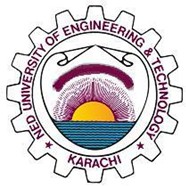
**UNDERGRADUATE FINAL YEAR PROJECT PROGRESS REPORT**

**Department of Telecommunication Engineering**

**NED University of Engineering and Technology**



RISC V SoC for Communication

#### Group Number: 11 Batch: 2021

**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

**© NED University of Engineering & Technology. All Rights Reserved – May 2025**

# 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](mailto:HASSAN4410300@cloud.neduet.edu.pk) | [KHAN4430173@cloud.neduet.edu.pk](mailto:KHAN4430173@cloud.neduet.edu.pk) | [UMER4404443@cloud.neduet.edu.pk](mailto: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

[**Author’s Declaration ii**](#_Toc198246239)

[**Statement of Contributions iii**](#_Toc198246240)

[**Executive Summary iv**](#_Toc198246241)

[**Acknowledgement vi**](#_Toc198246242)

[**Table of Contents 1**](#_Toc198246243)

[**List of Figures 8**](#_Toc198246244)

[**List of Abbreviations 9**](#_Toc198246245)

[**United Nations Sustainable Development Goals 10**](#_Toc198246246)

[**Similarity Index Report 11**](#_Toc198246247)

[**Project Title RISC V Soc for Communication 11**](#_Toc198246248)

[**Chapter 1 12**](#_Toc198246249)

[**Introduction 12**](#_Toc198246250)

[**1.1 Background Information 12**](#_Toc198246251)

[**1.2 Significance and Motivation 13**](#_Toc198246252)

[**1.2.1 Significance 13**](#_Toc198246253)

[**1.2.2 Motivation 14**](#_Toc198246254)

[**1.3 Aims and Objectives 15**](#_Toc198246255)

[**1.4 Methodology 15**](#_Toc198246256)

[**1.4.1 Literature Review 15**](#_Toc198246257)

[**1.4.2 Software Implementation 15**](#_Toc198246258)

[**1.5 Report Outline 17**](#_Toc198246259)

[**1.5.1 Scope 17**](#_Toc198246260)

[**1.5.2 Outline 17**](#_Toc198246261)

[**Chapter 2 19**](#_Toc198246262)

[**Literature Review 19**](#_Toc198246263)

[**2.1 Introduction 19**](#_Toc198246264)

[**2.2 RISC-V Processors 19**](#_Toc198246265)

[**2.3 Communication Protocols 19**](#_Toc198246266)

[**2.4 Interfacing 20**](#_Toc198246267)

[**2.4.1 UART: Universal Asynchronous Receiver/Transmitter 21**](#_Toc198246268)

[**2.4.2 I2C: Inter-Integrated Circuit 21**](#_Toc198246269)

[**2.4.3 Summary 22**](#_Toc198246270)

[**2.5 System Verilog 22**](#_Toc198246271)

[**2.6 Conclusion 23**](#_Toc198246272)

[**Chapter 3 24**](#_Toc198246273)

[**Single Cycle Processor 24**](#_Toc198246274)

[**3.1 Introduction 24**](#_Toc198246275)

[**3.2 Architecture of Single Cycle Processor 24**](#_Toc198246276)

[**3.2.1 Introduction 24**](#_Toc198246277)

[**3.2.2 Main Code 25**](#_Toc198246278)

[**3.3 Modeling of Single Cycle Processor 27**](#_Toc198246279)

[**3.3.1 Data Path 27**](#_Toc198246280)

[**3.3.2 Code of Data Path 28**](#_Toc198246281)

[**3.3.3 Flip-Flop-Based Register 28**](#_Toc198246282)

[**3.3.4 Adder 4 28**](#_Toc198246283)

[**3.3.5 Adder 28**](#_Toc198246284)

[**3.3.6 Mux 2to1: Pcmux 29**](#_Toc198246285)

[**3.3.7 Register File 29**](#_Toc198246286)

[**3.3.8 Extension Unit 29**](#_Toc198246287)

[**3.3.9 Mux 2to1: SrcB Mux 29**](#_Toc198246288)

[**3.3.10 Arithmetic and Logic Unit 29**](#_Toc198246289)

[**3.3.11 Mux 3to1: Result Mux 30**](#_Toc198246290)

[**3.3.12 Summary 30**](#_Toc198246291)

[**3.3.13 Control Unit 30**](#_Toc198246292)

[**3.3.14 Code of Control Unit 31**](#_Toc198246293)

[**3.3.15 Main Decoder 31**](#_Toc198246294)

[**3.3.16 ALU Decoder 31**](#_Toc198246295)

[**3.3.17 Summary 32**](#_Toc198246296)

[**3.4 Compilation of modules 32**](#_Toc198246297)

[**3.5 Testbenches 32**](#_Toc198246298)

[**3.6 Simulation of Single Cycle Processor 32**](#_Toc198246299)

[**3.7 Conclusion 33**](#_Toc198246300)

[**Chapter 4 34**](#_Toc198246301)

[**Pipelined Processor 34**](#_Toc198246302)

[**4.1 Introduction 34**](#_Toc198246303)

[**4.2 Modeling of Pipelined Processor 34**](#_Toc198246304)

[**4.2.1 Datapath 35**](#_Toc198246305)

[**4.2.2 Fetch Stage 38**](#_Toc198246306)

[**4.2.3 Decode Stage 38**](#_Toc198246307)

[**4.2.4 Execute Stage 38**](#_Toc198246308)

[**4.2.5 Memory Stage 39**](#_Toc198246309)

[**4.2.6 Writeback Stage 39**](#_Toc198246310)

[**4.2.7 Conclusion 39**](#_Toc198246311)

[**4.3 Control Unit 39**](#_Toc198246312)

[**4.3.1 Introduction 39**](#_Toc198246313)

[**4.3.2 Control Unit Code 41**](#_Toc198246314)

[**4.3.3 Decoding State Logic ( maindec and aludec) 42**](#_Toc198246315)

[**4.3.4 Execution State Logic (controlregE) 42**](#_Toc198246316)

[**4.3.5 Memory and WriteBack Stage Logic 42**](#_Toc198246317)

[**4.3.6 Conculsion 43**](#_Toc198246318)

[**4.4 Hazard Unit 43**](#_Toc198246319)

[**4.4.1 Introduction 43**](#_Toc198246320)

[**4.4.2 Hazard Unit Code 43**](#_Toc198246321)

[**4.4.3 Forwarding Logic 44**](#_Toc198246322)

[**4.4.4 Stall and Flush Control 44**](#_Toc198246323)

[**4.5 Testbench 45**](#_Toc198246324)

[**4.6 RTL Simulation 46**](#_Toc198246325)

[**4.7 Conclusion 48**](#_Toc198246326)

[**Chapter 5 49**](#_Toc198246327)

[**Rom IP 49**](#_Toc198246328)

[**5.1 Introduction 49**](#_Toc198246329)

[**5.2 ROM Module Configuration 49**](#_Toc198246330)

[**5.2.1 Introduction 49**](#_Toc198246331)

[**5.2.2 Rom IP Code 50**](#_Toc198246332)

[**5.3 Altsyncram Component Configuration 51**](#_Toc198246333)

[**5.4 Memory Initialization File 51**](#_Toc198246334)

[**5.5 Address and Clock Connection 52**](#_Toc198246335)

[**5.6 Conclusion 52**](#_Toc198246336)

[**5.7 Testbench 53**](#_Toc198246337)

[**5.7.1 Input and Output Signals Setup 53**](#_Toc198246338)

[**5.7.2 Monitoring Output and Address Increment 53**](#_Toc198246339)

[**5.7.3 Simulation Termination 54**](#_Toc198246340)

[**5.7.4 Conclusion 54**](#_Toc198246341)

[**5.8 RTL Simulation 54**](#_Toc198246342)

[**5.9 Conclusion 55**](#_Toc198246343)

[**Chapter 6 56**](#_Toc198246344)

[**UART (Universal Asynchronous Receiver Transmitter) 56**](#_Toc198246345)

[**6.1 Introduction 56**](#_Toc198246346)

[**6.2 Components of UART 56**](#_Toc198246347)

[**6.2.1 Transmit Shift Register (TSR) 56**](#_Toc198246348)

[**6.2.2 Receive Shift Register (RSR) 56**](#_Toc198246349)

[**6.2.3 Transmit Buffer 57**](#_Toc198246350)

[**6.2.4 Receive Buffer 57**](#_Toc198246351)

[**6.2.5 Baud Rate Generator 57**](#_Toc198246352)

[**6.2.6 Parity Generator/Checker 57**](#_Toc198246353)

[**6.3 UART Frame Format: 57**](#_Toc198246354)

[**6.4 UART Simulation 58**](#_Toc198246355)

[**6.4.1 Transmitter 58**](#_Toc198246356)

[**6.4.2 Receiver 60**](#_Toc198246357)

[**6.5. Conclusion: 61**](#_Toc198246358)

[**Chapter 7 62**](#_Toc198246359)

[**7.1 Introduction: 62**](#_Toc198246360)

[**7.2 UART FIFO Simulation: 63**](#_Toc198246361)

[**7.3 I2C FIFO Simulation: 64**](#_Toc198246362)

[**7.4 Conclusion: 65**](#_Toc198246363)

[**Chapter 8 66**](#_Toc198246364)

[**8.1 Introduction: 66**](#_Toc198246365)

[**8.2 Components of I²C (Single Master-Slave Configuration) 66**](#_Toc198246366)

[**8.2.1 I²C Master Controller: 66**](#_Toc198246367)

[**8.2.2 I²C Slave Device: 67**](#_Toc198246368)

[**8.2.3 SDA and SCL Lines: 67**](#_Toc198246369)

[**8.2.4 Top Module Wrapper: 67**](#_Toc198246370)

[**8.3 I2C Protocol Frame Format: 68**](#_Toc198246371)

[**8.4 I2C Simulation: 68**](#_Toc198246372)

[**8.5 Conclusion: 69**](#_Toc198246373)

[**Chapter 9 70**](#_Toc198246374)

[**9.1 Introduction: 70**](#_Toc198246375)

[**9.2 Embedded Systems in Industrial Automation: 70**](#_Toc198246376)

[**9.2.1 Application Overview: 70**](#_Toc198246377)

[**9.2.2 System Application: 71**](#_Toc198246378)

[**9.3 IoT (Internet of Things) Devices: 71**](#_Toc198246379)

[**9.3.1 Application Overview: 71**](#_Toc198246380)

[**9.3.2 System Application: 71**](#_Toc198246381)

[**9.4 Medical Devices: 72**](#_Toc198246382)

[**9.4.1 Application Overview: 72**](#_Toc198246383)

[**9.4.2 System Application: 72**](#_Toc198246384)

[**9.5 Automated Test Equipment: 73**](#_Toc198246385)

[**9.5.1 Application Overview: 73**](#_Toc198246386)

[**9.5.2 System Application: 73**](#_Toc198246387)

[**9.6 Wearable Devices: 74**](#_Toc198246388)

[**9.6.1 Application Overview: 74**](#_Toc198246389)

[**9.6.2 System Application: 74**](#_Toc198246390)

[**9.7 Conclusion: 75**](#_Toc198246391)

[**References 76**](#_Toc198246392)

[**Appendix 78**](#_Toc198246393)

[**Appendix A : SystemVerilog Codebase for RISC-V Single-Cycle Processor 78**](#_Toc198246394)

[**Appendix B : SystemVerilog Codebase for RISC-V Pipelined Processor 89**](#_Toc198246395)

[**Appendix C: UART Module Design Files 99**](#_Toc198246396)

[**Appendix D: I²C Module Design Files 104**](#_Toc198246397)

[**Appendix E : FIFO Buffer Implementation 109**](#_Toc198246398)

# List of Figures

[**Figure 1 RISC-V Core Interface using I2C and UART Protocol** 20](#_Toc198218622)

[**Figure 2 State Elements** 26](file:///C:\Users\RAO\Desktop\FYDP2021\FYPD%20MID%20FINAL%20REVIEW-------.docx#_Toc198218623)

[**Figure 3 Single Cycle Processor** 27](file:///C:\Users\RAO\Desktop\FYDP2021\FYPD%20MID%20FINAL%20REVIEW-------.docx#_Toc198218624)

[**Figure 4 Simulation of Single Cycle Processor** 33](file:///C:\Users\RAO\Desktop\FYDP2021\FYPD%20MID%20FINAL%20REVIEW-------.docx#_Toc198218625)

[**Figure 5 “Simulation Succeeded of Single Cycle” Message** 33](file:///C:\Users\RAO\Desktop\FYDP2021\FYPD%20MID%20FINAL%20REVIEW-------.docx#_Toc198218626)

[**Figure 6 Architecture of Pipelined Processor** 34](file:///C:\Users\RAO\Desktop\FYDP2021\FYPD%20MID%20FINAL%20REVIEW-------.docx#_Toc198218627)

[**Figure 7 Datapath of Pipelined Processor** 35](file:///C:\Users\RAO\Desktop\FYDP2021\FYPD%20MID%20FINAL%20REVIEW-------.docx#_Toc198218628)

[**Figure 8 Pipelined Processor With Control Signals** 40](file:///C:\Users\RAO\Desktop\FYDP2021\FYPD%20MID%20FINAL%20REVIEW-------.docx#_Toc198218629)

[**Figure 9 Successful Simulation** 46](file:///C:\Users\RAO\Desktop\FYDP2021\FYPD%20MID%20FINAL%20REVIEW-------.docx#_Toc198218630)

[**Figure 10 Simulation Results of Pipelined Processor** 47](file:///C:\Users\RAO\Desktop\FYDP2021\FYPD%20MID%20FINAL%20REVIEW-------.docx#_Toc198218631)

[**Figure 11 RTL Simulation of ROM IP** 54](file:///C:\Users\RAO\Desktop\FYDP2021\FYPD%20MID%20FINAL%20REVIEW-------.docx#_Toc198218632)

[**Figure 12 Successful Simulation** 55](file:///C:\Users\RAO\Desktop\FYDP2021\FYPD%20MID%20FINAL%20REVIEW-------.docx#_Toc198218633)

[**Figure 13 UART Frame Format** 57](#_Toc198218634)

[**Figure 14 Data driven on channel port ‘tx’ from port ‘din’** 59](#_Toc198218635)

[**Figure 15 Transcript showing the data being driven on the channel port** 59](#_Toc198218636)

[**Figure 16 Data captured from channel port ‘rx’ as seen in port ‘dout’** 60](#_Toc198218637)

[**Figure 17 Transcript showing the data captured from channel port** 61](#_Toc198218638)

[**Figure 18 UART FIFO Waveforms** 63](#_Toc198218639)

[**Figure 19 UART FIFO Transcript View** 63](#_Toc198218640)

[**Figure 20 I2C FIFO Waveforms** 64](#_Toc198218641)

[**Figure 21 I2C FIFO Transcript View** 64](#_Toc198218642)

[**Figure 22 I2C Timing Diagram** 68](#_Toc198218643)

[**Figure 23 I2C Top Module Simulation** 68](#_Toc198218644)

[**Figure 24 I2C Top Module Transcript View** 69](#_Toc198218645)

# 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

# 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.

## 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.

A diagram of a computer system

Description automatically generated

**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

A screenshot of a computer program

Description automatically generated

#### 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.

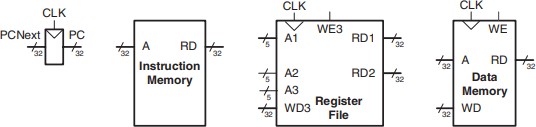
##### 3.2.2.1.2 Memory

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

A diagram of a computer system

Description automatically generatedOur 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

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

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

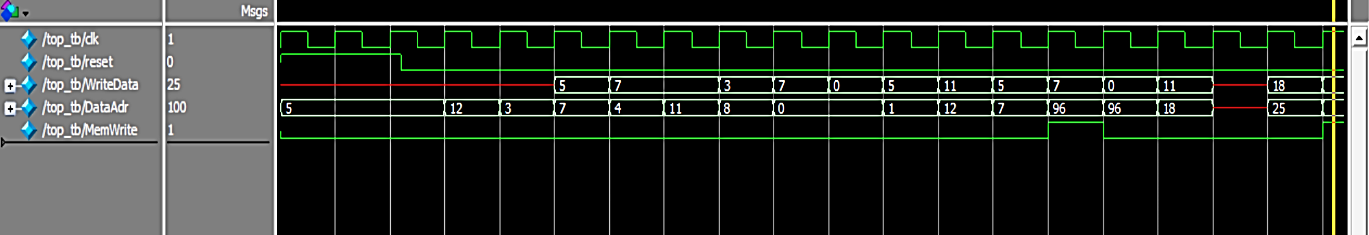
All the written modules are in System Verilog and are ‘compile ready’ in QUARTUS software with RTL simulation done.

## 3.5 Testbenches

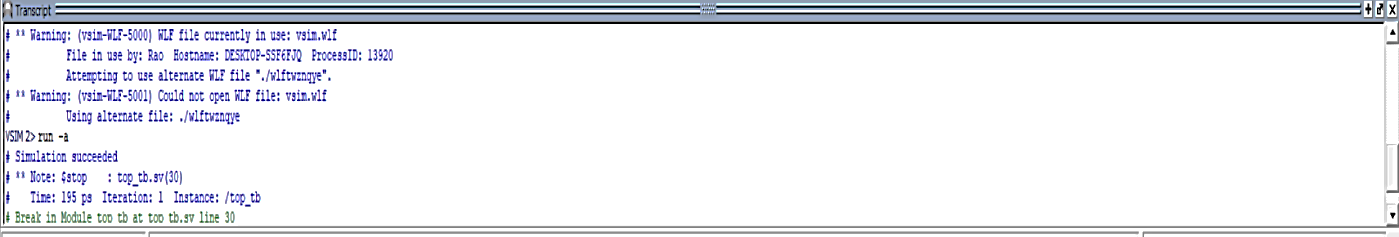
A test bench for each of the modules was also designed to test each of the modules on Modelsim. The testbench evaluates the designed functions and execution speed of both RISC-V processor and memory blocks.

## 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

A diagram of a computer

Description automatically generatedThe 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

A diagram of a machine

Description automatically generatedUsually 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).

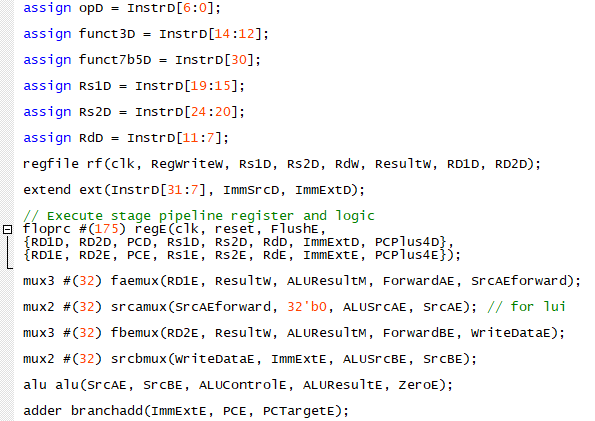
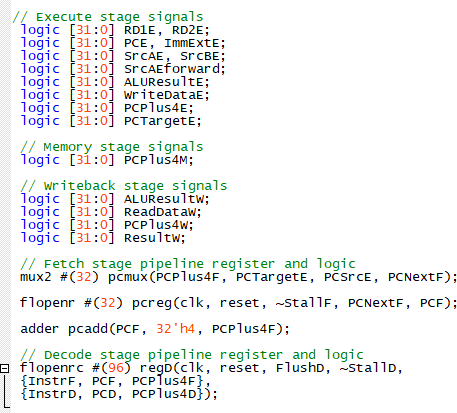
**Figure 7 Datapath of Pipelined Processor**

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.

#### A screenshot of a computer program Description automatically generatedA screen shot of a computer Description automatically generated4.2.1.1 Datapath Code

A computer code with black text

Description automatically generated



### 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 program counter (`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 unveil its structure and nuances. The ‘regD’ pipeline register emerges as a critical entity in this context. It captures three pivotal elements: the fetched instruction (`InstrF`), the ongoing program counter (`PCF`), and the increment program counter (`PCPlus4F`). Here, the instruction undergoes a process of parsing, where essential fields such as the operation type (`opD`), functional code (`funct3D`), and register identifiers (`Rs1D`, `Rs2D`, `RdD`) are extracted. These fields lay the foundation for understanding the nature of the instruction and the registers it operates on.

