Department of Electronics Electrical Engineering / Mechatronics Engineering

RISC-V Based Microcontroller using Arrow Deca Max 10 FPGA development Board

Dissertation

Kaung Min Khant

Supervisor: Dr. Thet Htun Khine

2^{nd} Marker:

6556-6555ELEMM Aug. 2023

Acknowledges

This work would not have been possible without the support of Auston University. I am especially indebted to Dr. Thet Htun Khine, who have been supportive of my research and who worked actively to provide me with the protected academic time to meet the required goals. I am grateful to all of those with whom I have had the pleasure to work during this and other related projects.

Nobody has been more important to me in the pursuit of this project than the members of my family, and friends. I would like to thank my parents, whose love and guidance are with me in whatever I pursue. They are the ultimate role models. Most importantly, I wish to thank my friends, Ko Htet Linn Aung, Ko Minn Bo Bo, Ko Naing Lin Aung for support when I damaged my laptop and when I was stuck during research.

Abstract

Microcontrollers have become a ubiquitous and vital component in a wide range of applications, from embedded systems to Internet of Things (IoT) devices. With the growing demand for efficient and versatile computing solutions, the RISC-V (Reduced Instruction Set Computing - V) Instruction Set Architecture (ISA) has emerged as a promising alternative to proprietary and traditional ISAs.

This thesis presents the design and implementation of a RISC-V microcontroller tailored to meet the demands of modern computing systems. The proposed microcontroller architecture leverages the inherent advantages of RISC-V, including simplicity, modularity, and extensibility, to deliver enhanced performance and versatility compared to existing solutions.

The research begins with a comprehensive study of the RISC-V ISA, identifying its key features and design principles. Based on this analysis, a custom microarchitecture is devised to optimize critical components such as instruction decoding, pipeline stages, and data forwarding mechanisms. The microcontroller is then synthesized and simulated using industry-standard Electronic Design Automation (EDA) tools to assess its performance, and area utilization.

The results of this research illustrate the feasibility and benefits of employing RISC-V as the foundation for designing microcontrollers. The proposed architecture's ability to achieve improved performance and adaptability to application-specific needs lays the groundwork for broader adoption of RISC-V in future computing systems.

In conclusion, this thesis contributes to the growing body of knowledge surrounding RISC-V microcontroller design and serves as a stepping stone for further advancements in the field of open-source and customizable processor architectures. As RISC-V gains momentum in the industry, the developed microcontroller has the potential to drive innovation and address the ever-evolving demands of the digital era.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Project Background

In recent years, the rapid advancement of technology has led to an exponential growth in the number of embedded systems and Internet of Things (IoT) devices. These systems demand efficient, low-power, and versatile microcontrollers capable of executing complex tasks while maintaining cost-effectiveness. To address these challenges, the development of a RISC-V-based microcontroller presents an attractive and innovative solution.

The Reduced Instruction Set Computing - V (RISC-V) Instruction Set Architecture (ISA) is an open standard, freely available for anyone to use, modify, and implement. Its modular and scalable design has gained significant attention within the semiconductor industry and academic community. Unlike proprietary ISAs, RISC-V fosters collaboration and fosters a rich ecosystem of open-source hardware and software projects, leading to reduced development costs and increased accessibility.

This project aims to design a RISC-V microcontroller that leverages the benefits of the RISC-V ISA, tailored to meet the specific requirements of modern computing applications. The motivation behind this work aims to provide the detail designs of several parts of the microcontroller, aiming to act a reference and a base for future design and implementation of the microcontroller core using RISC-V ISA.

1.2 Aims and Objectives

This research is proposed for construction of RISC-V Hardware using VHDL and implemented on FPGA. The objectives of this research are as follow.

1. To build a complete microcontroller, yet simple enough to follow, based on RISC-V RV32E ISA using VHDL.

2. To offer detail documentation of the implementation of the hardware

3. To offer hardware extension for future developement

4. To offer assembler for firmware creation

5. To offer implementation support for hardware designers to build their own hardwares based on the required projects

1.3 Expected Outcomes

The expected outcomes of the research are as follows

1. The CPU must work based on the specified ISA, in this case, RICS-V RV32E ISA
2. The CPU must be tested
3. The overall micro-controller must have standard peripherals and they must be tested
4. The documentation of all of the hardware implements must be provided
5. The research will be published on GitHub Repo

1.4 Report Structure

This report has been broken down into sections.

1. Chapter 1: Introduction - Introduction above this work
2. Chapter 2: Literature Review - A review of current microcontroller works around RISC-V
3. Chapter 3: Microcontroller CPU Design and Implementation - RISC-V CPU Core Design and Implementations
4. Chapter 4: Peripheral Designs and Implementation - Design of peripherals for CPU Core
5. Chapter 5: Simulation Testings - Testings of CPU and Peripherals in simulation
6. Chapter 6: Conclusion - Conclusion that can be drawn from this work

1.5 Conclusion

By successfully designing and implementing a RISC-V microcontroller with a focus on performance and versatility, this project seeks to make a valuable contribution to the field of embedded systems and IoT devices. Embracing the open-source philosophy, the project's outcomes will be shared with the broader community, encouraging collaboration and pushing the boundaries of what RISC-V-based microcontrollers can achieve.

Chapter 2

Literature Review

The development of a RISC-V microcontroller with a focus on simplicity and comprehensive documentation availability has gained significant interest in both academic and industrial circles. RISC-V's open-source nature allows for greater accessibility and collaboration, making it an attractive choice for microcontroller designs. This literature review aims to explore relevant research and publications related to RISC-V-based microcontroller designs, with an emphasis on simplicity in architecture and the availability of extensive documentation.

"The RISC-V Instruction Set Manual, Volume I: User-Level ISA" by Andrew Waterman provides an in-depth documentation of the RISC-V user-level instruction set architecture, offering a comprehensive understanding of the ISA's base instructions. It serves as a crucial reference for microcontroller designers seeking to implement a simple and standard-compliant RISC-V core. The clear and well-structured presentation of the ISA facilitates an uncomplicated design process, encouraging developers to create efficient and robust microcontrollers [Waterman et al., 2017]

The paper by Daniel Brisk, focuses on architectural simplicity while maintaining high performance and low power consumption. RI5CY's generator includes customization options, allowing designers to tailor the microcontroller for specific applications. The work underscores the importance of simplicity in the microcontroller design process while enabling adaptability to diverse use cases [Brisk, 2017].

Rocket Chip, RISC-V Processer by Yunsup Lee, provides design philosophy emphasizes simplicity, enabling ease of understanding, modification, and extension. Rocket Chip is highly configurable, allowing designers to create microcontrollers suited to their specific requirements. The paper highlights the significance of open-source projects in facilitating collaboration and fostering a community-driven approach to microcontroller design [Lee, 2016].

Frank K. Gürkaynak present PULPino, an open-source RISC-V processor designed for Internet of Things (IoT) applications. The focus of the project is on simplicity and energy efficiency, making it suitable for resource-constrained devices. PULPino's architecture is thoroughly documented, promoting ease of use and customization for specific IoT use cases [Gurkaynak, 2019].

Ibex, by Stefan Wallentowitz, is an open-source, parameterizable RISC-V processor core designed for flexibility and simplicity. The paper highlights the core's configurability and comprehensible architecture, which allows designers to optimize it for various applications. Ibex's extensive documentation and open-source nature facilitate community-driven improvements and encourage knowledge exchange [Wallentowitz, 2023].

VexRiscv by Charles Papon is an open-source RISC-V CPU implementation in SpinalHDL. The project emphasizes clarity and modularity, allowing developers to easily understand and modify the

processor core. The repository includes extensive documentation, making it a valuable resource for those interested in building customizable RISC-V microcontrollers [Papon, 2023].

The paper for AnnikaCore, by Yunrui Zhang, discusses the growing market demand for embedded IoT processors due to the booming IoT industry. It highlights the RISC-V instruction set architecture, known for its concise coding and modular extensions, as an ideal choice for embedded IoT processors. The paper presents the design of a 3-stage pipelined scalar micro-out-of-order processor based on RISC-V's RV32IMA instruction set. The processor has been verified through simulation and FPGA prototype, showing functional correctness with a Coremark performance of 2.93 Coremark/MHz. The final implementation used SMIC 180nm process with a main frequency of 50MHz, resulting in a core circuit of 35K gate and a power consumption of 0.20 mW/MHz [Zhang et al., 2021].

The above works, although rich in information and advances in RISC-V Microcontroller technology, lack in a way that prevents a beginner to start designing a customized microcontroller along with providing a comprehensive documentation. They lack either in the detail design steps or in the implementation itself using a hardware description language. This work forcus around designing a simple RISC-V microcontroller with a simple enough documentation so that a beginner will have an entry to start designing the desired microcontroller. This work also provides a way for producing a customized assembler for producing the machine code for the controller.

Chapter 3

Microcontroller CPU Design and Implementation

Chapter 4

Peripheral Designs and Implementations

4.1 UART - Universal Asynchronous Receiver Transmitter

This universal asynchronous receiver transmitter (UART) module is written for Field Programmable Logic Array (FPGA) implementation with VHDL. This module can be duplicated and controlled using external logic or controllers. The module can be initialized with desired clock and baudrate. Only one baudrate is used for both transmitter and receiver. Little-endianess is used for both transmission and reception. These can be modified if required.

4.1.1 Why use UART

The UART port provides a simplified UART interface to other peripherals or hosts, supporting full-duplex, DMA, and asynchronous transfer of serial data. The UART port includes support for five to eight data bits, and none, even, or odd parity. A frame is terminated by one and a half or two stop bits.[Pena and Legaspi, 2020]

4.1.2 Block Diagram and I/O

Figure 4.1: UART Module Block Diagram

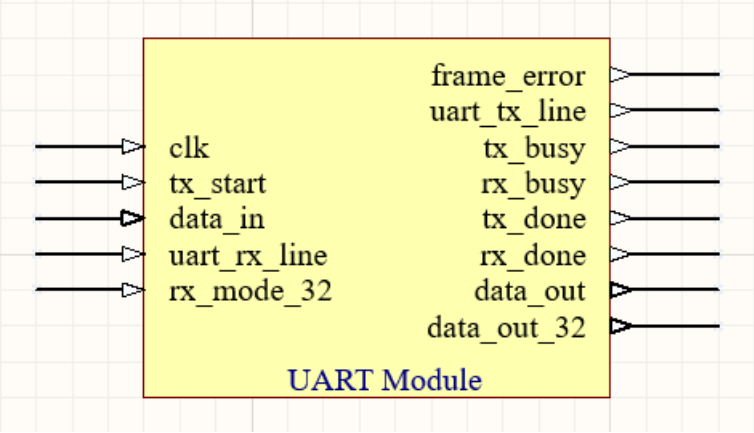


Table 4.1: Input/Output Table for UART Module

Input Name	Function	Output Name	Function
clk	main clock input	**uart_tx_line**	UART TX line
tx_start	transmitter start bit	**tx_busy**	transmitter busy bit
data_in	32 bit data input	**rx_busy**	receiver busy bit
uart_rx_line	UART RX line	**tx_done**	transmitter done bit
		rx_done	receiver busy bit
		data_out	8 bit receiver data out
		data_out_32	32 bit receiver data out
		frame_error	Frame Error

4.1.3 Usage

The UART module consists of transmitter and transmitter. The transmitter and receiver can work simultaneously without interfering with each other. The module has 32 bit data input for transmission with 8 bit or 32 bit data output for reception.

