude of peripheral equipment through the I2C control channel.

2.4.3 Summary

In conclusion, UART relies on a UART controller to manage serial communication, while I2C uses an I2C controller [25]. Both types of controllers share similar features, such as a baud rate generator, shift register, and FIFO buffer in the case of UART, and a state machine, clock generator, and data buffer for I2C.

2.5 System Verilog

System Verilog is an HDL used for designing and verification of digital systems [2]. It is an improved version of the Verilog HDL based on the standard IEEE 1364-2005.

System Verilog introduces several new features that Verilog lacks, including:

1. Object-oriented programming (OOP) features like classes, interfaces, and inheritance [5].
2. Enhanced concurrency modeling, enabling the representation of multiple execution threads within a single design.

To be precise, System Verilog is important in that it allows the designers to code at a higher level of abstraction than the raw hardware description language, and the code reusability results in a faster development cycle and reduced errors [3]. It also improve the actual verification time, which, in turn, reduces the verification expenses. It is currently in extensive use in the semiconductor industry for both digital circuit design and testing and supports various EDA tools [4]. As for us, System Verilog is used to describe and validate the processor for an FPGA. After this, the System Verilog code is compiled and mapped into a netlist form of the design required for FPGA. Other EDA tools used in the implementation recline Altera Quartus which supports System Verilog and offers system-wise RTL to bit-stream-based design system.

2.6 Conclusion

In this chapter, we discussed the sections that were part of our Literature Review. The first section covers all the basic knowledge about the core processor which has been designed using RISC-V Processor and its ISA [6]. The next section is about the protocols that we will use to carry out communication between the processor. In the last two sections, we discussed the HDL language that we have used to script our code for the Single Cycle processor, UART and I2C modules and also about the devices that will be used for interfacing.

Chapter 3
Single Cycle Processor

3.1 Introduction

Single-cycle processor design can be used when implementing the instruction set architecture known as RISC-V. A single-cycle processor communicates one instruction in one term, so the instructions are implemented efficiently. Even when these processors can offer high speeds they are more troublesome to implement because they are complex [7].

Single-cycle processors are normally employed as the core processors of microcontrollers, digital signal processor and other related integrated systems that demand high performance. They are also used in specific fields such as HPC, where performance beats aesthetics any given day. However, to generate such processors requires careful planning and optimization in a way that all elements run concurrently with the same clock cycle.

3.2 Architecture of Single Cycle Processor

3.2.1 Introduction

A Single cycle processor is the most elementary type of the Central Processing Unit and its structure is quite evident. It consists of specific functional blocks that function as one in order to perform a given command. At the center is the control unit that manages the working of the processor. It also has different regions for storage instruction or data instruction or data too.

The first step towards designing a single-cycle processor is to create parts in hardware that store important data [1]. These are the program counter, the analogue of the instruction pointer that points at the instruction being currently executed; and registers that are used for storing of the values produced at some stage of computations and used further. These compose the processor's state basis.

Next, combinational circuit blocks are interconnected so as to perform certain operations. Depending on the present state of the processor these blocks decide the new state of the processor. For example, instructions are read from a program memory specifically reserved for this purpose and load and store instructions, both access the data in different locations in the memory [9].

This architecture guarantees a systematic manner of working the instruction and data. The separation of memory from a function allows it to be used readily while blending combinational logic into processor performance without interruption.

3.2.2 Main Code

```
module riscvsingle(input logic clk, reset,
 output logic [31:0] PC,
input logic [31:0] Instr,
output logic MemWrite,
output logic [31:0] ALUResult, WriteData,
input logic [31:0] ReadData);


logic ALUSrc, RegWrite, Jump, Zero;
logic [1:0] ResultSrc, ImmSrc;
logic [2:0] ALUControl;


controller c(Instr[6:0], Instr[14:12], Instr[30],
Zero,ResultSrc, MemWrite, PCSrc,ALUSrc, RegWrite,
Jump,ImmSrc, ALUControl);


datapath dp(clk, reset, ResultSrc, PCSrc,
ALUSrc, RegWrite,
ImmSrc, ALUControl,
Zero, PC, Instr,
ALUResult, WriteData, ReadData);

endmodule
```

3.2.2.1 State Elements

3.2.2.1.1 Program Counter

A program counter or Program counter (PC) is charged with the responsibility of keeping track of the current instruction. It has an input called PCNext – the identifier of the memory address to the next instruction to execute.

3.2.2.1.2 Memory

Memory in a processor system is usually parted into two parts in order to make it easier to manage and execute. Instruction memory is the first sector mentioned; it contains instructions which the processor performs. It has only one read port and it receives a 32-bit instruction address input and the port is labelled as “A”. The Instruction Memory reads here the 32-bit instruction of the given address and transfers it through the “RD” read data line. This arrangement helps make sure that the processor can retrieve the instructions with ease and faster.

The second part is Data Memory area which is responsible for storage and access of data only. It is made with just one read/write port to enable both activities. If 'WE' is asserted the Data Memory 'WD' writes data to the addressed memory 'A' during the clock cycle. When "WE" is low, the Data Memory takes the data from the memory address "A" and, after that transfers it through the "RD" data bus. This dual functionality port helps the processor to load and unload the data as desired depending on its operation.

3.2.2.1.3 Register

The register file is an important component of the processor that has three different ports, two for reading and one for writing. The two read ports take a 5-bit address input, which indicates the specific register that will be used as a source operand. The data stored in these specified registers is then placed on the read data outputs "RD1" and "RD2".

The write port, which is port 3, is designed to store new data. It takes a 5-bit address input "A3" to specify the register where the data will be stored, a 32-bit write data input "WD3",a write enable signal "WE3", and the clock. If the write enable signal is asserted, the data "WD3" is written into the designated register "A3" during the rising edge of the clock cycle.

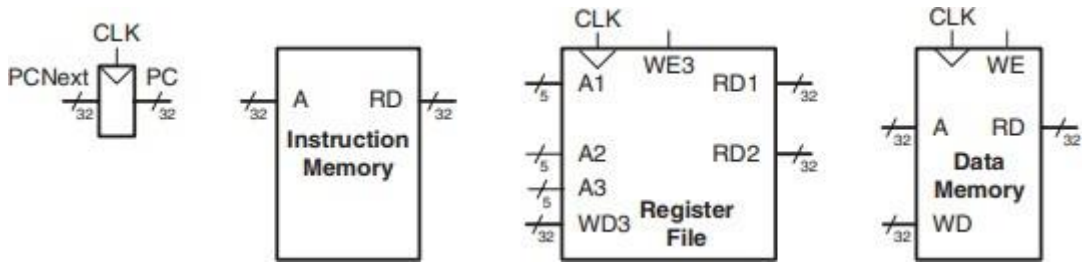

Figure 2 State Elements

3.2.2.1.4 Summary

In the processor, when the address is changed here what comes on RD is the data after a short delay not the clock. We can write within a particular time span, but the writing process is dictated by a schedule. These memories update their content only on the rising edge of the clock signal [1]. This enables changes to the system state to occur only at clock transition times.

3.3 Modeling of Single Cycle Processor

Our microarchitecture is split into two interconnected components: which includes the data path and the control unit.

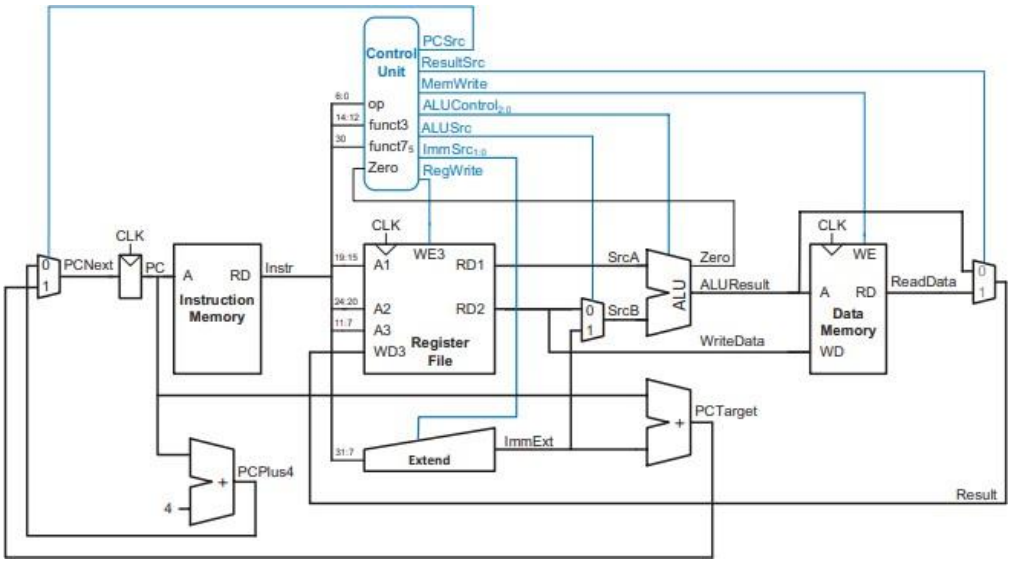

Figure 3 Single Cycle Processor

3.3.1 Data Path

The data path can process information units know as words and it has sub sections for multiplexers; registers; ALUs and memory units. In our design, we implement the RV32I profile that is a 32-bit extension of RISC-V, and therefore the data path is 32 bits wide [16].

It performs control functions of different processor components for instance the program counter (PC) [16]. Next is a storing place for the address that the program counter needs to grab from memory for the processor to execute. Following a given instruction, the PC is modified to position at the next of a program sequence.

3.3.2 Code of Data Path

```
module datapath(input logic clk, reset,
 input logic [1:0] ResultSrc,
input logic PCSrc, ALUSrc,
input logic RegWrite,
input logic [1:0] ImmSrc,
input logic [2:0] ALUControl,
output logic Zero,
output logic [31:0] PC,
input logic [31:0] Instr,
output logic [31:0] ALUResult, WriteData,
input logic [31:0] ReadData);
 logic [31:0] PCNext, PCPlus4, PCTarget;
 logic [31:0] ImmExt;
 logic [31:0] SrcA, SrcB;
 logic [31:0] Result;
 // next PC logic
 flopr #(32) pcreg(clk, reset, PCNext, PC);
 adder pcadd4(PC, 32'd4, PCPlus4);
 adder pcaddbranch(PC, ImmExt, PCTarget);
 mux2 #(32) pcmux(PCPlus4, PCTarget, PCSrc, PCNext);
 // register file logic
 regfile rf(clk, RegWrite, Instr[19:15], Instr[24:20],
 Instr[11:7], Result, SrcA, WriteData);
 extend ext(Instr[31:7], ImmSrc, ImmExt);
 // ALU logic
 mux2 #(32) srcbmux(WriteData, ImmExt, ALUSrc, SrcB);
 alu alu(SrcA, SrcB, ALUControl, ALUResult, Zero);
 mux3 #(32) resultmux(ALUResult, ReadData, PCPlus4,
ResultSrc, Result);
endmodule
```

The modules that are used in the above code are described as follows:

3.3.3 Flip-Flop-Based Register

The 32-bit flip-flop-based register called "pcreg" that is used to store the program counter value in the data path of a RISC-V processor.

3.3.4 Adder 4

The “pcadd4” is a 32-bit adder implemented in the data path of a RISC-V processor to increment a computer’s program counter by a constant four, so that it points to the next instruction for execution.

3.3.5 Adder

The “pcaddbranch” is a 32-bit adder in RISC -V data path that adds an extended immediate value to current program counter so that it points to target address of a branch instruction. This operation is intended for changing the flow of the program’s execution.

3.3.6 Mux 2to1: Pcmux

The “pcmux” is a further multiplexer on the data path of a RISC-V processor that chooses between the incremented value of the Program Counter (PC), coming from the “pcadd4” adder and the target value of PC, coming from the “pcaddbranch” adder. It passes on a chosen value to the output “PCNext” which is reflected on the program counter register. The selection is done by the “PCSrc” control signal which directs the program on its path.

3.3.7 Register File

The “rf” can be a part of the data path of a RISC-V processor that holds the register value associated with the processor. It receives the clock signal, control signal, address of registers and data to be written in the registers. These inputs are used for reading and writing a register content which is used as data sources and destinations during performing arithmetic and logic instructions in course of an instruction cycle.

3.3.8 Extension Unit

The “ext” is an extension implemented in the datapath of a RISC-V CPU. A zero or sign extension of a 25 -bit immediate value may be made depending on the control signal: The circuit takes as inputs a 25 -bit immediate value and a control signal. This enables the immediately value to be treated as a sign value and this enables the use of registers in operation.

3.3.9 Mux 2to1: SrcB Mux

The data path of the RISC-V processor consist of a 2 Input:1 Output multiplexer known as “srcbmux”. It receives data from two sources: the register file and the sign-extender are the two components in the.code section of a CPU. The ALUSrc control input is to select one of these two inputs as the output or SrcB input. It is then supplied to the ALU as one out of the inputs during the course of an instruction being executed. refers to immediate value or register file values as per the value of the ALU source control signal of the instruction.

3.3.10 Arithmetic and Logic Unit

An ALU is for keeping in mind that it is actually named “alu” and is inserted into the data path of a RISC-V processor [18]. The ALU is designed to perform arithmetic and logical operations; it receives two operands named SrcA, SrcB; and one control signal called ALUControl. It produces

two outputs: ALUResult which is the result of the last operation occurred and Zero, which is a signal that ALU result is zero. The ALU is totally charged with the responsibility of performing arithmetic operations including addition, subtraction, shifting procedures and other logical operations like, AND, OR, and NOT inclusive of every other instructing operation that might be assigned to the CPU hence making it part of the CPU that is vital in instructing operations.