### 4.2.4 Execute Stage

The Execute stage is the heart of instruction processing, where the actual computation and manipulation of data take place. The `regE` pipeline register assumes a pivotal role here, holding essential values such as source register data (`RD1E`, `RD2E`), immediate extension (`ImmExtE`), and the target program counter (`PCPlus4E`). To optimize performance, data forwarding is facilitated through the `faemux` and `fbemux` multiplexers, ensuring that pertinent data from preceding pipeline stages can be channeled effectively to the Arithmetic Logic Unit (ALU). The ALU (`alu`) serves as the computational engine, performing arithmetic and logical operations on operands (`SrcAE`, `SrcBE`) based on the control signal `ALUControlE`, leading to the generation of the ALU result (`ALUResultE`). Simultaneously, the `branchadd` adder calculates the target program counter (`PCTargetE`) by adding the immediate extension (`ImmExtE`) to the current program counter (`PCE`). This is a pivotal step for executing branch instructions.

### 4.2.5 Memory Stage

The Memory stage is pivotal for memory-related activities, encompassing both data retrieval and storage operations. The `regM` pipeline register serves as a reservoir for key data: the ALU result (`ALUResultE`), data earmarked for writing back (`WriteDataE`), destination register (`RdE`), and the increment program counter (`PCPlus4E`). In this stage, the processor interacts with the memory subsystem, which could potentially trigger memory reads or writes. This phase is also crucial for tackling memory-related hazards.

### 4.2.6 Writeback Stage

The Writeback stage culminates the journey of an instruction by committing outcomes to registers. The `regW` pipeline register plays a pivotal role here, housing vital information such as the ALU result (`ALUResultM`), data read from memory (`ReadDataM`), destination register (`RdM`), and the increment program counter (`PCPlus4M`). The multiplexer `resultmux` assumes the role of selecting the pertinent data for writing back to the register file. This decision is steered by the `ResultSrcW` control signal, which governs whether the value for writing back emanates from the ALU result, memory read data, or the increment program counter (`PCPlus4W`).

### 4.2.7 Conclusion

In summary, the datapath module acts as a central coordinator, seamlessly managing the flow of data and control signals across various pipeline stages. This orchestration ensures that multiple instructions are executed concurrently and efficiently [1]. Despite its advantages, the design of a pipelined processor introduces challenges such as handling data dependencies, ensuring proper synchronization, and managing the complexities of control flow [20]. As a fundamental component, the datapath module encapsulates the essence of parallel and efficient instruction execution, making it an indispensable part of pipeline architecture.

## 4.3 Control Unit

### 4.3.1 Introduction

The control unit within the pipelined processor (illustrated in Figure 8) assumes a vital role in overseeing the various pipelined stages and guaranteeing their smooth synchronization. This section outlines the blueprint of the pipelined control unit, elucidating how it formulates control signals tailored to each stage of the pipeline. These signals are generated based on the specific instruction currently undergoing processing [1], [8], [14].

A diagram of a machine

Description automatically generatedAdditionally, the control unit's task involves managing potential hazards and ensuring that instructions progress seamlessly through the pipeline without conflicts. By dynamically adapting control signals, the control unit optimizes the utilization of resources and facilitates efficient instruction execution.

**Figure 8 Pipelined Processor With Control Signals**

### A screenshot of a computer program Description automatically generatedA computer screen shot of a program code Description automatically generated4.3.2 Control Unit Code

This section deals with the details of the 'controller' module, which is at the core of the pipelined control unit. This module acts as a central point where various inputs and outputs converge to direct how instructions are executed. The clock signal ('clk') sets the pace for operations, and the reset signal ('reset') initiates the synchronization process.

### 4.3.3 Decoding State Logic ( maindec and aludec)

In this stage, the 'maindec' and 'aludec' modules become important. They uncover the complex details of instruction codes and functions. The 'maindec' decodes the code, creating control signals such as 'RegWriteD', 'ResultSrcD', 'MemWriteD', and 'BranchD'. These signals set the base for the upcoming stages. At the same time, 'aludec' focuses on function details, turning them into control signals like 'ALUOpD' and 'ALUControlD'. These signals guide arithmetic or logic operations as the instruction moves forward. The 'maindec' and 'aludec' create control signals, all in sync with the codes and the instructions.

### 4.3.4 Execution State Logic (controlregE)

The execution stage is responsible for generating essential control signals. The `controlregE` module, driven by inputs from the preceding decode stage, processes critical signals such as `RegWriteD`, `ResultSrcD`, `MemWriteD`, `JumpD`, `BranchD`, `ALUControlD`, `ALUSrcAD`, and `ALUSrcBD`.

Within this module, the control signals for the execution phase are meticulously generated. These include `RegWriteE`, `ResultSrcE`, `MemWriteE`, `JumpE`, `BranchE`, `ALUControlE`, `ALUSrcAE`, and `ALUSrcBE`, each fulfilling a distinct role during this stage. Among these, the `PCSrcE` signal plays a pivotal role, acting as the decision-maker for the program counter’s direction based on branching logic. This dynamic mechanism ensures instructions are executed as planned, maintaining the intended flow of the program.

### 4.3.5 Memory and WriteBack Stage Logic

This stage is responsible for memory access and write-back, including 'controlregM' and 'controlregW' modules. These modules protect data integrity, generating control signals for memory access and write-back stages.'controlregM' manages memory access, creating signals like 'RegWriteM', 'ResultSrcM', and 'MemWriteM'. Inputs from the execution stage contribute to memory access actions.This phase is crucial for data manipulation and retrieval, and careful control signal generation maintains data integrity. Simultaneously, 'controlregW' performs during write-back. It generates 'RegWriteW' and 'ResultSrcW' signals. These signals allow instructions to finalize, ensuring results are accurately preserved in registers.

### 4.3.6 Conculsion

In summary, the control unit plays a vital role in the pipelined processor, synchronizing stages and optimizing resources. The 'controller' module directs instruction execution, 'maindec' and 'aludec' decode, 'controlregE' generates execution signals. 'controlregM' and 'controlregW' protect data integrity. This dynamic management ensures smooth instruction flow, enhancing processor performance.

## 4.4 Hazard Unit

### 4.4.1 Introduction

Designing a pipelined processor comes with a challenge: handling hazards that can disrupt smooth instruction flow. One type is RAW data hazards, where an instruction waits for a result from another before it's ready [8], [14]. Forwarding can help if the result is ready in time, or we pause the pipeline (stall) until it is. Another issue is control hazards, happening when the next instruction isn't decided yet. We can pause the pipeline or guess the next instruction, sometimes needing to start over (flush) if the guess is wrong. Managing these challenges requires understanding how instructions interact and spotting potential problems.

### 4.4.2 Hazard Unit Code

A screen shot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

This System Verilog module is designed for hazard detection and resolution within a pipelined processor. The module is crucial for maintaining correct instruction execution by detecting hazards that can arise due to data dependencies and controlling the pipeline stages accordingly.

### 4.4.3 Forwarding Logic

The forwarding logic (ForwardAE and ForwardBE) enables seamless data forwarding to the execution stage to prevent data hazards. It allows data from previous stages to be directly used in the execution stage, minimizing stalls. The module determines if values need to be forwarded from memory (RdM) or write-back (RdW) stages to the execution stage (Rs1E and Rs2E).

### 4.4.4 Stall and Flush Control

The module also manages stalls (temporary halts in instruction processing) and flushes (clearing of instructions in the pipeline due to hazards). It identifies potential hazards, such as load-use hazards, indicated by the lwStallD signal. If a hazard is detected, StallD and StallF signals are activated, introducing stalls in the decode and fetch stages.

#### 4.4.4.1 Stall Detection

The lwStallD signal is set when a load-use hazard occurs, i.e., when the result being computed in the execution stage (RdE) matches an operand (Rs1D or Rs2D) being fetched in the decode stage. This prompts StallD and StallF signals to introduce stalls.

#### 4.4.4.2 Flush Control

The FlushD signal is activated in case of a potential hazard in the execute stage (PCSrcE), ensuring proper synchronization. The FlushE signal flushes the decode stage when a hazard is detected (lwStallD) or when an execute stage hazard is identified (PCSrcE).

#### 4.4.4.3 Conclusion

In conclusion, the hazard detection and resolution module presented here showcases its significance in maintaining data integrity and preventing hazards in pipelined processors [8]. Its ability to control stalls and flushes ensures a reliable and efficient execution of instructions, contributing to the overall success of a processor design [14].

## 4.5 Testbench

#### 4.5.1 Introduction

This section presents the Verilog code for a testbench module that facilitates the simulation of a device under test (DUT), which is referred to as 'top'. The testbench is designed to validate the functionality of the DUT by providing inputs, generating a clock, initializing the test conditions, and checking the results.

#### A screenshot of a computer program Description automatically generated4.5.2 Testbench Code

The testbench module serves a vital role in verifying the correct functionality of the device under test. It simulates various scenarios by providing input signals and generating a clock, enabling the examination of how the DUT responds [3]. The results are checked against expected conditions, and upon successful validation, the simulation is halted.

## A screenshot of a computer program Description automatically generated4.6 RTL Simulation

**Figure 9 Successful Simulation**

This process is responsible for monitoring and verifying the outcomes of the simulation based on specific conditions [10]. The first part of the condition verifies if the data address (DataAdr) matches the value and the second part of the condition verifies if the written data (WriteData) matches the hexadecimal value. If both conditions are satisfied, meaning the address and data meet the specified criteria, the message "Simulation succeeded" will be displayed.

As shown in Figure 9 and Figure 10, the successful simulation has been achieved as both the conditions.A screen shot of a computer screen

Description automatically generated

**Figure 10 Simulation Results of Pipelined Processor**

This segment of the code acts as the validation mechanism for the simulation along with the results snapshots. It checks whether the simulated actions have resulted in the expected values for the memory address and written data. Since the conditions are met, indicating accurate simulation behavior, a success message has been displayed, and the simulation is terminated.

## 4.7 Conclusion

In conclusion, pipelining stands as a potent strategy for amplifying digital system processing efficiency. By segmenting a single-cycle processor into five stages, concurrent execution of instructions is enabled, driving performance through parallelism [14]. While this approach offers substantial clock frequency gains, challenges arise in managing data dependencies and control flow intricacies. The datapath module acts as a conductor, orchestrating data and control signal flow across stages. 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 testbench module 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.

A screenshot of a computer code

Description automatically generatedA screen shot of a computer program

Description automatically generated

## 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. The flexibility of this approach allows for customization of ROM contents to handle a wide range of scenarios, from bootstrapping an embedded system to storing lookup tables for computational tasks.

Each instruction, which is like a command for the computer, is linked to a specific address. This address shows the exact spot in the ROM where the instruction is stored. In simple terms, the MIF file guides the ROM on what to save and where to put it, making sure the right instructions are ready to be used.

## 5.5 Address and Clock Connection

The "address\_a" input of the "altsyncram\_component" is linked to the address input of the ROM module, enabling the processor to send the address information needed to fetch instructions from the ROM. Additionally, the "clock0" input of the "altsyncram\_component" is connected to the clock input of the ROM module. This synchronization guarantees that the operations performed by the ROM are perfectly coordinated with the clock signal of the processor.

## 5.6 Conclusion

In conclusion, the ROM is loaded with instructions, and the testbench simulates the sequential fetching of instructions based on the address input and clock signal. The ROM contents are preloaded using a MIF file, and the simulation provides insight into the instruction-fetching process.

## 5.7 Testbench

A screenshot of a computer program

Description automatically generatedThis 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.

### 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.

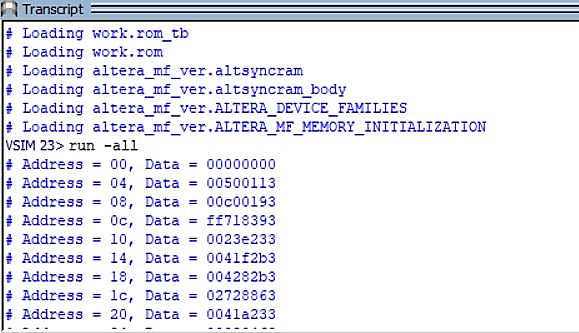
### 5.7.4 Conclusion

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.

## A screenshot of a computer Description automatically generated5.8 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.



**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

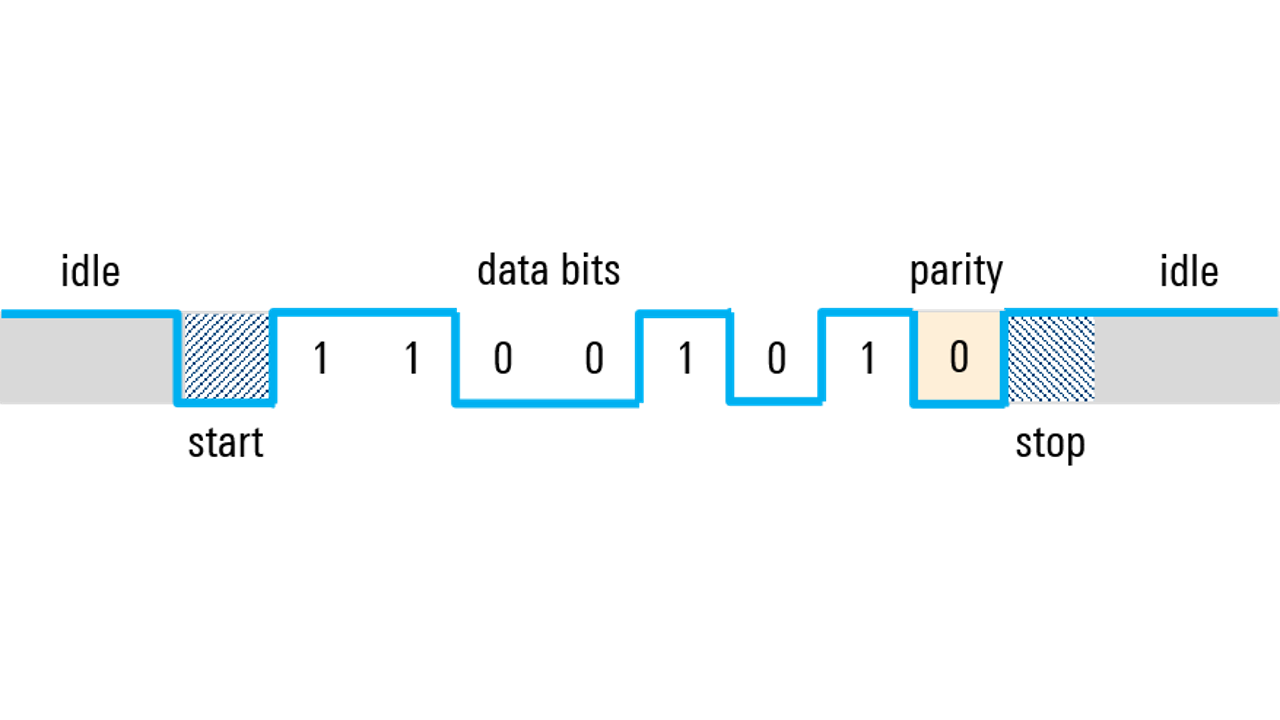
Controls the clock signal that drives the baud rate of data transfer rate in this case. Controls the data rate which in turn coordinate the working of the transmitter and the receiver sections.

### 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**

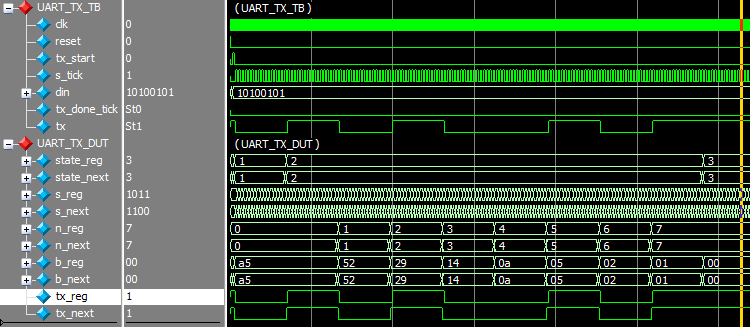
## 6.4 UART Simulation

### 6.4.1 Transmitter

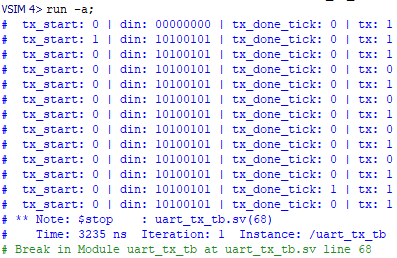
The waveform below illustrates the simulation of a UART (Universal Asynchronous Receiver/Transmitter) transmitter. On the left side of the image, the signals from the **UART\_TX\_TB (testbench)** include the **clk (clock)**, **reset**, **tx\_start**, **tx\_tick**, and **tx\_done\_tick**, which control 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’**



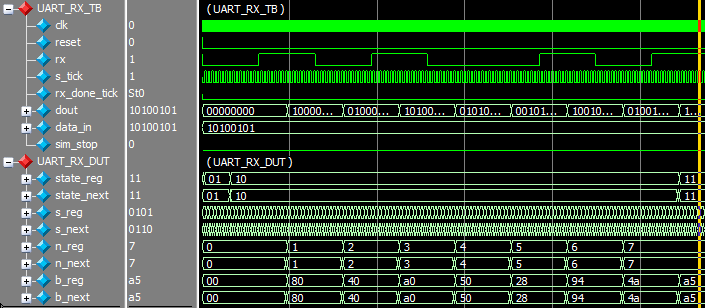
**Figure 15 Transcript showing the data being driven on the channel port**

### 6.4.2 Receiver

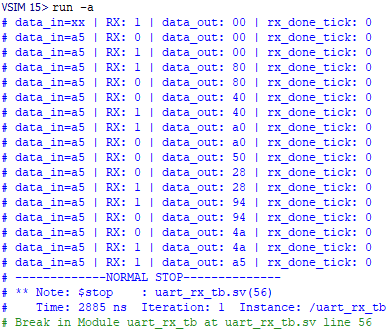
This waveform shows the simulation of a UART (Universal Asynchronous Receiver/Transmitter) receiver. The left side displays the testbench signals such as **clk (clock)**, **reset**, **rx (received data)**, **rx\_tick**, **rx\_done\_tick**, and **data\_in**. On the right side, the **UART\_RX\_DUT (device 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.

The **state\_reg** and **state\_next** represent the current and next states of the receiver, ensuring proper operation as data is received as shown if figure 16 and figure 17. The **s\_reg** stores the received data in its serialized form, which is shifted in and aligned properly during reception. The **n\_reg** holds the incoming bits, and **b\_reg** represents the data stored in the receiver buffer. The waveform shows how the bits are received, with the **UART\_RX\_DUT** receiving values such as 0xA5 and 0x52 as part of the received data. The simulation verifies the correct operation of the UART receiver, ensuring that the data is received and stored properly.



**Figure 16 Data captured from channel port ‘rx’ as seen in port ‘dout’**



**Figure 17 Transcript showing the data captured from channel port**

## 6.5. Conclusion:

The successful implementation of a Universal Asynchronous Receiver Transmitter (UART) in SystemVerilog demonstrates the practical application of digital design principles in enabling serial communication between hardware components. The design accurately models the key functionalities of UART, including configurable baud rate generation, data framing with start and stop bits, and reliable transmission and reception of serial data [15]. By leveraging the modular and concurrent capabilities of SystemVerilog, the UART module was structured in a clear and scalable manner, enabling efficient simulation and verification. The functionality was verified using a dedicated testbench that validated correct data transmission, reception, and error handling under various scenarios. This UART design can be integrated into larger SoC (System-on-Chip) or communication projects, making it a fundamental component for interfacing processors, memory, and peripherals over serial channels [20]. The project enhances understanding of timing synchronization, protocol handling, and hardware-level data communication, reinforcing the importance of UART in embedded and digital systems.