The transmission can be initiated by setting **tx_start** bit. The designer should clear **tx_start** bit after setting it to prevent it from triggering multiple times. Data will be transmitted via **uart_tx_line**. During transmission, **tx_busy** bit will be set to indicate transmission and will be cleared after transmission. **tx_done** bit will be pulsed high once the transmission is completed. The designer can detect the rising edge, falling edge or the pulse level to determine transmission completion.

The receiving is initiated automatically upon receiving the start bit on **uart_rx_line**. **rx_busy** is set during receiving and cleared upon completion. **rx_done** is pulsed once upon completion. The designer must use falling edge of the pulse to ensure the data has been stored within the receiver. If 32 bit receiver mode is selected, **rx_done** is pulsed only after receiving 4 bytes.

The received data will be available on **data_out** and 32 bit **data_out_32** lines based on **rx_mode_32** input. If **rx_mode_32** input is set, the receiver will expect four 8 bit data pack-

ets to complete 32 bit. If **rx_mode_32** input is cleared, the receiver will only expect a single 8 bit data packets. (*Note: Protection feature should be implemented to provide the failure of receiving four bytes to the system. Currently the system will always set* ***rx_busy*** *if it fails to complete the reception*)

4.1.3.1 Initialization

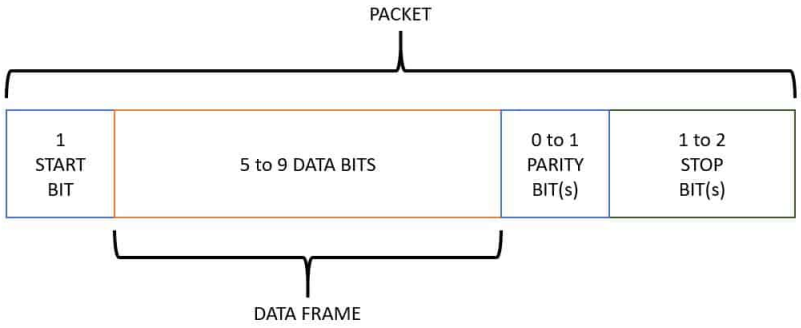
The module has two generic variables which needs to be set according to the desired clock frequency and baudrate.
input_clock_frequency : system input clock in Hz *Example: 10e6*
baudrate : desired baud rate for transmission and reception *Example: 9600*

4.1.4 UART Architecture

4.1.4.1 Data Frame

The data frame for the module is chosen for the standard data frame. It is composed as figure 4.2. Data frame error is checked for every receiving data frame to ensure data integrity. (*Parity bits can be implement as optional feature*)

Figure 4.2: Data Frame used in UART Module



4.1.4.2 Implementation

Implemetation can be found on the GitHub listed in A.1. The transmitter and receiver are implemented as separated modules which are then initialized and combined with top-level VHDL module. The implementation can be accessed through the following GitHub repo: https://github.com/kaung-minkhant/risc-v-micro/tree/master/peripherals/uart.

Transmitter Implementation The transmitter receives 32 bit data, which needs to be seperated, since standard UART only transmits 8 bit of data at a time. (*This behavior can be modified using necessary controls and implementation*). It is accomplished by simple data truncation to respective data groups.

The module is set to work every system clock cycle. However, in order to match the desired baudrate, counters are utilized. Once the transmitter start signal (in this case: **send**) is set, the necessary flags are raised and transmission is started on the next clock cycle. In order to transmit all 4 bytes of data, a counter **word** is utilized to keep track. Each segment is modified to fit into correct data frame with a single stop bit and start bit. According to the system clock cycle and desired baudrate, the count value for full bit can be obtained using

$$full\_bit\_count = \frac{System\_clock}{baudrate}$$

According to UART specifications, each data bit need to be transmitted at the middle of the full bit count to ensure data integrity. **index** is used to keep track of bit position. The data is transmitted on **tx** signal. Once all the bits are transmitted and transmission is completed. **tx_done** output is pulsed high as soon as the transmission is completed and transmission line is kept high until **send** signal is asserted.

Receiver Implementation The receiver will receive data on **rx** signal line at a given baudrate. The calculation is identical as transmitter baudrate calculation. Full bit counting is used the same way as the transmitter. At every rising edge of system clock cycle, the **rx** line is monitored, and once the start bit is properly detected, the process of receiving starts and necessary flags are raised. The start bit on the **rx** line must be held low for half bit count in order to be read as valid data, as per UART specification. **index** is used to keep track of bit addressing.

Currently, two modes of receiving is implemented: 8 bit mode and 32 bit mode. This mode can be changed using **mode32** signal. If **mode32** signal is set, 32 bit receiving mode is selected.

In 8 bit receiver mode, the data on the **rx** signal at half bit count to ensure data integrity. **rx_done** is pulsed high at the end of receiving 8 bit data.

index variable is used to keep track of bit addressing. In 32 bit receiver mode, **word** variable is used to keep track of 8 bit word addressing. **rx_done** is pulsed high at the end of receiving 4 bytes. All flags are reset after each completion.

In both modes, data packet integrity check is performed on the received data. In 8 bit mode, the defected data will be ignored and will assert **frame_error** signal and wait for resend. In 32 bit mode, the defected word will be ignored and will assert **frame_error** signal and wait for resend.

4.2 Inter-Integrated Circuit Protocol(I^2C)

This Inter-Integrated Circuit Protocol(I^2C) module is written for Field Programmable Logic Array (FPGA) implementation with VHDL. This module can be duplicated and controlled using external logic or controllers. This module can be initialized with desired system clock and desired bus clock frequency. This I^2C module supports peripheral clock stretching due to internal clock generation which is separated from system clock.

4.2.1 Why use I^2C

I^2C can be used if the application needs one or more controllers with multiple peripheral devices.

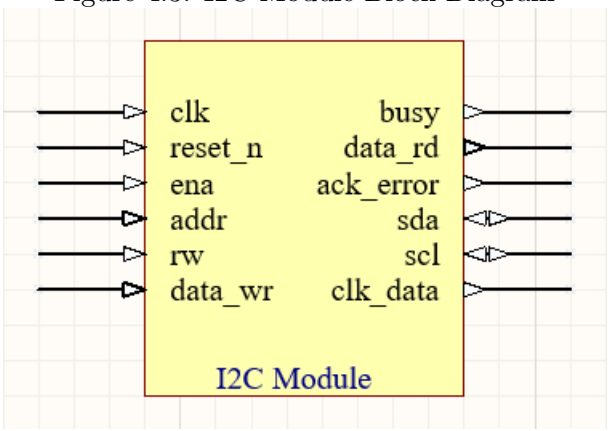
Other two popular choices for data communication are SPI and UART. However, SPI only allows one controller on the bus. The most obvious drawback of SPI is the number of pins required, requiring 4 lines for each peripheral device.

UART will work reliably only when the controller and the peripheral device agrees on the baudrate. Moreover, UART usually is utilized for communication between one controller and one peripheral device.

I^2C provides the best of both world, allowing multiple peripheral devices with relatively fast data rate using just two wire lines. [sparkfun and SFUPTOWNMAKER, 2015]

4.2.2 Block Diagram and I/O

Figure 4.3: I2C Module Block Diagram



4.2.3 Usage

The I^2C module only consists of one overall module, which is called a master or controller. The system clock of desired frequency need to be feed through **clk**. This module can be reset using active low **reset_n**.

To operate this module, the designer places 7 bit slave or peripheral address on **addr**, places read command(1) or write command(0) on **wr**, the data to be written if write command is chosen on **data_wr** while keeping **ena** low. If the module is ready to transmit, pulse **ena** high. This will initiate the communication between the peripheral device using **sda** and **scl** lines while **busy** will go high indicating the module is in use. If the peripheral doesn't acknowledge the command,

Table 4.2: Input/Output Table for I^2C Module

Input Name	Function	Output Name	Function
clk	main clock input	**busy**	indicate whether module is busy
reset_n	asynchronous reset	**data_rd**	data read from read operation
ena	to initiate communication	**ack_error**	peripheral acknowledge error
addr	peripheral address	**sda**	I^2C data line
rw	read/write command	**scl**	I^2C clock line
data_wr	data to be written to peripheral	**clk_data**	internal data clock for debugging

ack_error will be raised. If the data is read from the peripheral device, the data read will be available on **data_rd**. This data can be read by the controller after the **busy** becomes low.

If the designer wishes to write or read a single byte, **ena** should be pulsed, meaning **ena** should be pulled low, as soon as **busy** is set. If the designer wishes to write or read multiple bytes, **ena** should be kept high. New data needs to be updated on the rising edge of **busy**. After the last byte is updated, **ena** should be pulled low.

4.2.3.1 Initialization

The module has two generic variables which needs to be set according to the desired system clock frequency and scl bus clock frequency.
input_clk : system input clock in Hz *Example: 10e6*
bus_clk : desired scl clock rate *standards: 1e6, 3.4e6, 5e6*

4.2.4 I2C Architecture

4.2.4.1 I^2C Operation and Data Frame

The overall communication transactions for reading and writing operations of the controller module can be described as in figure 4.4

The data frame implemented in the I^2C module is shown in figure 4.5

Acknowledge bit is checked between each transaction to make sure the peripheral device responds correctly.