3.3.11 Mux 3to1: Result Mux

The data path of a RISC-V processor contains a 3-input multiplexer (mux) known as resultmux. The mux receives three operands, ALUResult to ReadData, and PCPlus4, along with one control signal called ResultSrc to determine which of the operands will go out. For the mux input of Result, the circuit selects one of the inputs. The resultmux has an important function of choosing the right data for use depending on the instruction they are about to process and is an integral part of CPU for processing of instruction.

3.3.12 Summary

The datapath of a single-cycle processor is a component of the processor that is used to complete the task [20]. As all of the mentioned modules, they all work together in order to fetch the instruction, then determine in decode mode whether the Instruction Byte has an arithmetical or logic operation then perform the said operation. After that, the results may be written into the register file or memory [16]. This process is performed until the program ends, and for each instruction in the program above one of the two movements is performed [13]. The single-cycle processor datapath presents itself well and is very lean and fast explanatory, nevertheless, it is capable of performing at most one instruction per a clock cycle which hampers it through overall proficiency.

3.3.13 Control Unit

It functions to read the current instruction from the data path and produce signals on how an instruction will be processed. These signals are multiplexer select, register enable and memory write, which control the path in the data path to perform the needed function. But in layman terms, the control unit gets the instruction and then issues directions to the data path as to how that instruction is to be processed.

3.3.14 Code of Control Unit

```
module controller(input logic [6:0] op,
 input logic [2:0] funct3,
 input logic funct7b5,
 input logic Zero,
 output logic [1:0] ResultSrc,
 output logic MemWrite,
 output logic PCSrc, ALUSrc,
 output logic RegWrite, Jump,
 output logic [1:0] ImmSrc,
 output logic [2:0] ALUControl);
logic [1:0] ALUOp;
logic Branch;


maindec md(op, ResultSrc, MemWrite, Branch,
 ALUSrc, RegWrite, Jump, ImmSrc, ALUOp);
aludec ad(op[5], funct3, funct7b5, ALUOp, ALUControl);
assign PCSrc = Branch & Zero | Jump;
endmodule
```

The modules used in above code are discussed as follows:

3.3.15 Main Decoder

Main decoder in any RISC-V processor is accountable of decoding the opcode of an instruction and creating control signals which are required for execution of the instruction. The opcode is a header in the instruction, which defines which operation needs to be performed. The main decoder utilizes the opcode to fetch either the correct control signals from a lookup table or through a circuit diagram [11]. These control signals control the operation of the processor's components in term of ALU, memory and register file to perform the instruction. Such signals may include, Data swap, branching, ALU operations and other tasks' signals. Main decoder also has an integral part in the instruction pipeline in the sense that the initial step for decoding is done by the main decoder apart from it determining other control signals for the rest of the pipeline stages to fetch and execute the instruction.

3.3.16 ALU Decoder

In this module, the data coming from the main decoder are entered when the program counter crosses the instruction. Next, the module details the operand type whether a register or immediate or whichever and what is present in the instruction. Also, the ALU decoder determines which operation in the ALU should be executed.

In RISC-V processors, ALU decoder identifies the specific function the ALU is to perform from the opcode and the function field of the instruction [1]. These fields are used by the ALU decoder to find out the required operation from a lookup table or via several logic circuits that produce control signals to allow the ALU to execute the operation of account. Sub-classes of instructions that are included in RISC-V are for example R-type instructions and I/B-type instructions Every of those has its own opcode as well as function fields. For each instruction type it is used to decode the appropriate ALU operation, out of the fields described above.

3.3.17 Summary

This part in the single-cycle processor is in charge of the instruction control in the processor. This decipher playfully the instructions which are found in the memory and produce the necessary control signals to execute it [20]. It also controls the fetch decode execute cycle and that instructions are properly executed, and data is properly shifted between the units of the processor.

3.4 Compilation of modules

All the written modules are in System Verilog and are ‘compile ready’ in QUARTUS software with RTL simulation done.

3.5 Testbenches

A test bench for each of the modules was also designed to test each of the modules on Modelsim. The testbench evaluates the designed functions and execution speed of both RISC-V processor and memory blocks.

3.6 Simulation of Single Cycle Processor

To ensure that all instructions in a RISC-V processor work as intended, we have developed a test program to conduct this test. The idea is to execute multiple operations which if all procedures work as expected should provide the desired result. More specifically if the program runs it should store the value 25 to the memory location 100 as evidenced by the following screen shot: The machine code for these instructions is shown in the file riscvtest.txt that is loaded in memory during simulations by the test bench. The highest level of the design contains the main module that provides the RISC-V processor and memory, the testbench is initializing the model under test,

generating the clock signal, and resetting the simulation process. It observes the writing to memory and raises success when the value of 25 is written to memory.

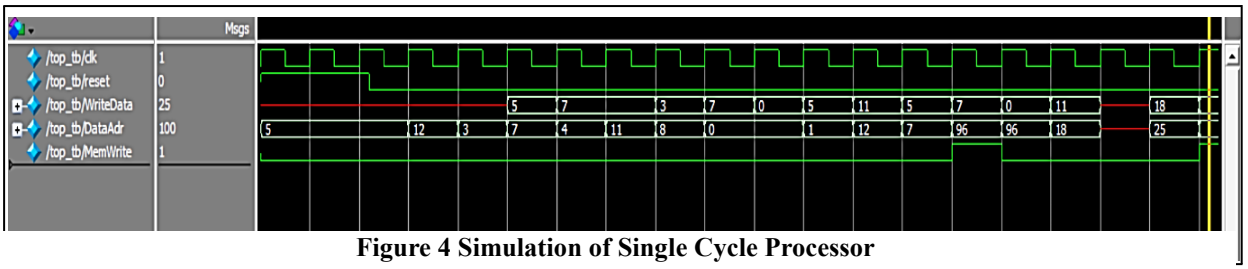

Figure 4 Simulation of Single Cycle Processor

As the simulation was successful, a message ‘Simulation Succeeded’ was displayed to let the seemingly executed written code work through without issues.

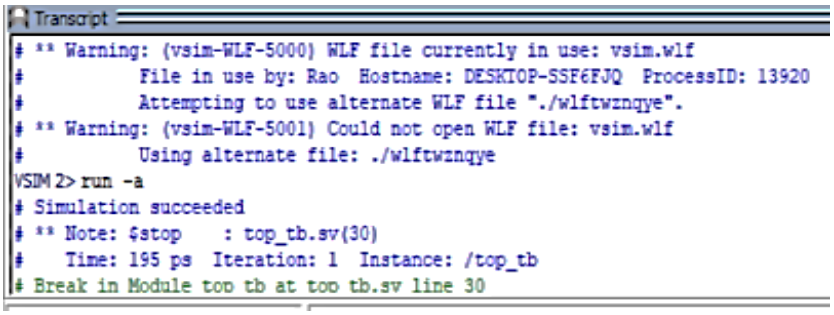

Figure 5 “Simulation Succeeded of Single Cycle” Message

3.7 Conclusion

Therefore, microarchitecture design of single-cycle processor is designed in such a way that one instruction per clock cycle is completed. The support of control unit is imperative here since it interprets the instructions and synthesizes the appropriate control signals for implementation.

Chapter 4
Pipelined Processor

4.1 Introduction

Pipelining is one of the complex techniques applied to optimize the turnaround time of digital systems. This is realized by partitioning of the single-cycle processor into five pipeline stages, where it employs a five-instruction, five-stage pipelined processor in which each stage performs a distinct operation [1]. This division means that the clock frequency is considerably higher than with the ordinary approach, where each stage contains one-fifth of the total logic.

The time taken to execute each instruction has not changed but the system output capacity has improved by five times. In modern microprocessor of the kind that executes billions of instructions per second then ensuring that the single instruction expenditure minimum is of little important than the throughput rate [8]. In spite of these issues, the advantages of pipelining make the latter a key concern in the concept of high-performance microprocessors of the present days (Qi et al., 2022).

4.2 Modeling of Pipelined Processor

The pipeline processor architecture is designed to extract out the most performance from modern microprocessors.

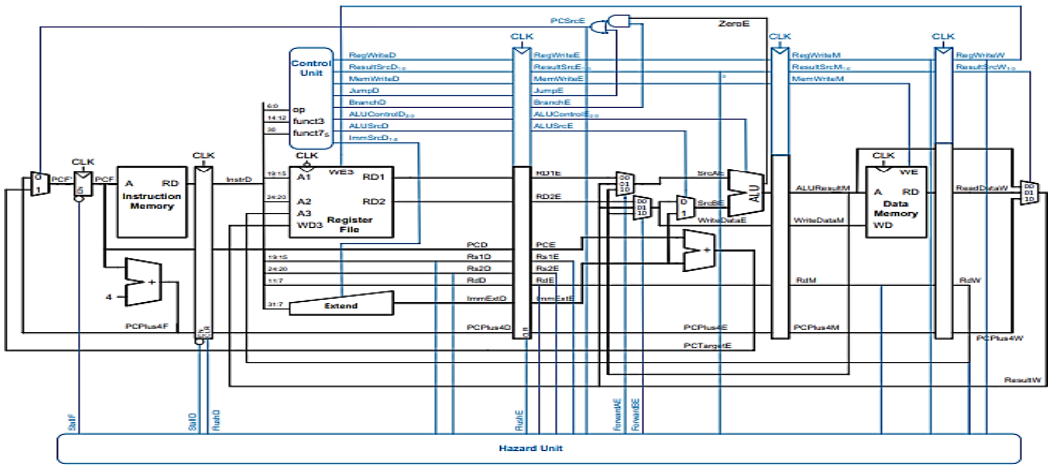

Figure 6 Architecture of Pipelined Processor

The pipelining divides the procedure of instruction execution into varied stages where each and every stage is incharge of a definite operation [8]. Such a division allows a number of instructions to be processed concurrently, thus increasing overall system productivity (Qi et al., 2022). Most of the

elements of a pipelined architecture are similar to those in any single-cycle processor; the data path, however, is divided into 5 stages as shown in figure 6. Moreover, if and how the hazards exist for the pipelined processor has also been incorporated into the discussion.

4.2.1 Datapath

Usually there is a low performance in modern processors due to the very time-consuming operations that include memory access, register files, and the actual operations performed in the arithmetic/logical unit. To solve these challenges, it was important to design pipelined architecture which could easily solve these issues [16]. To clarify, the various stages of the independent and self-contained pipeline microarchitecture mentioned above comprise five stages, where each stage of the processor's data path is dedicated to the performance of one of the five aforementioned tasks consuming quite a lot of resources for their completion, the pipeline structure also allows for instruction parallelism, which leads to a significant increase in total cycles per second [14]. Building upon the methodology of the multicycle processor, the pipelined processor incorporates five key stages: These are the five stages of the system which include; Fetch, Decode, Execute, Memory, and Write back. These stages correlate to the workings in multicycle processing, but with the bonus of being able to run at the same time (Qi et al., 2022).

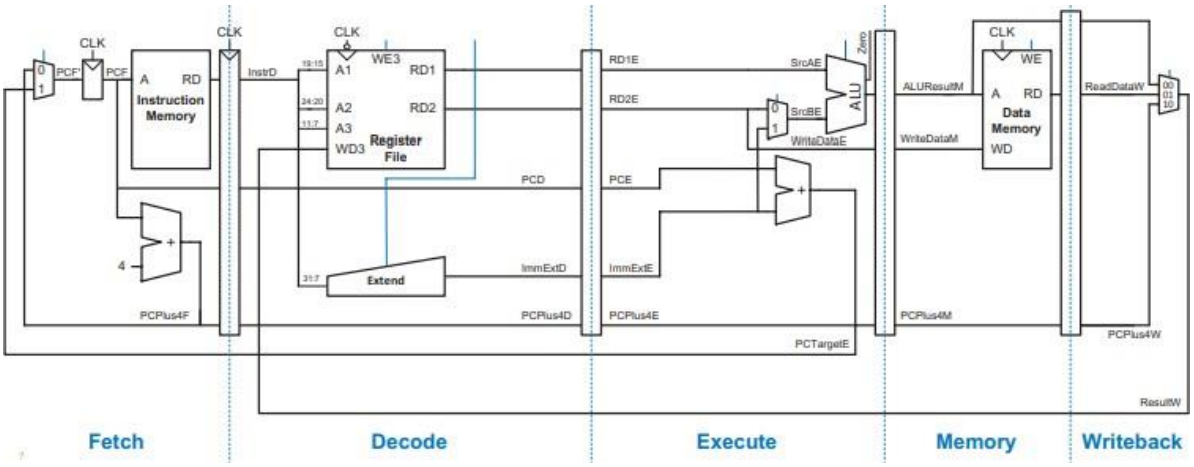

Figure 7 Datapath of Pipelined Processor

In Figure 7, the pipelined data path is illustrated, achieved by introducing four pipeline registers that effectively segment the data path into five distinct stages [14]. The demarcation of these stages, along with their boundaries, is visually depicted through dotted lines. To differentiate signals based on their

location within the pipeline stages, a suffix (F, D, E, M, or W) is appended, indicating the specific stage they belong to.