# Chapter 7

**FIFO (First In First Out)**

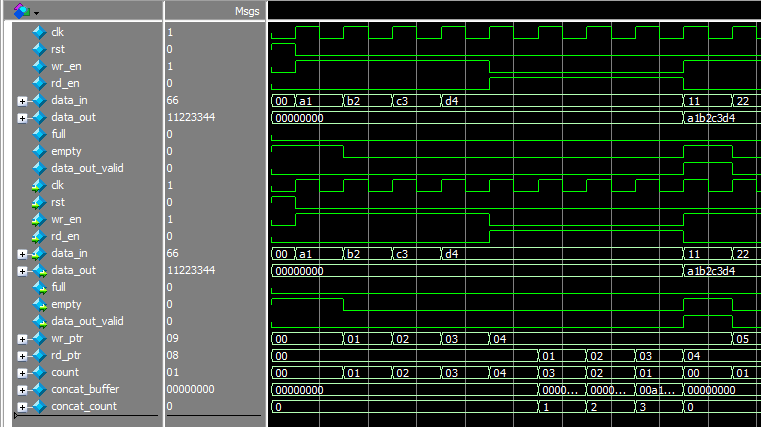
## 7.1 Introduction:

In digital systems, First-In, First-Out (FIFO) is a widely used data buffering technique that follows the queue principle where the first data element entered is the first to be removed. FIFOs are crucial in scenarios where data needs to be temporarily stored and transmitted between modules operating at different speeds or clock domains, ensuring smooth data flow without loss or overlap. [12]

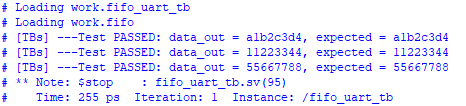
A FIFO buffer operates using two primary pointers: the write pointer, which indicates where new data is to be written, and the read pointer, which shows where data is to be read. The logic ensures that data is read in the exact order in which it was written, maintaining temporal integrity. FIFOs are implemented in hardware using registers or memory arrays along with control logic, and are commonly used in applications such as communication systems, data acquisition, and image processing pipelines.

In the context of this project, a FIFO module plays a vital role in managing data flow efficiently between subsystems. Its design, verification, and integration are essential steps to ensure reliable and synchronized operation within the overall system architecture.

## 7.2 UART FIFO Simulation:

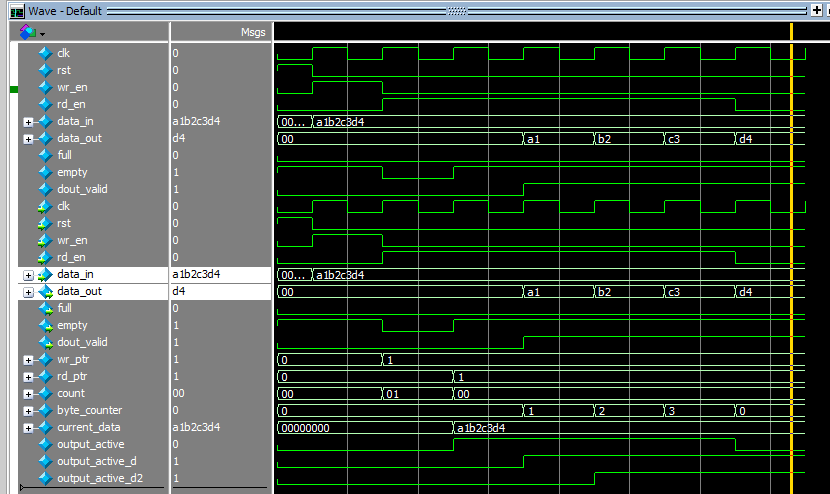


**Figure 18 UART FIFO Waveforms**

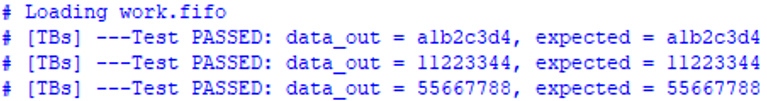


**Figure 19 UART FIFO Transcript View**

## 7.3 I2C FIFO Simulation:



**Figure 20 I2C FIFO Waveforms**



**Figure 21 I2C FIFO Transcript View**

## 7.4 Conclusion:

The implementation of the FIFO (First-In, First-Out) buffer in this project has proven to be an effective solution for managing data flow between different components of a digital system, especially when operating across varying clock domains or data rates. The FIFO design ensures orderly and lossless data transfer by maintaining the sequence of input and output operations. Developed using SystemVerilog, the FIFO module incorporates essential features such as configurable depth, full and empty flag detection, and synchronized read/write control logic. The simulation results confirm that the FIFO behaves correctly under normal and boundary conditions, handling data buffering and flow control with precision [13]. This design not only meets the functional requirements but also demonstrates good modularity and scalability, making it suitable for integration into more complex systems like communication interfaces, image processing pipelines, or SoC architectures. Overall, the FIFO buffer plays a critical role in ensuring efficient and reliable data handling within the digital system designed in this project.

# Chapter 8

**I2C (Inter-Integrated Circuit)**

## 8.1 Introduction:

Inter-Integrated Circuit (I²C) is a widely used serial communication protocol that enables efficient data exchange between multiple integrated circuits over just two wires: Serial Data (SDA) and Serial Clock (SCL). Designed by Philips (now NXP), I²C supports multi-master and multi-slave communication, making it ideal for embedded systems where several peripherals need to communicate with a microcontroller. In this project, the I²C protocol has been implemented using SystemVerilog, providing a hardware description of the I²C master and/or slave behavior [8]. The design includes key features such as start and stop condition generation, address recognition, data transfer synchronization, and acknowledgment handling. Special attention is given to the timing and state transitions required by the I²C standard to ensure reliable communication. SystemVerilog, with its rich set of modeling and simulation capabilities, offers a robust environment for designing and verifying the I²C controller. This implementation serves as a core communication module that can be integrated into larger digital systems requiring low-speed, short-distance serial data transfer, such as sensor interfacing, EEPROM communication, or configuration of peripheral devices.

## 8.2 Components of I²C (Single Master-Slave Configuration)

The I²C protocol implemented in this project follows a basic architecture with a single master and single slave operating at the lowest standard I²C speed, typically 100 kHz. Due to this simplicity, the design is easy to understand and ideal for educational and low-complexity communication purposes. The major components of this design are described below [5].

### 8.2.1 I²C Master Controller:

The I²C Master Controller is the central unit responsible for initiating and managing the entire communication process. It generates the start and stop conditions required to frame each I²C transaction. 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:

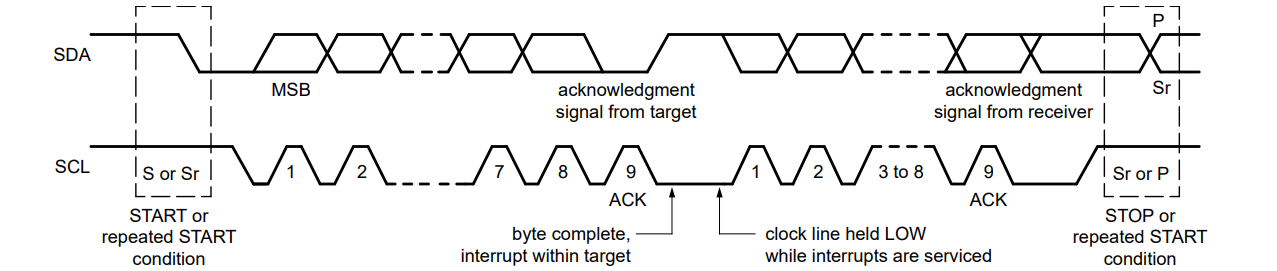
The SDA (Serial Data) Line Control Logic ensures that data transfer between the master and slave occurs on a single shared line. Since I²C uses open-drain signaling, the SDA line must be managed in such a way that only one device drives it at a time while the other remains in a high-impedance state. This logic guarantees correct timing, proper arbitration (if needed), and data integrity during transmission.

The SCL (Serial Clock) Generator is part of the master controller and is responsible for producing the timing signal required to coordinate bit-level communication over the SDA line. Operating at a low frequency, the clock ensures that data bits are sampled at the correct times, following the I²C standard’s timing constraints for setup and hold durations.

### 8.2.4 Top Module Wrapper:

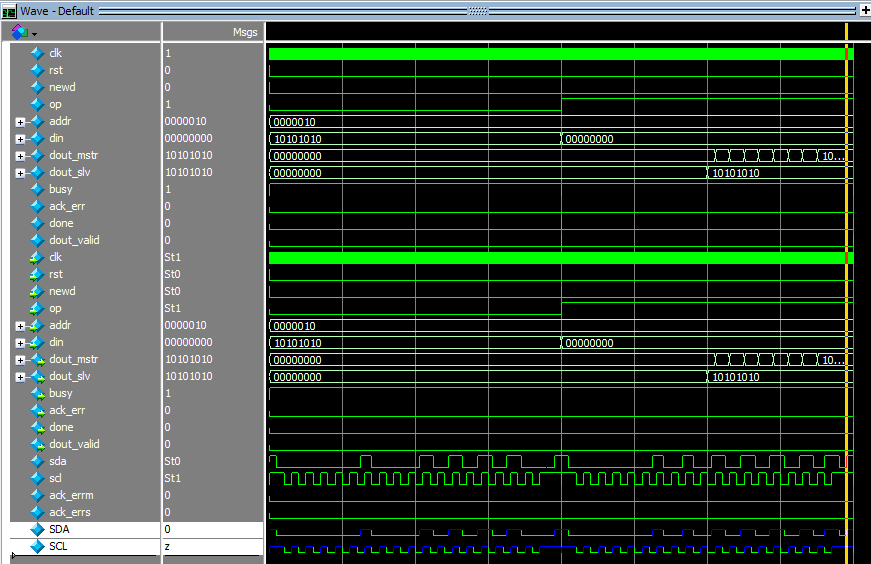
Lastly, both the master and slave operate using a Finite State Machine (FSM) that governs the protocol flow. The master's FSM transitions through states such as idle, start, address, data, acknowledgment, and stop. Similarly, the slave's FSM is designed to detect the start condition, match its address, send or receive data accordingly, and respond with acknowledgments. These FSMs ensure that each step of the I²C communication protocol is executed correctly and reliably.

## 8.3 I2C Protocol Frame Format:

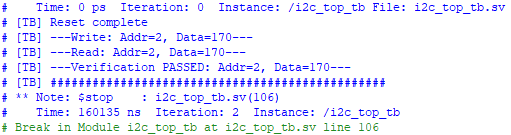


**Figure 22 I2C Timing Diagram**

## 8.4 I2C Simulation:



**Figure 23 I2C Top Module Simulation**



**Figure 24 I2C Top Module Transcript View**

## 8.5 Conclusion:

The implementation of the I²C protocol in SystemVerilog has provided a practical understanding of serial communication between digital devices using a simple and efficient interface. This project focused on designing a basic I²C system with a single master and single slave configuration operating at the lowest standard speed, making it ideal for applications requiring low data rates and minimal complexity. Through this design, core functionalities such as start/stop condition generation, address transmission, data transfer, acknowledgment handling, and clock synchronization were successfully achieved. The use of SystemVerilog enabled a clear and modular hardware description, allowing for precise control over timing and state transitions in accordance with the I²C standard. The developed I²C module serves as a foundational component that can be expanded in the future to support more advanced features such as multi-master communication, higher data rates (Fast Mode or High-Speed Mode), and more robust error handling. Overall, the project demonstrates the effectiveness of I²C as a reliable communication protocol for short-distance, low-speed digital communication, and reinforces the importance of protocol-level understanding in digital design.

# Chapter 9

**Applications**

## 9.1 Introduction:

The development of embedded systems has become integral to the evolution of numerous industries, enabling real-time data processing and communication across various devices and platforms. This project explores the design and implementation of a RISC-V processor-based system that communicates through UART and I2C interfaces, offering a robust framework for handling data in embedded applications [14], [18]. By integrating these communication protocols with data processing capabilities, this system is poised to support a wide array of practical applications in diverse fields.

The applications of such a system span across critical domains, including Industrial Automation, IoT Devices, Medical Devices, Automated Test Equipment, and Wearable Devices. Each of these areas benefits from the system's ability to receive sensor data, process it efficiently, and provide meaningful output, whether in the form of control signals, user feedback, or remote communication.

The modular design of the system, featuring UART for input, a RISC-V processor for computation, FIFO buffers for managing data flows, and I2C for output, provides a flexible and scalable solution that can be adapted to meet the specific requirements of each application. The following sections will detail the implementation of the system in these diverse application areas, illustrating its versatility and potential for real-world impact.

## 9.2 Embedded Systems in Industrial Automation:

### 9.2.1 Application Overview:

Industrial automation systems are pivotal in modern manufacturing environments, where they are used to control machinery and monitor various sensor parameters such as temperature, pressure, and motor speeds. These systems collect data from sensors and use it to optimize machine performance, minimize downtime, and ensure operational safety.

### 9.2.2 System Application:

1. **UART for Sensor Data Input**: Industrial machinery typically uses UART communication for receiving data from external sensors, such as temperature or vibration sensors. UART is ideal for industrial settings due to its simplicity and reliability over long-distance serial communication.
2. **RISC-V Processor for Data Processing**: The RISC-V processor processes the data received from the UART interface. For instance, temperature readings can be analyzed to check for any dangerous levels, triggering appropriate actions if thresholds are exceeded.
3. **I2C for Output Control**: Processed data is subsequently transmitted via I2C to control actuators, display systems, or other devices in the system. I2C is well-suited for controlling small devices such as motor controllers or status displays, given its efficiency and multi-device communication capabilities.
4. **FIFO for Buffering**: FIFO buffers ensure smooth data transfer between different stages of the system. In industrial applications, this buffering mechanism helps manage continuous sensor data streams and ensures no data is lost or delayed.

The system's modular design incorporating UART for input, the RISC-V processor for computation, FIFO buffers for data handling, and I2C for output—makes it an ideal solution for embedded systems in industrial automation. The system can efficiently monitor machinery, process real-time data, and trigger control actions, thereby enhancing both efficiency and safety in industrial settings [20].

## 9.3 IoT (Internet of Things) Devices:

### 9.3.1 Application Overview:

IoT devices are small, low-power systems designed to collect and communicate data with other devices or the cloud. These devices are widely used in applications such as smart home systems, environmental monitoring, and health tracking.

### 9.3.2 System Application:

1. **UART for Communication with Sensors**: IoT devices often rely on UART to communicate with external sensors. For example, a weather station might use UART to collect data from temperature, humidity, or air quality sensors. The data received via UART is forwarded to the RISC-V processor for processing.
2. **RISC-V Processor for Data Computation**: The RISC-V processor analyzes the sensor data, performing calculations or transformations such as determining average values, detecting irregularities, or generating alerts based on sensor readings.
3. **I2C for External Communication**: After processing, the results can be transmitted via I2C to an output device, such as a display, or sent to a cloud gateway for further analysis or user notifications. I2C’s efficiency in managing multiple devices makes it well-suited for communication within IoT networks.
4. **FIFO for Buffering**: FIFO buffers manage the continuous stream of incoming sensor data, ensuring smooth data flow and preventing delays or data loss. This is crucial in IoT systems that must process large amounts of real-time data.

The system’s efficient handling of data through UART, processing via the RISC-V processor, and output via I2C is highly suited for IoT devices. The modularity and low power consumption make it an ideal solution for embedded IoT applications, where communication with multiple sensors and devices is required in real-time.

## 9.4 Medical Devices:

### 9.4.1 Application Overview:

Medical devices such as ECG machines, blood glucose meters, and pulse oximeters rely on real-time data processing to monitor vital health parameters. These devices must provide accurate, reliable results, often under time-sensitive conditions. [6]

### 9.4.2 System Application:

1. **UART for Sensor Data Reception**: Medical devices commonly use UART to receive data from sensors, such as ECG electrodes or glucose meters. The UART interface allows the system to easily receive serial data from these sensors.
2. **RISC-V Processor for Data Analysis**: The RISC-V processor processes the received sensor data, performing tasks such as filtering ECG signals, calculating heart rate, or determining glucose levels from raw measurements.
3. **I2C for Output (Display/Peripheral Communication)**: Processed data is transmitted to an I2C-connected display or transmitted to external devices for further analysis. In medical applications, this could involve showing real-time results on a display for healthcare providers or transmitting data to a central monitoring system.
4. **FIFO for Data Buffering**: FIFO buffers handle incoming data streams from sensors, ensuring that the system can process data in real-time without loss. This is particularly important in continuous monitoring systems like ECG machines, where data integrity and timing are critical.

The ability to receive, process, and display or transmit medical data in real-time is crucial in medical devices. The system’s architecture utilizing UART for data input, the RISC-V processor for computation, FIFO buffers for managing data flow, and I2C for communication ensures efficient, timely processing of critical health data, contributing to patient safety and accurate monitoring.

## 9.5 Automated Test Equipment:

### 9.5.1 Application Overview:

Automated test equipment (ATE) is used in a variety of industries to test and verify the functionality of components and systems. These systems require precise data acquisition, analysis, and output to ensure the devices under test (DUTs) are functioning as expected.

### 9.5.2 System Application:

1. **UART for Test Equipment Communication**: ATE systems typically use UART to communicate with DUTs. The UART module receives test data or measurement results from the DUTs, which are then passed to the RISC-V processor for analysis.
2. **RISC-V Processor for Data Processing**: The processor performs necessary calculations on the data received from the DUTs, such as measuring voltage, current, or resistance values. This allows for precise and accurate testing of components in various scenarios.
3. **I2C for Test Results Output**: After processing, the results are sent to an I2C-connected display or logging system. I2C is ideal for this purpose, as it allows easy communication with multiple devices such as display screens or external data storage devices.
4. **FIFO for Data Buffering**: FIFO buffers facilitate the smooth handling of large amounts of incoming test data. This is particularly important in environments where multiple devices are being tested simultaneously, ensuring no data is lost and that the system can keep up with high-throughput data streams.

Automated test equipment requires fast, efficient handling of data to ensure the accuracy and speed of the testing process. The modular design, featuring UART for input, the RISC-V processor for analysis, FIFO for data management, and I2C for output, is well-suited for ATE applications where precise, real-time data processing is crucial for verifying the functionality of components and systems.

## 9.6 Wearable Devices:

### 9.6.1 Application Overview:

Wearable devices, such as fitness trackers, smartwatches, and health monitors, collect and process data related to the user’s health and activity. These devices are designed to be compact and power-efficient while providing real-time feedback to users.

### 9.6.2 System Application:

1. **UART for Sensor Input**: Wearable devices often rely on UART for communication with various sensors, such as accelerometers or heart rate monitors. These sensors send data to the processing unit, where it is analyzed to determine metrics like steps taken or heart rate.
2. **RISC-V Processor for Real-Time Processing**: The RISC-V processor processes the sensor data in real-time, performing computations like calculating steps taken from accelerometer data or monitoring the user’s heart rate for abnormalities.
3. **I2C for Output to Display**: The processed data is then sent to an I2C-connected display, such as an OLED or LCD screen, where users can view their health statistics. Additionally, the processed data can be sent via I2C to other devices for logging or further analysis.
4. **FIFO for Data Buffering**: FIFO buffers help manage the data received from sensors, ensuring smooth and uninterrupted processing. This is especially important in wearable devices that need to handle continuous data streams, such as tracking movement or heart rate.

The design utilizing UART for sensor input, the RISC-V processor for computation, FIFO buffers for managing data flow, and I2C for output—ensures that these devices can collect, process, and display user data efficiently, while maintaining low power consumption, making it an ideal solution for wearable applications.

## 9.7 Conclusion:

The applications described—Industrial Automation, IoT Devices, Medical Devices, Automated Test Equipment, and Wearable Devices—demonstrate the versatility of the system. The architecture, which integrates UART for input, the RISC-V processor for data processing, FIFO buffers for efficient data management, and I2C for output, is adaptable across various real-world domains. These applications highlight the practicality and utility of the system in handling real-time data processing and communication, ensuring its relevance in numerous 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.

# Appendix

### ****Appendix A :** SystemVerilog Codebase for RISC-V Single-Cycle Processor**

This appendix includes the SystemVerilog source files and testbenches used in the design and simulation of a RISC-V single-cycle processor. Each subsection corresponds to a specific module or component.