Figure 4.4: Communication Transactions for I^{2}C Operation

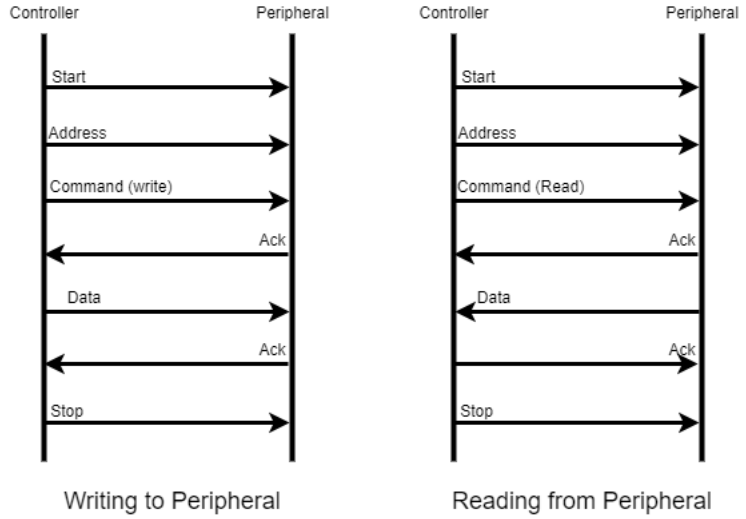
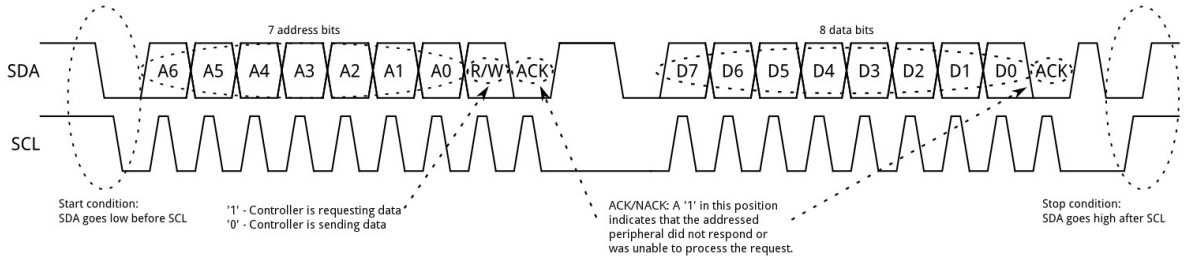


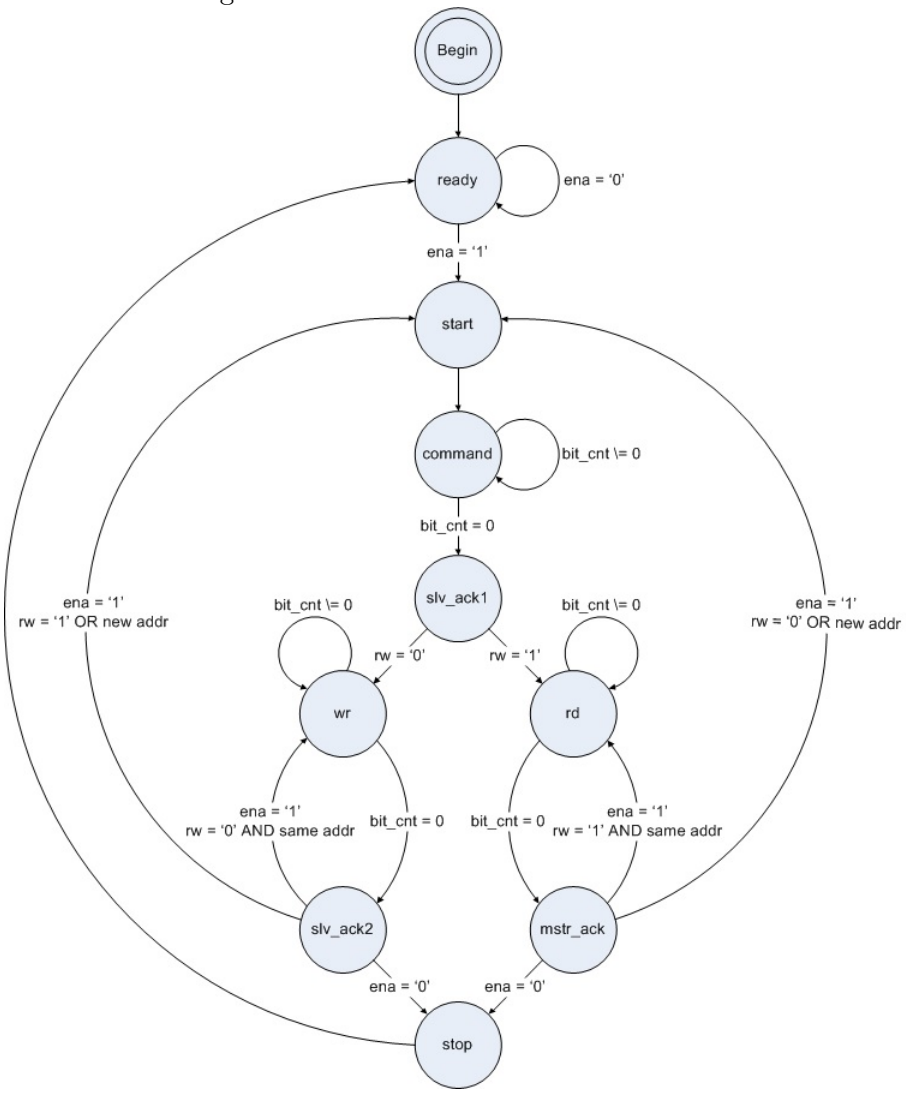
Figure 4.5: Data Frame used in I^{2}C Module



4.2.4.2 I^{2}C State Machine

The I^{2}C protocol can be viewed as working in states. First, a start state after the module is set to initialize. Then, the state machine goes through a couple of stages to complete the communication.

Figure 4.6: State Machine of I^2C Module



The machine stays in 'ready' state when **ena** is not set. If **ena** got set, the machine transition to 'start' state which generate the start condition on I^2C lines. Then, it moves to 'command' state. Depending on the **addr** and **wr**, the machine transmit the peripheral address and the read/write command to the peripheral. After all the bits have been transmitted, it transitions to 'slv_ack1' state, where the machine reads in the acknowledge bit send back by the peripheral device. Then, depending on the read/write command, the machine moves to respective states, and read or write data accordingly. Then it reads or writes acknowledge bit. Then, if **ena** is set for another round of operation, if the read/write command has been changed from the previous command or the address has been modified, the machine transitions to 'start' state and repeats the operation. Otherwise, the machine cycles as seen in figure 4.6. If **ena** is cleared, it transitions to 'stop' state, which generates stop condition on I^2C lines. Then, the machine went back to 'ready' state.

4.2.5 Implementation

Implemetation can be found on the GitHub listed in A.1. The controller is implemented as VHDL module. The implementation can be accessed through the following GitHub repo: https://github.com/kaung-minkhant/risc-v-micro/tree/master/peripherals/i2c.

4.2.5.1 Controller Implementation

The implementation can be divided in three sections. Each will be explained in detail below. The operation will be demonstrated visually in testing section with simulation waveforms.

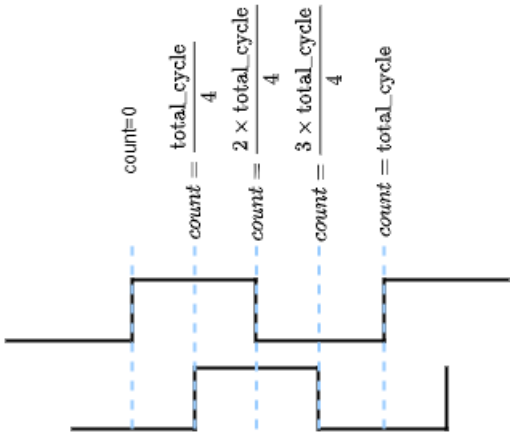
Declaration The first section declares the signal to be used in the implementation and most importantly, the state machine's states. The notable feature of the implementation is that, in this module, the clock for **scl** and the data clock, which is used for loading and updating data is generated internally using **clk**.

Clock Generation The second section generates two clock signals: **scl_clk** and **data_clk**. **scl_clk** is used for **scl** while **data_clk** is used for data updates. These two signals are set to be 90°out of phase. One of the reason is that, **scl** is received by the peripheral device and is used for data sampling. Thus, data must be ready and stable before the rising edge of **scl**. Counters are utilized here. The clock cycles need for the count is calculated based on the system clock frequency and the desired bus frequency with the following formular:

$$\text{Number of clock cycle} = \frac{\text{system clock frequency}}{\text{bus frequency}}$$

. To generate the 90°phase shift, number of clock cycle is divided by 4 to generate four regions of operation as in figure 4.7. The support for peripheral clock stretching can be seen here, as the count has been stopped when the peripheral is stretching the clock.

Figure 4.7: Clock cycle division for 90°Phase shift



State Machine The third section is the state machine itself. Here various states and signals have been set and cleared. Asynchronous reset has been utilized. As can be seen in the implementation, the state machine runs on internal data clock, instead of system clock. The detail implementation of each state is described below.

ready state: This is the state to which the machine normally sits in. If the transaction is initiated, i.e **ena**=1, the module is set to busy, and initialize all the required data, such as peripheral address, read/write command and the data to be transmitted. Then the state is transitioned to start state.

start state: This state generates start condition for the controller. **scl_ena** is used to enable **scl** output from the controller, because the controller is the one that is transmitting the start condition. The notable piece of code that needs to keep in mind when understanding the implementation is the mux implemented for the **sda** using **sda_ena_n** shown in figure 4.8. As can be seen, **sda** depends upon **sda_ena_n**, which in turn depends on states. **sda_ena_n** is set to generate the correct signal depending on the state, as will be illustrated in the testing of this module. The machine then moves to the command state.