4.2.1.1 Datapath Code

```
module datapath(

input logic clk, reset,

 // Fetch stage signals
 input logic StallF,
 output logic [31:0] PCF,
 input logic [31:0] InstrF,

 // Decode stage signals
 output logic [6:0] opD,
 output logic [2:0] funct3D,
 output logic funct7b5D,
 input logic StallD, FlushD,
 input logic [2:0] ImmSrcD,

 // Execute stage signals
 input logic FlushE,
 input logic [1:0] ForwardAE, ForwardBE,
 input logic PCSrcE,
 input logic [2:0] ALUControlE,
 input logic ALUSrcAE, // needed for lui
 input logic ALUSrcBE,
 output logic ZeroE,

 // Memory stage signals
 input logic MemWriteM,
 output logic [31:0] WriteDataM, ALUResultM,
 input logic [31:0] ReadDataM,

 // Writeback stage signals
 input logic RegWriteW,
 input logic [1:0] ResultSrcW,

 // Hazard Unit signals
 output logic [4:0] Rs1D, Rs2D, Rs1E, Rs2E,
 output logic [4:0] RdE, RdM, RdW);

 // Fetch stage signals
 logic [31:0] PCNextF, PCPlus4F;

  // Decode stage signals
  logic [31:0] InstrD;
  logic [31:0] PCD, PCPlus4D;
  logic [31:0] RD1D, RD2D;
  logic [31:0] ImmExtD;
  logic [4:0] RdD;
```

```
  // Execute stage signals
   logic [31:0] RD1E, RD2E;
   logic [31:0] PCE, ImmExtE;
   logic [31:0] SrcAE, SrcBE;
   logic [31:0] SrcAEforward;
   logic [31:0] ALUResultE;
   logic [31:0] WriteDataE;
   logic [31:0] PCPlus4E;
   logic [31:0] PCTargetE;

   // Memory stage signals
   logic [31:0] PCPlus4M;

   // Writeback stage signals
   logic [31:0] ALUResultW;
   logic [31:0] ReadDataW;
   logic [31:0] PCPlus4W;
   logic [31:0] ResultW;

   // Fetch stage pipeline register and logic
   mux2 #(32) pcmux(PCPlus4F, PCTargetE, PCSrcE, PCNextF);

   flopenr #(32) pcreg(clk, reset, ~StallF, PCNextF, PCF);

   adder pcadd(PCF, 32'h4, PCPlus4F);

   // Decode stage pipeline register and logic
   flopenrc #(96) regD(clk, reset, FlushD, ~StallD,
   {InstrF, PCF, PCPlus4F},
   {InstrD, PCD, PCPlus4D});
  assign opD = InstrD[6:0];

  assign funct3D = InstrD[14:12];

  assign funct7b5D = InstrD[30];

  assign Rs1D = InstrD[19:15];

  assign Rs2D = InstrD[24:20];

  assign RdD = InstrD[11:7];

  regfile rf(clk, RegWriteW, Rs1D, Rs2D, RdW, ResultW, RD1D, RD2D);

  extend ext(InstrD[31:7], ImmSrcD, ImmExtD);

  // Execute stage pipeline register and logic
  floprc #(175) regE(clk, reset, FlushE,
  {RD1D, RD2D, PCD, Rs1D, Rs2D, RdD, ImmExtD, PCPlus4D},
  {RD1E, RD2E, PCE, Rs1E, Rs2E, RdE, ImmExtE, PCPlus4E});

  mux3 #(32) faemux(RD1E, ResultW, ALUResultM, ForwardAE, SrcAEforward);

  mux2 #(32) srcamux(SrcAEforward, 32'b0, ALUSrcAE, SrcAE); // for lui

  mux3 #(32) fbemux(RD2E, ResultW, ALUResultM, ForwardBE, WriteDataE);

  mux2 #(32) srcbmux(WriteDataE, ImmExtE, ALUSrcBE, SrcBE);

  alu alu(SrcAE, SrcBE, ALUControlE, ALUResultE, ZeroE);

  adder branchadd(ImmExtE, PCE, PCTargetE);
 // Memory stage pipeline register
 flopr #(101) regM(clk, reset,
 {ALUResultE, WriteDataE, RdE, PCPlus4E},
 {ALUResultM, WriteDataM, RdM, PCPlus4M});

 // Writeback stage pipeline register and logic
 flopr #(101) regW(clk, reset,
 {ALUResultM, ReadDataM, RdM, PCPlus4M},
 {ALUResultW, ReadDataW, RdW, PCPlus4W});

 mux3 #(32) resultmux(ALUResultW, ReadDataW, PCPlus4W, ResultSrcW,ResultW);
endmodule
```

4.2.2 Fetch Stage

At the inception of the pipeline, the processor's foremost task is to retrieve the next instruction from memory. This pivotal task involves the `pcmux` multiplexer, which operates based on the`PCSrcE` control signal. It decides whether the next program counter (`PCNextF`) originates from the increment program counter (`PCPlus4F`) or the potential target program counter (`PCTargetE`) in cases where branch instructions are predicted. The chosen program counter is stored within the `pcreg` register. Meanwhile, the `pcadd` adder computes the increment programcounter (`PCPlus4F`) by adding 4 to the current program counter (`PCF`).

4.2.3 Decode Stage

The Decode stage assumes the responsibility of disassembling the fetched instruction to unveilits structure and nuances. The ‘regD’ pipeline register becomes very important when looking at how pipelining works. It captures three pivotal elements: the instruction that was fetched (`InstrF`), the program counter that’s keeping track of where we’re at (`PCF`), and the next place the program counter will go if it counters up (`PCPlus4F`). Here, the instruction goes through a step where important parts like the operation type, the function code, and the names of the registers are pulled out from the code. These fields help us figure out what type of instructions there are and what information the CPU works with.

4.2.4 Execute Stage

The Execute stage is the heart of instruction processing, where the actual computation and manipulation of data take place.During the Execute stage that data is actually computed and processed. The `regE` pipeline register is extremely important and stores key data such as the source registers (`RD1E`, `RD2E`) content, an immediate extension (`ImmExtE`), and the program counter ready for the next instruction (`PCPlus4E`). By means of the `faemux` and `fbemux` multiplexers, the information that the ALU requires is efficiently sent by data forwarding from previous stages of the pipeline. The ALU performs calculations and logic on the operands `SrcAE`, `SrcBE` using the control signal `ALUControlE`, resulting in the ALU result `ALUResultE`. At the same time, the `branchadd` adder figures out the target program counter (`PCTargetE`) by adding the immediate extension (`ImmExtE`) to the current program counter (`PCE`). This is an important step when the computer wants to do a jump to another instruction.

4.2.5 Memory Stage

Before data can be used again or stored, the Memory stage is critical for carrying out either process. The `regM` register is used to store important pieces of data. the result generated by the ALU (`ALUResultE`), the data to be written back (`WriteDataE`), the register where the written data will go (`RdE`), and the new program counter value (`PCPlus4E`). In this step, the processor works together with the memory system that may cause memory caches to be updated. It is also very important for dealing with problems related to memory.

4.2.6 Writeback Stage

This stage brings the instruction's path to an end by storing the results in registers. The `regW` register is very important, as it holds the ALU output, memory-read information, the next register to use, and the counter for stepping forward in the program. The multiplexer `resultmux` filters the data to make sure the necessary parts are written back to the register file. Deciding which value to write back is up to the `ResultSrcW` controller, which directs the value from the ALU output, memory read result, or PC increment.

4.2.7 Conclusion

All in all, the datapath module handles the flow of data and control signals from one stage of the pipeline to another. Because of this process, various instructions are carried out at the same time and efficiently [1]. While pipe lining improves performance, it also introduces problems like managing data dependencies, making sure everything is synchronized, and managing flow of control [20]. The main function of the datapath module is to execute instructions fast and in parallel, which makes it a necessary element in pipeline design.

4.3 Control Unit

4.3.1 Introduction

The control unit in the pipelined processor (pictured in Figure 8) is important for guiding the different sections of the pipelined design and ensuring everything happens in sync. The outline here explains how the pipelined control unit generates signals according to the different phases of the pipeline. The

signals are produced by analyzing the instruction that is being processed at the moment [1], [8], and [14].

Additionally, the control unit's job is to keep an eye out for any dangers and make sure that each step in the pipeline runs smoothly and doesn't have any problems during the process. By changing these control signals as needed, the controller helps the computer use its resources better and run instructions in a more efficient way.

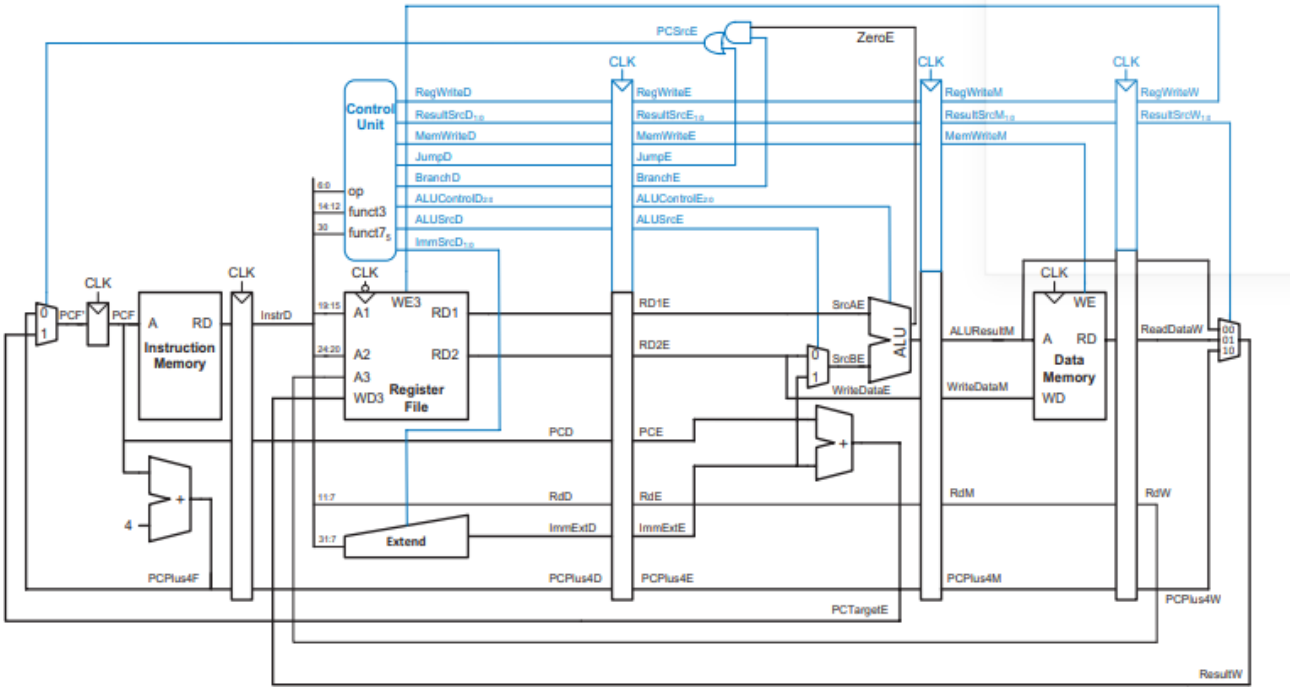

Figure 8 Pipelined Processor With Control Signals

4.3.2 Control Unit Code

```
module controller(
 input logic clk, reset,

 // Decode stage control signals
 input logic [6:0] opD,
 input logic [2:0] funct3D,
 input logic funct7b5D,
 output logic [2:0] ImmSrcD,

  // Execute stage control signals
 input logic FlushE,
 input logic ZeroE,
 output logic PCSrcE, // for datapath and Hazard Unit
 output logic [2:0] ALUControlE,
 output logic ALUSrcAE,
 output logic ALUSrcBE, // for lui
 output logic ResultSrcEb0, // for Hazard Unit

 // Memory stage control signals
 output logic MemWriteM,
 output logic RegWriteM, // for Hazard Unit

 // Writeback stage control signals
 output logic RegWriteW, // for datapath and Hazard Unit
 output logic [1:0] ResultSrcW);

 // pipelined control signals
 logic RegWriteD, RegWriteE;
 logic [1:0] ResultSrcD, ResultSrcE, ResultSrcM;
 logic MemWriteD, MemWriteE;
 logic JumpD, JumpE;
 logic BranchD, BranchE;
 logic [1:0] ALUOpD;
 logic [2:0] ALUControlD;
 logic ALUSrcAD;
 logic ALUSrcBD; // for lui

 // Decode stage logic
 maindec md(opD, ResultSrcD, MemWriteD, BranchD,ALUSrcAD, ALUSrcBD, RegWriteD, JumpD, ImmSrcD, ALUOpD);

 aludec ad(opD[5], funct3D, funct7b5D, ALUOpD, ALUControlD);

  // Execute stage pipeline control register and logic
  floprc #(11) controlregE(clk, reset, FlushE,
 {RegWriteD, ResultSrcD, MemWriteD, JumpD, BranchD,ALUControlD, ALUSrcAD, ALUSrcBD},
 {RegWriteE, ResultSrcE, MemWriteE, JumpE, BranchE,ALUControlE, ALUSrcAE, ALUSrcBE});

  assign PCSrcE = (BranchE & ZeroE) | JumpE;

  assign ResultSrcEb0 = ResultSrcE[0];

  // Memory stage pipeline control register
  flopr #(4) controlregM(clk, reset,
  {RegWriteE, ResultSrcE, MemWriteE},
  {RegWriteM, ResultSrcM, MemWriteM});

  // Writeback stage pipeline control register
  flopr #(3) controlregW(clk, reset,
  {RegWriteM, ResultSrcM},
  {RegWriteW, ResultSrcW});

 endmodule
```

This section focuses on the specifics of the controller, which is the key component of the pipelined control unit. This part of the CPU is responsible for coordinating how inputs and outputs work together to choose the instructions that should be carried out. The clock signal determines the speed of operations, and the reset signal initiates the process of synchronizing the data.