#### ****A.1 Adder Module****

**module** adder**(input** **[**31**:**0**]** a**,** b**,**

**output** **[**31**:**0**]** y**);**

**assign** y **=** a **+** b**;**

**endmodule**

**module** adder\_tb **;**

**logic** **[**31**-**0**:**0**]** a **,**b**,**c**;**

adder dut**(.**a**(**a**),.**b**(**b**),.**y**(**c**));**

**initial** **begin**

// Initialize Inputs

a **=** 0**;**

b **=** 0**;**

// Apply test vectors

**#**10 a **=** 32'h00000001**;** b **=** 32'h00000001**;**

**#**10 a **=** 32'hFFFFFFFF**;** b **=** 32'h00000001**;**

**#**10 a **=** 32'h12345678**;** b **=** 32'h87654321**;**

**#**10 a **=** 32'hA5A5A5A5**;** b **=** 32'h5A5A5A5A**;**

**#**10 a **=** 32'h0000FFFF**;** b **=** 32'hFFFF0000**;**

**#**10 $stop**;** // End the simulation

**end**

**initial** **begin**

$monitor**(**"At time %t, a = %h, b = %h, y = %h"**,** $time**,** a**,** b**,** y**);**

**end**

**endmodule**

#### ****A.2 ALU Module****

**module** alu**(input** **logic** **[**31**:**0**]** a**,** b**,**

**input** **logic** **[**2**:**0**]** alucontrol**,**

**output** **logic** **[**31**:**0**]** result**,**

**output** **logic** zero**);**

**logic** **[**31**:**0**]** condinvb**,** sum**;**

**logic** v**;** // overflow

**logic** isAddSub**;** // true when is add or subtract operation

**assign** condinvb **=** alucontrol**[**0**]** **?** **~**b **:** b**;**

**assign** sum **=** a **+** condinvb **+** alucontrol**[**0**];**

**assign** isAddSub **=** **~**alucontrol**[**2**]** **&** **~**alucontrol**[**1**]** **|**

**~**alucontrol**[**1**]** **&** alucontrol**[**0**];**

**always\_comb**

**case** **(**alucontrol**)**

3'b000**:** result **=** sum**;** // add

3'b001**:** result **=** sum**;** // subtract

3'b010**:** result **=** a **&** b**;** // and

3'b011**:** result **=** a **|** b**;** // or

3'b100**:** result **=** a **^** b**;** // xor

3'b101**:** result **=** sum**[**31**]** **^** v**;** // slt

3'b110**:** result **=** a **<<** b**[**4**:**0**];** // sll

3'b111**:** result **=** a **>>** b**[**4**:**0**];** // srl

**default:** result **=** 32'bx**;**

**endcase**

**assign** zero **=** **(**result **==** 32'b0**);**

**assign** v **=** **~(**alucontrol**[**0**]** **^** a**[**31**]** **^** b**[**31**])** **&** **(**a**[**31**]** **^** sum**[**31**])** **&** isAddSub**;**

**endmodule**

**module** alu\_tb**;**

// Inputs

**logic** **[**31**:**0**]** a**;**

**logic** **[**31**:**0**]** b**;**

**logic** **[**2**:**0**]** alucontrol**;**

// Outputs

**logic** **[**31**:**0**]** result**;**

**logic** zero**;**

// Instantiate the ALU module

alu uut **(**

**.**a**(**a**),**

**.**b**(**b**),**

**.**alucontrol**(**alucontrol**),**

**.**result**(**result**),**

**.**zero**(**zero**)**

**);**

**initial** **begin**

// Initialize Inputs

a **=** 0**;**

b **=** 0**;**

alucontrol **=** 0**;**

// Test vector 1: Add

**#**10 a **=** 32'h00000005**;** b **=** 32'h00000003**;** alucontrol **=** 3'b000**;**

// Test vector 2: Subtract

**#**10 a **=** 32'h00000008**;** b **=** 32'h00000003**;** alucontrol **=** 3'b001**;**

// Test vector 3: AND

**#**10 a **=** 32'hFFFFFFFF**;** b **=** 32'h0F0F0F0F**;** alucontrol **=** 3'b010**;**

// Test vector 4: OR

**#**10 a **=** 32'hAAAAAAAA**;** b **=** 32'h55555555**;** alucontrol **=** 3'b011**;**

// Test vector 5: XOR

**#**10 a **=** 32'h12345678**;** b **=** 32'h87654321**;** alucontrol **=** 3'b100**;**

// Test vector 6: SLT (Set Less Than)

**#**10 a **=** 32'h00000001**;** b **=** 32'h00000002**;** alucontrol **=** 3'b101**;**

// Test vector 7: SLL (Shift Left Logical)

#10 a = 32'h00000001; b = 32'h00000004; alucontrol = 3'b110;

// Test vector 8: SRL (Shift Right Logical)

#10 a = 32'h00000010; b = 32'h00000002; alucontrol = 3'b111;

// Test vector 9: Zero result

#10 a = 32'h00000002; b = 32'h00000002; alucontrol = 3'b001; // Subtract result is zero

#10 $stop; // End the simulation

end

initial begin

$monitor("At time %t, a = %h, b = %h, alucontrol = %b, result = %h, zero = %b", $time, a, b, alucontrol, result, zero);

end

endmodule

#### ****A.3 ALU Decoder Module****

**module** aludec**(input** **logic** opb5**,**

**input** **logic** **[**2**:**0**]** funct3**,**

**input** **logic** funct7b5**,**

**input** **logic** **[**1**:**0**]** ALUOp**,**

**output** **logic** **[**2**:**0**]** ALUControl**);**

**logic** RtypeSub**;**

**assign** RtypeSub **=** funct7b5 **&** opb5**;** // TRUE for R-type subtract instruction

**always\_comb**

**case(**ALUOp**)**

2'b00**:** ALUControl **=** 3'b000**;** // addition

2'b01**:** ALUControl **=** 3'b001**;** // subtraction

**default:** **case(**funct3**)** // R-type or I-type ALU

3'b000**:** **if** **(**RtypeSub**)**

ALUControl **=** 3'b001**;** // sub

**else**

ALUControl **=** 3'b000**;** // add, addi

3'b010**:** ALUControl **=** 3'b101**;** // slt, slti

3'b110**:** ALUControl **=** 3'b011**;** // or, ori

3'b111**:** ALUControl **=** 3'b010**;** // and, andi

**default:** ALUControl **=** 3'bxxx**;** // ???

**endcase**

**endcase**

**endmodule**

**module** aludec\_tb**;**

// Inputs

**logic** opb5**;**

**logic** **[**2**:**0**]** funct3**;**

**logic** funct7b5**;**

**logic** **[**1**:**0**]** ALUOp**;**

// Outputs

**logic** **[**2**:**0**]** ALUControl**;**

// Instantiate the ALU Decoder module

aludec uut **(**

**.**opb5**(**opb5**),**

**.**funct3**(**funct3**),**

**.**funct7b5**(**funct7b5**),**

**.**ALUOp**(**ALUOp**),**

**.**ALUControl**(**ALUControl**)**

**);**

**initial** **begin**

// Test vector 1: R-type ADD (ALUOp = 2'b10, funct3 = 3'b000, funct7b5 = 0)

**#**10 opb5 **=** 0**;** funct3 **=** 3'b000**;** funct7b5 **=** 0**;** ALUOp **=** 2'b10**;**

// Test vector 2: R-type SUB (ALUOp = 2'b10, funct3 = 3'b000, funct7b5 = 1)

**#**10 opb5 **=** 1**;** funct3 **=** 3'b000**;** funct7b5 **=** 1**;** ALUOp **=** 2'b10**;**

// Test vector 3: R-type SLT (ALUOp = 2'b10, funct3 = 3'b010, funct7b5 = 0)

**#**10 opb5 **=** 0**;** funct3 **=** 3'b010**;** funct7b5 **=** 0**;** ALUOp **=** 2'b10**;**

// Test vector 4: R-type OR (ALUOp = 2'b10, funct3 = 3'b110, funct7b5 = 0)

**#**10 opb5 **=** 0**;** funct3 **=** 3'b110**;** funct7b5 **=** 0**;** ALUOp **=** 2'b10**;**

// Test vector 5: R-type AND (ALUOp = 2'b10, funct3 = 3'b111, funct7b5 = 0)

**#**10 opb5 **=** 0**;** funct3 **=** 3'b111**;** funct7b5 **=** 0**;** ALUOp **=** 2'b10**;**

// Test vector 6: I-type ADDI (ALUOp = 2'b00)

**#**10 opb5 **=** 0**;** funct3 **=** 3'b000**;** funct7b5 **=** 0**;** ALUOp **=** 2'b00**;**

// Test vector 7: I-type SUBTRACT (ALUOp = 2'b01)

**#**10 opb5 **=** 0**;** funct3 **=** 3'b000**;** funct7b5 **=** 0**;** ALUOp **=** 2'b01**;**

// End simulation

**#**10 $stop**;**

end

initial begin

$monitor("At time %t, opb5 = %b, funct3 = %b, funct7b5 = %b, ALUOp = %b, ALUControl = %b", $time, opb5, funct3, funct7b5, ALUOp, ALUControl);

end

endmodule

#### ****A.4 Controller Module****

`include "maindec.sv"

`include "aludec.sv"

**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**

#### ****A.5 Datapath Module****

`include "flopr.sv"

`include "adder.sv"

`include "mux2.sv"

`include "mux3.sv"

`include "regfile.sv"

`include "extend.sv"

`include "alu.sv"

**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**

#### ****A.6 Data Memory Module****

**module** dmem**(input** **logic** clk**,** we**,**

**input** **logic** **[**31**:**0**]** a**,** wd**,**

**output** **logic** **[**31**:**0**]** rd**);**

**logic** **[**31**:**0**]** RAM**[**63**:**0**];**

**assign** rd **=** RAM**[**a**[**31**:**2**]];** // word aligned

**always\_ff** **@(posedge** clk**)**

**if** **(**we**)** RAM**[**a**[**31**:**2**]]** **<=** wd**;**

// initial begin

// mem[28] = 32'h00000020;

// mem[40] = 32'h00000002;

// end

**endmodule**

**module** dmem\_tb**;**

// Inputs

**bit** clk**;**

**logic** we**;**

**logic** **[**31**:**0**]** a**;**

**logic** **[**31**:**0**]** wd**;**

// Outputs

**logic** **[**31**:**0**]** rd**;**

// Instantiate the Data Memory module

dmem uut **(**

**.**clk**(**clk**),**

**.**we**(**we**),**

**.**a**(**a**),**

**.**wd**(**wd**),**

**.**rd**(**rd**)**

**);**

// Clock generation

**always** **#**5 clk **=** **~**clk**;**

**initial** **begin**

// Initialize Inputs

clk **=** 0**;**

we **=** 0**;**

a **=** 0**;**

wd **=** 0**;**

// Load initial memory content from hex file

$readmemh**(**"riscvtest.hex"**,** uut**.**RAM**);**

// Test Write and Read Operations

**#**10 we **=** 1**;** a **=** 32'h00000010**;** wd **=** 32'hDEADBEEF**;** // Write to address 0x10

**#**10 we **=** 0**;** a **=** 32'h00000010**;** // Read from address 0x10

**#**10 we **=** 1**;** a **=** 32'h00000014**;** wd **=** 32'hCAFEBABE**;** // Write to address 0x14

**#**10 we **=** 0**;** a **=** 32'h00000014**;** // Read from address 0x14

**#**10 $stop**;** // End simulation

**end**

**initial** **begin**

$monitor**(**"At time %t, clk = %b, we = %b, a = %h, wd = %h, rd = %h"**,** $time**,** clk**,** we**,** a**,** wd**,** rd**);**

end

endmodule

#### ****A.7 Immediate Extender Module****

**module** extend**(input** **logic** **[**31**:**7**]** instr**,**

**input** **logic** **[**1**:**0**]** immsrc**,**

**output** **logic** **[**31**:**0**]** immext**);**

**always\_comb**

**case(**immsrc**)**

// I-type

2'b00**:** immext **=** **{{**20**{**instr**[**31**]}},** instr**[**31**:**20**]};**

// S-type (stores)

2'b01**:** immext **=** **{{**20**{**instr**[**31**]}},** instr**[**31**:**25**],** instr**[**11**:**7**]};**

// B-type (branches)

2'b10**:** immext **=** **{{**20**{**instr**[**31**]}},** instr**[**7**],** instr**[**30**:**25**],** instr**[**11**:**8**],** 1'b0**};**

// J-type (jal)

2'b11**:** immext **=** **{{**12**{**instr**[**31**]}},** instr**[**19**:**12**],** instr**[**20**],** instr**[**30**:**21**],** 1'b0**};**

**default:** immext **=** 32'bx**;** // undefined

**endcase**

**endmodule**

**module** extend\_tb**;**

// Inputs

**logic** **[**31**:**7**]** instr**;**

**logic** **[**1**:**0**]** immsrc**;**

// Outputs

**logic** **[**31**:**0**]** immext**;**

// Instantiate the Extend module

extend uut **(**

**.**instr**(**instr**),**

**.**immsrc**(**immsrc**),**

**.**immext**(**immext**)**

**);**

**initial** **begin**

// Initialize Inputs

instr **=** 0**;**

immsrc **=** 0**;**

// Test vector 1: I-type instruction (e.g., addi)

**#**10 instr **=** 25'h00FFF**;** immsrc **=** 2'b00**;**

// Test vector 2: S-type instruction (e.g., sw)

**#**10 instr **=** 25'h1F123**;** immsrc **=** 2'b01**;**

// Test vector 3: B-type instruction (e.g., beq)

**#**10 instr **=** 25'h0FFFE**;** immsrc **=** 2'b10**;**

// Test vector 4: J-type instruction (e.g., jal)

**#**10 instr **=** 25'h07F45**;** immsrc **=** 2'b11**;**

// End simulation

**#**10 $stop**;**

**end**

**initial** **begin**

$monitor**(**"At time %t, instr = %h, immsrc = %b, immext = %h"**,** $time**,** instr**,** immsrc**,** immext**);**

**end**

**endmodule**

#### ****A.8 Flip-Flop Register (flopr)****

**module** flopr **#(parameter** WIDTH **=** 8**)**

**(input** **logic** clk**,** reset**,**

**input** **logic** **[**WIDTH**-**1**:**0**]** d**,**

**output** **logic** **[**WIDTH**-**1**:**0**]** q**);**

**always\_ff** **@(posedge** clk**,** **posedge** reset**)**

**if** **(**reset**)** q **<=** 0**;**

**else** q **<=** d**;**

**endmodule**

**module** flopr\_tb**;**

// Parameters

**parameter** WIDTH **=** 8**;**

// Inputs

**logic** clk**;**

**logic** reset**;**

**logic** **[**WIDTH**-**1**:**0**]** d**;**

// Outputs

**logic** **[**WIDTH**-**1**:**0**]** q**;**

// Instantiate the Flip-Flop module

flopr **#(**WIDTH**)** uut **(**

**.**clk**(**clk**),**

**.**reset**(**reset**),**

**.**d**(**d**),**

**.**q**(**q**)**

**);**

// Clock generation

**always** **#**5 clk **=** **~**clk**;**

**initial** **begin**

// Initialize Inputs

clk **=** 0**;**

reset **=** 0**;**

d **=** 0**;**

// Test vector 1: Apply reset

**#**10 reset **=** 1**;** d **=** 8'b10101010**;**

**#**10 reset **=** 0**;**

// Test vector 2: Normal operation

**#**10 d **=** 8'b11001100**;**

**#**10 d **=** 8'b11110000**;**

// Test vector 3: Apply reset again

**#**10 reset **=** 1**;**

**#**10 reset **=** 0**;**

// Test vector 4: Normal operation again

**#**10 d **=** 8'b00001111**;**

// End simulation

#10 $stop;

end

initial begin

$monitor("At time %t, clk = %b, reset = %b, d = %b, q = %b", $time, clk, reset, d, q);

end

endmodule

#### ****A.9 Instruction Memory Module****

**module** imem**(input** **logic** **[**31**:**0**]** a**,**

**output** **logic** **[**31**:**0**]** rd**);**

**logic** **[**31**:**0**]** RAM**[**63**:**0**];**

**assign** rd **=** RAM**[**a**[**31**:**2**]];** // word aligned

**initial** **begin**

//$readmemh("memfile.hex",mem);

// $readmemh("riscvtest.txt",RAM);

$readmemh**(**"riscvtest.hex"**,**RAM**);**

**end**

/\*

initial begin

// mem[0] = 32'hFFC4A303; //FOR LW

// mem[1] = 32'h00832383; // FOR LW

// mem[0] = 32'h0064A423; // FOR SW

// mem[1] = 32'h00B62423; // FOR SW

// mem[0] = 32'h0062E233; // FOR R Type

// mem[1] = 32'h00B62423; // FOR R Type

end \*/

**endmodule**

**module** imem\_tb**;**

// Testbench signals

**logic** **[**31**:**0**]** a**;**

**logic** **[**31**:**0**]** rd**;**

// Instantiate the imem module

imem dut **(**

**.**a**(**a**),**

**.**rd**(**rd**)**

**);**

// Test procedure

**initial** **begin**

// Initialize the address signal

a **=** 32'b0**;**

// Wait for the memory to be initialized

**#**10**;**

$display**(**"Testing memory reads..."**);**

// Apply different addresses and check the read data

a **=** 32'd0**;**

**#**10**;**

$display**(**"Address: %h, Data: %h"**,** a**,** rd**);**

a **=** 32'd4**;** // Since the address is word-aligned (4 bytes)

**#**10**;**

$display**(**"Address: %h, Data: %h"**,** a**,** rd**);**

a **=** 32'd8**;**

**#**10**;**

$display**(**"Address: %h, Data: %h"**,** a**,** rd**);**

a **=** 32'd12**;**

**#**10**;**

$display**(**"Address: %h, Data: %h"**,** a**,** rd**);**

// Test with another address to read data

a **=** 32'd16**;**

**#**10**;**

$display**(**"Address: %h, Data: %h"**,** a**,** rd**);**

// End simulation

$finish**;**

end

// Monitor changes in address and data

initial begin

$monitor("At time %t, Address: %h, Read Data: %h", $time, a, rd);

end

endmodule

#### ****A.10 Main Decoder Module****

**module** maindec**(input** **logic** **[**6**:**0**]** op**,**

**output** **logic** **[**1**:**0**]** ResultSrc**,**

**output** **logic** MemWrite**,**

**output** **logic** Branch**,** ALUSrc**,**

**output** **logic** RegWrite**,** Jump**,**

**output** **logic** **[**1**:**0**]** ImmSrc**,**

**output** **logic** **[**1**:**0**]** ALUOp**);**

**logic** **[**10**:**0**]** controls**;**

**assign** **{**RegWrite**,** ImmSrc**,** ALUSrc**,** MemWrite**,**

ResultSrc**,** Branch**,** ALUOp**,** Jump**}** **=** controls**;**

**always\_comb**

**case(**op**)**

// RegWrite\_ImmSrc\_ALUSrc\_MemWrite\_ResultSrc\_Branch\_ALUOp\_Jump

7'b0000011**:** controls **=** 11'b1\_00\_1\_0\_01\_0\_00\_0**;** // lw

7'b0100011**:** controls **=** 11'b0\_01\_1\_1\_00\_0\_00\_0**;** // sw

7'b0110011**:** controls **=** 11'b1\_xx\_0\_0\_00\_0\_10\_0**;** // R-type

7'b1100011**:** controls **=** 11'b0\_10\_0\_0\_00\_1\_01\_0**;** // beq

7'b0010011**:** controls **=** 11'b1\_00\_1\_0\_00\_0\_10\_0**;** // I-type ALU

7'b1101111**:** controls **=** 11'b1\_11\_0\_0\_10\_0\_00\_1**;** // jal

**default:** controls **=** 11'bx\_xx\_x\_x\_xx\_x\_xx\_x**;** // non-implemented instruction

**endcase**

**endmodule**

**module** maindec\_tb**;**

// Testbench signals

**logic** **[**6**:**0**]** op**;**

**logic** **[**1**:**0**]** ResultSrc**;**

**logic** MemWrite**;**

**logic** Branch**,** ALUSrc**;**

**logic** RegWrite**,** Jump**;**

**logic** **[**1**:**0**]** ImmSrc**;**

**logic** **[**1**:**0**]** ALUOp**;**

// Instantiate the maindec module

maindec dut **(**

**.**op**(**op**),**

**.**ResultSrc**(**ResultSrc**),**

**.**MemWrite**(**MemWrite**),**

**.**Branch**(**Branch**),**

**.**ALUSrc**(**ALUSrc**),**

**.**RegWrite**(**RegWrite**),**

**.**Jump**(**Jump**),**

**.**ImmSrc**(**ImmSrc**),**

**.**ALUOp**(**ALUOp**)**

**);**

// Test procedure

**initial** **begin**

// Test for lw instruction (opcode = 0000011)

op **=** 7'b0000011**;**

**#**10**;**

$display**(**"Opcode: lw"**);**

$display**(**"RegWrite: %b, ImmSrc: %b, ALUSrc: %b, MemWrite: %b, ResultSrc: %b, Branch: %b, ALUOp: %b, Jump: %b"**,**