Figure 4.8: Implementation of **sda** mux

```
--set sda output
WITH state SELECT
  sda_ena_n <= data_clk WHEN start, --generate start condition
  NOT data_clk WHEN stop, --generate stop condition
  sda_int WHEN OTHERS; --set to internal sda signal

--set scl and sda outputs
scl <= '0' WHEN (scl_ena = '1' AND scl_clk = '0') ELSE
  'Z';
sda <= '0' WHEN sda_ena_n = '0' ELSE
  'Z';
```

command state: In this state, the machine transmits the address and the read/write command to the peripheral. The notation implementation is that the data is transmitted with the most significant bit first, as per specification. After all the bits are transmitted, then the machine moves to 'slv_ack1' state.

slv_ack1 state: This is one of the acknowledge states. During this state, the machine decides which states to transition based on the read/write command. If the command is write command, **sda** line is used by the controller. If the command is read command, **sda** line is released so that the peripheral can use the line. The implementation can be better understood when keep in mind of the **sda** mux described above in figure 4.8. The notable piece of code is how the machine handles and checks for the acknowledge error shown in figure 4.9. If the peripheral successfully acknowledges, the **sda** will be pulled low by the peripheral device. In the 'slv_ack1' state, this condition is checked for the error.

wr state: If the command is write command, this state is used after 'slv_ack1' state. Here, the data to be sent is transmitted. If the transmission is completed, then the machine moves to 'slv_ack2' state.

rd state: If the command is read command, this state is used after 'slv_ack1' state. Here the read is performed for all the bits. This can be seen in figure 4.9. After all the bits have been received, depending on **ena**, acknowledge or no-acknowledge bit is sent. If the user wants to read

Figure 4.9: Implementation for checking Acknowledge Error

```
--reading from sda during scl high (falling edge of data_clk)
IF (reset_n = '0') THEN --reset asserted
  ack_error <= '0';
ELSIF (data_clk'EVENT AND data_clk = '0') THEN
  CASE state IS
    WHEN start =>
      IF (scl_ena = '0') THEN --starting new transaction
        ack_error <= '0'; --reset acknowledge error output
      END IF;
    WHEN slv_ack1 => --receiving slave acknowledge (command)
      IF (sda /= '0' OR ack_error = '1') THEN --no-acknowledge or previous no-acknowledge
        ack_error <= '1'; --set error output if no-acknowledge
      END IF;
    WHEN rd => --receiving slave data
      IF (sda = '0') THEN
        data_rx(bit_cnt) <= '0';
      ELSE
        data_rx(bit_cnt) <= '1';
      END IF;
      -- data_rx(bit_cnt) <= sda; --receive current slave data bit
    WHEN slv_ack2 => --receiving slave acknowledge (write)
      IF (sda /= '0' OR ack_error = '1') THEN --no-acknowledge or previous no-acknowledge
        ack_error <= '1'; --set error output if no-acknowledge
      END IF;
    WHEN OTHERS =>
      NULL;
  END CASE;
END IF;
```

from the same address again, then **ena** is set, thus acknowledge bit is sent. If the user wants to stop reading, then no-acknowledge bit is sent. This is done as per I^2C specifications.

slv_ack2 state: This state is used after 'wr' state. Within this state, if the user wants to write to the same address, the machine goes back to 'wr' state. If the user wants to change the command or write to different address, the new data is latched in and the machine moves to 'start' state. If **ena** is still cleared, then the communication is finished and the machine transitions to 'stop' state. The acknowledge error is checked the same way in figure 4.9.

mstr_ack state: This state is used after 'rd' state. Within this state, if the user wants to read from the same address, the machine goes back to 'rd' state. If the user wants to change the command or read from different address, the new data is latched in and the machine moves to 'start' state. If **ena** is still cleared, then the communication is finished and the machine transitions to 'stop' state.

stop state: This state generates the stop condition for the controller. The correct output for **sda** is generated within the mux in figure 4.8.

4.3 Serial Peripheral Interface (SPI)

This Serial Peripheral Interface (SPI) module is written for Field Programmable Logic Array (FPGA) implementation with VHDL. This module can be duplicated and controlled using external logic or controllers. Big-edianess is used according to specifications. This module can be initialized

with desired system clock and desired bus clock frequency and spi mode. This SPI module support 4 standards mode, however, default mode 0 will be the default. These modes are defined with CPOL and CPHA bits.

Table 4.3: Modes of SPI

Mode	CPOL	CPHA	Clock Polarity in Idle State	Data Sampling and Output
0	0	0	Logic Low	Data sampled on rising edge and shifted out on the falling edge
1	0	1	Logic Low	Data sampled on the falling edge and shifted out on the rising edge
2	1	0	Logic High	Data sampled on the rising edge and shifted out on the falling edge
3	1	1	Logic High	Data sampled on the falling edge and shifted out on the rising edge

4.3.1 Why use SPI

SPI is a synchronous protocol which also support full duplex data transfer. Due to the nature of the protocol, the controller and peripheral can transmit and receive data at the same time. Since the bus clock is generate only by the controller unlike bus clock in I^2C, the receiving peripheral hardware can be easily designed.

Here lists the advantages of SPI. [sparkfun and MIKEGRUSIN, 2018]

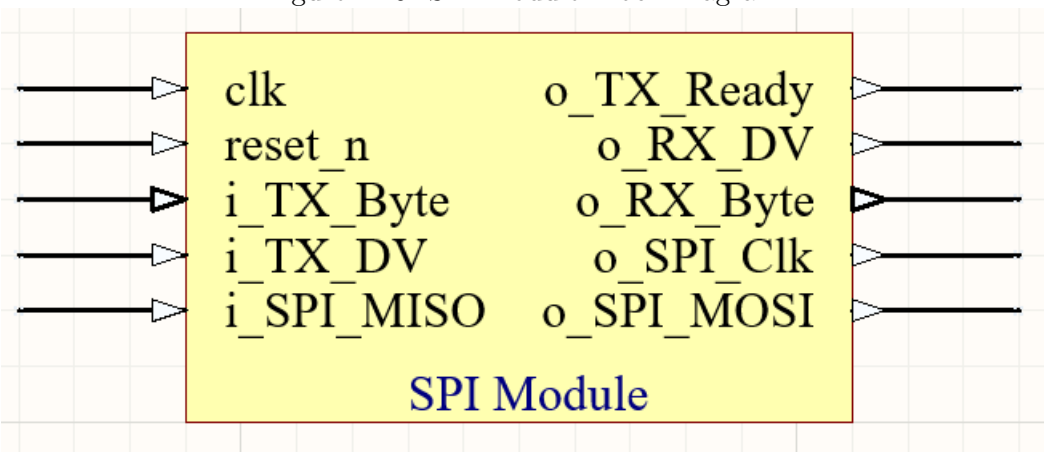
- It's faster than asynchronous serial
- The receive hardware can be a simple shift register
- It supports multiple peripherals

4.3.2 Block Diagram and I/O

Table 4.4: Input/Output Table for SPI Module

Input Name	**Function**	**Output Name**	**Function**
clk	main clock input	**o_TX_Ready**	indicate whether module is ready for next byte
reset_n	asynchronous reset	**o_RX_DV**	received data is valid to use
i_TX_Byte	Data to transmit	**o_RX_Byte**	Received data
i_TX_DV	Data is ready to transmit	**o_SPI_Clk**	SPI Clock Line
i_SPI_MISO	SPI MISO Line	**o_SPI_MOSI**	SPI MOSI Line

Figure 4.10: SPI Module Block Diagram



4.3.3 Usage

The SPI module only consists of one overall module, which is called a master or controller. The system clock of desired frequency need to be feed through **clk**. This module can be reset using active low **reset_n**.

This module does not contain Chip Select Pin. Thus, the designer will need to add chip select pin during his own overall implementation. The number of chip select pin is not limited as required by the number of peripherals that are to be connected to the bus.

To operate this module, if chip select is implemented, as per SPI standard, this pin needs to be pulled low to select a particular peripheral. Then, the data to be transmitted is placed on **i_TX_Byte**. After the data has been placed, **i_TX_DV** will need to be pulsed high in order to start the transmission. During transmission, **o_TX_Ready** will be pulled low, indicating the module is transmitting. After, successful transmission, **o_TX_Ready** will be pulled high, at that point, new data can be placed and repeat the transmission.

Due to the full-duplex nature of SPI Protocol, during transmission, reception of data also takes place at the same time. After successful reception, **o_RX_DV** will be pulsed high, and the received data will be available on **o_RX_Byte**. Thus, if conditional data reception is intended, first, data is needed to be transmitted, then discard the received data, then empty data is transmitted then read the received data.

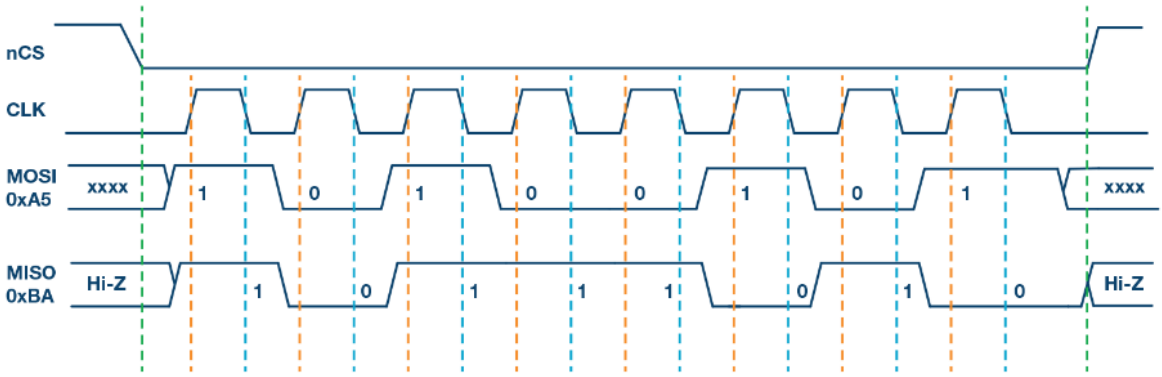
4.3.3.1 Initialization

The module has three generic variable which needs to be set according to the desired system clock frequency, bus clock frequency and SPI operating mode.
SPI_MODE : SPI Operating Mode
INPUT_CLK : system input clock in Hz
BUS_CLK : desired bus clock in Hz

4.3.4 SPI Architecture

The most common SPI operating Mode is mode 0, CPOL=0 and CPHA=0 where the clock is normally set to low, and upon transmission, the first clock edge is a rising edge. Data sampled on rising edge and shifted out on the falling edge of the clock.

Figure 4.11: SPI Mode 0, CPOL = 0, CPHA = 0: CLK idle state = low, data sampled on rising edge and shifted on falling edge



This mode is commonly used with most sensors, thus default mode is set to 0. The transmission and reception of data is simple in SPI protocol. The challenging part is to support all 4 modes of spi.

4.3.5 Implementation

Implemetation can be found on the GitHub listed in A.1. The controller is implemented as VHDL module. The implementation can be accessed through the following GitHub repo: https://github.com/kaung-minkhant/risc-v-micro/tree/master/peripherals/spi.

4.3.5.1 Controller Implementation

The implementation can be divided in four sections. Each will be explained in detail below. The operation will be demonstrated visually in testing section with simulation waveforms.

Clock Generation and Data Latch This section generates the requires clock edges at the desired bus frequency and latch the transmit data. As per specification, the number of clock cycle for the bus is limited by the number of bits to transmit. Typically 8 bit data is used, thus, there will be 8 clock cycle per transaction. To support all SPI modes, clock edges are also generated. With 8 clock cycles, there will be 16 clock edges, namely leading edges and trailing edges. This clock generation is synchronized with system clock.