4.3.3 Decoding State Logic (maindec and aludec)

At this point, maindec and aludec are the focus of the decoder. They research the details involved in the coding and key functions of computers. Its function is to inspect the instruction and create control signals including ‘RegWriteD’, ‘ResultSrcD’, ‘MemWriteD’, and ‘BranchD’. The signals determine what will happen in the following steps. At the same time, the 'aludec' phase processes functions into two control signals called 'ALUOpD' and 'ALUControlD'. They help the computer know what operations it should do when the instruction is executed. The ‘maindec’ and ‘aludec’ develop control signals that operate correctly with the other codes and instructions.

4.3.4 Execution State Logic (controlregE)

In this stage, control signals needed for the process are produced. This stage is responsible for `RegWriteD`, `ResultSrcD`, `MemWriteD`, `JumpD`, `BranchD`, `ALUControlD`, `ALUSrcAD`, and `ALUSrcBD` using inputs it gets from the previous decode stage. This module helps to generate the necessary signals that ensure the execution phase proceeds smoothly. There are `RegWriteE`, `ResultSrcE`, `MemWriteE`, `JumpE`, `BranchE`, `ALUControlE`, `ALUSrcAE`, and `ALUSrcBE`, each with a different function during this step. While all these signals are important, `PCSrcE` is key because it makes the decision about which way the program counter will move after a branch. Since this system enforces the instructions, the program functions as intended and follows its proper order.

4.3.5 Memory and WriteBack Stage Logic

It is at this stage that controlregM and controlregW handle memory access and write-back tasks. Modules in this group ensure that data remains intact, generating commands required for reading from and writing to memory. The results generated at the execution stage help manage and utilize data. This phase must be managed properly, as it is key in handling and retaining information. Simultaneously, 'controlregW' performs during write-back. This module assembles and outputs the 'RegWriteW' and 'ResultSrcW' signals. The signals are present to ensure that instructions complete and the results are correctly saved in the registers.

4.3.6 Conculsion

Overall, the control unit manages stages and ensures efficient resource use in the pipelined processor. ‘maindec’ and ‘aludec’ translate the code, ‘controlregE’ controls when parts of the program are carried

out, and ‘controlregM’ and ‘controlregW’ keep data integrity. As a result of this, instructions can move smoothly between different parts, boosting the performance of the processor.

4.4 Hazard Unit

4.4.1 Introduction

For pipelined processors, the main challenge is designing them properly. dealing with hazards that disturb the teaching process. Another problem is RAW hazards, where an instruction must wait until the result of another instruction is ready [8], [14]. If the result arrives on time, we can proceed forward, and otherwise, we can pause the process (stall) until the result is ready. If the processor doesn’t know what to access next, that is a case of control hazards. We can either stop the pipeline or estimate what other command will be next, and if the guess is wrong, we must restart the process. You must know how instructions influence each other and identify any possible flaws.

4.4.2 Hazard Unit Code

```
module hazard(
 input logic [4:0] Rs1D, Rs2D, Rs1E, Rs2E, RdE, RdM, RdW,
 input logic PCSrcE, ResultSrcEb0,
 input logic RegWriteM, RegWriteW,
 output logic [1:0] ForwardAE, ForwardBE,
 output logic StallF, StallD, FlushD, FlushE);

 logic lwStallD;

    // forwarding logic

 always_comb begin

 ForwardAE = 2'b00;
 ForwardBE = 2'b00;
```

```
 if (Rs1E != 5'b0)
 if ((Rs1E == RdM) & RegWriteM) ForwardAE = 2'b10;
 else if ((Rs1E == RdW) & RegWriteW) ForwardAE = 2'b01;
 if (Rs2E != 5'b0)
 if ((Rs2E == RdM) & RegWriteM) ForwardBE = 2'b10;
 else if ((Rs2E == RdW) & RegWriteW) ForwardBE = 2'b01;
 end
    // stalls and flushes
 assign lwStallD = ResultSrcEb0 & ((Rs1D == RdE) | (Rs2D == RdE));
 assign StallD = lwStallD;
 assign StallF = lwStallD;
 assign FlushD = PCSrcE;
 assign FlushE = lwStallD | PCSrcE;
endmodule
```

This System Verilog module helps detect and fix hazards that may occur in a pipelined processor. The role of this module is to check for hazards that might come from data dependencies and to control how stages in the pipeline run accordingly.

4.4.3 Forwarding Logic

ForwardsAE and ForwardBE allow the data to be processed without hazards in the execution stage. By doing so, it reduces interruptions that can happen in other stages. The module identifies whether the value should be sent forward either to write-back (RdW) or to execution stage (Rs1E or Rs2E) from the memory (RdM).

4.4.4 Stall and Flush Control

The module also looks after handling stalls and clearing instructions from the pipeline caused by hazards. It finds possible risks, such as those involved with load usage, which are marked by the lwStallD light. When a hazard is found, StallD and StallF signals come on and cause the computer to stall during decode and fetch stages.

4.4.4.1 Stall Detection

A lwStallD signal is issued when a load-use hazard happens, which means the result being processed in the execution phase becomes the same as an operand being read at the decode phase. It also leads to the activation of StallD and StallF, which causes the engine to stall.

4.4.4.2 Flush Control

When there is a hazard detected in the execute stage, the FlushD signal is enabled to keep things synchronized. If a hazard is found in the lwStallD or PCSrcE, the FlushE signal flushes the decode stage.

4.4.4.3 Conclusion

In conclusion, The hazard detection and resolution process demonstrated has a significant role in ensuring data is intact and contributes to preventing hazards in pipelined processors [8]. Meaningful stalls and flushes during execution help a processor run smoothly, ultimately contributing a lot to the success of its design [14].

4.5 Testbench

4.5.1 Introduction

Here we go over the Verilog code for the testbench that makes it possible to simulate the DUT, called 'top'. The purpose of the testbench is to make sure the DUT operates as expected by setting up the test environment and testing the results.

4.5.2 Testbench Code

```
module tb();

    reg clk=0, rst;

    always begin
        clk = ~clk;
        #50;
    end

    initial begin
        rst <= 1'b0;
        #200;
        rst <= 1'b1;
        #1000;
        $finish;
    end

    initial begin
        $dumpfile("dump.vcd");
        $dumpvars(0);
    end

    Pipeline_top dut (.clk(clk), .rst(rst));
endmodule
```

The testbench module plays a key part in checking that the device under test is working correctly. It makes it possible to study the DUT by generating input and a clock, allowing us to observe its behavior [3]. The expected results are compared to the results obtained, and when the match is correct, the simulation comes to a stop.

4.6 RTL Simulation

The process ensures that the results of the simulation meet the requirements for the specific case [10]. The initial part of the condition checks if DataAdr has the same value as the input, and the second part checks if WriteData matches the hexadecimal value. Should both the address and data comply with the conditions, a "Simulation succeeded" message will show up.

```
Transcript
# vsim tb
# Start time: 21:23:53 on Nov 23,2024
# Loading work.tb
# Loading work.Pipeline_top
# Loading work.fetch_cycle
# Loading work.Mux
# Loading work.PC_Module
# Loading work.Instruction_Memory
# Loading work.PC_Adder
# Loading work.decode_cycle
# Loading work.Control_Unit_Top
# Loading work.Main_Decoder
# Loading work.ALU_Decoder
# Loading work.Register_File
# Loading work.Sign_Extend
# Loading work.execute_cycle
# Loading work.Mux_3_by_1
# Loading work.ALU
# Loading work.memory_cycle
# Loading work.Data_Memory
# Loading work.writeback_cycle
# Loading work.hazard_unit
add wave -position insertpoint sim:/tb/*
add wave -position insertpoint sim:/tb/dut/*
VSIM 3> run -all
# ** Note: $finish    : pipeline_tb.v(15)
#    Time: 1200 ps  Iteration: 0  Instance: /tb
# 1
# Break in Module tb at pipeline_tb.v line 15
```

Figure 9 Successful Simulation

As shown in Figure 9 and Figure 10, the successful simulation has been achieved as both the conditions.

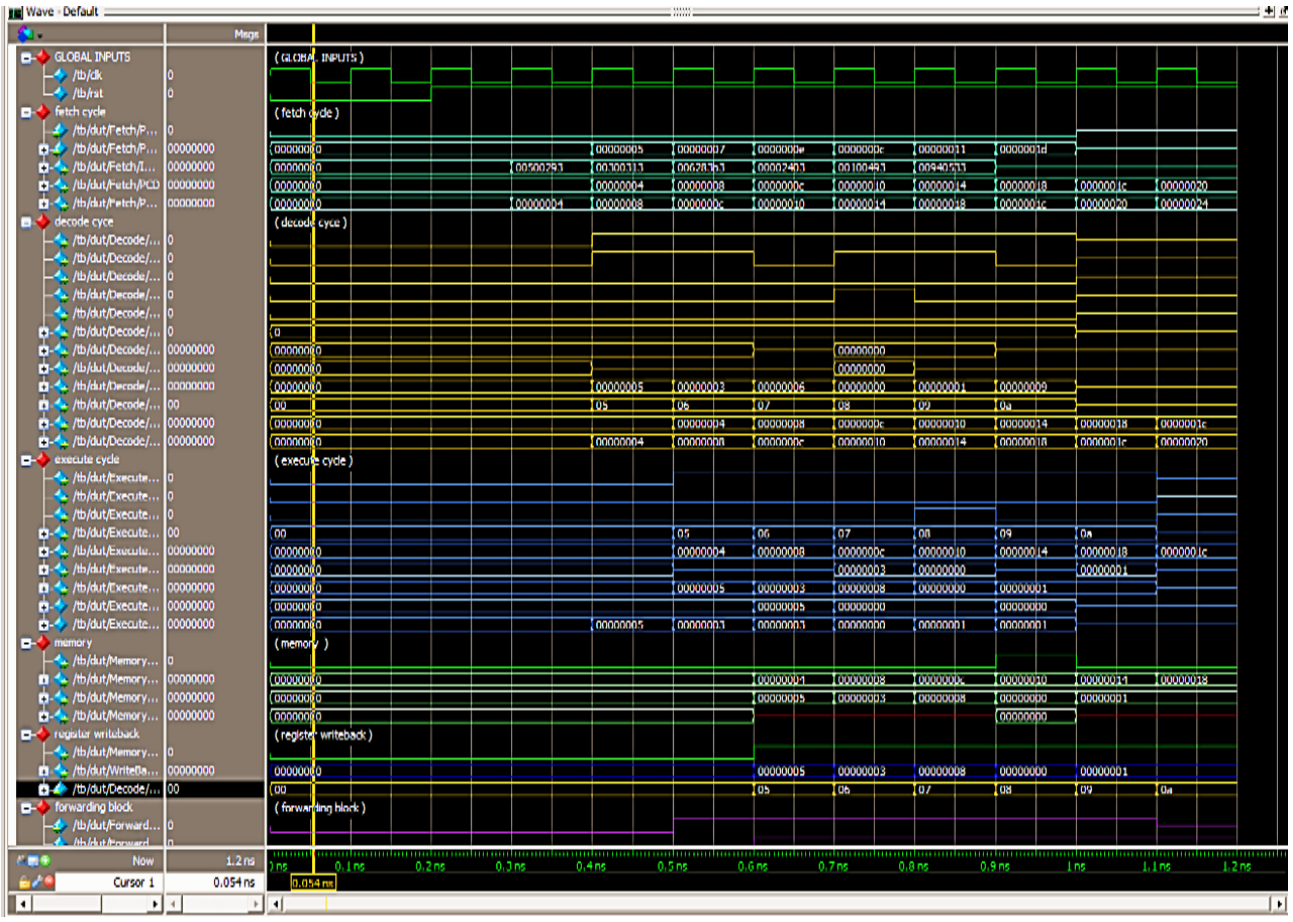

Figure 10 Simulation Results of Pipelined Processor

It serves as the guarantee that everything was simulated during the experiment and is proved by the results snapshots. It looks to see that the actions taken by the simulated program result in the expected memory values and written data. As the simulation is acting accurately based on the given details, the success message has been shown, and the simulation is stopped.

4.7 Conclusion

In short, the use of pipelining helps boosts the efficiency of processing in digital systems. Single-cycle processors are improved in speed by separating them into five stages, which allows several instructions to be ran at the same time [14]. Using this approach, it is possible to get a high clock frequency, but facing data dependency and control flow problems is difficult. This module directs the movement of data and signals from one stage to another. The control unit's role is pivotal, synchronizing and optimizing resources, with various modules ensuring smooth instruction flow and data integrity [8]. Notably, the hazard detection module safeguards against hazards, enhancing execution reliability. The testbenchmodule serves as a validation mechanism, confirming accurate simulation behavior. Overall, pipelining's benefits are underscored by efficient instruction execution and robust processor design.

Chapter 5
Rom IP

5.1 Introduction

But, it's not as simple as that when it comes to memory in computer systems. It appears in many packages and each package has its own functionality. One such form is Read-Only Memory, ROM. In the information hierarchy of the digital systems, ROM forms an essential component that is quite different from other types of memory. But think of it as a conceptual 'safe,' so to speak, for information that isn't subject to constant updating, one that is necessary in order to maintain systems as logical, reliable, and consistently functional. As with RAM, the power is a major issue with ROM because even though it can retain key information it is more suitable for storing program instructions, constant values, or look-up tables.

What differentiate ROM is its credibility in passing out information and data [1]. For instance, in a digital system, all data required in designing the system are input in the ROM. This data is not altered in any manner, or changed dynamically; this makes it a stable data that serves as a support system. It is similar to having a reliable navigator that guarantee every branch of the system what to do, with the help of that stable foundation of the unchangeable data.