RegWrite**,** ImmSrc**,** ALUSrc**,** MemWrite**,** ResultSrc**,** Branch**,** ALUOp**,** Jump**);**

// Test for sw instruction (opcode = 0100011)

op **=** 7'b0100011**;**

**#**10**;**

$display**(**"Opcode: sw"**);**

$display**(**"RegWrite: %b, ImmSrc: %b, ALUSrc: %b, MemWrite: %b, ResultSrc: %b, Branch: %b, ALUOp: %b, Jump: %b"**,**

RegWrite**,** ImmSrc**,** ALUSrc**,** MemWrite**,** ResultSrc**,** Branch**,** ALUOp**,** Jump**);**

// Test for R-type instruction (opcode = 0110011)

op **=** 7'b0110011**;**

**#**10**;**

$display**(**"Opcode: R-type"**);**

$display**(**"RegWrite: %b, ImmSrc: %b, ALUSrc: %b, MemWrite: %b, ResultSrc: %b, Branch: %b, ALUOp: %b, Jump: %b"**,**

RegWrite, ImmSrc, ALUSrc, MemWrite, ResultSrc, Branch, ALUOp, Jump);

// Test for beq instruction (opcode = 1100011)

op = 7'b1100011;

#10;

$display("Opcode: beq");

$display("RegWrite: %b, ImmSrc: %b, ALUSrc: %b, MemWrite: %b, ResultSrc: %b, Branch: %b, ALUOp: %b, Jump: %b",

RegWrite, ImmSrc, ALUSrc, MemWrite, ResultSrc, Branch, ALUOp, Jump);

// Test for I-type ALU instruction (opcode = 0010011)

op = 7'b0010011;

#10;

$display("Opcode: I-type ALU");

$display("RegWrite: %b, ImmSrc: %b, ALUSrc: %b, MemWrite: %b, ResultSrc: %b, Branch: %b, ALUOp: %b, Jump: %b",

RegWrite, ImmSrc, ALUSrc, MemWrite, ResultSrc, Branch, ALUOp, Jump);

// Test for jal instruction (opcode = 1101111)

op = 7'b1101111;

#10;

$display("Opcode: jal");

$display("RegWrite: %b, ImmSrc: %b, ALUSrc: %b, MemWrite: %b, ResultSrc: %b, Branch: %b, ALUOp: %b, Jump: %b",

RegWrite, ImmSrc, ALUSrc, MemWrite, ResultSrc, Branch, ALUOp, Jump);

// Test for an unknown opcode

op = 7'b1111111;

#10;

$display("Opcode: unknown");

$display("RegWrite: %b, ImmSrc: %b, ALUSrc: %b, MemWrite: %b, ResultSrc: %b, Branch: %b, ALUOp: %b, Jump: %b",

RegWrite, ImmSrc, ALUSrc, MemWrite, ResultSrc, Branch, ALUOp, Jump);

// End the simulation

$finish;

end

endmodule

#### ****A.11 Multiplexers (MUX2 & MUX3)****

**module** mux2 **#(parameter** WIDTH **=** 8**)**

**(input** **logic** **[**WIDTH**-**1**:**0**]** d0**,** d1**,**

**input** **logic** s**,**

**output** **logic** **[**WIDTH**-**1**:**0**]** y**);**

**assign** y **=** s **?** d1 **:** d0**;**

**endmodule**

**module** mux2\_tb**;**

// Parameter for the testbench

**parameter** WIDTH **=** 8**;**

// Testbench signals

**logic** **[**WIDTH**-**1**:**0**]** d0**,** d1**;**

**logic** s**;**

**logic** **[**WIDTH**-**1**:**0**]** y**;**

// Instantiate the mux2 module

mux2 **#(**WIDTH**)** dut **(**

**.**d0**(**d0**),**

**.**d1**(**d1**),**

**.**s**(**s**),**

**.**y**(**y**)**

**);**

// Test procedure

**initial** **begin**

// Test case 1: s = 0, d0 should be selected

d0 **=** 8'hAA**;** // Set d0 = 0xAA

d1 **=** 8'h55**;** // Set d1 = 0x55

s **=** 0**;** // Select d0

**#**10**;**

$display**(**"Test 1 - s = %b, y = %h (Expected: AA)"**,** s**,** y**);**

// Test case 2: s = 1, d1 should be selected

s **=** 1**;** // Select d1

**#**10**;**

$display**(**"Test 2 - s = %b, y = %h (Expected: 55)"**,** s**,** y**);**

// Test case 3: Change d0 and d1 values

d0 **=** 8'h0F**;** // Set d0 = 0x0F

d1 **=** 8'hF0**;** // Set d1 = 0xF0

s **=** 0**;** // Select d0

**#**10**;**

$display**(**"Test 3 - s = %b, y = %h (Expected: 0F)"**,** s**,** y**);**

// End the simulation

$stop**;**

**end**

**endmodule**

**module** mux3 **#(parameter** WIDTH **=** 8**)**

**(input** **logic** **[**WIDTH**-**1**:**0**]** d0**,** d1**,** d2**,**

**input** **logic** **[**1**:**0**]** s**,**

**output** **logic** **[**WIDTH**-**1**:**0**]** y**);**

**assign** y **=** s**[**1**]** **?** d2 **:** **(**s**[**0**]** **?** d1 **:** d0**);**

**endmodule**

**module** mux3\_tb**;**

// Parameter for the testbench

**parameter** WIDTH **=** 8**;**

// Testbench signals

**logic** **[**WIDTH**-**1**:**0**]** d0**,** d1**,** d2**;**

**logic** **[**1**:**0**]** s**;**

**logic** **[**WIDTH**-**1**:**0**]** y**;**

// Instantiate the mux3 module

mux3 **#(**WIDTH**)** dut **(**

**.**d0**(**d0**),**

**.**d1**(**d1**),**

**.**d2**(**d2**),**

**.**s**(**s**),**

**.**y**(**y**)**

**);**

// Test procedure

**initial** **begin**

// Initialize inputs

d0 **=** 8'h11**;** // Set d0 = 0x11

d1 **=** 8'h22**;** // Set d1 = 0x22

d2 **=** 8'h33**;** // Set d2 = 0x33

s **=** 2'b00**;** // Select d0

// Test case 1: s = 00, d0 should be selected

**#**10**;**

$display**(**"Test 1 - s = %b, y = %h (Expected: 11)"**,** s**,** y**);**

// Test case 2: s = 01, d1 should be selected

s **=** 2'b01**;** // Select d1

**#**10**;**

$display**(**"Test 2 - s = %b, y = %h (Expected: 22)"**,** s**,** y**);**

// Test case 3: s = 10, d2 should be selected

s **=** 2'b10**;** // Select d2

**#**10**;**

$display**(**"Test 3 - s = %b, y = %h (Expected: 33)"**,** s**,** y**);**

// End simulation

$stop**;**

**end**

endmodule

#### ****A.12 Register File Module****

**module** regfile**(input** **logic** clk**,**

**input** **logic** we3**,**

**input** **logic** **[** 4**:**0**]** a1**,** a2**,** a3**,**

**input** **logic** **[**31**:**0**]** wd3**,**

**output** **logic** **[**31**:**0**]** rd1**,** rd2**);**

**logic** **[**31**:**0**]** rf**[**31**:**0**];**

// three ported register file

// read two ports combinationally (A1/RD1, A2/RD2)

// write third port on rising edge of clock (A3/WD3/WE3)

// register 0 hardwired to 0

**always\_ff** **@(posedge** clk**)**

**if** **(**we3**)** rf**[**a3**]** **<=** wd3**;**

**assign** rd1 **=** **(**a1 **!=** 0**)** **?** rf**[**a1**]** **:** 0**;**

**assign** rd2 **=** **(**a2 **!=** 0**)** **?** rf**[**a2**]** **:** 0**;**

**endmodule**

**module** regfile\_tb**;**

// Testbench signals

**logic** clk**;**

**logic** we3**;**

**logic** **[**4**:**0**]** a1**,** a2**,** a3**;**

**logic** **[**31**:**0**]** wd3**;**

**logic** **[**31**:**0**]** rd1**,** rd2**;**

// Instantiate the regfile module

regfile dut **(**

**.**clk**(**clk**),**

**.**we3**(**we3**),**

**.**a1**(**a1**),**

**.**a2**(**a2**),**

**.**a3**(**a3**),**

**.**wd3**(**wd3**),**

**.**rd1**(**rd1**),**

**.**rd2**(**rd2**)**

**);**

// Clock generation

**always** **#**5 clk **=** **~**clk**;** // Clock with period of 10ns

// Test procedure

**initial** **begin**

// Initialize signals

clk **=** 0**;**

we3 **=** 0**;**

a1 **=** 0**;**

a2 **=** 0**;**

a3 **=** 0**;**

wd3 **=** 0**;**

// Wait for reset

**#**10**;**

// Write test case 1: Write 32'hABCD1234 to register 1

a3 **=** 5'd1**;** // Select register 1 for writing

wd3 **=** 32'hABCD1234**;** // Write data

we3 **=** 1**;** // Enable write

**#**10**;** // Wait for one clock cycle

we3 **=** 0**;** // Disable write

// Read test case 1: Read from register 1

a1 = 5'd1; // Select register 1 for reading on rd1

#10;

$display("Test 1 - Read register 1: rd1 = %h (Expected: ABCD1234)", rd1);

// Write test case 2: Write 32'h12345678 to register 2

a3 = 5'd2; // Select register 2 for writing

wd3 = 32'h12345678; // Write data

we3 = 1; // Enable write

#10; // Wait for one clock cycle

we3 = 0; // Disable write

// Read test case 2: Read from register 2

a2 = 5'd2; // Select register 2 for reading on rd2

#10;

$display("Test 2 - Read register 2: rd2 = %h (Expected: 12345678)", rd2);

// Test register 0 (which should always read 0)

a1 = 5'd0; // Select register 0 for reading

#10;

$display("Test 3 - Read register 0: rd1 = %h (Expected: 0)", rd1);

// Write test case 3: Write 32'hDEADBEEF to register 3

a3 = 5'd3; // Select register 3 for writing

wd3 = 32'hDEADBEEF; // Write data

we3 = 1; // Enable write

#10; // Wait for one clock cycle

we3 = 0; // Disable write

// Read test case 3: Read from register 3

a1 = 5'd3; // Select register 3 for reading

a2 = 5'd1; // Also read register 1 at the same time

#10;

$display("Test 4 - Read register 3: rd1 = %h (Expected: DEADBEEF)", rd1);

$display("Test 4 - Read register 1: rd2 = %h (Expected: ABCD1234)", rd2);

// End simulation

$finish;

end

endmodule

#### ****A.13 Top-Level RISC-V Single-Cycle Processor****

`include "controller.sv"

`include "datapath.sv"

**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**

### ****Appendix B :** SystemVerilog Codebase for RISC-V Pipelined Processor**

This appendix includes SystemVerilog modules, testbenches, memory files, and supporting data for the pipelined implementation of a RISC-V processor. The pipelined architecture is structured in stages: Fetch, Decode, Execute, Memory, and Writeback.

#### ****B.1 ALU and ALU Decoder****

**module** ALU**(**A**,**B**,**Result**,**ALUControl**,**OverFlow**,**Carry**,**Zero**,**Negative**);**

**input** **[**31**:**0**]**A**,**B**;**

**input** **[**2**:**0**]**ALUControl**;**

**output** Carry**,**OverFlow**,**Zero**,**Negative**;**

**output** **[**31**:**0**]**Result**;**

**wire** Cout**;**

**wire** **[**31**:**0**]**Sum**;**

**assign** Sum **=** **(**ALUControl**[**0**]** **==** 1'b0**)** **?** A **+** B **:**

**(**A **+** **((~**B**)+**1**))** **;**

**assign** **{**Cout**,**Result**}** **=** **(**ALUControl **==** 3'b000**)** **?** Sum **:**

**(**ALUControl **==** 3'b001**)** **?** Sum **:**

**(**ALUControl **==** 3'b010**)** **?** A **&** B **:**

**(**ALUControl **==** 3'b011**)** **?** A **|** B **:**

**(**ALUControl **==** 3'b101**)** **?** **{{**32**{**1'b0**}},(**Sum**[**31**])}** **:**

**{**33**{**1'b0**}};**

**assign** OverFlow **=** **((**Sum**[**31**]** **^** A**[**31**])** **&**

**(~(**ALUControl**[**0**]** **^** B**[**31**]** **^** A**[**31**]))** **&**

**(~**ALUControl**[**1**]));**

**assign** Carry **=** **((~**ALUControl**[**1**])** **&** Cout**);**

**assign** Zero **=** **&(~**Result**);**

**assign** Negative **=** Result**[**31**];**

**endmodule**

**module** ALU\_Decoder**(**ALUOp**,**funct3**,**funct7**,**op**,**ALUControl**);**

**input** **[**1**:**0**]**ALUOp**;**

**input** **[**2**:**0**]**funct3**;**

**input** **[**6**:**0**]**funct7**,**op**;**

**output** **[**2**:**0**]**ALUControl**;**

// Method 1

// assign ALUControl = (ALUOp == 2'b00) ? 3'b000 :

// (ALUOp == 2'b01) ? 3'b001 :

// (ALUOp == 2'b10) ? ((funct3 == 3'b000) ? ((({op[5],funct7[5]} == 2'b00) | ({op[5],funct7[5]} == 2'b01) | ({op[5],funct7[5]} == 2'b10)) ? 3'b000 : 3'b001) :

// (funct3 == 3'b010) ? 3'b101 :

// (funct3 == 3'b110) ? 3'b011 :

// (funct3 == 3'b111) ? 3'b010 : 3'b000) :

// 3'b000;

// Method 2

**assign** ALUControl **=** **(**ALUOp **==** 2'b00**)** **?** 3'b000 **:**

**(**ALUOp **==** 2'b01**)** **?** 3'b001 **:**

**((**ALUOp **==** 2'b10**)** **&** **(**funct3 **==** 3'b000**)** **&** **({**op**[**5**],**funct7**[**5**]}** **==** 2'b11**))** **?** 3'b001 **:**

**((**ALUOp **==** 2'b10**)** **&** **(**funct3 **==** 3'b000**)** **&** **({**op**[**5**],**funct7**[**5**]}** **!=** 2'b11**))** **?** 3'b000 **:**

**((**ALUOp **==** 2'b10**)** **&** **(**funct3 **==** 3'b010**))** **?** 3'b101 **:**

**((**ALUOp **==** 2'b10**)** **&** **(**funct3 **==** 3'b110**))** **?** 3'b011 **:**

**((**ALUOp **==** 2'b10**)** **&** **(**funct3 **==** 3'b111**))** **?** 3'b010 **:**

3'b000 **;**

**endmodule**

**module** Main\_Decoder**(**Op**,**RegWrite**,**ImmSrc**,**ALUSrc**,**MemWrite**,**ResultSrc**,**Branch**,**ALUOp**);**

**input** **[**6**:**0**]**Op**;**

**output** RegWrite**,**ALUSrc**,**MemWrite**,**ResultSrc**,**Branch**;**

**output** **[**1**:**0**]**ImmSrc**,**ALUOp**;**

**assign** RegWrite **=** **(**Op **==** 7'b0000011 **|** Op **==** 7'b0110011 **|** Op **==** 7'b0010011 **)** **?** 1'b1 **:**

1'b0 **;**

**assign** ImmSrc **=** **(**Op **==** 7'b0100011**)** **?** 2'b01 **:**

**(**Op **==** 7'b1100011**)** **?** 2'b10 **:**

2'b00 **;**

**assign** ALUSrc **=** **(**Op **==** 7'b0000011 **|** Op **==** 7'b0100011 **|** Op **==** 7'b0010011**)** **?** 1'b1 **:**

1'b0 **;**

**assign** MemWrite **=** **(**Op **==** 7'b0100011**)** **?** 1'b1 **:**

1'b0 **;**

**assign** ResultSrc **=** **(**Op **==** 7'b0000011**)** **?** 1'b1 **:**

1'b0 **;**

**assign** Branch **=** **(**Op **==** 7'b1100011**)** **?** 1'b1 **:**

1'b0 **;**

**assign** ALUOp **=** **(**Op **==** 7'b0110011**)** **?** 2'b10 **:**

**(**Op **==** 7'b1100011**)** **?** 2'b01 **:**

2'b00 **;**

**endmodule**

#### ****B.2 Pipeline Stages****

**module** decode\_cycle**(**clk**,** rst**,** InstrD**,** PCD**,** PCPlus4D**,** RegWriteW**,** RDW**,** ResultW**,** RegWriteE**,** ALUSrcE**,** MemWriteE**,** ResultSrcE**,**

BranchE**,** ALUControlE**,** RD1\_E**,** RD2\_E**,** Imm\_Ext\_E**,** RD\_E**,** PCE**,** PCPlus4E**,** RS1\_E**,** RS2\_E**);**

// Declaring I/O

**input** clk**,** rst**,** RegWriteW**;**

**input** **[**4**:**0**]** RDW**;**

**input** **[**31**:**0**]** InstrD**,** PCD**,** PCPlus4D**,** ResultW**;**

**output** RegWriteE**,**ALUSrcE**,**MemWriteE**,**ResultSrcE**,**BranchE**;**

**output** **[**2**:**0**]** ALUControlE**;**

**output** **[**31**:**0**]** RD1\_E**,** RD2\_E**,** Imm\_Ext\_E**;**

**output** **[**4**:**0**]** RS1\_E**,** RS2\_E**,** RD\_E**;**

**output** **[**31**:**0**]** PCE**,** PCPlus4E**;**

// Declare Interim Wires

**wire** RegWriteD**,**ALUSrcD**,**MemWriteD**,**ResultSrcD**,**BranchD**;**

**wire** **[**1**:**0**]** ImmSrcD**;**

**wire** **[**2**:**0**]** ALUControlD**;**

**wire** **[**31**:**0**]** RD1\_D**,** RD2\_D**,** Imm\_Ext\_D**;**

// Declaration of Interim Register

**reg** RegWriteD\_r**,**ALUSrcD\_r**,**MemWriteD\_r**,**ResultSrcD\_r**,**BranchD\_r**;**

**reg** **[**2**:**0**]** ALUControlD\_r**;**

**reg** **[**31**:**0**]** RD1\_D\_r**,** RD2\_D\_r**,** Imm\_Ext\_D\_r**;**

**reg** **[**4**:**0**]** RD\_D\_r**,** RS1\_D\_r**,** RS2\_D\_r**;**

**reg** **[**31**:**0**]** PCD\_r**,** PCPlus4D\_r**;**

// Initiate the modules

// Control Unit

Control\_Unit\_Top control **(**

**.**Op**(**InstrD**[**6**:**0**]),**

**.**RegWrite**(**RegWriteD**),**

**.**ImmSrc**(**ImmSrcD**),**

**.**ALUSrc**(**ALUSrcD**),**

**.**MemWrite**(**MemWriteD**),**

**.**ResultSrc**(**ResultSrcD**),**

**.**Branch**(**BranchD**),**

**.**funct3**(**InstrD**[**14**:**12**]),**

**.**funct7**(**InstrD**[**31**:**25**]),**

**.**ALUControl**(**ALUControlD**)**

**);**

// Register File

Register\_File rf (

.clk(clk),

.rst(rst),

.WE3(RegWriteW),

.WD3(ResultW),

.A1(InstrD[19:15]),

.A2(InstrD[24:20]),

.A3(RDW),

.RD1(RD1\_D),

.RD2(RD2\_D)

);

// Sign Extension

Sign\_Extend extension (

.In(InstrD[31:0]),

.Imm\_Ext(Imm\_Ext\_D),

.ImmSrc(ImmSrcD)

);

// Declaring Register Logic

always @(posedge clk or negedge rst) begin

if(rst == 1'b0) begin

RegWriteD\_r <= 1'b0;

ALUSrcD\_r <= 1'b0;

MemWriteD\_r <= 1'b0;

ResultSrcD\_r <= 1'b0;

BranchD\_r <= 1'b0;

ALUControlD\_r <= 3'b000;

RD1\_D\_r <= 32'h00000000;

RD2\_D\_r <= 32'h00000000;

Imm\_Ext\_D\_r <= 32'h00000000;

RD\_D\_r <= 5'h00;

PCD\_r <= 32'h00000000;

PCPlus4D\_r <= 32'h00000000;