To get the required bus clock frequency, counters are used. The count for each bit is calculated using the following formula:

$$\text{count} = \frac{\text{system clock frequency}}{\text{bus frequency}}$$

.

Using these leading and trailing clock edges, all the required SPI modes can be implemented. Clock Edges will be tested with simulation.

Then there is the latch for transmit data. As soon as **i_TX_DV** is pulsed, the data is latched into **r_TX_Byte**.

MOSI Process This section concerns with the process for transmission on MOSI line. Depending on SPI Mode, using the leading and trailing edges, the data is transmitted accordingly. Counters are used to track the number of bits already transmitted. Big-endianess can be seen in this case. The process can be seen in figure 4.12.

Figure 4.12: MOSI Process

```
MOSI_Data : PROCESS (clk, reset_n)
BEGIN
    IF reset_n = '0' THEN
        o_SPI_MOSI <= '0';
        r_TX_Bit_Count <= "111"; -- Send MSB first -- 7
    ELSIF rising_edge(clk) THEN
        -- If ready is high, reset bit counts to default
        IF o_TX_Ready = '1' THEN
            r_TX_Bit_Count <= "111"; -- 7

            -- Catch the case where we start transaction and CPHA = 0
        ELSIF (r_TX_DV = '1' AND w_CPHA = '0') THEN
            o_SPI_MOSI <= r_TX_Byte(7);
            r_TX_Bit_Count <= "110"; -- 6
        ELSIF (r_Leading_Edge = '1' AND w_CPHA = '1') OR (r_Trailing_Edge = '1' AND w_CPHA = '0') THEN
            r_TX_Bit_Count <= r_TX_Bit_Count - 1;
            o_SPI_MOSI <= r_TX_Byte(to_integer(r_TX_Bit_Count));
        END IF;
    END IF;
END PROCESS MOSI_Data;
```

MISO Process This section concerns with the process for receiving from MISO line. Depending on SPI Mode, using the leading and trailing edges, the data is sampled accordingly from MISO line. Counters are again used to track the number of bits received. Big-endianess can be seen. Since these implementations are seperated process, they works in parallel. Thus, full-duplex data transfer can be achieved relatively fast. The process can be seen in figure 4.13.

Figure 4.13: MISO Process

```
MISO_Data : PROCESS (clk, reset_n)
BEGIN
    IF reset_n = '0' THEN
        o_RX_Byte <= X"00";
        o_RX_DV <= '0';
        r_RX_Bit_Count <= "111"; -- Starts at 7
    ELSIF rising_edge(clk) THEN
        -- Default Assignments
        o_RX_DV <= '0';

        IF o_TX_Ready = '1' THEN -- Check if ready, if so reset count to default
            r_RX_Bit_Count <= "111"; -- Starts at 7
        ELSIF (r_Leading_Edge = '1' AND w_CPHA = '0') OR (r_Trailing_Edge = '1' AND w_CPHA = '1') THEN
            o_RX_Byte(to_integer(r_RX_Bit_Count)) <= i_SPI_MISO; -- Sample data
            r_RX_Bit_Count <= r_RX_Bit_Count - 1;
            IF r_RX_Bit_Count = "000" THEN
                o_RX_DV <= '1'; -- Byte done, pulse Data Valid
            END IF;
        END IF;
    END IF;
END PROCESS MISO_Data;
```

SPI Clock Process This process generate clock signals on SPI Clock line. The clock alignment is done using the system clock. The correct clock polarity is generated using CPOL bit.

Chapter 5

Simulation Testings

This chapter documents the testings performed on the implemented modules. ModelSim is used for simulation testings.

5.1 Testings of UART Modules

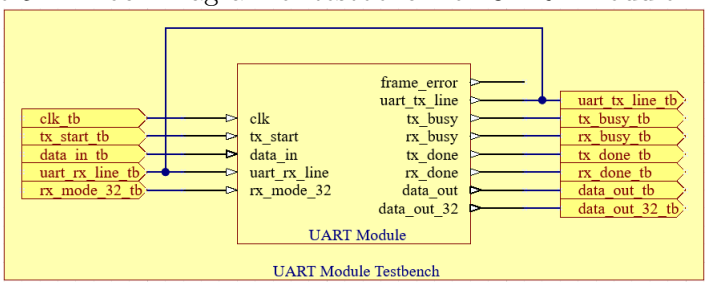
The first test for the UART module is performed using a data loop back test. This test requires the the transmitted data to be fed back to the receiving line, making a loop. The second test perform testing on detection of frame error for the receiving data. The follwing tests will be performed.

1. Transmitting same data 2 times with 32 bit mode off
2. Transmitting different data with 32 bit mode off
3. Transmitting same data 2 times with 32 bit mode on
4. Transmitting different data with 32 bit mode on
5. Testing detection of frame error
6. Baudrate Testing

5.1.1 Testbench Setup

Testbench for first test includes the transmitter module, transmitting data on the tx line and the tx line is connected into the rx data line, creating a loop.

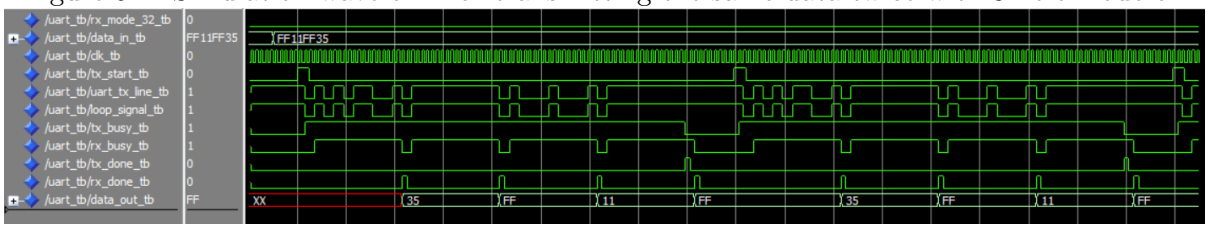
Figure 5.1: Block Diagram of testbench for UART Module Testing



5.1.2 Transmitting same data 2 times with 32 bit mode off

In this loop back test, the module is set to transmit the same data 2 times, one after another. 32 bit receive mode is turned off (**rx_mode_32_tb** = 0), thus, **rx_done_tb** is pulsed high after receiving every byte without frame error. In figure 5.2, it can be seen transmitting the correct data and the same data has been received without frame error. **rx_done_tb** pulsing pattern can also be seen. Little-endian effect can be seen clearly.

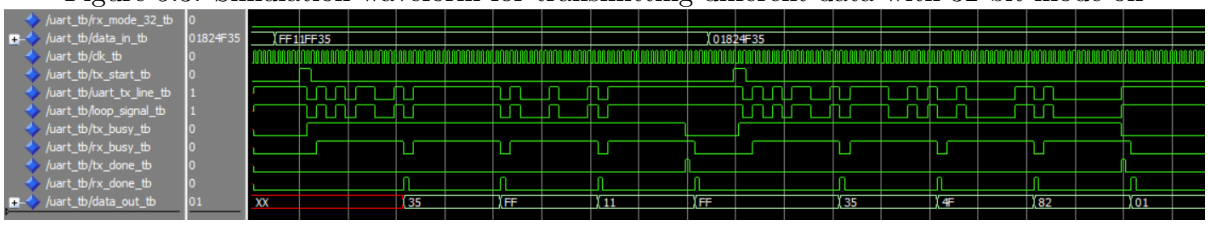
Figure 5.2: Simulation waveform for transmitting the same data twice with 32 bit mode off



5.1.3 Transmitting different data with 32 bit mode off

In this loop back test, the module is set to transmit the two different data, one after another. 32 bit receive mode is turned off (**rx_mode_32_tb** = 0), thus, **rx_done_tb** is pulsed high after receiving every byte without frame error. In figure 5.3, it can be seen transmitting the correct data and the same data has been received without frame error. **rx_done_tb** pulsing pattern can also be seen. Little-endian effect can be seen clearly.

Figure 5.3: Simulation waveform for transmitting different data with 32 bit mode off

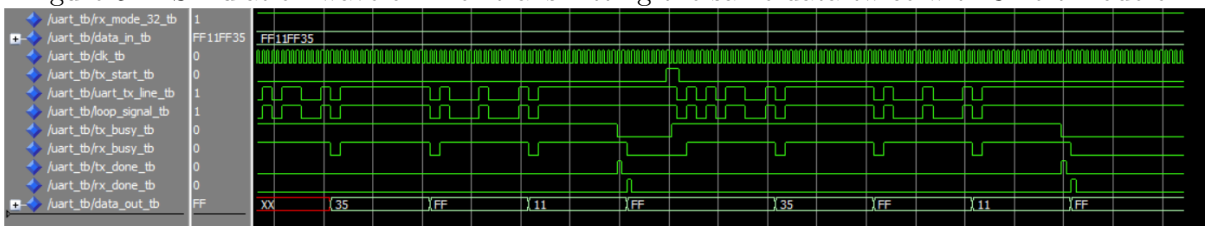


5.1.4 Transmitting same data 2 times with 32 bit mode on

In this loop back test, the module is set to transmit the same data 2 times, one after another. 32 bit receive mode is turned on (**rx_mode_32_tb** = 1), thus, **rx_done_tb** is pulsed high only after

receiving four bytes without frame error. In figure 5.4, it can be seen transmitting the correct data and the same data has been received without frame error. **rx_done_tb** pulsing pattern can also be seen. Little-endian effect can be seen clearly.

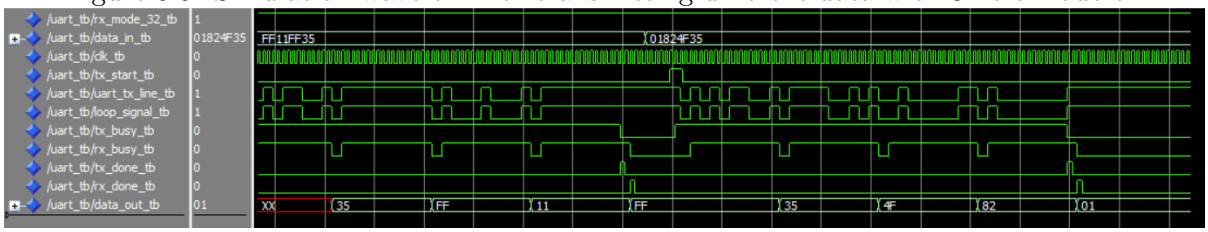
Figure 5.4: Simulation waveform for transmitting the same data twice with 32 bit mode on



5.1.5 Transmitting different data with 32 bit mode on

In this loop back test, the module is set to transmit the two different data, one after another. 32 bit receive mode is turned on (**rx_mode_32_tb** = 1), thus, **rx_done_tb** is pulsed high after receiving four bytes without frame error. In figure 5.5, it can be seen transmitting the correct data and the same data has been received without frame error. **rx_done_tb** pulsing pattern can also be seen. Little-endian effect can be seen clearly.