ROMs have both a broad variety of shapes and functions and cannot be left unmentioned in terms of the significance to the design of digital systems. Based on the experience of developing ROM, starting from the roots of its creation and ending with the latest practices, it is possible to conclude that it continues to act as an innovational organism in the world of high technologies.

5.2 ROM Module Configuration

5.2.1 Introduction

The strategic implementation of ROM exemplifies how digital designs leverage specialized memory components to optimize performance, reliability, and data management. The generation of this module signifies a pivotal step in digital design. By incorporating a Read-Only Memory (ROM) element, the module contributes to the overall functionality of the system [10].

5.2.2 Rom IP Code

This module is produced using the "altsyncram" wizard and serves to create a ROM element within a digital design. The ROM component is tasked with storing and furnishing data according to a provided address.

```
module rom (
    address,
    clock,
    q);

    input [7:0]  address;
    input    clock;
    output   [31:0]  q;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
    tri1     clock;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
`endif

    wire [31:0] sub_wire0;
    wire [31:0] q = sub_wire0[31:0];

    altsyncram  altsyncram_component (
                .address_a (address),
                .clock0 (clock),
                .q_a (sub_wire0),
                .aclr0 (1'b0),
                .aclr1 (1'b0),
                .address_b (1'b1),
                .addressstall_a (1'b0),
                .addressstall_b (1'b0),
                .byteena_a (1'b1),
                .byteena_b (1'b1),
                .clock1 (1'b1),
                .clocken0 (1'b1),
                .clocken1 (1'b1),
                .clocken2 (1'b1),
                .clocken3 (1'b1),
                .data_a ({32{1'b1}}),
                .data_b (1'b1),
                .eccstatus (),
                .q_b (),
                .rden_a (1'b1),
                .rden_b (1'b1),
                .wren_a (1'b0),
                .wren_b (1'b0));
    defparam
        altsyncram_component.address_aclr_a = "NONE",
        altsyncram_component.clock_enable_input_a = "BYPASS",
        altsyncram_component.clock_enable_output_a = "BYPASS",
        altsyncram_component.init_file = "rom.mif",
        altsyncram_component.intended_device_family = "Cyclone V",
        altsyncram_component.lpm_hint = "ENABLE_RUNTIME_MOD=NO",
        altsyncram_component.lpm_type = "altsyncram",
        altsyncram_component.numwords_a = 256,
        altsyncram_component.operation_mode = "ROM",
        altsyncram_component.outdata_aclr_a = "NONE",
        altsyncram_component.outdata_reg_a = "UNREGISTERED",
        altsyncram_component.widthad_a = 8,
        altsyncram_component.width_a = 32,
        altsyncram_component.width_byteena_a = 1;

endmodule
```

5.3 Altsyncram Component Configuration

The instance of the altsyncram_component represents the ROM memory in practice. It is positioned by setting several factors that describe its behavior and characteristics. These parameters include the width of addresses, the width of data, Clock is controlled, data is initialized and other attributes. The “defparam” statement is used to define the parameters of the “altsyncram_component” instance. Some more parameters include address_aclr_a, clock_enable_input_a, and clock_enable_output_a with values NONE or BYPASS for addressing clearing and other stuffs. The “init_file” parameter describes the MIF file, “intended_device_family” denotes that the used FPGA family as “Cyclone V”, and the “lpm_hint” states that the RAM should not be modified at runtime “lpm_type” describes the low-power mode as “altsyncram”, “numwords_a” and “widthad_a” influences the word count of the ROM and the width of the address bus Other parameters regulate data operations with "outdata_aclr_a" and "outdata_reg_a” defining the output data operation. The width by the name “width_a” is set to 32 and “width_byteena_a” is a 1-bit byte enabled. These parameter configurations acting in concert optimize the “altsyncram_component” instance for fulfilling the design requirements that are necessary for incorporating it into the broader digital system environment.

5.4 Memory Initialization File

Inside ROM, there is a Memory Initialization File (MIF) that serves as a plan of the data that is to be written in ROM and where exactly it is to be written. This MIF is essential for writing the required data into the ROM when the design is being developed and for the correct functioning of the system that this MIF supports. The data in the MIF is stored in hexadecimal form, which is a very convenient way of coding numerical data by the help of numbers from 0 to 9 and letters from A to F. This hexadecimal representation is especially appropriate for digital systems, because it coincides with the use of binary based memory addressing and data storage. By leveraging the MIF, designers can precisely map out the contents of the ROM module, tailoring it to meet specific application requirements.Since this approach is flexible, it can be customized to use the ROM for various purposes, from building up an embedded system to saving tables used for computations. Each command given to the computer is tied to a particular address. It shows where the instruction is stored in the ROM. Basically, the MIF file controls the ROM and helps store the necessary instructions in the proper order.

5.5 Address and Clock Connection

Because the address_a input of altsyncram_component is connected to the address input of the ROM module, the processor can give the address needed to get instructions from the ROM. A second step is to link the clock0 bus in “altsyncram_component” to the clock input of the ROM. Both the ROM and the processor’s timing are always synchronized by this process.

5.6 Conclusion

In conclusion, The ROM stores instructions, and the testbench uses the input address and the clock signal to fetch them one at a time. A MIF file includes the preloaded ROM, and the simulator can reveal the method of fetching instructions.

5.7 Testbench

This module represents the testbench for simulating the functionality of a ROM module. The "ADDRESS_WIDTH" and "DATA_WIDTH" parameters are established to determine the width of the address and data signals used within the testbench.

```
`timescale 1ns / 1ps
module rom_tb;

  // Parameters
  parameter ADDRESS_WIDTH = 8;
  parameter DATA_WIDTH = 32;

  // Inputs
  reg [ADDRESS_WIDTH-1:0] address;
  reg clock;

  // Outputs
  wire [DATA_WIDTH-1:0] q;

  // Instantiate the DUT (Design Under Test)
  rom dut (
    .address(address),
    .clock(clock),
    .q(q)
  );

  // Initialize inputs
  initial begin
    address = 0;
    clock = 0;
  end

  // Toggle the clock every 5 time units
  always #5 clock = ~clock;

  // Monitor the outputs
  always @(posedge clock) begin
    $display("Address = %h, Data = %h", address, q);
    // Increment the address by 4 for each clock cycle
    address = address + 4;
  end

  // Stop the simulation after a certain number of clock cycles (optional)
  initial #100 $stop;

endmodule
```

5.7.1 Input and Output Signals Setup

The module also declares input signals “address” – the input through which the memory address is supplied to the ROM module under test, and “clock” – the clock input. Also, an output signal “q” has been specified to denote the data out signal of the ROM.

5.7.2 Monitoring Output and Address Increment

In another “always” block which is activated on positive clock edge, the simulation observes the outputs by printing out the current values of “address” and “q”. During each clock cycles the value of “address” increases by 4 in order to emulate clock cycles movements through the memory addresses.

5.7.3 Simulation Termination

The simulation is deliberately set up in a manner whereby it shall only take 100 clock cycles to complete by use of the $stop command. Thus it can deliberately stop the simulation after a certain clock cycle has been specified to have completed. This feature allows a selected scrutiny on the sequence of instructions fetched and processed within the simulated digital system which can provide significant knowledge of the system’s performance and operations.

5.7.4 Conclusion

Overall, the testbench creates a model of a system in which a ROM module is a part to check its functionality when reading data based on addresses and during successive clock cycles. This arrangement allows the identification of a change in the address values of the data, as well as the reading of the corresponding data in the ROMs in each clock cycle.

5.8 RTL Simulation

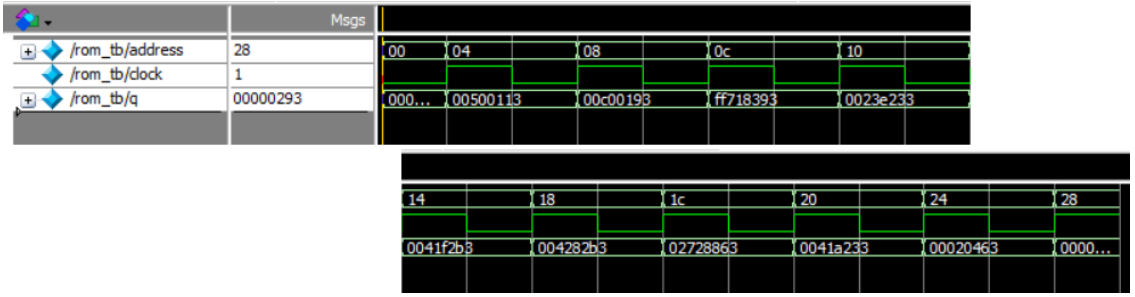

Figure 11 RTL Simulation of ROM IP

As shown in Figure 11 and Figure 12, the instructions are successfully stored on the respective addresses as given in the .mif file.

```
Transcript
# Loading work.rom_tb
# Loading work.rom
# Loading altera_mf_ver.altsyncram
# Loading altera_mf_ver.altsyncram_body
# Loading altera_mf_ver.ALTERA_DEVICE_FAMILIES
# Loading altera_mf_ver.ALTERA_MF_MEMORY_INITIALIZATION
VSIM 23> run -all
# Address = 00, Data = 00000000
# Address = 04, Data = 00500113
# Address = 08, Data = 00c00193
# Address = 0c, Data = ff718393
# Address = 10, Data = 0023e233
# Address = 14, Data = 0041f2b3
# Address = 18, Data = 004282b3
# Address = 1c, Data = 02728863
# Address = 20, Data = 0041a233
```

Figure 12 Successful Simulation

5.9 Conclusion

To help us to understand it let us remember that in the world of the digital systems, various types of memory have their work to do. Rom is one important memory type; it stays constant, holds data and makes up strong systems since the information is system information that does not change. ROM, even without operating power, contains such significant data as instructions and constants. In system design phase, ROM has significant function as it provides continuant information to different sections. ROM is whatever type, has numerous functions, and is needed from the onset until now.

In other words, through the loaded instruction and test bench simulations, the manner in which data is taken is exemplified in ROM. The program interacts with the test to determine how it provides

information with addresses and time. This planned setup verifies how addresses transform and data is retrieved, which reemphasizes that ROM plays a large part within the system architecture.

Chapter 6

UART (Universal Asynchronous Receiver Transmitter)

6.1 Introduction

They are referred as the Universal Asynchronous Receiver-Transmitter (UART) that is a serial hardware interface used among devices [23]. UART works in an asynchronous mode, so there no need for a shift clock signal to be sent from the transmitting device to the receiving one. Both devices, however, use a fixed baud rate to ensure that the sender and receiver are at a similar data transferring frequency [12].

UART helps in shifting data from parallel form that is available from a computer or microcontroller into serial form of data which can be transmitted or to shift back data form serial form to parallel form in the case of receiving data [10]. Data that is transmitted comes in a format; often, it has a start bit, data bits, parity bit, and one or more than one stop bits as shown in figure 13.

6.2 Components of UART

All the sub parts of a UART (Universal Asynchronous Receiver-Transmitter) enable the data to be transmitted serially between two devices [15], [23]. These components include the conversion of parallel data, into serial formats for transmitting and vice versa for receiving. The key components of a UART are:

6.2.1 Transmit Shift Register (TSR)

This register retains the data to be transmitted. Data is transferred out in parallel form shifted serially starting with the least significant bit (LSB). It also guarantees that the data are in the correct format with start bits, data bits, optional parity, and stop bits.

6.2.2 Receive Shift Register (RSR)

This register takes in serial data in terms of bits at a time. If the complete frame (start bit, data bits, parity, and stop bit) is ascertained then the received data is transferred to the Receive Buffer, where it is processed in parallel.

6.2.3 Transmit Buffer

A holder of copies of parallel data which are to be transmitted. Data access is also controlled by queuing data before it is transmitted by the Transmit Shift Register which makes the flow of the data more efficient.

6.2.4 Receive Buffer

Saves the complete frame received by the Receive Shift Register. When it is completely received the parallel data is processed or passed to the connected device.

6.2.5 Baud Rate Generator

Controls the clock signal that drives the baud rate of data transfer rate in this case. Controls the data rate which in turn coordinate the working of the transmitter and the receiver sections.

6.2.6 Parity Generator/Checker

Originally used to generate, or to verify the carriers of error checking bits known as parity bits. Assists in detection of single bit error in the transmitted or received data bits.

6.3 UART Frame Format:

Figure 13 shows the arrangement of states of UART's frame (idle, start, stop, data bits and parity) discussed above:

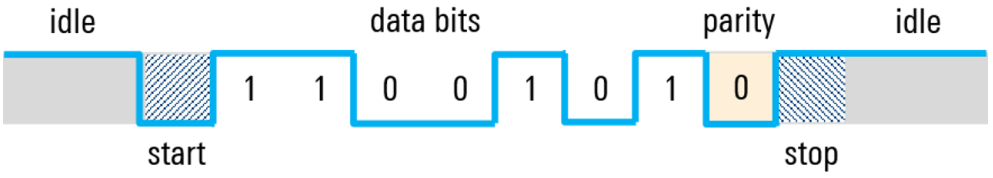

Figure 13 UART Frame Format

6.4 UART Simulation

6.4.1 Transmitter

The waveform below illustrates the simulation of a UART (Universal Asynchronous Receiver/Transmitter) transmitter. On the left side of the image, the signals from the **UART_TX_TB (testbench)** include the **clk (clock)**, **reset**, **tx_start**, **tx_tick**, and **tx_done_tick**, which control and monitor the transmission process. On the right side, the **UART_TX_DUT (device under test)** signals are displayed, showing the **state_reg**, **state_next**, and **tx_reg**, which help manage the flow and execution of data transmission as shown in figure 14 and figure 15.