RS1\_D\_r <= 5'h00;

RS2\_D\_r <= 5'h00;

end

else begin

RegWriteD\_r <= RegWriteD;

ALUSrcD\_r <= ALUSrcD;

MemWriteD\_r <= MemWriteD;

ResultSrcD\_r <= ResultSrcD;

BranchD\_r <= BranchD;

ALUControlD\_r <= ALUControlD;

RD1\_D\_r <= RD1\_D;

RD2\_D\_r <= RD2\_D;

Imm\_Ext\_D\_r <= Imm\_Ext\_D;

RD\_D\_r <= InstrD[11:7];

PCD\_r <= PCD;

PCPlus4D\_r <= PCPlus4D;

RS1\_D\_r <= InstrD[19:15];

RS2\_D\_r <= InstrD[24:20];

end

end

// Output asssign statements

assign RegWriteE = RegWriteD\_r;

assign ALUSrcE = ALUSrcD\_r;

assign MemWriteE = MemWriteD\_r;

assign ResultSrcE = ResultSrcD\_r;

assign BranchE = BranchD\_r;

assign ALUControlE = ALUControlD\_r;

assign RD1\_E = RD1\_D\_r;

assign RD2\_E = RD2\_D\_r;

assign Imm\_Ext\_E = Imm\_Ext\_D\_r;

assign RD\_E = RD\_D\_r;

assign PCE = PCD\_r;

assign PCPlus4E = PCPlus4D\_r;

assign RS1\_E = RS1\_D\_r;

assign RS2\_E = RS2\_D\_r;

endmodule

**module** execute\_cycle**(**clk**,** rst**,** RegWriteE**,** ALUSrcE**,** MemWriteE**,** ResultSrcE**,** BranchE**,** ALUControlE**,**

RD1\_E**,** RD2\_E**,** Imm\_Ext\_E**,** RD\_E**,** PCE**,** PCPlus4E**,** PCSrcE**,** PCTargetE**,** RegWriteM**,** MemWriteM**,** ResultSrcM**,** RD\_M**,** PCPlus4M**,** WriteDataM**,** ALU\_ResultM**,** ResultW**,** ForwardA\_E**,** ForwardB\_E**);**

// Declaration I/Os

**input** clk**,** rst**,** RegWriteE**,**ALUSrcE**,**MemWriteE**,**ResultSrcE**,**BranchE**;**

**input** **[**2**:**0**]** ALUControlE**;**

**input** **[**31**:**0**]** RD1\_E**,** RD2\_E**,** Imm\_Ext\_E**;**

**input** **[**4**:**0**]** RD\_E**;**

**input** **[**31**:**0**]** PCE**,** PCPlus4E**;**

**input** **[**31**:**0**]** ResultW**;**

**input** **[**1**:**0**]** ForwardA\_E**,** ForwardB\_E**;**

**output** PCSrcE**,** RegWriteM**,** MemWriteM**,** ResultSrcM**;**

**output** **[**4**:**0**]** RD\_M**;**

**output** **[**31**:**0**]** PCPlus4M**,** WriteDataM**,** ALU\_ResultM**;**

**output** **[**31**:**0**]** PCTargetE**;**

// Declaration of Interim Wires

**wire** **[**31**:**0**]** Src\_A**,** Src\_B\_interim**,** Src\_B**;**

**wire** **[**31**:**0**]** ResultE**;**

**wire** ZeroE**;**

// Declaration of Register

**reg** RegWriteE\_r**,** MemWriteE\_r**,** ResultSrcE\_r**;**

**reg** **[**4**:**0**]** RD\_E\_r**;**

**reg** **[**31**:**0**]** PCPlus4E\_r**,** RD2\_E\_r**,** ResultE\_r**;**

// Declaration of Modules

// 3 by 1 Mux for Source A

Mux\_3\_by\_1 srca\_mux **(**

**.**a**(**RD1\_E**),**

**.**b**(**ResultW**),**

**.**c**(**ALU\_ResultM**),**

**.**s**(**ForwardA\_E**),**

**.**d**(**Src\_A**)**

**);**

// 3 by 1 Mux for Source B

Mux\_3\_by\_1 srcb\_mux **(**

**.**a**(**RD2\_E**),**

**.**b**(**ResultW**),**

**.**c**(**ALU\_ResultM**),**

**.**s**(**ForwardB\_E**),**

.d(Src\_B\_interim)

);

// ALU Src Mux

Mux alu\_src\_mux (

.a(Src\_B\_interim),

.b(Imm\_Ext\_E),

.s(ALUSrcE),

.c(Src\_B)

);

// ALU Unit

ALU alu (

.A(Src\_A),

.B(Src\_B),

.Result(ResultE),

.ALUControl(ALUControlE),

.OverFlow(),

.Carry(),

.Zero(ZeroE),

.Negative()

);

// Adder

PC\_Adder branch\_adder (

.a(PCE),

.b(Imm\_Ext\_E),

.c(PCTargetE)

);

// Register Logic

always @(posedge clk or negedge rst) begin

if(rst == 1'b0) begin

RegWriteE\_r <= 1'b0;

MemWriteE\_r <= 1'b0;

ResultSrcE\_r <= 1'b0;

RD\_E\_r <= 5'h00;

PCPlus4E\_r <= 32'h00000000;

RD2\_E\_r <= 32'h00000000;

ResultE\_r <= 32'h00000000;

end

else begin

RegWriteE\_r <= RegWriteE;

MemWriteE\_r <= MemWriteE;

ResultSrcE\_r <= ResultSrcE;

RD\_E\_r <= RD\_E;

PCPlus4E\_r <= PCPlus4E;

RD2\_E\_r <= Src\_B\_interim;

ResultE\_r <= ResultE;

end

end

// Output Assignments

assign PCSrcE = ZeroE & BranchE;

assign RegWriteM = RegWriteE\_r;

assign MemWriteM = MemWriteE\_r;

assign ResultSrcM = ResultSrcE\_r;

assign RD\_M = RD\_E\_r;

assign PCPlus4M = PCPlus4E\_r;

assign WriteDataM = RD2\_E\_r;

assign ALU\_ResultM = ResultE\_r;

endmodule

**module** fetch\_cycle**(**clk**,** rst**,** PCSrcE**,** PCTargetE**,** InstrD**,** PCD**,** PCPlus4D**);**

// Declare input & outputs

**input** clk**,** rst**;**

**input** PCSrcE**;**

**input** **[**31**:**0**]** PCTargetE**;**

**output** **[**31**:**0**]** InstrD**;**

**output** **[**31**:**0**]** PCD**,** PCPlus4D**;**

// Declaring interim wires

**wire** **[**31**:**0**]** PC\_F**,** PCF**,** PCPlus4F**;**

**wire** **[**31**:**0**]** InstrF**;**

// Declaration of Register

**reg** **[**31**:**0**]** InstrF\_reg**;**

**reg** **[**31**:**0**]** PCF\_reg**,** PCPlus4F\_reg**;**

// Initiation of Modules

// Declare PC Mux

Mux PC\_MUX **(.**a**(**PCPlus4F**),**

**.**b**(**PCTargetE**),**

**.**s**(**PCSrcE**),**

**.**c**(**PC\_F**)**

**);**

// Declare PC Counter

PC\_Module Program\_Counter **(**

**.**clk**(**clk**),**

**.**rst**(**rst**),**

**.**PC**(**PCF**),**

**.**PC\_Next**(**PC\_F**)**

**);**

// Declare Instruction Memory

Instruction\_Memory IMEM **(**

**.**rst**(**rst**),**

**.**A**(**PCF**),**

**.**RD**(**InstrF**)**

**);**

// Declare PC adder

PC\_Adder PC\_adder **(**

**.**a**(**PCF**),**

.b(32'h00000004),

.c(PCPlus4F)

);

// Fetch Cycle Register Logic

always @(posedge clk or negedge rst) begin

if(rst == 1'b0) begin

InstrF\_reg <= 32'h00000000;

PCF\_reg <= 32'h00000000;

PCPlus4F\_reg <= 32'h00000000;

end

else begin

InstrF\_reg <= InstrF;

PCF\_reg <= PCF;

PCPlus4F\_reg <= PCPlus4F;

end

end

// Assigning Registers Value to the Output port

assign InstrD = (rst == 1'b0) ? 32'h00000000 : InstrF\_reg;

assign PCD = (rst == 1'b0) ? 32'h00000000 : PCF\_reg;

assign PCPlus4D = (rst == 1'b0) ? 32'h00000000 : PCPlus4F\_reg;

endmodule

**module** memory\_cycle**(**clk**,** rst**,** RegWriteM**,** MemWriteM**,** ResultSrcM**,** RD\_M**,** PCPlus4M**,** WriteDataM**,**

ALU\_ResultM**,** RegWriteW**,** ResultSrcW**,** RD\_W**,** PCPlus4W**,** ALU\_ResultW**,** ReadDataW**);**

// Declaration of I/Os

**input** clk**,** rst**,** RegWriteM**,** MemWriteM**,** ResultSrcM**;**

**input** **[**4**:**0**]** RD\_M**;**

**input** **[**31**:**0**]** PCPlus4M**,** WriteDataM**,** ALU\_ResultM**;**

**output** RegWriteW**,** ResultSrcW**;**

**output** **[**4**:**0**]** RD\_W**;**

**output** **[**31**:**0**]** PCPlus4W**,** ALU\_ResultW**,** ReadDataW**;**

// Declaration of Interim Wires

**wire** **[**31**:**0**]** ReadDataM**;**

// Declaration of Interim Registers

**reg** RegWriteM\_r**,** ResultSrcM\_r**;**

**reg** **[**4**:**0**]** RD\_M\_r**;**

**reg** **[**31**:**0**]** PCPlus4M\_r**,** ALU\_ResultM\_r**,** ReadDataM\_r**;**

// Declaration of Module Initiation

Data\_Memory dmem **(**

**.**clk**(**clk**),**

**.**rst**(**rst**),**

**.**WE**(**MemWriteM**),**

**.**WD**(**WriteDataM**),**

**.**A**(**ALU\_ResultM**),**

**.**RD**(**ReadDataM**)**

**);**

// Memory Stage Register Logic

**always** **@(posedge** clk **or** **negedge** rst**)** **begin**

**if** **(**rst **==** 1'b0**)** **begin**

RegWriteM\_r **<=** 1'b0**;**

ResultSrcM\_r **<=** 1'b0**;**

RD\_M\_r **<=** 5'h00**;**

PCPlus4M\_r **<=** 32'h00000000**;**

ALU\_ResultM\_r **<=** 32'h00000000**;**

ReadDataM\_r **<=** 32'h00000000**;**

**end**

**else** **begin**

RegWriteM\_r **<=** RegWriteM**;**

ResultSrcM\_r **<=** ResultSrcM**;**

RD\_M\_r <= RD\_M;

PCPlus4M\_r <= PCPlus4M;

ALU\_ResultM\_r <= ALU\_ResultM;

ReadDataM\_r <= ReadDataM;

end

end

// Declaration of output assignments

assign RegWriteW = RegWriteM\_r;

assign ResultSrcW = ResultSrcM\_r;

assign RD\_W = RD\_M\_r;

assign PCPlus4W = PCPlus4M\_r;

assign ALU\_ResultW = ALU\_ResultM\_r;

assign ReadDataW = ReadDataM\_r;

endmodule

**module** writeback\_cycle**(**clk**,** rst**,** ResultSrcW**,** PCPlus4W**,** ALU\_ResultW**,** ReadDataW**,** ResultW**);**

// Declaration of IOs

**input** clk**,** rst**,** ResultSrcW**;**

**input** **[**31**:**0**]** PCPlus4W**,** ALU\_ResultW**,** ReadDataW**;**

**output** **[**31**:**0**]** ResultW**;**

// Declaration of Module

Mux result\_mux **(**

**.**a**(**ALU\_ResultW**),**

**.**b**(**ReadDataW**),**

**.**s**(**ResultSrcW**),**

**.**c**(**ResultW**)**

**);**

**endmodule**

#### ****B.3 Pipeline Top-Level Integration****

`include "Fetch\_Cycle.v"

`include "Decode\_Cycle.v"

`include "Execute\_Cycle.v"

`include "Memory\_Cycle.v"

`include "Writeback\_Cycle.v"

`include "PC.v"

`include "PC\_Adder.v"

`include "Mux.v"

`include "Instruction\_Memory.v"

`include "Control\_Unit\_Top.v"

`include "Register\_File.v"

`include "Sign\_Extend.v"

`include "ALU.v"

`include "Data\_Memory.v"

`include "Hazard\_unit.v"

**module** Pipeline\_top**(**clk**,** rst**);**

// Declaration of I/O

**input** clk**,** rst**;**

// Declaration of Interim Wires

**wire** PCSrcE**,** RegWriteW**,** RegWriteE**,** ALUSrcE**,** MemWriteE**,** ResultSrcE**,** BranchE**,** RegWriteM**,** MemWriteM**,** ResultSrcM**,** ResultSrcW**;**

**wire** **[**2**:**0**]** ALUControlE**;**

**wire** **[**4**:**0**]** RD\_E**,** RD\_M**,** RDW**;**

**wire** **[**31**:**0**]** PCTargetE**,** InstrD**,** PCD**,** PCPlus4D**,** ResultW**,** RD1\_E**,** RD2\_E**,** Imm\_Ext\_E**,** PCE**,** PCPlus4E**,** PCPlus4M**,** WriteDataM**,** ALU\_ResultM**;**

**wire** **[**31**:**0**]** PCPlus4W**,** ALU\_ResultW**,** ReadDataW**;**

**wire** **[**4**:**0**]** RS1\_E**,** RS2\_E**;**

**wire** **[**1**:**0**]** ForwardBE**,** ForwardAE**;**

// Module Initiation

// Fetch Stage

fetch\_cycle Fetch **(**

**.**clk**(**clk**),**

**.**rst**(**rst**),**

**.**PCSrcE**(**PCSrcE**),**

**.**PCTargetE**(**PCTargetE**),**

**.**InstrD**(**InstrD**),**

**.**PCD**(**PCD**),**

**.**PCPlus4D**(**PCPlus4D**)**

**);**

// Decode Stage

decode\_cycle Decode (

.clk(clk),

.rst(rst),

.InstrD(InstrD),

.PCD(PCD),

.PCPlus4D(PCPlus4D),

.RegWriteW(RegWriteW),

.RDW(RDW),

.ResultW(ResultW),

.RegWriteE(RegWriteE),

.ALUSrcE(ALUSrcE),

.MemWriteE(MemWriteE),

.ResultSrcE(ResultSrcE),

.BranchE(BranchE),

.ALUControlE(ALUControlE),

.RD1\_E(RD1\_E),

.RD2\_E(RD2\_E),

.Imm\_Ext\_E(Imm\_Ext\_E),

.RD\_E(RD\_E),

.PCE(PCE),

.PCPlus4E(PCPlus4E),

.RS1\_E(RS1\_E),

.RS2\_E(RS2\_E)

);

// Execute Stage

execute\_cycle Execute (

.clk(clk),

.rst(rst),

.RegWriteE(RegWriteE),

.ALUSrcE(ALUSrcE),

.MemWriteE(MemWriteE),

.ResultSrcE(ResultSrcE),

.BranchE(BranchE),

.ALUControlE(ALUControlE),

.RD1\_E(RD1\_E),

.RD2\_E(RD2\_E),

.Imm\_Ext\_E(Imm\_Ext\_E),

.RD\_E(RD\_E),

.PCE(PCE),

.PCPlus4E(PCPlus4E),

.PCSrcE(PCSrcE),

.PCTargetE(PCTargetE),

.RegWriteM(RegWriteM),

.MemWriteM(MemWriteM),

.ResultSrcM(ResultSrcM),

.RD\_M(RD\_M),

.PCPlus4M(PCPlus4M),

.WriteDataM(WriteDataM),

.ALU\_ResultM(ALU\_ResultM),

.ResultW(ResultW),

.ForwardA\_E(ForwardAE),

.ForwardB\_E(ForwardBE)

);

// Memory Stage

memory\_cycle Memory (

.clk(clk),

.rst(rst),

.RegWriteM(RegWriteM),

.MemWriteM(MemWriteM),

.ResultSrcM(ResultSrcM),

.RD\_M(RD\_M),

.PCPlus4M(PCPlus4M),

.WriteDataM(WriteDataM),

.ALU\_ResultM(ALU\_ResultM),

.RegWriteW(RegWriteW),

.ResultSrcW(ResultSrcW),

.RD\_W(RDW),

.PCPlus4W(PCPlus4W),

.ALU\_ResultW(ALU\_ResultW),

.ReadDataW(ReadDataW)

);

// Write Back Stage

writeback\_cycle WriteBack (

.clk(clk),

.rst(rst),

.ResultSrcW(ResultSrcW),

.PCPlus4W(PCPlus4W),

.ALU\_ResultW(ALU\_ResultW),

.ReadDataW(ReadDataW),

.ResultW(ResultW)

);

// Hazard Unit

hazard\_unit Forwarding\_block (

.rst(rst),

.RegWriteM(RegWriteM),

.RegWriteW(RegWriteW),

.RD\_M(RD\_M),

.RD\_W(RDW),

.Rs1\_E(RS1\_E),

.Rs2\_E(RS2\_E),

.ForwardAE(ForwardAE),

.ForwardBE(ForwardBE)

);

endmodule

**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**

#### ****B.4 Control Units****

`include "ALU\_Decoder.v"

`include "Main\_Decoder.v"

**module** Control\_Unit\_Top**(**Op**,**RegWrite**,**ImmSrc**,**ALUSrc**,**MemWrite**,**ResultSrc**,**Branch**,**funct3**,**funct7**,**ALUControl**);**

**input** **[**6**:**0**]**Op**,**funct7**;**

**input** **[**2**:**0**]**funct3**;**

**output** RegWrite**,**ALUSrc**,**MemWrite**,**ResultSrc**,**Branch**;**

**output** **[**1**:**0**]**ImmSrc**;**

**output** **[**2**:**0**]**ALUControl**;**

**wire** **[**1**:**0**]**ALUOp**;**

Main\_Decoder Main\_Decoder**(**

**.**Op**(**Op**),**

**.**RegWrite**(**RegWrite**),**

**.**ImmSrc**(**ImmSrc**),**

**.**MemWrite**(**MemWrite**),**

**.**ResultSrc**(**ResultSrc**),**

**.**Branch**(**Branch**),**

**.**ALUSrc**(**ALUSrc**),**

**.**ALUOp**(**ALUOp**)**

**);**

ALU\_Decoder ALU\_Decoder**(**

**.**ALUOp**(**ALUOp**),**

**.**funct3**(**funct3**),**

**.**funct7**(**funct7**),**

**.**op**(**Op**),**

**.**ALUControl**(**ALUControl**)**

**);**

**Endmodule**

#### ****B.5 Memory and Registers****

**module** Data\_Memory**(**clk**,**rst**,**WE**,**WD**,**A**,**RD**);**

**input** clk**,**rst**,**WE**;**

**input** **[**31**:**0**]**A**,**WD**;**

**output** **[**31**:**0**]**RD**;**

**reg** **[**31**:**0**]** mem **[**1023**:**0**];**

**always** **@** **(posedge** clk**)**

**begin**

**if(**WE**)**

mem**[**A**]** **<=** WD**;**

**end**

**assign** RD **=** **(~**rst**)** **?** 32'd0 **:** mem**[**A**];**

**initial** **begin**

mem**[**0**]** **=** 32'h00000000**;**

//mem[40] = 32'h00000002;

**end**

**endmodule**

**module** Instruction\_Memory**(**rst**,**A**,**RD**);**

**input** rst**;**

**input** **[**31**:**0**]**A**;**

**output** **[**31**:**0**]**RD**;**

**reg** **[**31**:**0**]** mem **[**1023**:**0**];**

**assign** RD **=** **(**rst **==** 1'b0**)** **?** **{**32**{**1'b0**}}** **:** mem**[**A**[**31**:**2**]];**

**initial** **begin**

$readmemh**(**"memfile.hex"**,**mem**);**

**end**

/\*

initial begin

//mem[0] = 32'hFFC4A303;

//mem[1] = 32'h00832383;

// mem[0] = 32'h0064A423;

// mem[1] = 32'h00B62423;

mem[0] = 32'h0062E233;