Figure 5.5: Simulation waveform for transmitting different data with 32 bit mode on



Usernote

As can be seen in the simulation wave forms, in order to read in the received data, the falling edge of the **rx_done_tb** has to be used to avoid metastability.

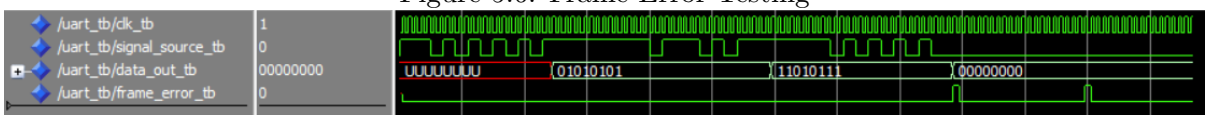
5.1.6 Testing detection of frame error

In this test, only the receiver is utilized to test frame error detection. A series of bit data will be generated by the testbench, without frame error first, and then with the frame error. In figure 5.6, the first data packet is sent without frame error. The second one is sent without start bit and the third one is without stop bit. As can be seen, the module checks for each received frame and respond the error through **frame_error**.

5.1.7 Baudrate Testing

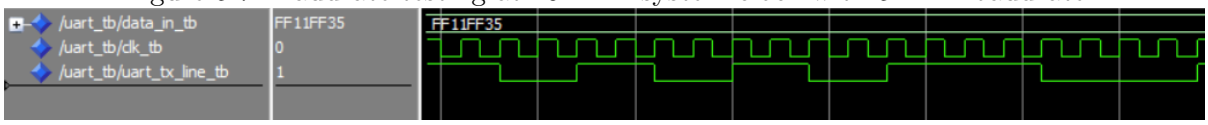
In this test, baudrate testing is performed. Two frequencies are used for this UART Module. Input system clock frequency delivered on **clk** and the required baudrate set by the user when initializing

Figure 5.6: Frame Error Testing



the module. Note that baudrate must always be smaller than input system clock frequency. In this test only, input system frequency of 10 MHz and baudrate of 5 MHz is used. This will translate to transmitting each bit every two system clock cycles as can be seen in figure 5.7.

Figure 5.7: Baudrate testing at 10 MHz system clock with 5 MHz baudrate



5.2 Testings of I^{2}C Module

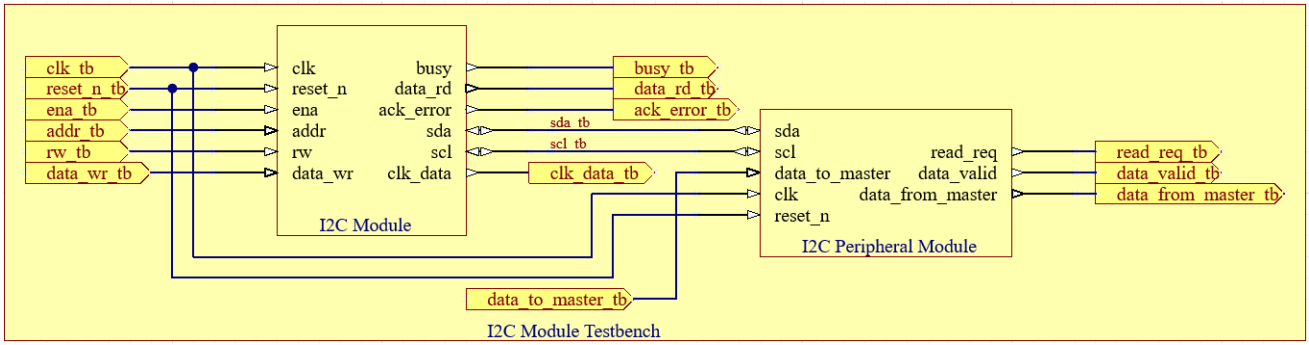
Various tests will be performed on the I^{2}C controller module. Unlike UART Module testings, we cannot do data loop back tests. A peripheral device needs to be built in order to simulate the overall operation. The implementation of this peripheral can be found on A.4. The following tests will be performed.

1. Testing of Start and Stop Conditions
2. Transmitting Data Once
3. Transmitting Three Data Consecutively
4. Transmitting Two Data to Different Peripheral Devices
5. Reading Data Once
6. Reading Two Data Consecutively
7. Reading Two Data from Different Peripheral Devices
8. A Read after a Write
9. Testing Peripheral Device Acknowledge

5.2.1 Testbench Setup

The testbench is setup with I^{2}C controller and I^{2}C peripheral device. The necessary signals will be generated within the testbench.

Figure 5.8: Block Diagram of testbench for I^{2}C Module Testing

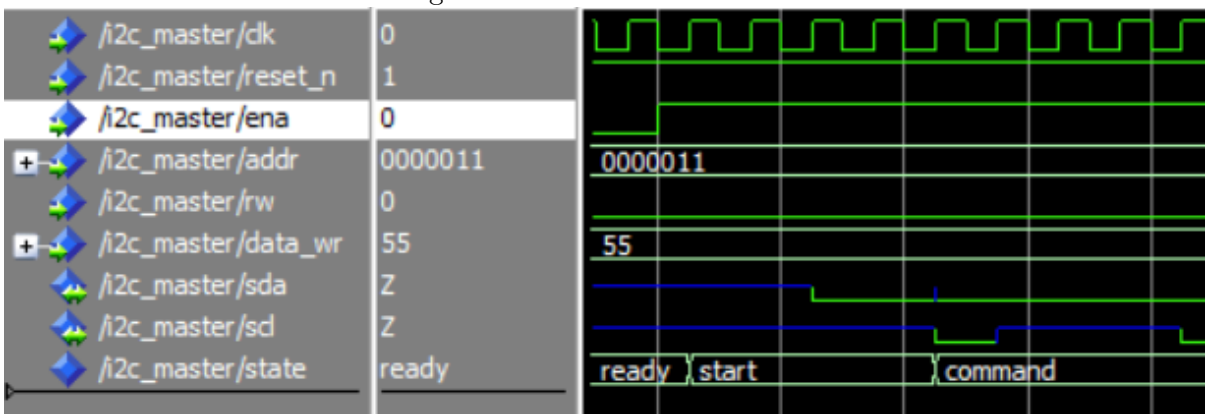


5.2.2 Testing of Start and Stop Conditions

In this test, the controller is set to transmit some data, and the goal of this test is to check the start and stop conditions are correctly generated. The settings for this test are **rw** = 0, **data_wr** = 0x55, **addr** = 0b000011. **ena** will be pulsed to initiate the communication.

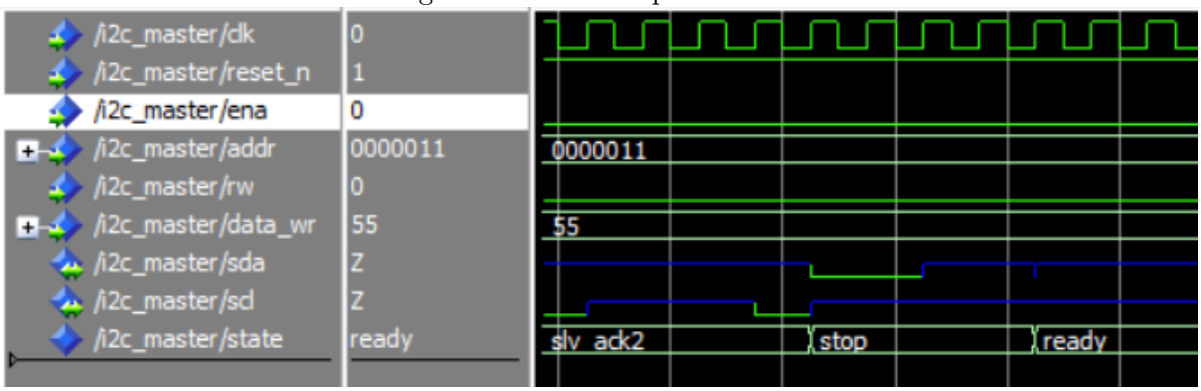
In figure 5.9, it can be observed then the start condition is correctly transmitted. As per I^{2}C specifications, to have a correct start condition, **sda** needs to go low first, then **scl** go low.

Figure 5.9: I^{2}C Start Condition



In figure 5.10, it can be observed then the stop condition is correctly transmitted. As per I^{2}C specifications, to have a correct stop condition, **scl** needs to go low first, then **sda** go low.

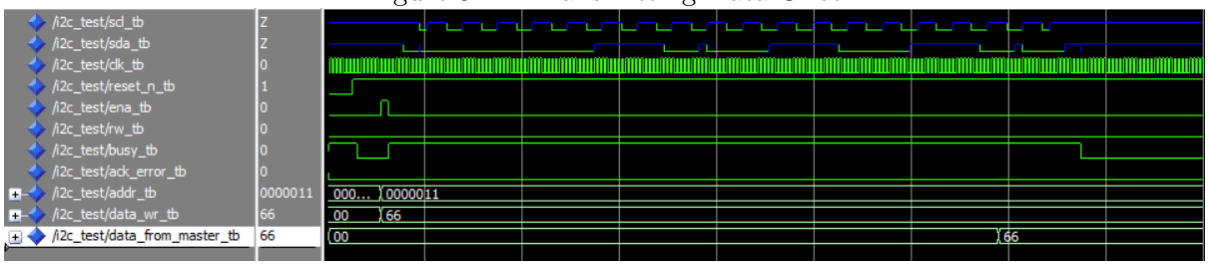
Figure 5.10: I^{2}C Stop Condition



5.2.3 Transmitting Data Once

In this test, one data is sent to the peripheral device once. This test is to check the operation of write for just one data.