The **clk** signal is the main timing reference that controls when actions occur, while the **reset** signal initializes the system to a known state. The **tx_start** signal is asserted to trigger the start of the transmission, and **tx_tick** determines the timing of bit shifts during data transmission. The **tx_done_tick** indicates when the transmission has been completed. The **tx** signal represents the actual serial data output, where individual data bits are transmitted one at a time.

Data to be transmitted is stored in the **tx_reg**, with examples such as 0xA5 and 0x52 appearing in the waveform. The transmission begins once **tx_start** is asserted, and the bits are shifted out serially. The **state_reg** reflects the various states of the transmitter, such as being idle or actively sending data bits. This process follows the UART frame format, which typically includes a start bit, several data bits, an optional parity bit, and a stop bit, ensuring reliable communication between devices. The simulation verifies the correct operation of the transmitter, with proper timing and sequencing of the transmitted data.

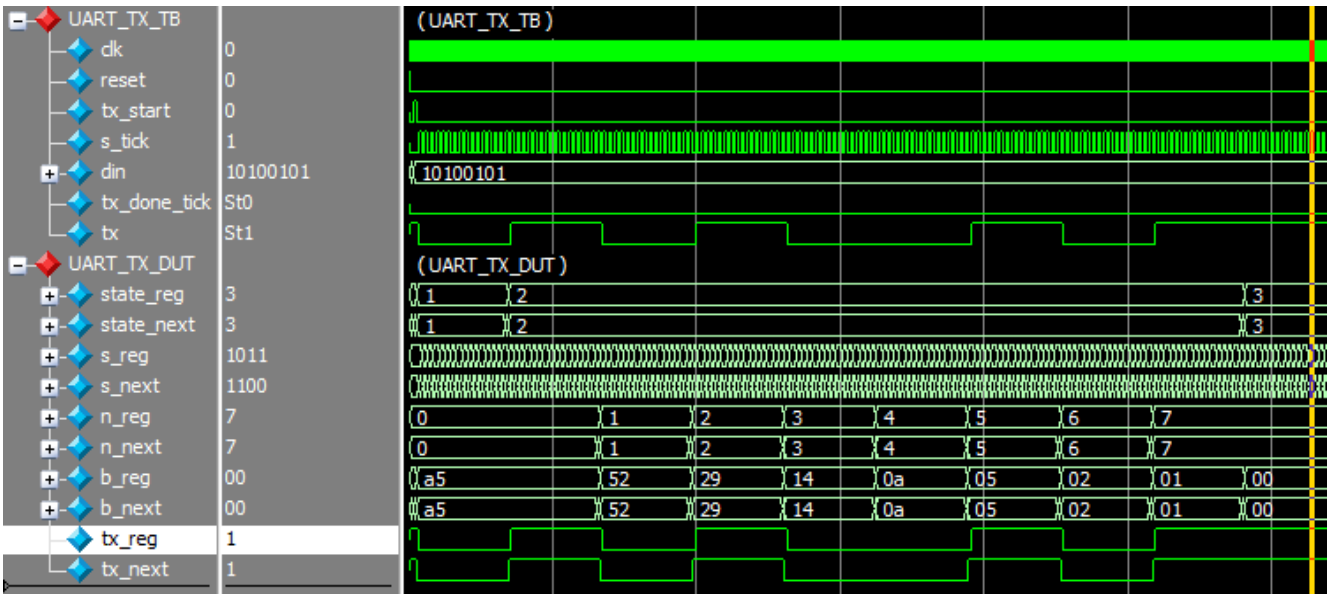

Figure 14 Data driven on channel port 'tx' from port 'din'

```
VSIM 4> run -a;
#  tx_start: 0 | din: 00000000 | tx_done_tick: 0 | tx: 1
#  tx_start: 1 | din: 10100101 | tx_done_tick: 0 | tx: 1
#  tx_start: 0 | din: 10100101 | tx_done_tick: 0 | tx: 1
#  tx_start: 0 | din: 10100101 | tx_done_tick: 0 | tx: 0
#  tx_start: 0 | din: 10100101 | tx_done_tick: 0 | tx: 1
#  tx_start: 0 | din: 10100101 | tx_done_tick: 0 | tx: 0
#  tx_start: 0 | din: 10100101 | tx_done_tick: 0 | tx: 1
#  tx_start: 0 | din: 10100101 | tx_done_tick: 0 | tx: 0
#  tx_start: 0 | din: 10100101 | tx_done_tick: 0 | tx: 1
#  tx_start: 0 | din: 10100101 | tx_done_tick: 0 | tx: 0
#  tx_start: 0 | din: 10100101 | tx_done_tick: 0 | tx: 1
#  tx_start: 0 | din: 10100101 | tx_done_tick: 1 | tx: 1
#  tx_start: 0 | din: 10100101 | tx_done_tick: 0 | tx: 1
# ** Note: $stop    : uart_tx_tb.sv(68)
#    Time: 3235 ns  Iteration: 1  Instance: /uart_tx_tb
# Break in Module uart_tx_tb at uart_tx_tb.sv line 68
```

Figure 15 Transcript showing the data being driven on the channel port

6.4.2 Receiver

This waveform shows the simulation of a UART (Universal Asynchronous Receiver/Transmitter) receiver. The left side displays the testbench signals such as **clk (clock)**, **reset**, **rx (received data)**, **rx_tick**, **rx_done_tick**, and **data_in**. On the right side, the **UART_RX_DUT (device under test)** signals are shown, including **state_reg**, **state_next**, **s_reg**, **n_reg**, **b_reg**, and the received data bits.

The **clk** signal the timing reference, while **reset** initializes the system. The **rx** signal represents the serial data received by the receiver, and the **rx_tick** controls the timing of the bits as they are shifted in. The **rx_done_tick** indicates when the reception of the data is complete. The **data_in** signal shows the received data as a sequence of bits, such as 1010101.

These two variables, state_reg and state_next, control how the receiver functions and guarantees data is received as demonstrated in figure 16 and 17. Data is stored in its serialized format in the s_reg, and then it is aligned during the reception process. The incoming bits are stored in n_reg and b_reg is what is stored in the receiver buffer. The diagram reveals that UART_RX_DUT is receiving bits in the order 0xA5 and 0x52. The simulation makes sure that the data coming from the UART is handled and stored as expected.

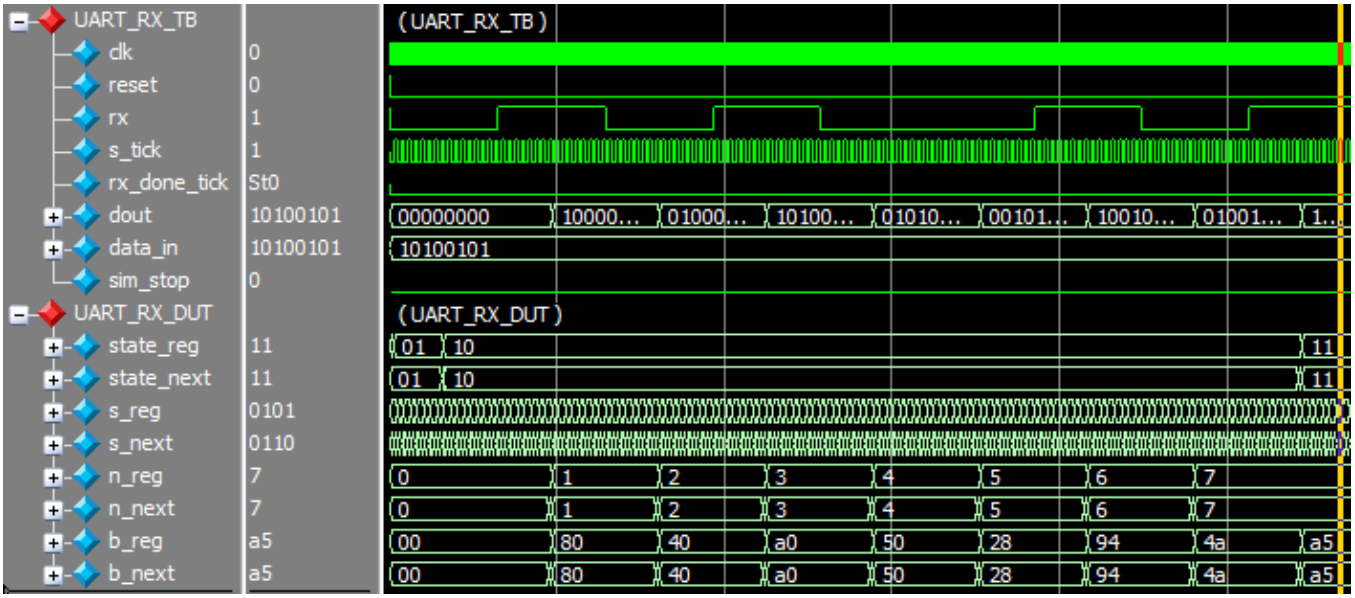

Figure 16 Data captured from channel port 'rx' as seen in port 'dout'

```
VSIM 15> run -a
# data_in=xx | RX: 1 | data_out: 00 | rx_done_tick: 0
# data_in=a5 | RX: 0 | data_out: 00 | rx_done_tick: 0
# data_in=a5 | RX: 1 | data_out: 00 | rx_done_tick: 0
# data_in=a5 | RX: 1 | data_out: 80 | rx_done_tick: 0
# data_in=a5 | RX: 0 | data_out: 80 | rx_done_tick: 0
# data_in=a5 | RX: 0 | data_out: 40 | rx_done_tick: 0
# data_in=a5 | RX: 1 | data_out: 40 | rx_done_tick: 0
# data_in=a5 | RX: 1 | data_out: a0 | rx_done_tick: 0
# data_in=a5 | RX: 0 | data_out: a0 | rx_done_tick: 0
# data_in=a5 | RX: 0 | data_out: 50 | rx_done_tick: 0
# data_in=a5 | RX: 0 | data_out: 28 | rx_done_tick: 0
# data_in=a5 | RX: 1 | data_out: 28 | rx_done_tick: 0
# data_in=a5 | RX: 1 | data_out: 94 | rx_done_tick: 0
# data_in=a5 | RX: 0 | data_out: 94 | rx_done_tick: 0
# data_in=a5 | RX: 0 | data_out: 4a | rx_done_tick: 0
# data_in=a5 | RX: 1 | data_out: 4a | rx_done_tick: 0
# data_in=a5 | RX: 1 | data_out: a5 | rx_done_tick: 0
# ------------NORMAL STOP-------------
# ** Note: $stop    : uart_rx_tb.sv(56)
#    Time: 2885 ns  Iteration: 1  Instance: /uart_rx_tb
# Break in Module uart_rx_tb at uart_rx_tb.sv line 56
```

Figure 17 Transcript showing the data captured from channel port

6.5. Conclusion:

By including a UART in SystemVerilog, you can see how digital design principles are used to allow serial communication between hardware parts. The design works just like UART, letting you adjust the sweetheart, send information grouped by start and stop bits, and make sure data gets sent and received reliably. By using the ability to write modules in isolation and run them at once in SystemVerilog, we made the UART structure both understandable and scalable for testing and verification. The functionality was checked by running tests on a special test setup that made sure the data went through properly, was caught when it didn't come in right, and worked as expected in different situations. It can be part of bigger SoC projects or used in communication systems since it helps connect processors, memory, and peripherals over serial channels [20]. By working on the project, people can see that timing synchronization, protocol handling, and hardware-level communication are important in digital and embedded systems because of UART.

Chapter 7

FIFO (First In First Out)

7.1 Introduction:

In digital systems, First-In, First-Out (FIFO) is a popular way of temporarily holding data, and it works by letting the first piece of data you add in come out first. FIFOs are important in situations where data needs to be held and moved between parts of a system that work at different speeds, helping to make sure the data goes through without getting lost or mixed up.

A FIFO buffer uses two main markers, or pointers, to keep track of where data is coming in and where it is going out. the write pointer shows where the next piece of data will go, and the read pointer shows where the program should get a piece of data from. The logic sees to it that data is read from start to finish just as it was written, keeping things in the right order. FIFOs are put to use in hardware by using registers or memory banks together with simple control circuits, and you can find them being used in areas like communication systems, data collection, and processing images.

In the context of this project, a FIFO module helps make sure data is shared between parts of the program in an organized and smooth way. Its design, verification, and integration are important parts to make sure all its pieces work well with the rest of the system.