// mem[1] = 32'h00B62423;

end

\*/

**endmodule**

**module** Register\_File**(**clk**,**rst**,**WE3**,**WD3**,**A1**,**A2**,**A3**,**RD1**,**RD2**);**

**input** clk**,**rst**,**WE3**;**

**input** **[**4**:**0**]**A1**,**A2**,**A3**;**

**input** **[**31**:**0**]**WD3**;**

**output** **[**31**:**0**]**RD1**,**RD2**;**

**reg** **[**31**:**0**]** Register **[**31**:**0**];**

**always** **@** **(posedge** clk**)**

**begin**

**if(**WE3 **&** **(**A3 **!=** 5'h00**))**

Register**[**A3**]** **<=** WD3**;**

**end**

**assign** RD1 **=** **(**rst**==**1'b0**)** **?** 32'd0 **:** Register**[**A1**];**

**assign** RD2 **=** **(**rst**==**1'b0**)** **?** 32'd0 **:** Register**[**A2**];**

**initial** **begin**

Register**[**0**]** **=** 32'h00000000**;**

**end**

**endmodule**

#### ****B.6 Program Counter and Support Modules****

**module** Main\_Decoder**(**Op**,**RegWrite**,**ImmSrc**,**ALUSrc**,**MemWrite**,**ResultSrc**,**Branch**,**ALUOp**);**

**input** **[**6**:**0**]**Op**;**

**output** RegWrite**,**ALUSrc**,**MemWrite**,**ResultSrc**,**Branch**;**

**output** **[**1**:**0**]**ImmSrc**,**ALUOp**;**

**assign** RegWrite **=** **(**Op **==** 7'b0000011 **|** Op **==** 7'b0110011 **|** Op **==** 7'b0010011 **)** **?** 1'b1 **:**

1'b0 **;**

**assign** ImmSrc **=** **(**Op **==** 7'b0100011**)** **?** 2'b01 **:**

**(**Op **==** 7'b1100011**)** **?** 2'b10 **:**

2'b00 **;**

**assign** ALUSrc **=** **(**Op **==** 7'b0000011 **|** Op **==** 7'b0100011 **|** Op **==** 7'b0010011**)** **?** 1'b1 **:**

1'b0 **;**

**assign** MemWrite **=** **(**Op **==** 7'b0100011**)** **?** 1'b1 **:**

1'b0 **;**

**assign** ResultSrc **=** **(**Op **==** 7'b0000011**)** **?** 1'b1 **:**

1'b0 **;**

**assign** Branch **=** **(**Op **==** 7'b1100011**)** **?** 1'b1 **:**

1'b0 **;**

**assign** ALUOp **=** **(**Op **==** 7'b0110011**)** **?** 2'b10 **:**

**(**Op **==** 7'b1100011**)** **?** 2'b01 **:**

2'b00 **;**

**endmodule**

**module** hazard\_unit**(**rst**,** RegWriteM**,** RegWriteW**,** RD\_M**,** RD\_W**,** Rs1\_E**,** Rs2\_E**,** ForwardAE**,** ForwardBE**);**

// Declaration of I/Os

**input** rst**,** RegWriteM**,** RegWriteW**;**

**input** **[**4**:**0**]** RD\_M**,** RD\_W**,** Rs1\_E**,** Rs2\_E**;**

**output** **[**1**:**0**]** ForwardAE**,** ForwardBE**;**

**assign** ForwardAE **=** **(**rst **==** 1'b0**)** **?** 2'b00 **:**

**((**RegWriteM **==** 1'b1**)** **&** **(**RD\_M **!=** 5'h00**)** **&** **(**RD\_M **==** Rs1\_E**))** **?** 2'b10 **:**

**((**RegWriteW **==** 1'b1**)** **&** **(**RD\_W **!=** 5'h00**)** **&** **(**RD\_W **==** Rs1\_E**))** **?** 2'b01 **:** 2'b00**;**

**assign** ForwardBE **=** **(**rst **==** 1'b0**)** **?** 2'b00 **:**

**((**RegWriteM **==** 1'b1**)** **&** **(**RD\_M **!=** 5'h00**)** **&** **(**RD\_M **==** Rs2\_E**))** **?** 2'b10 **:**

**((**RegWriteW **==** 1'b1**)** **&** **(**RD\_W **!=** 5'h00**)** **&** **(**RD\_W **==** Rs2\_E**))** **?** 2'b01 **:** 2'b00**;**

**endmodule**

**module** Mux **(**a**,**b**,**s**,**c**);**

**input** **[**31**:**0**]**a**,**b**;**

**input** s**;**

**output** **[**31**:**0**]**c**;**

**assign** c **=** **(~**s**)** **?** a **:** b **;**

**endmodule**

**module** Mux\_3\_by\_1 **(**a**,**b**,**c**,**s**,**d**);**

**input** **[**31**:**0**]** a**,**b**,**c**;**

**input** **[**1**:**0**]** s**;**

**output** **[**31**:**0**]** d**;**

**assign** d **=** **(**s **==** 2'b00**)** **?** a **:** **(**s **==** 2'b01**)** **?** b **:** **(**s **==** 2'b10**)** **?** c **:** 32'h00000000**;**

**endmodule**

**module** PC\_Module**(**clk**,**rst**,**PC**,**PC\_Next**);**

**input** clk**,**rst**;**

**input** **[**31**:**0**]**PC\_Next**;**

**output** **[**31**:**0**]**PC**;**

**reg** **[**31**:**0**]**PC**;**

**always** **@(posedge** clk**)**

**begin**

**if(**rst **==** 1'b0**)**

PC **<=** **{**32**{**1'b0**}};**

**else**

PC **<=** PC\_Next**;**

**end**

**endmodule**

**module** PC\_Adder **(**a**,**b**,**c**);**

**input** **[**31**:**0**]**a**,**b**;**

**output** **[**31**:**0**]**c**;**

**assign** c **=** a **+** b**;**

**endmodule**

**module** Sign\_Extend **(**In**,**ImmSrc**,**Imm\_Ext**);**

**input** **[**31**:**0**]** In**;**

**input** **[**1**:**0**]** ImmSrc**;**

**output** **[**31**:**0**]** Imm\_Ext**;**

**assign** Imm\_Ext **=** **(**ImmSrc **==** 2'b00**)** **?** **{{**20**{**In**[**31**]}},**In**[**31**:**20**]}** **:**

**(**ImmSrc **==** 2'b01**)** **?** **{{**20**{**In**[**31**]}},**In**[**31**:**25**],**In**[**11**:**7**]}** **:** 32'h00000000**;**

**endmodule**

### ****Appendix C: UART Module Design Files****

#### ****C.1 uart\_rx****

// Example: 10 MHz Clock, 115200 baud UART

// CLKS\_PER\_BIT = (Frequency of clk) / (Frequency of UART)

// (10000000)/(115200) = 87

**module** uart\_rx **#(**

**parameter** TICKS\_PER\_BIT **=** 87 // Clock ticks per baud period

**)(**

**input** **logic** clk**,** // System clock

**input** **logic** reset**,** // Synchronous reset

**input** **logic** rx**,** // Serial data input

**output** **logic** **[**7**:**0**]** data\_out**,** // 8-bit parallel data received

**output** **logic** valid // Indicates if received data is valid

**);**

**logic** **[**7**:**0**]** baud\_counter**;** // Counter for baud rate timing

**logic** **[**3**:**0**]** bit\_index**;** // Current bit being received

**logic** **[**9**:**0**]** shift\_reg**;** // Shift logic register (start + data + stop bits)

**logic** **[**7**:**0**]** d\_out**;** // Data output

**typedef** **enum** **logic** **[**1**:**0**]** **{**

IDLE**,** // Idle state

RX**,** // Receiving state

CLEANUP // Cleanup state

**}** state\_t**;**

state\_t state**;**

**always** **@(posedge** clk **or** **posedge** reset**)** **begin**

**if** **(**reset**)** **begin**

d\_out **<=** 0**;**

valid **<=** 1'b0**;**

baud\_counter **<=** 0**;**

bit\_index **<=** 0**;**

shift\_reg **<=** 0**;**

state **<=** IDLE**;**

**end** **else** **begin**

**case** **(**state**)**

IDLE**:** **begin**

valid **<=** 1'b0**;**

**if** **(**rx **==** 1'b0**)** **begin** // Start bit detected

baud\_counter **<=** TICKS\_PER\_BIT **/** 2**;** // To sample in the middle of the bit

bit\_index **<=** 0**;**

state **<=** RX**;**

**end** **else** **begin**

state **<=** IDLE**;**

**end**

**end**

RX**:** **begin**

**if** **(**baud\_counter **==** TICKS\_PER\_BIT **-** 1**)** **begin**

**if** **(**bit\_index **==** 9**)** **begin**

valid **<=** 1'b1**;**

d\_out **<=** shift\_reg**[**8**:**1**];**

state **<=** CLEANUP**;**

**end** **else** **begin**

baud\_counter **<=** 0**;**

shift\_reg**[**bit\_index**]** **<=** rx**;**

bit\_index **<=** bit\_index **+** 1**;**

state **<=** RX**;**

**end**

**end** **else** **begin**

baud\_counter **<=** baud\_counter **+** 1**;**

**end**

**end**

CLEANUP**:** **begin**

**if** **(**baud\_counter **==** TICKS\_PER\_BIT **-** 1**)** **begin**

baud\_counter **<=** 0**;**

**if** **(**rx **==** 1'b1**)** **begin**

state **<=** IDLE**;**

**end**

**end** **else** **begin**

baud\_counter **<=** baud\_counter **+** 1**;**

**end**

**end**

**endcase**

**end**

**end**

**assign** data\_out **=** d\_out**;**

**endmodule**

#### ****C.2 uart\_rx\_tb****

**module** uart\_rx\_tb**();**

**parameter** TICKS\_PER\_BIT **=** 87**;** // Clock ticks per baud period

// Clock period for 50 MHz clock

**localparam** CLK\_PERIOD **=** 20**;** // 20 ns

**logic** clk**;**

**logic** reset**;**

**logic** rx**;**

**logic** **[**7**:**0**]** data\_out**;**

**logic** valid**;**

**logic** **[**7**:**0**]** din**;** // Data to be transmitted

**logic** **[**9**:**0**]** shift\_reg**;**

uart\_rx **#(**

**.**TICKS\_PER\_BIT**(**TICKS\_PER\_BIT**)**

**)** uart\_rx\_DUT **(**

**.**clk**(**clk**),**

**.**reset**(**reset**),**

**.**rx**(**rx**),**

**.**data\_out**(**data\_out**),**

**.**valid**(**valid**)**

**);**

// Clock generation

**initial** **begin**

clk **=** 0**;**

**forever** **#(**CLK\_PERIOD **/** 2**)** clk **=** **~**clk**;**

**end**

// Task to send data

**task** **automatic** send**();**

// reset the DUT

reset **=** 1**;**

rx **=** 1**;**

**@(posedge** clk**);**

reset **=** 0**;**

**@(posedge** clk**);**

din **=** $random**();**

// din = 8'b10010011;

shift\_reg **=** **{**1'b1**,**din**,**1'b0**};**

**for(int** i**=**0**;**i**<**9**;**i**++)** **begin**

rx **=** shift\_reg**[**i**];**

**repeat(**TICKS\_PER\_BIT**)** **@(posedge** clk**);** // means give a delay of 1 baud period after every bit.

**end**

**wait(**valid**);**

**if(**data\_out **==** din**)** **begin**

$display**(**"------------SUCCESS: Data Tx: %h, Data Rx: %h------------"**,** din**,** data\_out**);**

**end** **else** **begin**

$display**(**"------------FAILED: Data Tx: %h, Data Rx: %h------------"**,** din**,** data\_out**);**

**end**

**endtask** //automatic

**initial** **begin**

// transmitt the data

**repeat(**10**)** **begin**

send**();**

**end**

$stop**;**

**end**

**endmodule**

#### ****C.3 uart\_tx****

// Example: 10 MHz Clock, 115200 baud UART

// CLKS\_PER\_BIT = (Frequency of clk) / (Frequency of UART)

// (10000000)/(115200) = 87

**module** uart\_tx **#(**

**parameter** TICKS\_PER\_BIT **=** 87 // Clock ticks per baud period

**)(**

**input** **logic** clk**,** // System clock

**input** **logic** reset**,** // Synchronous reset

**input** **logic** start**,** // Start signal for transmission

**input** **logic** **[**7**:**0**]** data\_in**,** // 8-bit parallel data to transmit

**output** **logic** tx**,** // Serial data output

**output** **logic** busy // Indicates if transmission is ongoing

**);**

**logic** **[**7**:**0**]** baud\_counter**;** // Counter for baud rate timing

**logic** **[**3**:**0**]** bit\_index**;** // Current bit being transmitted

**logic** **[**9**:**0**]** shift\_reg**;** // Shift logic register (start + data + stop bits)

**typedef** **enum** **logic** **{**

IDLE**,** // Idle state

TX // Transmission state

**}** state\_t**;**

state\_t state**;**

**always** **@(posedge** clk **or** **posedge** reset**)** **begin**

**if** **(**reset**)** **begin**

tx **<=** 1'b1**;** // Idle state (UART line high)

busy **<=** 1'b0**;**

baud\_counter **<=** 0**;**

bit\_index **<=** 0**;**

shift\_reg **<=** 0**;**

state **<=** IDLE**;**

**end** **else** **begin**

**case** **(**state**)**

IDLE**:** **begin**

tx **<=** 1'b1**;** // Line remains high

busy **<=** 1'b0**;**

**if** **(**start**)** **begin**

busy **<=** 1'b1**;**

shift\_reg **<=** **{**1'b1**,** data\_in**,** 1'b0**};** // Stop, data, Start bits

bit\_index **<=** 0**;**

state **<=** TX**;**

**end**

**end**

TX**:** **begin**

**if** **(**baud\_counter **==** TICKS\_PER\_BIT **-** 1**)** **begin** // 1 tick before the end of the baud period

baud\_counter **<=** 0**;**

tx **<=** shift\_reg**[**bit\_index**];**

bit\_index **<=** bit\_index **+** 1**;**

**if** **(**bit\_index **==** 10**)** **begin**

state **<=** IDLE**;** // All bits transmitted

tx **<=** 1'b1**;**

**end**

**end** **else** **begin**

baud\_counter **<=** baud\_counter **+** 1**;**

**end**

**end**

**endcase**

**end**

**end**

**endmodule**

#### ****C.4 uart\_tx\_tb****

**module** uart\_tx\_tb**;**

// Parameters for simulation

**parameter** TICKS\_PER\_BIT **=** 87**;** // Clock ticks per baud period

// Clock period for 50 MHz clock

**localparam** CLK\_PERIOD **=** 20**;** // 20 ns

**logic** clk**;**

**logic** reset**;**

**logic** start**;**

**logic** **[**7**:**0**]** data\_in**;**

**logic** tx**;**

**logic** busy**;**

// Instantiate the DUT

uart\_tx **#(**

**.**TICKS\_PER\_BIT**(**TICKS\_PER\_BIT**)**

**)** uart\_tx\_DUT **(**

**.**clk**(**clk**),**

**.**reset**(**reset**),**

**.**start**(**start**),**

**.**data\_in**(**data\_in**),**

**.**tx**(**tx**),**

**.**busy**(**busy**)**

**);**

// Clock generation

**initial** **begin**

clk **=** 0**;**

**forever** **#(**CLK\_PERIOD **/** 2**)** clk **=** **~**clk**;**

**end**

// Test procedure to send data

**task** **automatic** send**();**

// Initialize inputs

reset **=** 1**;**

start **=** 0**;**

// Apply reset

**@(posedge** clk**);**

reset **=** 0**;**

// data\_in = 8'b11111111; // Worst case scenarios

// data\_in = 8'b00000000; // Worst case scenarios

// data\_in = 8'b10101010; // Worst case scenarios

// data\_in = 8'b01010101; // Worst case scenarios

// data\_in = 8'b01111110; // Worst case scenarios

// data\_in = 8'b10000001; // Worst case scenarios

// data\_in = 8'b11110000; // Worst case scenarios

data\_in **=** $random**()** **%** $random**()** **+** $random**();**

start **=** 1**;**

**@(posedge** clk**);**

start **=** 0**;**

**@(posedge** clk**);**

**wait(~**busy**);**

**endtask** //automatic

**initial** **begin**

**repeat(**10**)** // Number of transmissions

send**();** // Send data

$stop**;** // Stop simulation

**end**

// Monitor outputs

**initial** **begin**

// $monitor("rst=%b start=%b data=%b tx=%b busy=%b",

// reset, start, data\_in, tx, busy);

**end**

**endmodule**

// Example: 10 MHz Clock, 115200 baud UART

// CLKS\_PER\_BIT = (Frequency of clk) / (Frequency of UART)

// (10000000)/(115200) = 87

**module** uart\_top **#(**

**parameter** TICKS\_PER\_BIT **=** 87 // Clock ticks per baud period

**)(**

**input** **logic** clk**,** // System clock

**input** **logic** reset**,** // Synchronous reset

**input** **logic** start**,** // Start signal for transmission

**input** **logic** **[**7**:**0**]** data\_in**,** // 8-bit parallel data to transmit

**output** **logic** **[**7**:**0**]** data\_out**,** // 8-bit parallel data output

**output** **logic** busy // Busy signal

**);**

**logic** line**;** // UART transmission line signal (output of tx and input of rx)

// Instantiate the UART transmitter

uart\_tx **#(**

**.**TICKS\_PER\_BIT**(**TICKS\_PER\_BIT**)**

**)** uart\_tx **(**

**.**clk**(**clk**),** // System clock

**.**reset**(**reset**),** // Synchronous reset

**.**start**(**start**),** // Start signal for transmission

**.**data\_in**(**data\_in**),** // 8-bit parallel data to transmit

**.**tx**(**line**),** // UART transmission line signal

**.**busy**(**busy**)** // Busy signal

**);**

// Instantiate the UART receiver

uart\_rx **#(**

**.**TICKS\_PER\_BIT**(**TICKS\_PER\_BIT**)**

**)** uart\_rx **(**

**.**clk**(**clk**),** // System clock

**.**reset**(**reset**),** // Synchronous reset

**.**rx**(**line**),** // UART transmission line signal

**.**valid**(**valid**),** // Data valid signal

**.**data\_out**(**data\_out**)** // 8-bit parallel data output

**);**

**endmodule**

**module** uart\_top\_tb**;**

// Parameters for simulation

**parameter** TICKS\_PER\_BIT **=** 87**;** // Clock ticks per baud period

// Clock period for 50 MHz clock

**localparam** CLK\_PERIOD **=** 20**;** // 20 ns

**logic** clk**;**

**logic** reset**;**

**logic** start**;**

**logic** **[**7**:**0**]** data\_in**;**

**logic** **[**7**:**0**]** data\_out**;**

**logic** busy**;**

// Instantiate the DUT

uart\_top **#(**

**.**TICKS\_PER\_BIT**(**TICKS\_PER\_BIT**)**

**)** uart\_top **(**

**.**clk**(**clk**),**

**.**reset**(**reset**),**

**.**start**(**start**),**

**.**data\_in**(**data\_in**),**

**.**data\_out**(**data\_out**),**

**.**busy**(**busy**)**

**);**

// Clock generation

**initial** **begin**

clk **=** 0**;**

**forever** **#(**CLK\_PERIOD **/** 2**)** clk **=** **~**clk**;**

**end**

// Test procedure to send data

**task** **automatic** send**();**

// Initialize inputs

reset **=** 1**;**

start **=** 0**;**

// Apply reset

**@(posedge** clk**);**

reset **=** 0**;**

// data\_in = 8'b11111111; // Worst case scenarios

// data\_in = 8'b00000000; // Worst case scenarios

// data\_in = 8'b10101010; // Worst case scenarios

// data\_in = 8'b01010101; // Worst case scenarios

// data\_in = 8'b01111110; // Worst case scenarios

// data\_in = 8'b10000001; // Worst case scenarios

// data\_in = 8'b11110000; // Worst case scenarios

data\_in **=** $random**()** **%** $random**()** **+** $random**();**

start **=** 1**;**

**@(posedge** clk**);**

start **=** 0**;**

**@(posedge** clk**);**

**wait(~**busy**);**

**if(**data\_in **!=** data\_out**)**

$display**(**"-----FAILED-----Data tx: %h, Data rx: %h"**,** data\_in**,** data\_out**);**

**else**

$display**(**"-----SUCCESS-----Data tx: %h, Data rx: %h"**,** data\_in**,** data\_out**);**