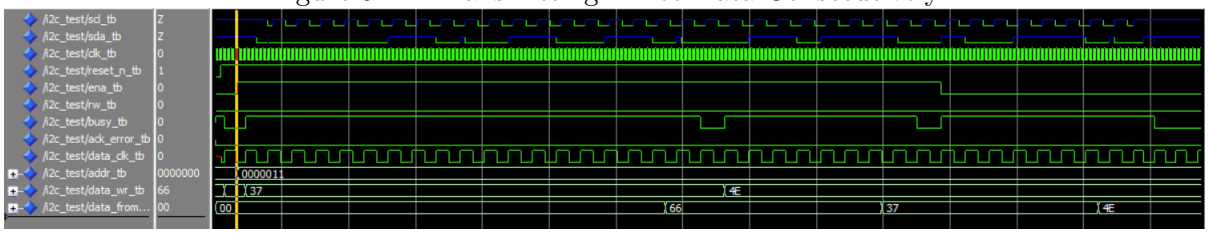
Figure 5.11: Transmitting Data Once



5.2.4 Transmitting Three Data Consecutively

In this test, three pieces of data is transmitted consecutively to the same address. This will demonstrate the address is not sent more than once and the data are sent correctly.

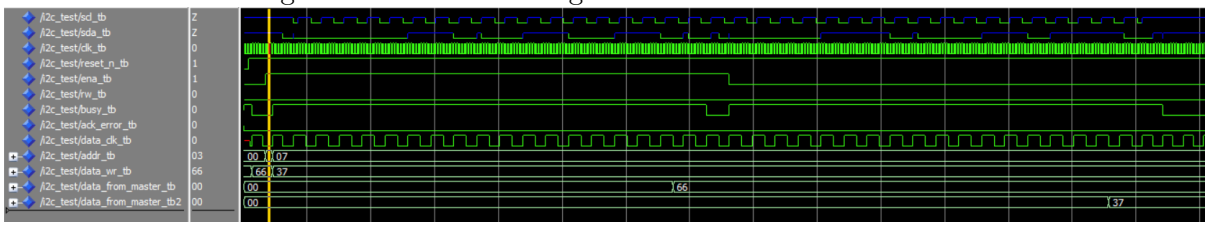
Figure 5.12: Transmitting Three Data Consecutively



5.2.5 Transmitting Two Data to Different Peripheral Devices

In this test, another peripheral device has been added with different address. The controller is set to transmit different data to these different peripheral devices. In this test, since there are two different address, the new address needs to be sent before transmitting another data.

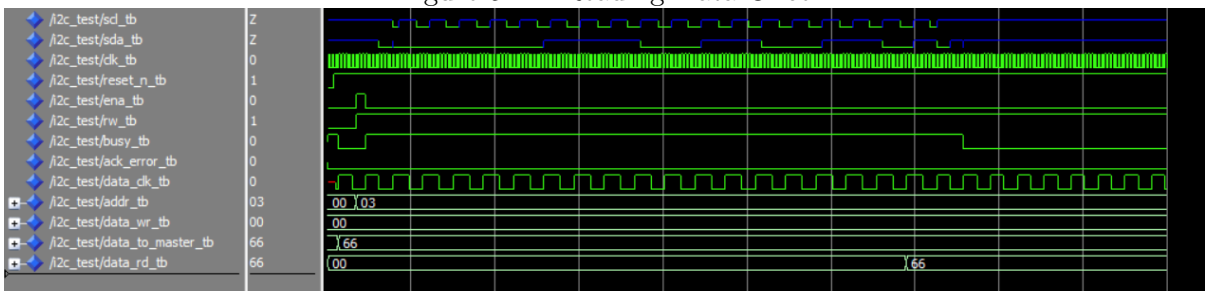
Figure 5.13: Transmitting two data to different devices



5.2.6 Reading Data Once

In this test, one piece of data is read from the peripheral device. **data_to_master** holds the data that is to be sent when requested. The received data is available on **data_rd**.

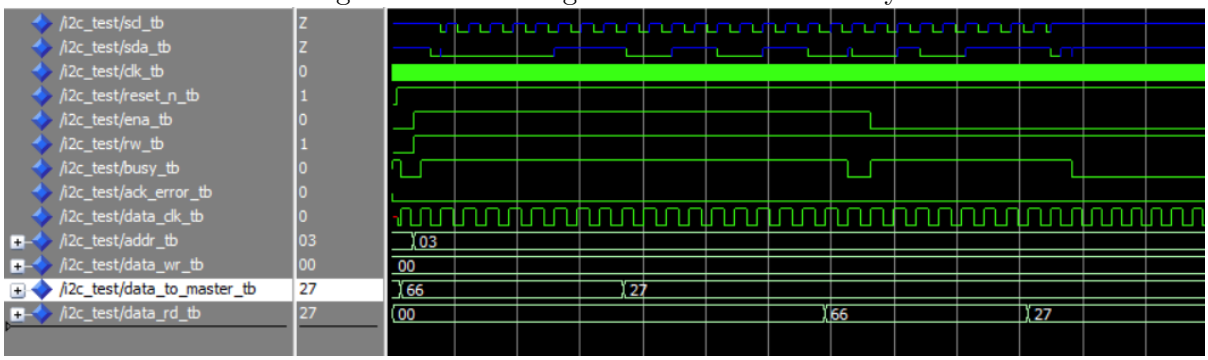
Figure 5.14: Reading Data Once



5.2.7 Reading Two Data Consecutively

In this test, two pieces of data is read from the same peripheral device. **data_to_master** holds the data that is to be sent when requested. The received data is available on **data_rd**. It can be seen that, since the address is the same, address is only sent once at the start of transaction.

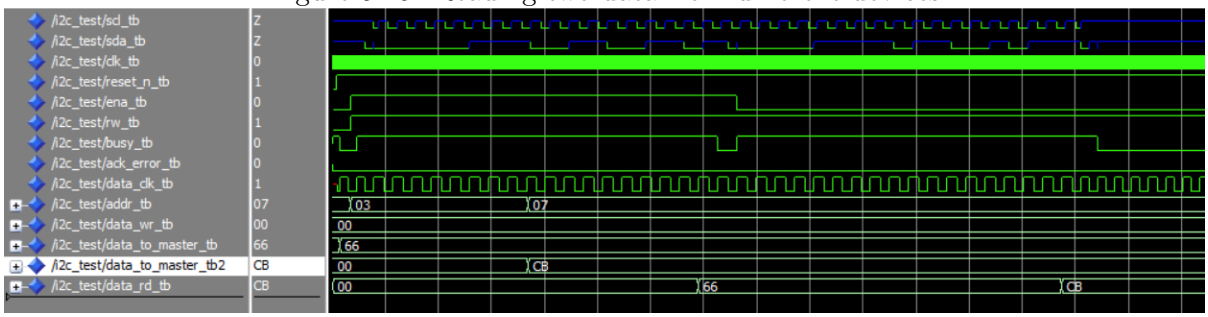
Figure 5.15: Reading Two Data Consecutively



5.2.8 Reading Two Data from Different Peripheral Devices

In this test, another peripheral device has been added with different address. The controller is set to read different data from these different peripheral devices. In this test, since there are two different address, the new address needs to be sent before reading another data.

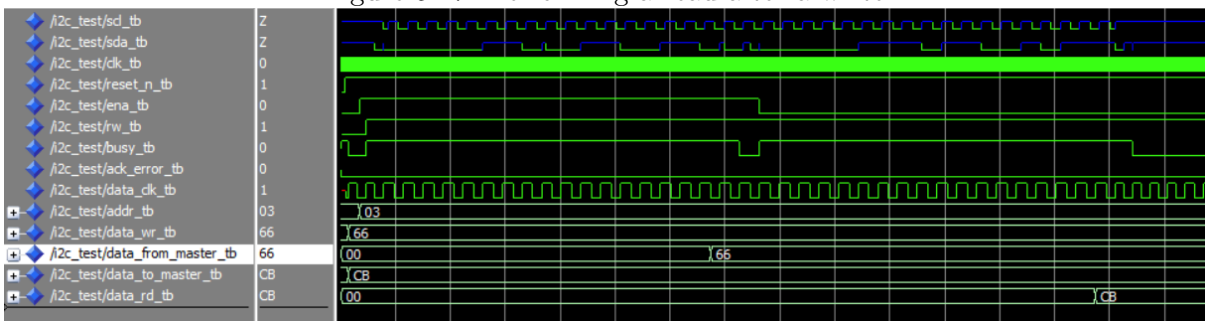
Figure 5.16: Reading two data from different devices



5.2.9 A Read after a Write

In this test, a standard I^2C communication is performed, which is a read after a write operation. The data to be written is available on **data_wr** and the data to be read from the peripheral device is available in **data_to_master_tb**. After successful communication, **data_from_master_tb** should have the data written, and **data_rd_tb** should contain the data read.

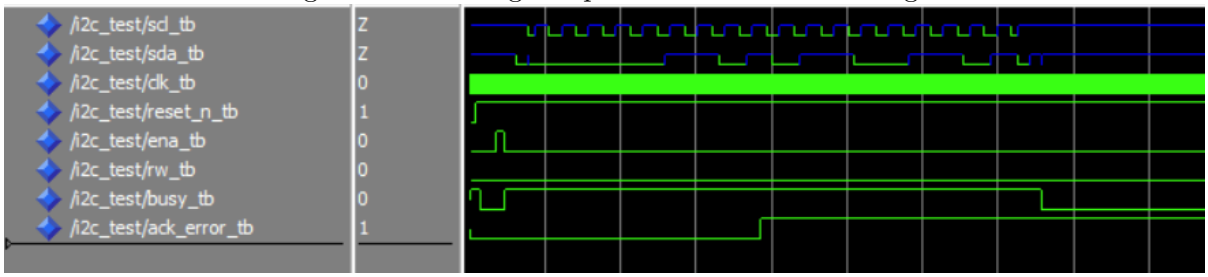
Figure 5.17: Performing a read after a write



5.2.10 Testing Peripheral Device Acknowledge

In this test, the detection for peripheral device failing to send acknowledge signal is tested. To achieve this, the code for sending acknowledge signal in the peripheral device implementation is removed. It can be seen from figure 5.18 that the error is maintained unless cleared by user.

Figure 5.18: Tesing Peripheral Device Acknowledge



5.3 Testings of SPI Module

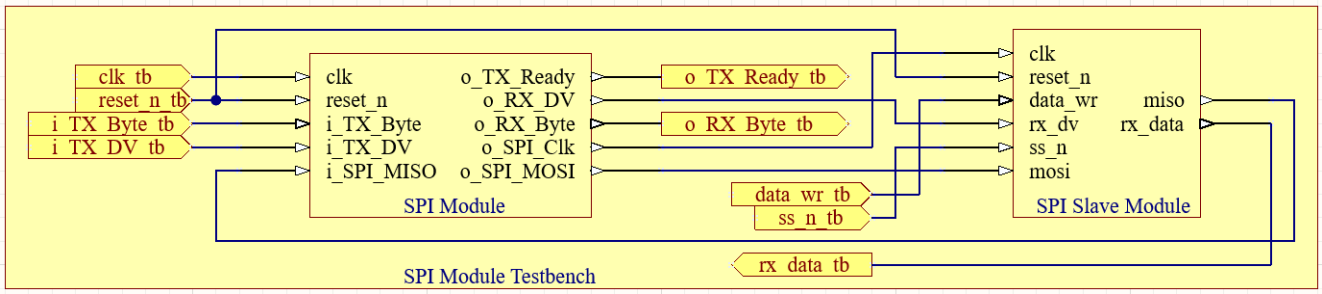
Various tests will be performed on theSPI controller module. Unlike UART Module testings, we cannot do data loop back tests. A peripheral device needs to be built in order to simulate the overall operation. The implementation of this peripheral can be found on A.5. The following tests will be performed.