7.2 UART FIFO Simulation:

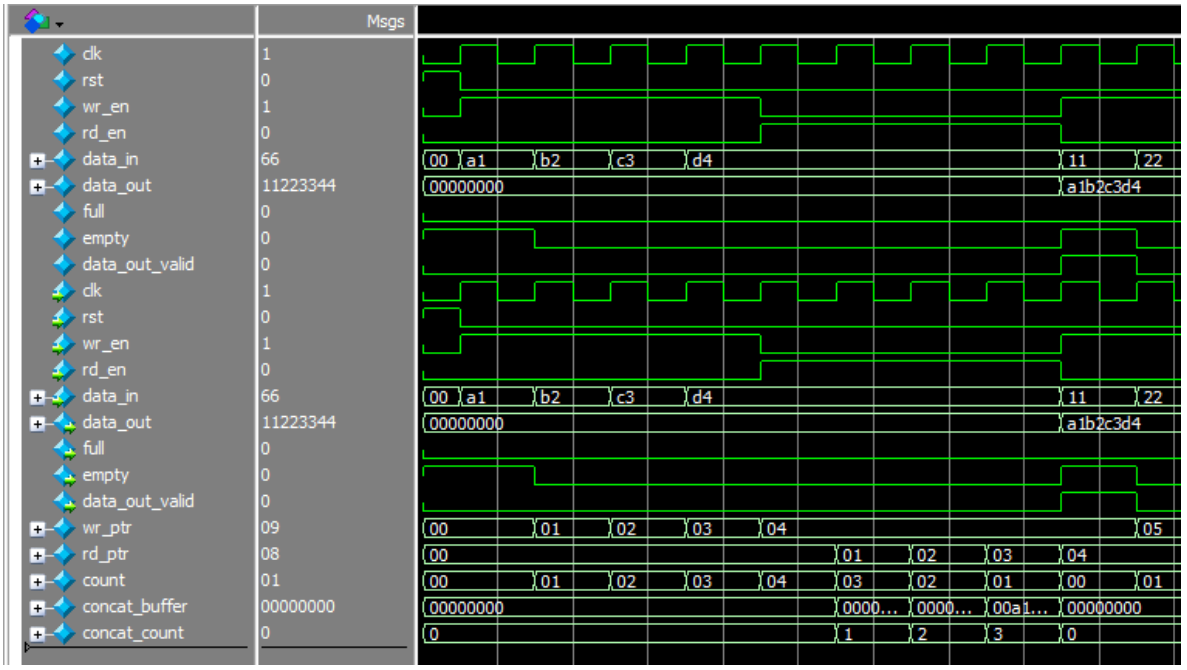

Figure 18 UART FIFO Waveforms

```
# Loading work.fifo_uart_tb
# Loading work.fifo
# [TBs] ---Test PASSED: data_out = a1b2c3d4, expected = a1b2c3d4
# [TBs] ---Test PASSED: data_out = 11223344, expected = 11223344
# [TBs] ---Test PASSED: data_out = 55667788, expected = 55667788
# ** Note: $stop    : fifo_uart_tb.sv(95)
#    Time: 255 ps  Iteration: 1  Instance: /fifo_uart_tb
```

Figure 19 UART FIFO Transcript View

7.3 I2C FIFO Simulation:

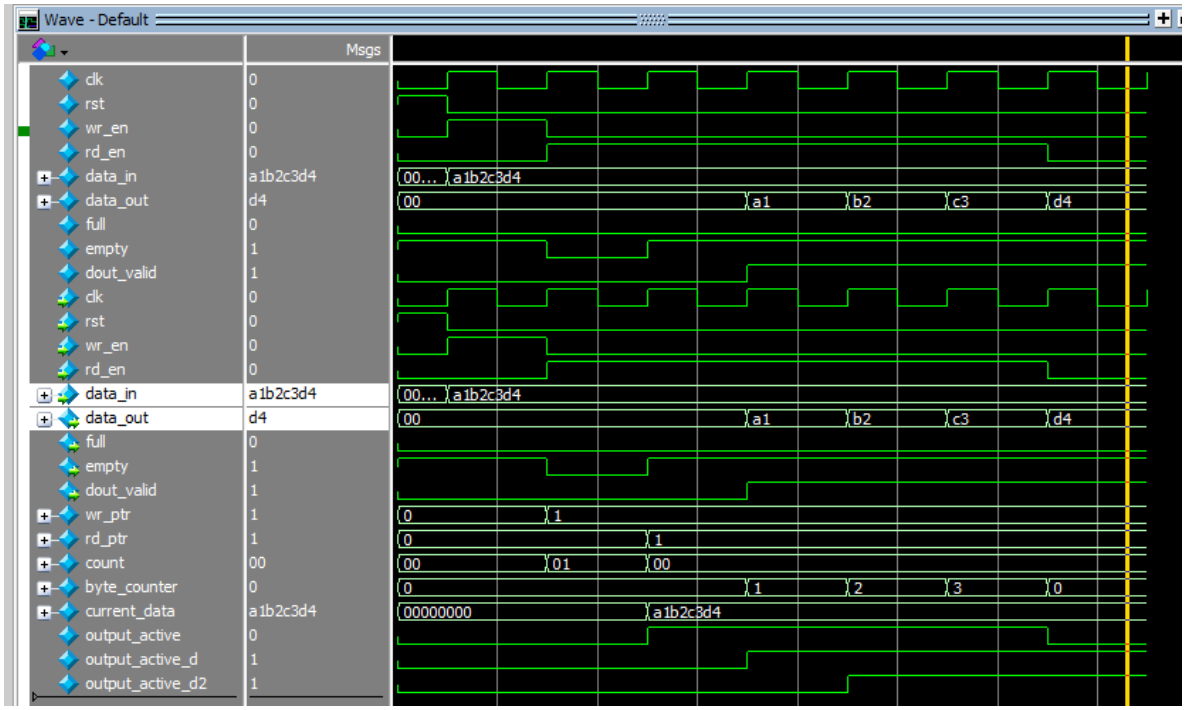

Figure 20 I2C FIFO Waveforms

```
# Loading work.fifo
# [TBs] ---Test PASSED: data_out = a1b2c3d4, expected = a1b2c3d4
# [TBs] ---Test PASSED: data_out = 11223344, expected = 11223344
# [TBs] ---Test PASSED: data_out = 55667788, expected = 55667788
```

Figure 21 I2C FIFO Transcript View

7.4 Conclusion:

The use of the FIFO buffer in this project has shown to be a good way to handle data moving between different parts of a digital system, especially when dealing with clock signals or speeds that aren't the same for everything. The FIFO design helps make sure data gets transferred in the right order and without losing any bits, because it keeps track of the input and output in the right order. Developed using SystemVerilog, the FIFO module has features that can be changed like how many elements it holds, knowing when it's full or empty, and making sure read and write happening at the right times. The simulation results show that the FIFO works properly when things are normal or near the edges, including managing how data is stored and moved right on time [13]. This design doesn't just work well for what it's supposed to do, but it also has a simple structure that can handle more work in the future and be added to things like communication tools, picture processing systems, or microprocessor designs. Overall, the FIFO buffer helps make sure the computer can handle data smoothly and without problems in the system we built for this project.

Chapter 8

I2C (Inter-Integrated Circuit)

8.1 Introduction:

The I²C protocol, a popular way for devices to communicate, only uses two cables to connect different integrated circuits. SDA and SCL are the two main pins found in I^{2}C protocols. Thanks to the design by Philips (now NXP), I²C allows multiple masters and slaves to talk to each other, making it fit for communicating several peripherals with a microcontroller. For this project, the I²C protocol was modeled in SystemVerilog, resulting in a hardware description of how any device might act as an I²C master or slave [8]. Among the features of the design are the start and stop conditions, the ability to recognize addresses, data transfer synchronization, and handling acknowledgments. Great care is taken to make sure the timing and state changes needed by I²C are accurate. System Verilog is well-equipped for designing and checking the I²C controller due to its rich set of modeling and simulation tools. It forms a main link for low-speed data transmission over short distances and is used with bigger systems for working with sensors, reading EEPROM data, or setting up devices.

8.2 Components of I²C (Single Master-Slave Configuration)

The I²C protocol used in this project consists of one master and one slave communicating at a speed of 100 kHz. Since the design is simple, it is easy to grasp and suitable for teaching and simple communication. What this design consists of will be discussed below [5].

8.2.1 I²C Master Controller:

The I²C Master Controller leads the way and takes charge of carrying out all necessary communication activities. It provides the necessary signals to guide the beginning and end of all I²C communications. The master sends the 7-bit address of the slave device along with a read/write control bit, followed by one or more data bytes. It also monitors the acknowledgment (ACK) bit sent by the slave after each byte to ensure successful communication. Additionally, the master generates the clock signal (SCL), which is shared with the slave to synchronize data transmission.

8.2.2 I²C Slave Device:

The I²C Slave Device, implemented in its most basic form, passively listens for communication initiated by the master. It compares the received address with its own and responds with an acknowledgment if there is a match. Depending on the operation specified (read or write), the slave either receives data from the master or sends data back to it. The slave also includes control logic to manage the SDA line properly during data and acknowledgment phases, ensuring open-drain compliance and collision-free communication.

8.2.3 SDA and SCL Lines:

The SDA (Serial Data) Line Control Logic ensures that data transfer between the master and slave occurs on a single shared line. Since I²C uses open-drain signaling, the SDA line must be managed in such a way that only one device drives it at a time while the other remains in a high-impedance state. With this logic, the right moment for transmitting data is ensured, along with proper arbitration and security for the data.

The SCL Generator is a part of the master controller and supplies the master clock needed for synchronizing communication over the SDA line. Working at a low frequency, the clock makes sure that data is collected only at the correct moments, keeping within the setup and hold rules of the I2C standardand hold durations.

8.2.4 Top Module Wrapper:

Lastly, Both the master and slave use a Finite State Machine (FSM) to direct the steps of the protocol. There are six stages in the FSM that the master handles, namely idle, start, address, data, acknowledgment, and stop. In the same way, the slave's FSM searches for the beginning of transmission, recognizes its address, moves data back and forth, and answers with acknowledgments. That means these FSMs manage the process so that all the actions in the I²C communication protocol are taken as planned and checked.

8.3 I2C Protocol Frame Format:

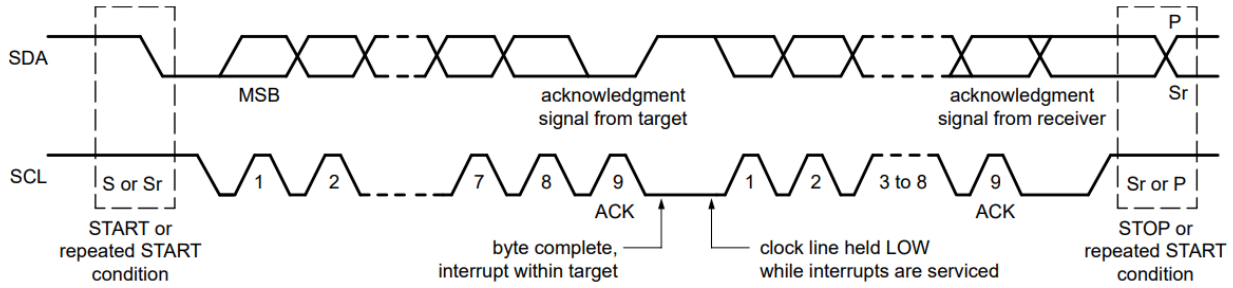

Figure 22 I2C Timing Diagram

8.4 I2C Simulation:

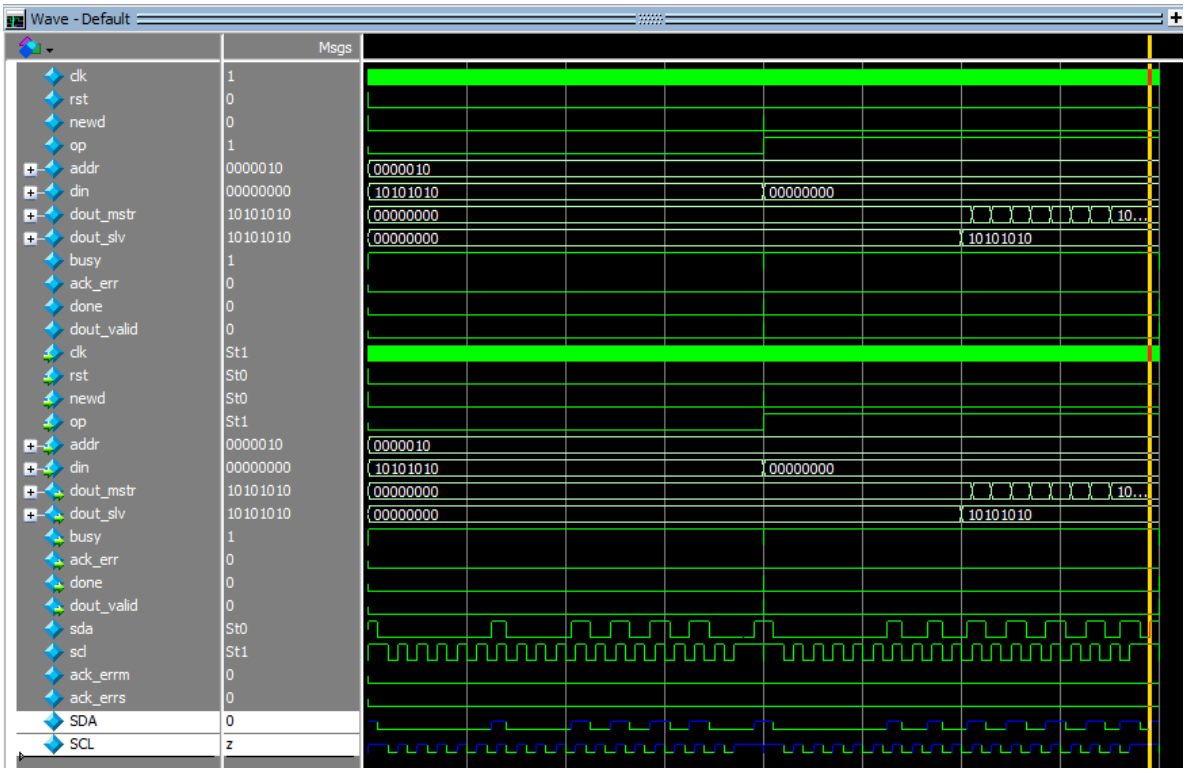

Figure 23 I2C Top Module Simulation

```
#    Time: 0 ps  Iteration: 0  Instance: /i2c_top_tb File: i2c_top_tb.sv
# [TB] Reset complete
# [TB] ---Write: Addr=2, Data=170---
# [TB] ---Read: Addr=2, Data=170---
# [TB] ---Verification PASSED: Addr=2, Data=170---
# [TB] ################################################
# ** Note: $stop    : i2c_top_tb.sv(106)
#    Time: 160135 ns  Iteration: 2  Instance: /i2c_top_tb
# Break in Module i2c_top_tb at i2c_top_tb.sv line 106
```

Figure 24 I2C Top Module Transcript View

8.5 Conclusion:

Using the I²C protocol in SystemVerilog makes it easy to understand how two digital devices can exchange information using a simple link. The project was intended for small tasks that only need a basic I²C setup with a single master and single slave communicating at the lowest available speed. With this design, the abilities to generate start/stop conditions, send addresses, exchange data, deal with acknowledgments, and synchronize clocks were achieved. The I²C standard was met by designing with SystemVerilog since it enabled a clear and reliable description of the circuit's functions and timing. A developed I²C module forms the base that can later be enhanced to allow for multi-master communication, higher data transmission speeds, plus better management of errors. In essence, the project proves that I²C performs well as a low-speed short-range digital communication protocol and highlights that knowledge of the protocol is valuable in digital designs.