**endtask** //automatic

**initial** **begin**

**repeat(**10**)** // Number of transmissions

send**();** // Send data

$stop**;** // Stop simulation

**end**

**endmodule**

### ****Appendix D: I²C Module Design Files****

#### ****D.1 i2c\_master (SystemVerilog)****

`timescale 1ns **/** 10ps

**module** i2c\_master**(**

**input** **wire** clk**,**

**input** **wire** rst**,**

**input** **wire** **[**6**:**0**]** addr**,**

**input** **wire** **[**7**:**0**]** data\_in**,**

**input** **wire** enable**,**

**input** **wire** rw**,**

**output** **reg** **[**7**:**0**]** read\_data**,**

**output** **wire** ready**,**

**output** **reg** **[**3**:**0**]** state**,**

**output** **reg** i2c\_clk **=** 0**,**

**inout** i2c\_sda**,**

**inout** **wire** i2c\_scl

**);**

// typedef enum logic [3:0] {

// IDLE,

// START,

// ADDRESS,

// RW,

// WRITE\_DATA,

// WRITE\_ACK,

// READ\_DATA,

// READ\_ACK2,

// STOP

// } state\_t;

// state\_t state; // Declare a variable of type state\_t

**localparam** IDLE **=** 0**;**

**localparam** START **=** 1**;**

**localparam** ADDRESS **=** 2**;**

**localparam** RW **=** 3**;**

**localparam** WRITE\_DATA **=** 4**;**

**localparam** WRITE\_ACK **=** 5**;**

**localparam** READ\_DATA **=** 6**;**

**localparam** READ\_ACK2 **=** 7**;**

**localparam** STOP **=** 8**;**

**localparam** DIVIDE\_BY **=** 1**;**

//reg [7:0] state;

**reg** **[**7**:**0**]** saved\_addr**;**

**reg** **[**7**:**0**]** saved\_data**;**

**reg** **[**7**:**0**]** counter**;**

**reg** **[**7**:**0**]** counter2 **=** 0**;**

**reg** write\_enable**;**

**reg** sda\_out**;**

**reg** i2c\_scl\_enable **=** 0**;**

//reg i2c\_clk = 1;

**assign** ready **=** **((**rst **==** 0**)** **&&** **(**state **==** IDLE**))** **?** 1 **:** 0**;**

**assign** i2c\_scl **=** **(**i2c\_scl\_enable **==** 0 **)** **?** 1 **:** i2c\_clk**;**

**assign** i2c\_sda **=** **(**write\_enable **==** 1**)** **?** sda\_out **:** 1'bz**;**

**always** **@(posedge** clk **or** **negedge** clk**)** **begin**

//if (counter2 == (DIVIDE\_BY/2) - 1) begin

// i2c\_clk <= ~i2c\_clk;

// counter2 <= 0;

// end

// else counter2 <= counter2 + 1;

i2c\_clk **<=** **~**i2c\_clk**;**

**end**

**always** **@(negedge** i2c\_clk**,** **posedge** rst**)** **begin**

**if(**rst **==** 1**)** **begin**

i2c\_scl\_enable **<=** 0**;**

**end** **else** **begin**

**if** **((**state **==** IDLE**)** **||** **(**state **==** START**)** **||** **(**state **==** STOP**))** **begin**

i2c\_scl\_enable **<=** 0**;**

**end** **else** **begin**

i2c\_scl\_enable **<=** 1**;**

**end**

**end**

**end**

**always** **@(posedge** i2c\_clk**,** **posedge** rst**)** **begin**

**if(**rst **==** 1**)** **begin**

state **<=** IDLE**;**

**end**

**else** **begin**

**case(**state**)**

IDLE**:** **begin**

**if** **(**enable**)** **begin**

state **<=** START**;**

saved\_addr **<=** **{**addr**,** rw**};**

saved\_data **<=** data\_in**;**

**end**

**else** state **<=** IDLE**;**

**end**

START**:** **begin**

counter **<=** 7**;**

state **<=** ADDRESS**;**

**end**

ADDRESS**:** **begin**

**if** **(**counter **==** 0**)** **begin**

state **<=** RW**;**

**end** **else** counter **<=** counter **-** 1**;**

**end**

RW**:** **begin**

**if** **(**i2c\_sda **==** 0**)** **begin**

counter **<=** 7**;**

**if(**saved\_addr**[**0**]** **==** 0**)** state **<=** WRITE\_DATA**;**

**else** state **<=** READ\_DATA**;**

**end** **else** state **<=** STOP**;**

**end**

WRITE\_DATA**:** **begin**

**if(**counter **==** 0**)** **begin**

state **<=** READ\_ACK2**;**

**end** **else** counter **<=** counter **-** 1**;**

**end**

READ\_ACK2**:** **begin**

**if** **((**i2c\_sda **==** 0**)** **&&** **(**enable **==** 1**))** state **<=** IDLE**;**

**else** state **<=** STOP**;**

**end**

READ\_DATA**:** **begin**

read\_data**[**counter**]** **<=** i2c\_sda**;**

**if** **(**counter **==** 0**)** state **<=** WRITE\_ACK**;**

**else** counter **<=** counter **-** 1**;**

**end**

WRITE\_ACK**:** **begin**

state **<=** STOP**;**

**end**

STOP**:** **begin**

**if(**rst **==** 0**)** **begin**

state **<=** IDLE**;**

i2c\_scl\_enable **<=** 0**;**

**end**

**else**

state **<=** START**;**

**end**

**endcase**

**end**

**end**

**always** **@(negedge** i2c\_clk**,** **posedge** rst**)** **begin**

**if(**rst **==** 1**)** **begin**

write\_enable **<=** 1**;**

sda\_out **<=** 1**;**

**end** **else** **begin**

**case(**state**)**

START**:** **begin**

write\_enable **<=** 1**;**

sda\_out **<=** 0**;**

**end**

ADDRESS**:** **begin**

sda\_out **<=** saved\_addr**[**counter**];**

**end**

RW**:** **begin**

write\_enable **<=** 0**;**

**end**

WRITE\_DATA**:** **begin**

write\_enable **<=** 1**;**

sda\_out **<=** saved\_data**[**counter**];**

**end**

WRITE\_ACK**:** **begin**

write\_enable **<=** 1**;**

sda\_out **<=** 0**;**

**end**

READ\_DATA**:** **begin**

write\_enable **<=** 0**;**

//read\_data[counter] <= i2c\_sda;

**end**

STOP**:** **begin**

write\_enable **<=** 1**;**

sda\_out **<=** 1**;**

**end**

**endcase**

**end**

**end**

**endmodule**

#### ****D.2 i2c\_slave (SystemVerilog)****

`timescale 1ns **/** 10ps

**module** i2c\_slave**(**

**output** **reg** **[**7**:**0**]** recived\_addr**,**

**output** **reg** **[**7**:**0**]** write\_data**,**

**inout** sda**,**

**inout** scl

**);**

**localparam** ADDRESS **=** 7'b0101010**;**

**localparam** READ\_ADDR **=** 0**;**

**localparam** SEND\_ACK **=** 1**;**

**localparam** READ\_DATA **=** 2**;**

**localparam** WRITE\_DATA **=** 3**;**

**localparam** SEND\_ACK2 **=** 4**;**

**reg** **[**7**:**0**]** addr**;**

**reg** **[**7**:**0**]** counter**;**

**reg** **[**7**:**0**]** state **=** 0**;**

**reg** **[**7**:**0**]** data\_in **=** 0**;**

**reg** **[**7**:**0**]** data\_out **=** 8'b11001100**;**

**reg** sda\_out **=** 0**;**

**reg** sda\_in **=** 0**;**

**reg** start **=** 0**;**

**reg** write\_enable **=** 0**;**

**assign** sda **=** **(**write\_enable **==** 1**)** **?** sda\_out **:** 1'bz**;**

**always** **@(negedge** sda**)** **begin**

**if** **((**start **==** 0**)** **&&** **(**scl **==** 1**))** **begin**

start **<=** 1**;**

counter **<=** 7**;**

**end**

**end**

**always** **@(posedge** sda**)** **begin**

**if** **((**start **==** 1**)** **&&** **(**scl **==** 1**))** **begin**

state **<=** READ\_ADDR**;**

start **<=** 0**;**

write\_enable **<=** 0**;**

**end**

**end**

**always** **@(posedge** scl**)** **begin**

**if** **(**start **==** 1**)** **begin**

**case(**state**)**

READ\_ADDR**:** **begin**

addr**[**counter**]** **<=** sda**;**

recived\_addr**[**counter**]** **<=** sda**;**

**if(**counter **==** 0**)** state **<=** SEND\_ACK**;**

**else** counter **<=** counter **-** 1**;**

**end**

SEND\_ACK**:** **begin**

**if(**addr**[**7**:**1**]** **==** ADDRESS**)** **begin**

counter **<=** 7**;**

**if(**addr**[**0**]** **==** 0**)** **begin**

state **<=** READ\_DATA**;**

**end**

**else** state **<=** WRITE\_DATA**;**

**end**

**end**

READ\_DATA**:** **begin**

data\_in**[**counter**]** **<=** sda**;**

write\_data**[**counter**]** **<=** sda**;**

**if(**counter **==** 0**)** **begin**

state **<=** SEND\_ACK2**;**

**end** **else** counter **<=** counter **-** 1**;**

**end**

SEND\_ACK2**:** **begin**

state **<=** READ\_ADDR**;**

**end**

WRITE\_DATA**:** **begin**

**if(**counter **==** 0**)** state **<=** READ\_ADDR**;**

**else** counter **<=** counter **-** 1**;**

**end**

**endcase**

**end**

**end**

**always** **@(negedge** scl**)** **begin**

**case(**state**)**

READ\_ADDR**:** **begin**

write\_enable **<=** 0**;**

**end**

SEND\_ACK**:** **begin**

sda\_out **<=** 0**;**

write\_enable **<=** 1**;**

**end**

READ\_DATA**:** **begin**

write\_enable **<=** 0**;**

**end**

WRITE\_DATA**:** **begin**

write\_enable **<=** 1**;**

sda\_out **<=** data\_out**[**counter**];**

**end**

SEND\_ACK2**:** **begin**

sda\_out **<=** 0**;**

write\_enable **<=** 1**;**

**end**

**endcase**

**end**

**endmodule**

#### ****D.3 i2c\_testbench (SystemVerilog)****

`timescale 1ns **/** 10ps

**module** i2c\_testbench**;**

// Inputs

**reg** clk**;**

**reg** rst**;**

**reg** **[**6**:**0**]** addr**;**

**reg** **[**7**:**0**]** data\_in**;**

**reg** enable**;**

**reg** rw**;**

// Outputs

**wire** **[**7**:**0**]** read\_data**;**

**wire** **[**7**:**0**]** recived\_addr**;**

**wire** **[**7**:**0**]** write\_data**;**

**wire** ready**;**

**wire** **[**3**:**0**]** state**;**

**wire** i2c\_clk**;**

// Bidirs

**wire** i2c\_sda**;**

**wire** i2c\_scl**;**

// Instantiate the Unit Under Test (UUT)

i2c\_master master **(**

**.**clk**(**clk**),**

**.**rst**(**rst**),**

**.**addr**(**addr**),**

**.**data\_in**(**data\_in**),**

**.**enable**(**enable**),**

**.**rw**(**rw**),**

**.**read\_data**(**read\_data**),**

**.**ready**(**ready**),**

**.**state**(**state**),**

**.**i2c\_clk**(**i2c\_clk**),**

**.**i2c\_sda**(**i2c\_sda**),**

**.**i2c\_scl**(**i2c\_scl**)**

**);**

i2c\_slave slave **(**

**.**recived\_addr**(**recived\_addr**),**

**.**write\_data**(**write\_data**),**

**.**sda**(**i2c\_sda**),**

**.**scl**(**i2c\_scl**)**

**);**

**initial** **begin**

clk **=** 0**;**

**forever** **begin**

clk **=** **#**5 **~**clk**;**

**end**

**end**

**initial** **begin**

// Initialize Inputs

clk **=** 0**;**

rst **=** 1**;**

// Wait 100 ns for global reset to finish

**#**10**;**

// Add stimulus here

rst **=** 0**;**

addr **=** 7'b0101010**;**

data\_in **=** 8'b11101011**;**

rw **=** 0**;**

enable **=** 1**;**

**#**10**;**

//enable = 0;

**#**190

rw**=**1**;**

**#**300

$finish**;**

**end**

**endmodule**

### ****Appendix E : FIFO Buffer Implementation****

This appendix contains the SystemVerilog source code and testbench for a FIFO (First-In-First-Out) memory buffer. FIFO buffers are commonly used in UART communication and processor designs to manage temporary data storage and synchronization between modules.

#### ****E.1 FIFO Design****

**module** FIFO **#(parameter** WIDTH **=** 4**,**

**parameter** DEPTH **=** 8 //8

**)(** clk**,** rst**,** buf\_in**,** buf\_out**,** wr\_en**,** rd\_en**,** buf\_empty**,** buf\_full**,** fifo\_counter **);**

**input** rst**,** clk**,** wr\_en**,** rd\_en**;**

**input** **[**WIDTH**-**1**:**0**]** buf\_in**;**

**output[**WIDTH**-**1**:**0**]** buf\_out**;**

**output** buf\_empty**,** buf\_full**;**

//output[6:0] fifo\_counter;

**output[**3**:**0**]** fifo\_counter**;**

**logic** **[**WIDTH**-**1**:**0**]** buf\_out**;**

**logic** buf\_empty**,** buf\_full**;**

//logic [6:0] fifo\_counter;

//logic [6:0] rd\_ptr, wr\_ptr;

**logic** **[**3**:**0**]** fifo\_counter**;**

**logic** **[**3**:**0**]** rd\_ptr**,** wr\_ptr**;**

**logic** **[**WIDTH**-**1**:**0**]** buf\_mem**[**DEPTH**-**1**:**0**];**

**always** **@(**fifo\_counter**)** // gives the status flage

**begin**

buf\_empty **=** **(**fifo\_counter**==** 0**);**

buf\_full **=** **(**fifo\_counter**==** DEPTH**-**1**);**

**end**

// this is fifo conuter which counts data

**always** **@(posedge** clk **or** **posedge** rst**)**

**begin**

**if(** rst **)**

fifo\_counter **<=** 0**;**

**else** **if(** **(!**buf\_full **&&** wr\_en**)** **&&** **(** **!**buf\_empty **&&** rd\_en **)** **)**

fifo\_counter **<=** fifo\_counter**;**

**else** **if(** **!**buf\_full **&&** wr\_en **)**

fifo\_counter **<=** fifo\_counter **+** 1**;**

**else** **if(** **!**buf\_empty **&&** rd\_en **)**

fifo\_counter **<=** fifo\_counter **-** 1**;**

**else**

fifo\_counter **<=** fifo\_counter**;**

**end**

//fecthing data from the fifo

**always** **@(** **posedge** clk **or** **posedge** rst**)**

**begin**

**if(** rst **)**

buf\_out **<=** 0**;**

**else**

**begin**

**if(** rd\_en **&&** **!**buf\_empty **)**

buf\_out **<=** buf\_mem**[**rd\_ptr**];**

**else**

buf\_out **<=** buf\_out**;**

**end**

**end**

//writing data into the fifo

**always** **@(posedge** clk**)**

**begin**

**if(** wr\_en **&&** **!**buf\_full **)**

buf\_mem**[** wr\_ptr **]** **<=** buf\_in**;**

**else**

buf\_mem**[** wr\_ptr **]** **<=** buf\_mem**[** wr\_ptr **];**

**end**

//manage the pointer and buffer location

**always@(posedge** clk **or** **posedge** rst**)**

**begin**

**if(** rst **)**

**begin**

wr\_ptr **<=** 0**;**

rd\_ptr **<=** 0**;**

**end**

**else** **begin**

**if(** **!**buf\_full **&&** wr\_en **)begin**

wr\_ptr **<=** wr\_ptr **+** 1**;**

**end**

**else** **begin**

wr\_ptr **<=** wr\_ptr**;**

**end**

**if(** **!**buf\_empty **&&** rd\_en **)begin**

rd\_ptr **<=** rd\_ptr **+** 1**;**

**end**

**else** **begin**

rd\_ptr **<=** rd\_ptr**;**

**end**

**end**

**end**

**endmodule**

#### ****E.2 FIFO Testbench****

//////////TESTBENCH///////////////

`timescale 1ns**/**1ps

**module** FIFO\_tb **();**

**parameter** WIDTH **=** 4**;**//4

**parameter** DEPTH **=** 8**;** //8

**bit** clk**;** // clock

**logic** reset**,** wr\_en**,** rd\_en**;** // input signals

**logic** **[**WIDTH**-**1**:**0**]** data\_in**;** // 32bit input word length

**logic** **[**WIDTH**-**1**:**0**]** data\_out**;** // 32bit output word length

**logic** full**,** empty**;** // output indicator signals for full / empty

// logic [6:0] fifo\_counter;

**logic** **[**3**:**0**]** fifo\_counter**;**

**logic** **[**WIDTH**-**1**:**0**]** data\_out\_svd\_fifo **[**DEPTH**:**0**];** //temporary variable to store the contents of the read mode of FIFO

**logic** **[**WIDTH**-**1**:**0**]** data\_out\_svd\_q **[**DEPTH**:**0**];** //temporary variable to store the contents of the POP mode of QUEUE

//generating CLOCK

**always** **#**10 clk **=** **~**clk**;**

// // Instantiate the FIFO module

FIFO dut **(**

**.**clk**(**clk**),**

**.**rst**(**reset**),**

**.**buf\_in**(**data\_in**),**

**.**buf\_out**(**data\_out**),**

**.**wr\_en**(**wr\_en**),**

**.**rd\_en**(**rd\_en**),**

**.**buf\_empty**(**empty**),**

**.**buf\_full**(**full**),**

**.**fifo\_counter**(**fifo\_counter**)**

**);**

// Queue implementation

**logic** **[**31**:**0**]** queue **[**$**];**

//logic [WIDTH-1:0] queue[$:DEPTH-1]; // sample bounded empty queue

**int** i **=** 0**;**

**integer** ridle**;**

// initial block

**initial** **begin**

reset **=** 1'b1**;**

wr\_en **=** 0**;**

rd\_en **=** 0**;**

**#**16**;**

reset **=** 1'b0**;**

**@(posedge** clk**);**

**fork**

**begin**

**repeat(**10000**)** **begin**

wr\_en **=** 1**;**

data\_in **=** i**+**1**;**

**@(posedge** clk**);**

wr\_en **=** 0**;**

ridle **=** $random**%**10**;**

**if** **(**ridle **>** 0**)**

**repeat** **(**ridle**)**

**@(posedge** clk**);**

i**++;**

**end**

**end**

// repeat (5)

// @(posedge clk);

**begin**

**repeat(**10000**)** **begin**

rd\_en **=** **~**empty**;**

**@(posedge** clk**);**

rd\_en **=** 0**;**

ridle **=** $random**%**10**;**

**if** **(**ridle **>** 0**)**

**repeat** **(**ridle**)**

**@(posedge** clk**);**

**end**

**end**

**join**

**#**1000

$display**(**"---------------- NORMAL STOP-------------"**,);**

$stop**;**

**end**

**logic** rd\_en\_d**;**

**always** **@(posedge** clk**)**

rd\_en\_d **<=** rd\_en **&** **~**empty**;**

**int** rcnt**;**

**logic** **[**WIDTH**-**1**:**0**]** mdl\_data**;**

**always** **@(posedge** clk**)**

**if** **(**rd\_en\_d**)** **begin**

rcnt **<=** rcnt **+** 1**;**

mdl\_data **=** queue**.**pop\_front**();**

**if** **(**mdl\_data **!=** data\_out**)**

$display**(**"RD %3d MISMATCH RTL %d EXPTD %d"**,**rcnt**,**data\_out**,**mdl\_data**);**

**if** **(**mdl\_data **!=** data\_out**)**

$stop**;**

**end**

**GitHub:** [**https://github.com/RAOUMERTC73/FYDP2021-RISCV-for-SOC?tab=readme-ov-file**](https://github.com/RAOUMERTC73/FYDP2021-RISCV-for-SOC?tab=readme-ov-file)

// if wr\_en asserted

**always\_ff** **@(posedge** clk**)**

**begin**

**if(**wr\_en **&** **!**full**)**

**begin**

queue**.**push\_back**(**data\_in**);**

$display**(**"queue = %p"**,**queue**);**

**end**

**end**

**endmodule** **:** FIFO\_tb