1. Clock Edge Generation
2. Transmitting Data Once
3. Transmitting Data Twice
4. Reading Data Once

5.3.1 Testbench Setup

The testbench is setup with SPI controller and SPI peripheral device. The necessary signals will be generated within the testbench.

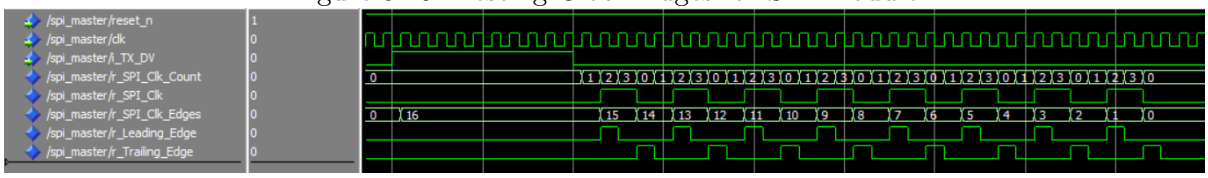
Figure 5.19: Block Diagram of testbench for SPI Module Testing



5.3.2 Clock Edge Generation

This test is to demonstrate the generation of the require clock edges and SPI clock for the module. The number of clock edges has to be 16, since there will be 8 bit data, so 8 clock cycles with 2 clock edges each. The counters can also be seen in figure 5.20

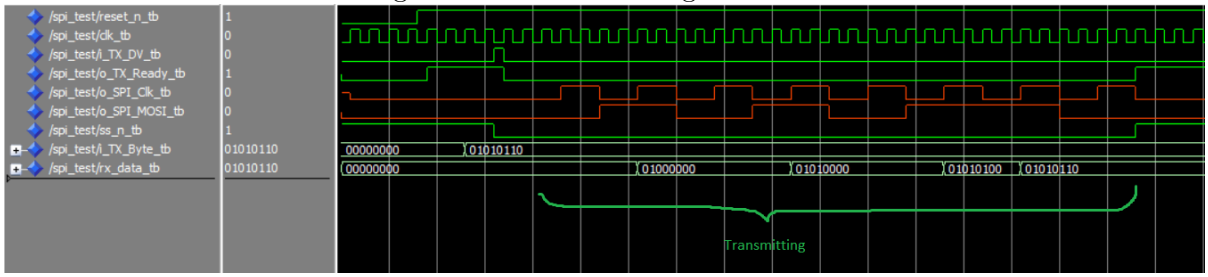
Figure 5.20: Testing Clock Edges for SPI Module



5.3.3 Transmitting Data Once

In this test, one data is sent to the peripheral device once. This test is to check the operation of write for one data.

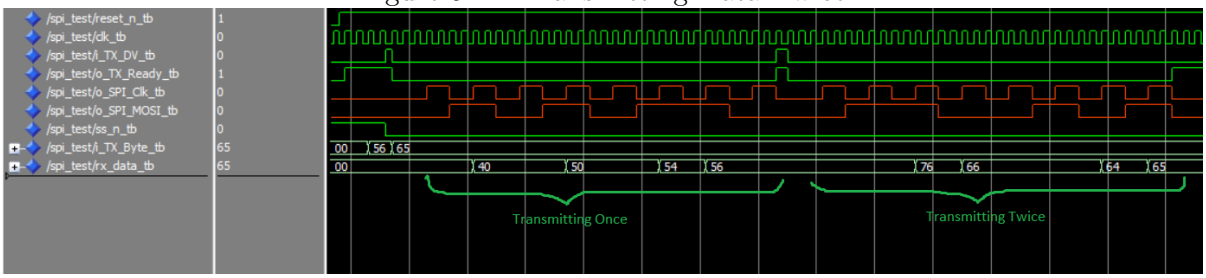
Figure 5.21: Transmitting Data Once



5.3.4 Transmitting Data Twice

In this test, two pieces of data are transmitted consecutively to the same peripheral device. To choose the desired peripheral device, respective **ss_n** pin is pulled low during transmission.

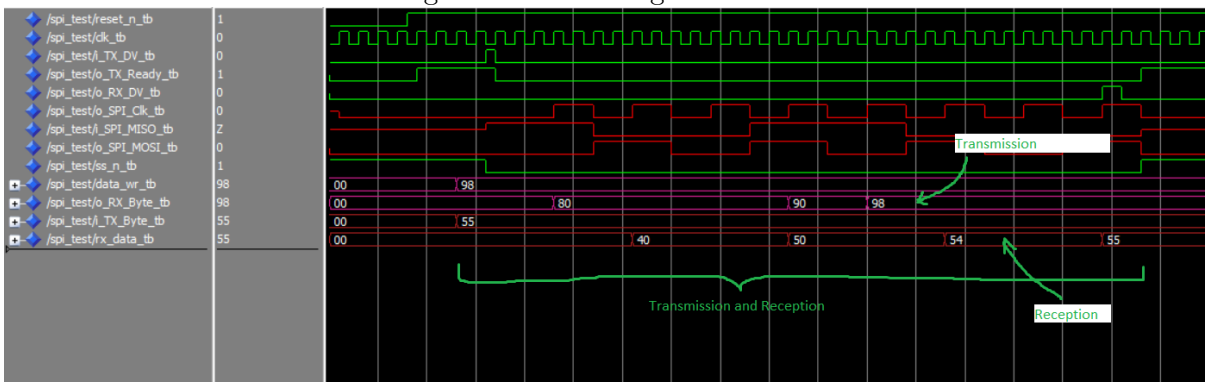
Figure 5.22: Transmitting Data Twice



5.3.5 Reading Data Once

In this test, a read is performed. Notable feature of SPI while reading is that it is not possible to read data without a write, due to simultaneous full-duplex nature of the protocol. Some form of data, either empty data or right data, need to be sent and the transmission and reception will take place at the same time on **mosi** and **miso** lines.

Figure 5.23: Reading Data Once



Chapter 6

Conclusion

To be filled when the research can be concluded.

Appendices

Appendix A

Some Appendix

A.1 Source Code

Source Codes can be found on the GitHub Page: Dummy Link

A.2 Requirements

A.2.1 Hardware and Software Requirements

Hardware and software requirements are as follow

1. Arrow Deca FPGA Development Board
2. Intel Quartus Prime Lite Software
3. ModelSim Lite simultation Software

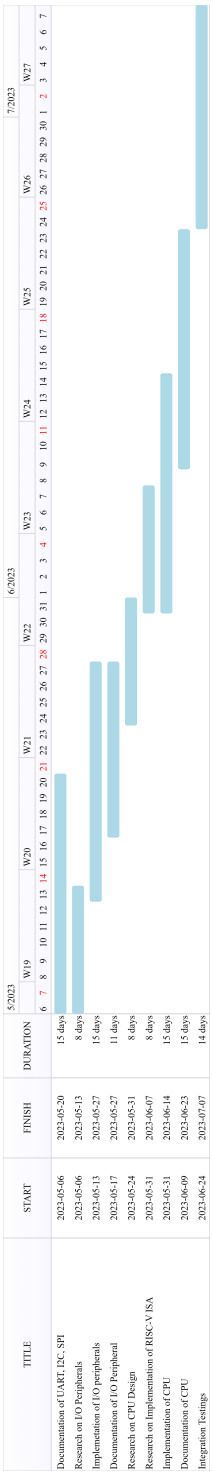
A.2.2 Skill Requirements

The following skills are required.

1. VHDL Programming
2. Computer Organization and Designe for RISC-V ISA
3. Micro-controller Peripheral Design
4. Micro-controller Communication Design
5. Assembly for RISC-V ISA

A.3 Timeline

Figure A.1: Timeline of the project



A.4 Implementation of I^{2}C Peripheral Device

A.5 Implementation of SPI Peripheral Device

Bibliography

[Brisk, 2017] Brisk, D. (2017). Ri5cy: A risc-v isa-based energy-efficient processor generator. https://www.researchgate.net/publication/321340163_Experimental_Study_on_Water_Sensitivity_Difference_Based_on_Oiliness_of_Porous_Medium_Rock.

[Gurkaynak, 2019] Gurkaynak, F. K. (2019). pulpino. https://github.com/pulp-platform/pulpino.

[Lee, 2016] Lee, Y. (2016). Risc-v rocket chip soc generator in chisel. https://riscv.org/wp-content/uploads/2015/01/riscv-rocket-chip-generator-workshop-jan2015.pdf.

[Papon, 2023] Papon, C. (2023). Vexriscv: A 32-bit risc-v cpu implementation in spinalhdl. https://github.com/lowRISC/ibex.

[Pena and Legaspi, 2020] Pena, E. and Legaspi, M. G. (2020). Uart: A hardware communication protocol understanding universal asynchronous receiver/transmitter. https://www.analog.com/en/analog-dialogue/articles/uart-a-hardware-communication-protocol.html#:~:text=The%20UART%20port%20provides%20a,half%20or%20two%20stop%20bits. Accessed: 2023-05-19.

[sparkfun and MIKEGRUSIN, 2018] sparkfun and MIKEGRUSIN (2018). Serial peripheral interface (spi). https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi/all. Accessed: 2023-05-27.

[sparkfun and SFUPTOWNMAKER, 2015] sparkfun and SFUPTOWNMAKER (2015). I2c. https://learn.sparkfun.com/tutorials/i2c/all. Accessed: 2023-05-19.

[Wallentowitz, 2023] Wallentowitz, S. (2023). Ibex: An open-source, parameterizable risc-v processor core. https://github.com/lowRISC/ibex.

[Waterman et al., 2017] Waterman, A., Asanovic, K., and Inc., S. (2017). The risc-v instruction set manual volume i: User-level isa. https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf.

[Zhang et al., 2021] Zhang, Y., Guo, Z., Li, J., Cai, F., and Zhou, J. (2021). Annikacore: Risc-v architecture processor design and implementation for iot. https://ieeexplore.ieee.org/document/9651690.