Chapter 9

Applications

9.1 Introduction:

Development in embedded systems has played a key role in progressing many industries by supporting real-time information processing and communication between different devices and systems. This project looks at setting up a RISC-V processor-based system using UART and I2C, giving it the ability to handle data in embedded systems effectively [14]. The fact that this system connects communication with data processing opens the door for its use in many different fields.

Industrial Automation, IoT Devices, Medical Devices, Automated Test Equipment, and Wearable Devices are among the main fields where such a system is applied. Every aspect of the system gains from having sensor data received, processed promptly, and then displayed as control signals, feedback, or remote communication.

Thanks to the use of UART for input, a RISC-V processor for computations, FIFO buffers for data control, and I2C for sending outputs, the system can be adjusted to the needs of any application. The next sections will explain how FS comes into play in different fields, highlighting its capability and how useful it is in practice.its versatility and potential for real-world impact.

9.2 Embedded Systems in Industrial Automation:

9.2.1 Application Overview:

Industries today rely on industrial automation, as it handles machinery and reviews data from sensors that record temperature, pressure, and motor operation. With the help of data collected from sensors, these systems improve machine performance, keep machines running minimally, and guarantee safety during operations.

9.2.2 System Application:

1. **UART for Sensor Data Input:** UART is a common method used by industrial machinery to receive information from sensors installed outside of the equipment. Industries find UART useful because it is easy to use and can communicate over long distances.

2. **RISC-V Processor for Data Processing:** The data transmitted from the UART becomes processed by the RISC-V processor. Should the temperature in the reactor go above a certain point, sensors detect this and actions are taken to keep the situation safe.

3. **I2C for Output Control:** Processed info is then sent using I2C buses to run the actuators, display screens, or other important devices within the system. Since I2C is efficient and allows many devices to interact, it is well-suited for managing things like motor controllers and displays.

4. **FIFO for Buffering:** The use of FIFO buffers results in easy and fast movement of data among system stages. In industrial situations, this buffering process manages continuous incoming information and prevents any loss or delays.

Thanks to having UART, RISC-V, FIFO buffers, and I2C in a modular system, it is very helpful for embedded systems in the field of industrial automation. Using a system, one can efficiently monitor systems, handle real-time data, and activate safety measures, which leads to more efficiency and better safety in factories [20].

9.3 IoT (Internet of Things) Devices:

9.3.1 Application Overview:

These devices are built to be small and energy efficient and to connect and transfer data with different devices or to the cloud. Smart home systems, monitoring the environment, and fitness tracking are just some areas where these devices are found.

9.3.2 System Application:

1. **UART for Communication with Sensors:** Most of the time, IoT devices use UART for exchanging data with sensors placed outside the device. As a simple use case, a weather station can use UART to receive data from sensors that measure temperature, humidity, and air quality. The data sent through UART is transferred to the RISC-V processor to be processed.

2. RISC-V Processor for Data Computation: The RISC-V chip looks at the sensor data and performs operations like calculating averages, spotting strange results on sensors, or generating warnings depending on the information it receives.

3. I2C for External Communication: After processed, you can output the results to a display or deliver them to a cloud gateway for further processing or alerts to the user. The ability of I2C to control many devices at the same time means it is perfect for communication in IoT systems.

4. FIFO for Buffering: It is the role of FIFO buffers to manage sensor data in real time and prevents any data from being missed or delayed. IoT systems that receive a lot of real-time data rely greatly on this connection.

Being able to communicate via UART, process information with the RISC-V processor, and output using I2C is an excellent match for IoT devices. It's well suited for use in IoT for embedded systems, as it handles communication with multiple sensors and devices quickly, while also saving power.

9.4 Medical Devices:

9.4.1 Application Overview:

For example, ECG machines, blood glucose meters, and pulse oximeters use real-time data analyses to track vital health markers. They have to give accurate and reliable outcomes even when there is little time to do the tests.

9.4.2 System Application:

1. UART for Sensor Data Reception: Often, doctors use UART to transport information from sensors in devices, for example ECG electrodes and glucose meters. Using UART, the system is able to pick up serial data from these sensors with ease.

2. RISC-V Processor for Data Analysis: After receiving data from the sensors, the RISC-V processor carries out various tasks such as filtering ECG, calculating heart rate, or determining the amount of glucose from the recorded measurements.

3. I2C for Output (Display/Peripheral Communication): The system transmits the processed data to a display connected by I2C or to external devices for further use. It could also mean showing the outcomes in real time for medical staff, or shipping the data to a centralized system for observation.

4. **FIFO for Data Buffering:** FIFO buffers help the system handle data incoming from sensors so the system can process it in real-time without missing anything. It is essential for systems that always read data, since the accuracy and timing of the ECG machine matter a lot.

It is very important for medical devices to process and show or send medical data in real-time. Having UART for data input, RISC-V to carry out computations, FIFO buffers to organize data, and I2C for connection makes sure that important health data is processed quickly, safely, and with no errors.

9.5 Automated Test Equipment:

9.5.1 Application Overview:

ATE is used in a number of sectors to examine and confirm the operation of different components and systems. These systems work best when there is precise data collection, analysis, and presentation of output to verify that the tested devices are behaving properly.

9.5.2 System Application:

1. **UART for Test Equipment Communication:** ATE systems communicate with DUTs mainly using the UART interface. Data or outcomes from the DUTs are first transferred to the UART module and then to the RISC-V processor for processing.

2. **RISC-V Processor for Data Processing:** The processor processes the data it gets from the DUTs, specifically by recording voltage, current, and resistance levels. This helps to ensure that testing is accurate when components are used under different circumstances.

3. **I2C for Test Results Output:** After that, the outcomes are transmitted to a display or recorder connected via I2C. I2C is used here because many different devices, such as screens and external storage, can be easily connected and exchanged data over it.

4. **FIFO for Data Buffering:** FIFO buffers help smooth out the processing of numerous incoming test results. It is especially vital in settings where many devices are running tests together, avoiding mistakes where data may be dropped and allowing the system to handle and pass large amounts of data.

The efficient management of data is vital for automated test equipment to test accurately and fast. The set-up of UART for receiving input, RISC-V for processing, FIFO for handling data, and I2C for output

is ideal for ATE applications where exact and timely data handling matters for confirming the correct operation of systems and components.

9.6 Wearable Devices:

9.6.1 Application Overview:

Fitness trackers, smartwatches, and health devices are some examples of wearables that gather their users' health and fitness data. The units are meant to be compact and power-saving and to give users instant feedback.

9.6.2 System Application:

1. **UART for Sensor Input:** Wearable devices usually use UART for sending and receiving data from different sensors, such as those for measuring acceleration or the heart rate. Data from these sensors goes to the processing unit to calculate how many steps have been taken or the current heart rate.

2. **RISC-V Processor for Real-Time Processing:** The RISC-V processor instantly analyses information sent by sensors, for example, counting the user's steps using accelerometer data or tracking their heart rate for unusual changes.

3. **I2C for Output to Display:** After being processed, the data goes to an I2C-connected screen, such as an OLED or LCD, so users can monitor their health measurements. The processed data can also be shared over I2C with other devices for additional recording or analyzing.

4. **FIFO for Data Buffering:** With FIFO buffers, data coming from sensors is controlled, leading to smooth and non-stop handling of the information. This applies mainly to wearable devices that constantly handle data on movement or heart rate.

Thanks to using UART for sensors, the RISC-V processor for calculations, I2C for output, and FIFO for data control, these devices are both efficient and power-saving in wearable applications.

9.7 Conclusion:

By looking at Industrial Automation, IoT Devices, Medical Devices, Automated Test Equipment, and Wearable Devices, we can see how flexible the system is. This architecture brings together UART, RISC-V, FIFO buffers, and I2C, making it suitable for several real-life domains. They demonstrate how

the system is helpful for real-time data processing and helps explain its value in several industries that rely on embedded data. industries requiring embedded data solutions.

References

[1] S L Harris and D M Harris, Digital Design and Computer Architecture, RISC-V ed., United States of America: Elsevier. Accessed: Jan. 23, 2023. [Online].

[2] J. Bhasker, "A Verilog HDL Primer," Star Galaxy Publishing, 2005.

[3] P. Flake, "SystemVerilog for Verification: A Guide to Learning the Testbench Language Features," Springer, 2008.

[4] K. Agarwal and S. Singh, "SystemVerilog for Verification: A Guide to Component Design and Verification with SystemVerilog," Springer, 2011.

[5] P. Wilson and A. Wajs, "Advanced SystemVerilog Assertions," Springer, 2013.

[6] R. Chaboyer, "SystemVerilog Assertions and Constraints," Springer, 2016

[7] K. Morris and J. Rowson, "Evaluating Single Cycle Implementations of Superscalar and VLIW Processors," in Proceedings of the 36th International Symposium on Computer Architecture (ISCA '09), pp. 105-116, 2009.

[8] P. P. Pande and K. Roy, "A Single Cycle MIPS Processor with Pipelined Data Paths," in Proceedings of the 11th ACM Great Lakes Symposium on VLSI (GLSVLSI '01), pp. 73-78, 2001.

[9] J. Hennessy and D. Patterson, "Computer Architecture: A Quantitative Approach," 5th edition, Morgan Kaufmann Publishers Inc., 2011.

[10] P. P. Chu, FPGA Prototyping by Verilog Examples: Xilinx Spartan-3 Version. Hoboken, NJ, USA: Wiley-Interscience, 2008.

[11] L. Poli, S. Saha, X. Zhai, and K. D. Maier, "Design and Implementation of a RISC V Processor on FPGA," 2021 17th International Conference on Mobility, Sensing and Networking (MSN), Exeter, United Kingdom, 2021.

[12] J. Chen and S. Huang, "Analysis and Comparison of UART, SPI and I2C," 2023 IEEE 2nd International Conference on Electrical Engineering, Big Data and Algorithms (EEBDA), Changchun, China, 2023.

[13] J. -Y. Lai, C. -A. Chen, S. -L. Chen, and C. -Y. Su, "Implement 32-bit RISC-V Architecture Processor using Verilog HDL," 2021 International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS), Hualien City, Taiwan.

[14] F. Hussain and S. Sarkar, "Design and FPGA Implementation of Five Stage Pipelined RISC-V Processor," 2024 IEEE 9th International Conference for Convergence in Technology (I2CT), Pune, India.

[15] A. K. Gupta, A. Raman, N. Kumar, and R. Ranjan, "Design and Implementation of High-Speed Universal Asynchronous Receiver and Transmitter (UART)," 2020.

[16] E. Cui, T. Li, and Q. Wei, "RISC-V Instruction Set Architecture Extensions: A Survey," IEEE Access, vol. 11, pp. 24696–24711, 2023.

[17] B. K. Dakua, M. S. Hossain, and F. Ahmed, "Design and Implementation of UART Serial Communication Module Based on FPGA," International Conference on Materials, Electronics & Information Engineering (ICMEIE-2019).

[18] K. Dennis, G. Borriello, M. Gahlinger, and J. Montrym, "Single cycle RISC-V micro architecture processor and its FPGA prototype," 2017 7th International Symposium on Embedded Computing and System Design (ISED), pp. 1–5.

[19] Z. Du, Y. Liu, C. Qiu, and X. Zhang, "Verilog implementation of configurable UART module," Proc. SPIE 12597, Second International Conference on Statistics, Applied Mathematics, and Computing Science (CSAMCS 2022), 2023.

[20] S. L. Harris and D. M. Harris, Digital Design and Computer Architecture, RISC-V ed. United States of America: Elsevier, 2021.

[21] S. Qi, S. Jin, Y. Xu, and Y. Dai, "A five-stage pipeline processor using the RISC-V instruction set architecture designed by System Verilog," 2022.

[22] "Analysis and Comparison of Asynchronous FIFO and Synchronous FIFO," 2023 IEEE 2nd International Conference on Electrical Engineering, Big Data and Algorithms (EEBDA), 2023.

[23] N. RS, S. S., S. M. A., S. P. M., and M. C., "Implementation Of I2C Protocol With Adaptive Baud Rate Using Verilog," 2024 7th International Conference on Devices, Circuits and Systems (ICDCS), Coimbatore, India, 2024.

[24] C. Huang and S. Yang, "A mini I2C bus interface circuit design and its VLSI implementation," J. Supercomput., vol. 80, pp. 23794–23814, 2024.

[25] H. Wang and P. Qiao, "Design of IIC Interface Controller based on FPGA," 2024 9th International Conference on Electronic Technology and Information Science (ICETIS), Hangzhou, China, 2024.